ELEN4020A: Data Intensive Computing Laboratory Exercise No 3

Kopantsho Mathafa (849038) Chizeba Maulu (900968) James Phillips (1036603)

April 16, 2019

I. Introduction

MapReduce is a programming model which enables the processing of large amounts of data for distributed computing. This two step model is composed of a map task and $reduce$ task [1]. As the name suggests, reduce is always performed after map [1]. The first task, map, receives a set of data and breaks it up into key-value pairs or tuples. The reduce task receives these tuples as input and it proceeds to combine those tuples into smaller sets of tuples [1].

This programming model has gained popularity with many programmers due to its ability to easily scale data processing over multiple computing nodes. The process of decomposing data processing into $mappers$ and $reducers$, which are MapReduce primitives, is sometimes not trivial. It is the developer's responsibility to decide how the tuples will be formed and what information will be contained The ability of MapReduce to scale over hundreds to thousands of machines in a cluster, is as simple as changing configuration file, after the mapper and reducer functions have been successfully written.

This report will document an implementation of MapReduce using the Phoenix++ framework. This framework is a shared-memory implementation of MapReduce.

II. MapReduce Algorithm - Exercise 1

To demonstrate the implemented MapReduce algorithm, a common example is used for counting and indexing a text file in C++. The process first consists of *map* where the amount of words are counted, and then *reduce* where the counted words are all joined together to form a word count for all words in the text. For accuracy of the data, all punctuation is removed and the words are all provided in lower case. A sample text is obtained using *Lorem Ipsum* [2].

A. Word Count Algorithm

As mentioned above, this word count method consists of a *map* and *reduce* phase.
To map the words in a provided text files, all the contents are first read into memory using *ifstream*. All the lines from a text file are stored into a vector for formatting. Formatting then takes place. Formatting does a few steps:

- Separates the words by spaces into individual strings. This is is achieved using a *istringstream* which is able to detect space as a delimiter [3].
- Removes punctuation from the words. This is achieved by iterating each character in a string and using the *ispunct* C++ method. If this evaluates true, the character is removed. This method is inefficient, but did achieve the objective [4].
- Determines whether a word is a stop word or not. A list of English stop words was obtained on Github and is used [5]. The stop words are stored in an additional file and are read in when needed for comparison. Each word is compared and removed if it is evaluated as a stop word. This method is not efficient, but it does achieve the objective.

Once the above separation process is complete, a struct is defined as a word and a frequency.

To complete the mapping phase, the reduction and the final steps take place together. Each word is iterated through and compared to the current map of words. If the word already exists in the map, the frequency is increased. This process of increasing the frequency will thus result in the reduction phase. After this step is complete, the word list is sorted into descending order by word frequency. This descending is achieved by means of a bubble sort.

B. K frequent words

Once the mapping and reduction phase has been complete, it is a matter of selecting the top K elements, provided there are enough elements for K.

C. Inverted Index

To created an inverted index from a text file, a map is first required[1]. In this implementation, the map is provided from the Word Counting Algorithm above, and the initial input text file is supplied. From here, the lines from the text file are loaded into another vector and formatted in a similar manner to above. Each word in the mapping is then compared to the lines and the line position is added to the mapping once a match is found[1]. Each word will be compared to each line, which is not an efficient process, but does achieve the objective.

D. Demonstration

To demonstrate the above implementation, a short text example is used. Here, Shakespeare's Sonnet 134 is used as a sample text, and the following output is presented [6]: Figure 1 presents the mapping of words and their frequencies, and Figure 2 presents the line count for the first 10 words in the map.

III. MapReduce with Phoenix

Phoenix is a shared-memory implementation of the MapReduce model written in C++ [7]. The functionality provided by the environment is used to implement MapReduce using the discussed algorithms. The `split` function is used to split the text file contents between a number of threads. This ideal for processing large datasets as is the case with *File2ForLab3.txt*. The `map` function is overloaded to define the required output of each mapping, in this case key-value pairs representing a word and its count are required. The combiner collects the counts of a specific word into a container and adds them. The running times of each algorithm are illustrated in Table I. Running times are for both using *File1ForLab3.txt* and *File2ForLab3.txt* as an input dataset.

TABLE I
TABLE INDICATING ALGORITHM RUNTIMES PER TEXT FILE

File	Word count duration (ms)	Top 10 duration (ms)	Top 20 duration (ms)
File1ForLab3.txt	317.8	0.019	0.039
File2ForLab3.txt	10.312	0.01	0.025

It is evident that the running times of the word count algorithm with *File1ForLab3.txt* as an input are shorter than those for *File2ForLab3.txt*. This is as expected as *File2ForLab3.txt* is the larger of the two text files therefore more data is required to be split, mapped and then combined. The top 10 and 20 queries are relatively similar. This is valid as the algorithm simply selects the top 10 and 20 of an already sorted word count. Although, the inverted index algorithm is implemented serially, the MapReduce implementation was unsuccessful. Thus, no runtimes are indicated for this algorithm. It is possible to measure the runtime using a serial inverted index using *File1ForLab3.txt*. This would be impossible for *File2ForLab3.txt* due to its size. Thus, the MapReduce implementation of the inverted index would have been capable of handling the size of *File2ForLab3.txt*.

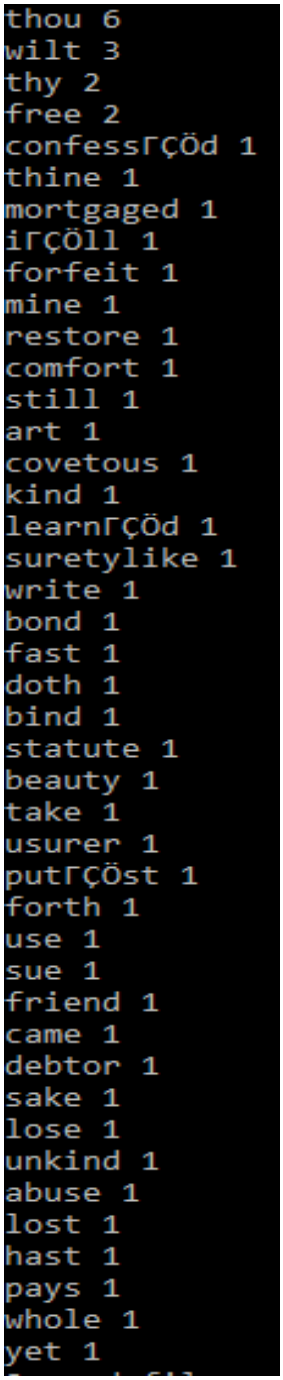

Fig. 1. Word Mapping Sonnet 134
labelfig:SonnetMap

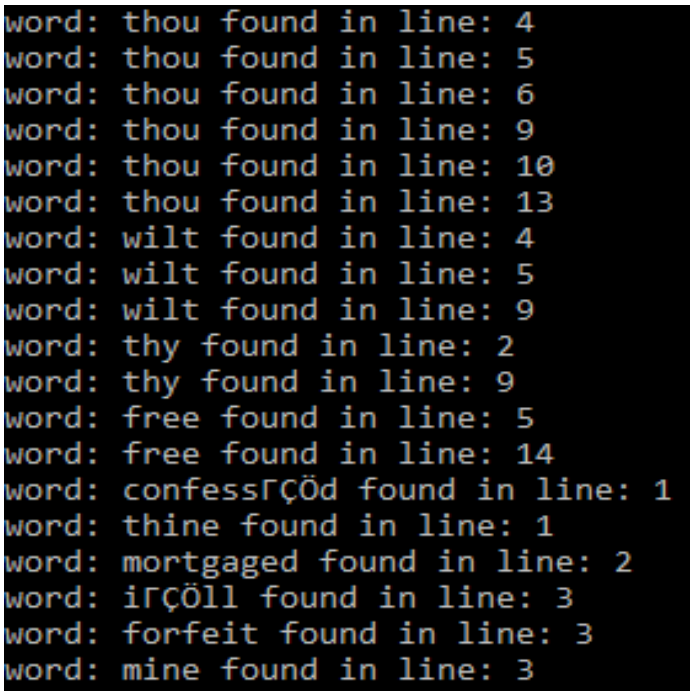

Fig. 2. Word Line Count Sonnet 134

IV. Pseudocode

A. *Word count algorithm*

Data: Any data type object and a container
Result: An intermediate key/value pair
Convert all characters in objects text data to uppercase;
while *While not at the end of the text data* **do**
  while *While not reading an uppercase char alphabet* **do**
    Increment a counter;
    If the counter has increased in value Store the word;
    if *If it the word is a stop word* **then**
      Do nothing;
    else
      Send out intermediate key/value pair for further processing;
    end
  end
end

Algorithm 1: Phoenix++ map function implementation

B. Inverted Index algorithm

Data: K value
Result: List of top K occuring words and the lines they appear in
for *From 0 to K* **do**
  for *From 0 until the end of the text data* **do**
    for *Each non-stop word found in the text* **do**
      Store the line which the word is found on;
    end
  end
end
Print the top most appearing words and the lines they are found on;

Algorithm 2: Inverted index function implementation

V. Conclusion

The MapReduce solution of the word count, top-K query and inverted index problems is presented. MapReduce decomposes large datasets and divides their reading and processing between processing units. The MapReduce functionality of the Phoenix framework is utilised in the solution. The word count algorithm of the large dataset takes longer than that of the small dataset. The top 10 and 20 queries are determined in relatively similar times for both datasets. Although a serial inverted index algorith is implemented, the MapReduce algorithm was unsuccessful. No running time tests are performed due to this.

References

[1] Shital Kat, "Word Count Program With MapReduce and Java - DZone Big Data." [Online]. Available: https://dzone.com/articles/word-count-hello-word-program-in-mapreduce

[2] blindtextgenerator, "|| Dummy Text Generator | Lorem ipsum for webdesigners ||." [Online]. Available: https://www.blindtextgenerator.com/lorem-ipsum

[3] kartik, "Split a sentence into words in C++," May 2018. [Online]. Available: https://www.geeksforgeeks.org/split-a-sentence-into-words-in-cpp/

[4] P0W, "C++ Remove punctuation from String." [Online]. Available: https://stackoverflow.com/questions/19138983/c-remove-punctuation-from-string

[5] sebleier, "NLTK's list of english stopwords." [Online]. Available: https://gist.github.com/sebleier/554280

[6] LW Willingham and William Shakespeare, "Top 10 Shakespeare Sonnets |," Jun. 2015. [Online]. Available: https://www.tweetspeakpoetry.com/2015/06/25/top-ten-shakespeare-sonnets/

[7] C. Kozyrakis, "phoenix," September 2016. [Online]. Available: https://github.com/kozyraki/phoenix