CS5600 Final Project
December 16, 2021
By: Mitchell Lawson

A Concurrent Implementation of map

Project Details

I gained inspiration for this project from the OSTEP project repository located here: https://github.com/remzi-arpacidusseau/ostep-projects/tree/master/concurrency-mapreduce. Here I was given outline for the project and a header/interface file to give students some direction. The specific function and paradigm that I implemented is called the MapReduce, created by engineers at Google for large scale parallel data processing. MapReduce has the capability to make programming tasks on large-scale clusters easier for developers by alleviating the worries of managing parallelism, crashes and other issues associated with clusters of machines.

A simplification that is made for this project as suggested by the outline provided is creating MapReduce for a single machine in a concurrent manner. There are 3 learning objectives for this personal end of term project.

1. Learn about and understand the nature of the MapReduce paradigm.
2. Implement a correct and efficient version of MapReduce that utilizes threads.
3. Gain more experience and familiarity creating concurrent code.

I had to become more comfortable with concurrency in order to understand how to create threads, implement mutual exclusion with locks and how to use condition variables (although they were not really used here). There were 5 areas or chapters that were introduced near the end of the course that proved to be essential in understanding a lot of the concepts necessary to complete this project.

- Intro to Threads
- Threads API
- Locks
- Using Locks
- Condition Variables

What is MapReduce?

MapReduce can perform distributed and parallel computations using large datasets across a large number of nodes. A MapReduce job usually splits the input datasets and then processes each of them independently using a Map() function in a completely parallel manner… BUT in my implementation this is done in a concurrent manner. As such some of the performance gains due to parallelism are not fully realized.

I thought of the general structure of the MapReduce to consist of 4 areas of focus:

1. Mapper
    - Some logic is applied to ”n” data blocks spread across different node
    - The data blocks in my implementation are words and the nodes are files
2. Shuffle and Sort
    - The output of the mapper then gets sorted
    - Repeated values get removed and the values get grouped together based on keys
    - This was done using quick sort and a concurrent hash table/ map
3. Reducer
    - The output from the previous step is used in this phase
    - “Consolidates” the outputs and creates the finalized product
    - Counts the words in my case
4. Combiner (Optional)
    - Provides performance increase by combining inputs from the same node

An excellent figure that aided in my understanding of what was going on came from Google’s “MapReduce: Simplified Data Processing on Large Clusters” article that this project was centered around.

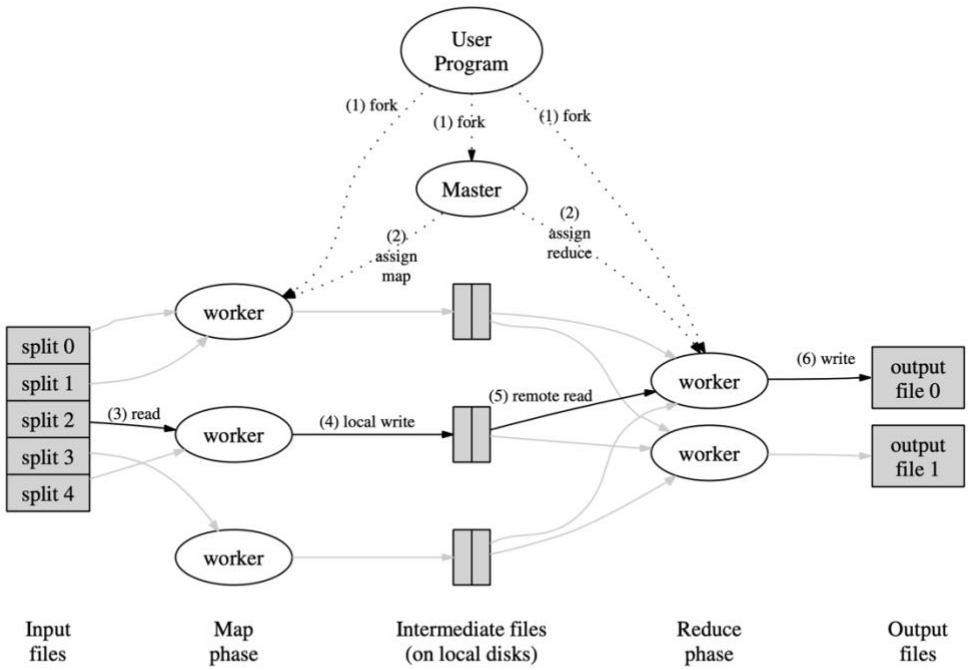

Figure 1: Execution overview

To simplify this explanation, let's look at this in terms of the word count of multiple documents. We have multiple files that are passed onto workers or "threads", where the Map() function is applied, the information is stored in some intermediate location (in some type of data structure), then the data is passed onto additional workers or "threads" with the Reduce() function being applied to combine the data and achieve a final result.

Code Structure (High Level Overview):

Concurrent_MapReduce – mapreduce.h

```h
#ifndef __mapreduce_h__
#define __mapreduce_h__

// Different function pointer types used by MR
typedef char *(*Getter)(char *key, int partition_number);
typedef void (*Mapper)(char *file_name);
typedef void (*Reducer)(char *key, Getter get_func, int partition_number);
typedef unsigned long (*Partitioner)(char *key, int num_partitions);

// External functions: these are what you must define
void MR_Emit(char *key, char *value);

unsigned long MR_DefaultHashPartition(char *key, int num_partitions);

void MR_Run(int argc, char *argv[],
        Mapper map, int num_mappers,
        Reducer reduce, int num_reducers,
        Partitioner partition);

#endif // __mapreduce_h__

```

```c
void Map(char *file_name) {
    FILE *fp = fopen(file_name, mode: "r");
    assert(fp != NULL);

    char* line = NULL;
    size_t size = 0;
    while (getline(&line, &size, fp) != -1) {
        char *token, *dummy = line;
        while ((token = strsep(&dummy, delim: " \t\n\r")) != NULL) {
            MR_Emit(token, value: "1");
        }
    }

    fclose(fp);
}

void Reduce(char *key, Getter get_next, int partition_number) {
    int count = 0;
    char *value;
    value = get_next(key, partition_number);
    while (value != NULL) {
        count++;
        value = get_next(key, partition_number);
    }
    printf("%s %d\n", key, count);
}

int main(int argc, char *argv[]) {
    MR_Run(argc, argv, Map, num_mappers: 10, Reduce, num_reducers: 10, MR_DefaultHashPartition);
}

```

We can delve into important functions and their purpose provided by the interface/header file, as well as the mapper and reducer functions used for word counting from files. Map() is simply used with a file name and Map(0 funs over the entire contents of the file.

For the given mapping function, strsep() splits the lines into tokens. Each token is supposed to be emitted (interesting new word I learned for the following function name) using the MR_Emit() function. This function takes two strings, a key and value... suggesting that a hash table/map would be super useful. The key is equal to the token whereas the value will always be 1 in the mapping function because we count the word once at the instance. Once finished, the file can be closed.

Because the MR_Emit() function is such a big part of this program that utilizes key/value pairs, it was super important for mappers to be able to store key/value pairs and for reducers to access them. My solution was a concurrent hash table with certain functionality for MapReduce.

Once the files have been scanned and key/value pairs have been added to a data structure (in my case a hash table), the Reduce() function is used on every key. The product or the output of the Reduce() prints the words and total number of times that it occurred in all documents.

The whole application begins with the MR_Run() which is contained in the main() portion of the program. A default partitioning function for our data structure was provided. Interestingly, it can be passed into the MR_Run() function but I implemented it in the mapreduce.c file. This function decides which thread a mapped item gets passed to. In the case of my word count application, it does not appear to be important but in other applications it may be more important.

The last requirement that was implemented, was that each partition must have its keys sorted in ascending order which I simply used quick sort to do. This is so that the Reduce() function is called on the keys in order for the partition.

Experiment Setup:

For me personally, the concept of concurrency can be quite confusing and the more functions and complexity that's added into the mix causes quite a few issues. For testing, I made sure to test a few different scenarios to ensure that I was obtaining the correct results… however when I arrived at the topic of performance, I struggled quite a bit in figuring out what my BEST, AVERAGE and WORSE cases are because these are difficult to determine and depend on what I'm specifically looking at or tuning. There are few routes that I chose in order to fully examine the impacts that certain parameters have on this implementation because it's difficult to test them all at once. Although, I understand the concurrent implementation of MapReduce would perform differently that the parallel version but there were some results surprised me greatly.

Test Set 1:

The first tests that I set up were some preliminary tests to determine that my code was returning the right output and specifically getting the right word counts for words. The methodology…

1. 1 file containing non-repeating words (file1.txt).
2. 1 file containing repeating and non-repeating words (file2.txt).
3. Multiple files containing repeating, non-repeating words and overlapping words (file3.txt).

Test Set 2:

In these tests I used different contents to see how the hash table or reducer function is able to handle them. Each file used contains 100,000 words in this test.

1. Best Case: 1 file all the same word (file4.txt).
2. Average Case: 1 file with a real-world example from free published text (file5.txt).
3. Worst Case: 1 file with all different words (file6.txt).

Test Set 3:

In this test I wanted to see how changing the number of mapping threads would affect the speed of computation in relation to one another. I kept the reducing threads greater than the number of files. Each file contains 100,000 words in this test and 3 files are used from the previous test were used.

1. Best Case: 3 files, 10 mapping threads.
2. Average Case: 3 files, 3 mapping threads.
3. Worst Case: 3 files, 2 mapping threads.

Test Set 4:

In this test I wanted to see how changing the number of reducing threads would affect the speed of computation in relation to one another. I kept the mapping threads greater than the number of files. Each file contains 100,000 words in this test and 3 files are used from the previous test were used.

1. Best Case: 3 files, 2 reducing threads.
2. Average Case: 3 files, 3 reducing threads.
3. Worst Case: 3 files, 10 reducing threads.

Instead, I decided that it would be interesting to observe 3 scenarios to see how my program would behave. These different tests were outlined as the following:

Test Results and Analysis:

Test Set 1:

These tests confirmed that I obtained the correct results based on the word counts printed out afterwards and my implementation was working as intended.

Test Set 2:

Test	Time (s)
1 (Best)	0.025456
2 (Average)	0.199620
3 (Worst)	0.953125

In these tests we can observe that the file containing all different words performed much worse that the file containing all of the same words. This is surprising to me because I expected that when everything is stored into 1 bucket of the hash table that the program would become more inefficient due to it becoming a linked list, but that is not the case because for a few different reasons. Firstly, the other buckets are empty so there is no reason to check them. Secondly and more importantly, the reducer function does not need to be used on multiple words, it is just invoked once on the single word whereas in the worst case it must be invoked for each word. Now imagine this one multiple partitions of a hash table…

Test Set 3:

Test	Time (s)
1 (Best)	1.005772
2 (Average)	1.018589
3 (Worst)	1.031590

In this test I was yet again surprised by these results as changing the number of mapping threads did not change my results even though I used multiple files and the number of threads was less than my total number of files used. My implementation has the Map() function complete in its entirety but in industry applications for parallel MapReduce they do something super cool… Multiple Map() functions can be invoked at once with different cores, CPU's or machines, then when one is finished the other process can be killed early saving immensely on resources! I could have had something similar done for my concurrent implementation but it would have added greatly to the complexity of the project.

Test Set 4:

Test	Time (s)
1 (Best)	0.530489
2 (Average)	0.878015
3 (Worst)	1.047338

In my last test, this was another surprise to me because the worst case occurred when I actually increased the number of threads for the Reduce() function. This one is also due to my specific implementation as having more threads increases the number of partitions for the hash table, resulting in an increase in iteration time for the Reduce() function. It will have to scan more partitions in their entirety. I originally thought that having more "workers" or threads would allow greater efficiency but this is not the case. In the future, I could improve this by skipping the hash table partition if its empty or work on using a combiner function to save on Reduce(0 time in general.

Notes:

1. I had memory leaks handled for program but found that my solution was caused issues when I scaled the sizing of my test files up. For example, when using text files with >=100,000 words I had issues compiling. When using smaller text files, I had no issues with my solution for freeing memory and being free of memory leaks. After investigating I found that this issue had to do with how memory is actually "reallocated" internally causing issues to how I was handling memory previously.
2. The test.c and mapreduce.c files can be compiled to build an application that can be ran. The runTests.c and mapreduce.c files can be compiled to automatically run the tests that I created for this project with timing information printed out. I also compiled the runTests.c file in Clion so folder path naming may be different.

Future Work:

I am extremely proud of the MapReduce program that I was able to implement but there are still many areas that I could improve upon if given more time. Most obviously is testing a creating a parallel implementation for multiple machines. The other is applying optimizations used in industry and making my program more robust (using a combine function, "killing" map early, etc.). The most understandable example was revealed in my tests when changing the threads to use for mapping and reducing. Conversely, something to look into would be whether my intermediate data structure, the concurrent hash table, could benefit from being able to dynamically resize itself as necessary or whether the size should be specific to application.

References:

https://github.com/remzi-arpacidusseau/ostep-projects/tree/master/concurrency-mapreduce

https://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf

https://www.talend.com/resources/what-is-mapreduce/

https://www.guru99.com/introduction-to-mapreduce.html