Assignment 1: Multi-Core MapReduce

By Mohit Khattar, Jason Scot, and Rishab Chawla

Due Date: October 9, 2018

Architecture

There were 4 main components to our architecture for the application.
From start to end, our program takes a file with words or integers as input and sorts them with their respective counts using multiple threads or processes. The file can have any n number of words or integers. The program has the following order of execution.

- **Parsing.** This part of the program will take a list of words or integers, delete any unnecessary non-alphanumeric characters, and store it in a data structure.
- **Map.** This part of the program will take n words or integers and uses threads and processes to assign a count of 1 to every word or integer. Each thread or process will handle a certain number of words or integers and this all depends on the number of maps the user inputs.
- **Shuffle/Sort.** This part of the program will take the output from map and arrange all the words or integers in the correct order. This is equivalent to sorting. We do this so that reduce can combine similar words or integers.
- **Reduce.** This part of the program will take n words or integers and use threads and processes to combine counts for similar words. Each process and thread has to handle a certain number of words and this all depends on the number of reduces the user inputs.
- **Combine.** This part of the program does one last sweep and combines the counts for any words or integers that were missed in reduce due to similar words or integers being handled by different threads or processes.
- **Output.** This part of the program takes the final output from combine and writes it to a text file of the users choice.

Parsing

The parser's role was to split up the input into a series of tokens. The tokens were split based on non-alphanumeric characters. To store the tokens, we decided on a linked list structure due to the unknown number of words in the input. Also, since the number of tokens would be important for future components, we decided to implement a "container" structure for the linked list. This container would store the size of the linked list (equivalent to the number of tokens) and maintain a pointer to the head of the linked list.

One notable thing about the parser is the way it handles large tokens. In C, we have to allocate a fixed chunk of memory that we can read into. However, if the token overflows past the end of the chunk, we need a way to construct the full token. This was done by performing a concatenation on a “partial” string, which we used to represent the overflowed part of the token we were reading, with the part that we were currently reading in. This allowed for very large sized tokens to be read in without risk of a buffer overflow.

Note: we also used the delimiters posted on Sakai and \n and \r to check for new lines for parsing tokens.

Map

The role of map is to take all parsed tokens and assign a count of 1 to every token. The tokens were split up in parsing stage. Our map function calls a parsing function which will produce a linked list containing all the tokens with a count of 0. After parsing, we use the number of maps inputted by the user to build a TpTable or thread process table to tell us which process or thread will handle which nodes or tokens in the linked list. To do this, we use a function called determineMapSize, which returns an array of size number of maps and each index contains the number of tokens that each thread or process assigned to that index will handle. After this, we decide whether to use processes or threads to map our input.

If processes are chosen, then we call a separate function that creates an array named data_array that will represent our shared memory segment that the processes will read from and write to. We decided to use the shm_open system to open a shared memory segment, which returns a file descriptor that can be used by ftruncate to declare the number of bytes needed for the shared memory segment. Then, we set an unsigned char pointer to the beginning of the shared memory segment. Afterwards, we translate our linked list to an unsigned char array, which stores the number of bytes in the array, the number of words, the length of each linked list node, the count of the token and the token itself. Since it is unstable to use a linked list in shared memory, we decided to take this array approach for our mapping using processes. Afterwards, we copy all this data into the shared memory segment and spawn a bunch of processes. We assign each process a start_index and end_index that they are allowed to access to prevent any need for semaphores and barriers for synchronization. Each process will find the count for each word and set it equal to 1. Afterwards, we wait for all the processes to die, convert the array to a linked list, and unlink the shared memory segment.

If threads are used, then we remain with the linked list approach and use the thread process table earlier to manage what each thread will handle. We opted to use the pthread library of functions to create threads, join those threads, set the counts for each token to 1. We use mutex locks when we need to concatenate the tokens that the thread has to handle to the global linked list structure that is used by shuffle/sort.

Sort/Shuffle

In the sort/shuffle phase, our goal is to take the global linked list structure produced from map and use string comparison to reorder all tokens in a sorted order. To start, we call a function called build_reduce which takes the entire linked list input and converts it to an array of type Node to be used by the library quicksort function. This array holds pointers to the element of each token in the linked list. Depending on whether we are using word count or integer sort, we have built two separate comparison functions to be used by the library qsort function. In the case of word count, we use string comparison. Because we use the string comparison function, our output may be in a different order than the TAs output. The TA mentioned that this is okay.

After using qsort, we start building a reduce_table, which contains all the tokens that each respective process or thread will handle. We free the array from earlier and return the reduce_table to be used by reduce to combine the counts of similar words.

Reduce

The goal of reduce is to take the reduce_table generate from sorting/shuffling and further combine the same tokens and their counts. Since we already know which nodes each thread or process will handle, all we have to do is decide whether the user wants to user threads or processes.

If the user wants to use processes, then we create an array that contains the number of words that each process will reduce. This is of size number of reduces. We pass this into a function called reduce_processes, which stores the output into a global linked list containing all the tokens and their new counts. The function starts by converting the reduce_table into an array and creating a data_array that is used for shared memory. We call shm_open to retrieve a file descriptor of our shared memory segment. This will then be passed to ftruncate, which allocates enough bytes for the array to be copied. Before we copy the array, we create an unsigned char pointer using mmap that starts at the beginning of the shared memory segment. Then, we copy over the array of bytes into the data_array shared memory segment. Afterwards, we spawn processes using fork. Each process is given a start and end index that they can use to give it a sense of what region of the shared_memory segment they can access. Once that is completed, we call process_reduce to compare tokens and combine counts. The first thing that it does is gather information on the first token. This includes the count, the length of the token, and the starting index of the token. Then we use a while loop that goes till the end of the end_index given or till we have no more tokens left to check.

- The while loop finds the next token and gathers the same information as the first token.
- Then we use a while loop to compare both tokens linearly in the array. If one index in the token is not equal to the other index in the token, then we set a flag variable to indicate that the strings are unequal.
    - If the strings are unequal, then there are two cases:
        - If there are no more words for the process to handle, then it breaks out of the function.
        - If there are more words for the process to handle, then we move the pointer for the first token to the location of the second token. When we call the while loop again, the second token will be set to the next token. Then we rerun to check for equality.
    - If the string are equal, then we set the count for second token equal to the count of the first token + count of the second token. We then set the first token equal to zero and move the first token to the location of the second token. The while loop will rerun for the next comparison.

Once we have finished combining counts, we wait for all the processes to end, convert the array back to a linked list, and unlink the shared memory segment using shm_unlink. The function returns the final reduced_list.

If the user wants to use threads, then the first thing we do is setup which index in the reduce_table the thread is allowed to touch. Afterwards, we create a bunch of threads, then we join those threads for 0 to number of reduces iterations. The reduce_thread_handler function will take the given arguments and decide if it is integer sort or word count. If it is integer sort, there is no point in running reduce because we don’t need to worry about the counts for integers. After that, a new list is created and we traverse the segment of the list we are given to combine counts in nodex and store it in the new linked list. We use the string compare library function to compare two tokens. After that is completed, each thread will create a mutex lock to concatenate the smaller list with all the counts combined for the same tokens to a larger global reduced list. Then this list is returned for combine to handle one last time.

Combine

To account for the case that the same word is in two different buckets of the reduce_table, we do one final combine. This combine will give us our final linked list with all the counts ready for output.

Performance of Threads vs Processes

1. Wordcount with threads
A) map: 0.010852 sec
B) reduce: 0.001623 sec
C) total: 0.012475 sec

2. Wordcount with processes
A) map: 0.016244 sec
B) reduce: 0.006289 sec
C) total: 0.018677 sec

3. Sort with threads
A) map: 0.118990 sec
B) reduce: 0.039749 sec
C) total: 0.125558 sec

4. Sort with processes
A) map: 0.295283 sec
B) reduce: 0.214339 sec
C) total: 0.509622 sec

We can see that on average, threads are faster than processes. Even though we have to have a mutex lock to maintain shared memory resources, threads still run faster than processes. With processes, we have to worry about context switches and converting linked lists to array to prevent one process from overwriting another process's shared memory segment. Overall, the program runs fast on the input of 10000 numbers.

Output

To output to a file, we take the final linked list produced from combine and pass it into our output_list function along with the name of the file the user wants to output to and an indicator to tell us whether we are doing integer sort or word count. If we are doing integer sort, then we output all the numbers in order without the counts. If we are doing word count, then we output all the words with their counts. We do this by traversing the linked list and using the write system call to write to the file.

Difficulties

There were many difficulties with this assignment. I think understanding the mapreduce algorithm was very confusing at first because we were unsure of the role of map or the role of reduce. Everything seemed to do the same thing. The next difficulty was just making sure that all group members were on the same page. It was difficult to decide which data structures we were going to use for every part, however, we were able to come to a general consensus. Lastly, the concept of shared memory was very unclear to us. In Systems Programming, our professor had told us to never use processes in condition of shared memory. Therefore, a lot of research had to be done because our linked list structure was difficult to manage in shared memory. Our solution was to convert the linked list to an array and perform our operations of map and reduce on the byte level.

How to Run the Program

To run the program, we used the exact same command line argument structure as the instructions for the assignment.

For example, if you want to run integer sort with threads and 4 maps and 4 reduces, do the following:

```
./mapred --app wordcount --impl threads --maps 4 --reduces 4 --input test/sort_input.csv
--output test/heyo.txt
```

Arguments:
--app: tells the program whether you want to run integer sort or word count
--impl: tells the program whether to use processes or threads
--maps: tells the program the number of maps
--reduces: tells the program the number of reducers to use
--input: the file path to the input file
--output: the file path to the output file

GitHub Repository

The GitHub repository can be found at:
https://github.com/Rishab/Assignment-1-Map-Reduce.

The GitHub repository is set to private. If you want to request access to the repository, please email rishab.chawla@rutgers.edu.