KNeighborhood Scorer - Serial, MultiThreading, MapReduce Benchmark

Rashmi Dwaraka

10/06/2017

KNeighborhood Scorer - Serial, MultiThreading, MapReduce Benchmark

Neighborhood Score program

The Neighborhood Score program description is as below:

Letters are scored according to the rarity with which they appear in a given corpus (a large body of texts). Rare letters are worth more than common letters. Specifically, if a given letter makes up 10% or more of the letters in the entire corpus of text, its score is 0. If the letter makes up [8,10)% of the corpus (at least 8 but not more than 10 percent of the corpus), its score is 1. If it makes up [6,8)%, its score is 2. If it makes up [4,6)%, its score is 4. If it makes up [2,4)%, its score is 8. If the letter makes up [1,2)%, its score is 16. Otherwise, the score is 32. Whitespaces, numbers, and punctuation do not receive scores. Scoring is case-insensitive.

A word’s score is the sum of the scores of all its letters. A word’s k-neighborhood is the bag of k words appearing before it and k words appearing after it in the corpus.If a word appears in the corpus multiple times, it will have multiple k-neighborhoods. The score of a k-neighborhood is the sum of the scores of the words that are members of the k neighborhood.

Benchmark - Experimental design

Since we know that the computer systems donot execute the same program at the same execution speeds as there are many uncontrollable factors that include other applications running on the system, kernel operations and hardware heating etc.,. Hence, I have benchmarked the program by considering the runtime for 100 Iterations. This should give us a fair idea about any stragglers and performance of each programming technique. In this Benchmark experiment, I will be comparing serial, multi-threaded with thread count ranging from 2-16 and hadoop mapreduce programs in pseudo distributed mode and cluster mode. The configurable parameter k is set to 3 for this experiment.

Algorithm Design

KNeighborhood Scoring Algorithm

The previous algorithm was not scalable for huge data and I was unable to run the code for the big-corpus. The below algorithm makes the program scalable for huge corpus. The algorithm maintains a buffer of k words for each line and enables us to run any size data, as the in memory data structure is very small at all points of runtime.

Sliding Window Algorithm for KNeighborhood

- Step1 : Make 2 passes over each line of data. In first pass, find all the forrward neighbors excluding last k words In Second pass, find all the backword neighbors for all words keep last k words in a buffer, so that we can find its k neighbors while parsing next line
- Step2 : After each processing each line, we calculate the scores and merge to a global hashmap The hashmap stores {word,(score,count)}
- Step3 : The last k words of the file is merged to the hashmap and written to the output

Modified Serial program for Scalability

- Step1 : For all files in the input folder: For all lines in the file: Calculate the letter score based on its occurence
- Step2 : For all files in the input folder: For all lines in the file: Process the lines using above Sliding window algorithm Print the KNeighborhoodScore mean for each word alphabhetical order to CSV

Modified Parallel program for Scalability

- Step1 : For all files in the input folder: Created a callable instance to process the letter occurrence (No sharing of data) The result of each callable is passed to a future instance Once all futures are evaluated, merge the results to calculate global letter score
- Step2 : For all files in the input folder: Created a callable instance to process the lines using Sliding window algorithm with no sharing of data The result of each callable is passed to a future instance Once all futures are evaluated, merge the results to calculate global KNeighborhood word score Print the KNeighborhoodScore mean for each word alphabhetical order to CSV

Modified Map-Reduce program (MEAN) for Accuracy and efficiency

- Step1: The Letter count mapper is run on the entire corpus. For each mapper, create a global hashmap to keep track of count. Emit the count of the letters during the cleanup i,e, at the end of the Mapper task This reduces the number of for reducer and hence reduces sort and shufle time. Reducer, reduces each letter counts and aggregates to give us letter counts and total letters in the corpus.
- Step2: The letters are scored in the driver program, by reading the hdfs output file. The letter scores are written to file and placed in hdfs. (Alternate way to explore is to set the letter scores as configuration key-value. To be explored.)
- Step3: Use non-splitable files i,e, assign entire file to 1 mapper to maintain accuracy. Data Cleanser mapper, cleans the data by removing extra spaces and special characters and tokensizes the words. Each line with 10 words is emitted.
- Step4: The cleansed data is passed to kNeighborhood mapper and the KneighborScore mappers calculate the k neighbors and score them using sliding window approach. Each map emits the words with its (score,count) Reducer, reduces the word scores by combining all the possible kNeighbors for the word and calculates the mean - which is its Kneighborscore.

Modified Map-Reduce program (MEDIAN) for Accuracy and efficiency

- Repeat steps 1 to 3 from above.
- Step4: Each map emits count as 1. The combiner create a histogram of scores by combining results from the mapper to the form The reducer finds the median based on the histogram. (The histogram approach reduces the data size for shuffle)

Hardware Specification

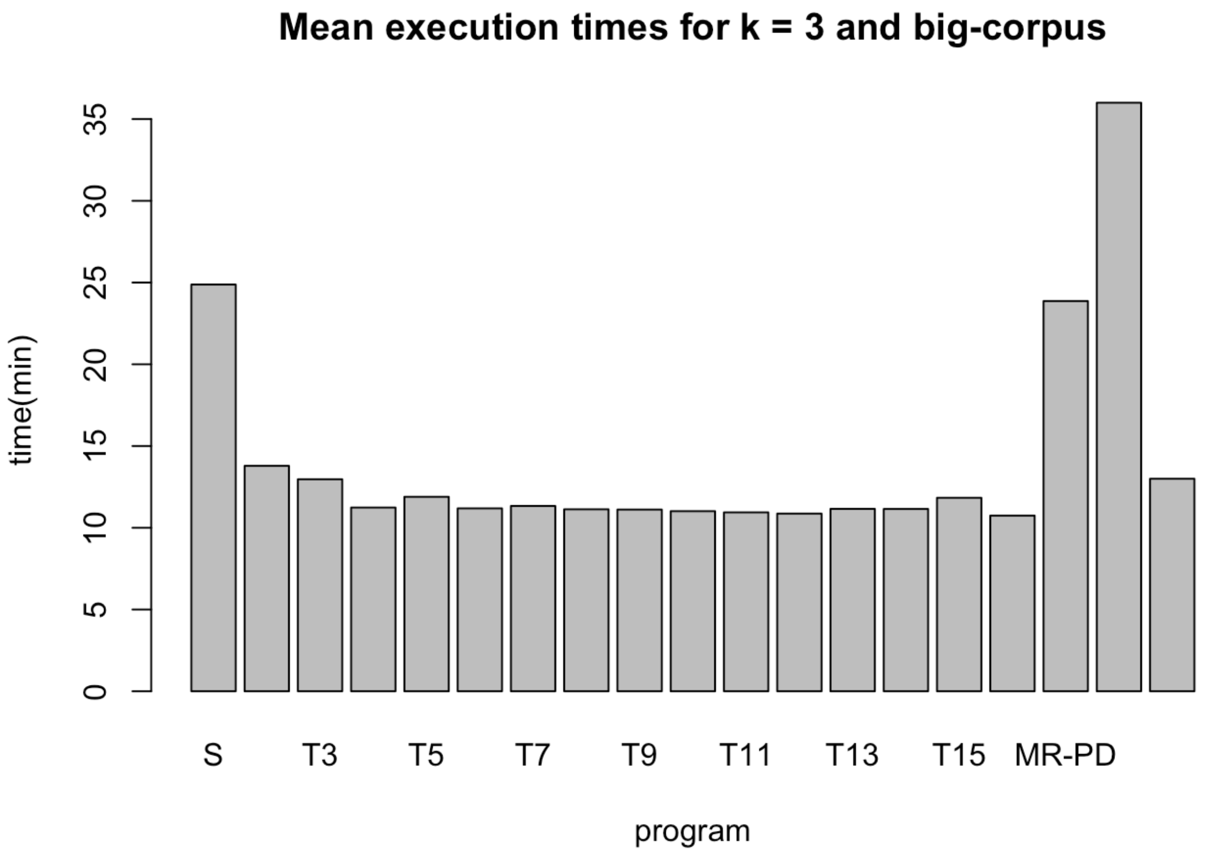
- Processor - 3.1 GHz Intel Core i7 with 4MB shared L3 cache.
- OS: MacOS Sierra - 10.12.4
- Memory: 16 GB 1867 MHz DDR3
- Number of cores - 2
- Number of threads - 4
- Hadoop: 2.8.1 (Pseudo Distribution mode)
- For Cluster - Amazon EMR - Amazon 2.7.3 (4 nodes and 2 nodes)

Observations from the Benchmark Experiment

Serial vs Parallel vs MapReduce

The modified algorithm, the mean execution runtime for 5 iterations each are as below: *Input:* big-corpus *K:* 3 * Serial - 1492700 ms = 24.8783 minutes * Parallel - + Thread count 2 is 827286 ms => 13.7881 min + Thread count 3 is 778083 ms => 12.968 min + Thread count 4 is 674260 ms => 11.2376 min + Thread count 5 is 713594 ms => 11.8932 min + Thread count 6 is 671255 ms => 11.1875 min + Thread count 7 is 680043 ms => 11.3340 min + Thread count 8 is 667858 ms => 11.1310 min + Thread count 9 is 666829 ms => 11.1138 min + Thread count 10 is 660950 ms => 11.0158 min + Thread count 11 is 656544 ms => 10.9424 min + Thread count 12 is 652012 ms => 10.8668 min + Thread count 13 is 669372 ms => 11.1562 min + Thread count 14 is 669078 ms => 11.1513 min + Thread count 15 is 710025 ms => 11.8337 min + Thread count 16 is 644622 ms => 10.7437 min * MapReduce - Pseudo Distribution mode - 1432112 ms => 23.8685 min * MapReduce - Cluster 2 nodes - * MapReduce - Cluster 4 nodes - 13 min

```
x <-  vector();
x <-  c("S","T2","T3","T4","T5","T6","T7","T8","T9","T10","T11","T12","T13","T14"
,"T15","T16","MR-PD","MR_4N","MR_2N")
t <- vector();
t <- c(24.8783,13.7881,12.968,11.2376,11.8932,11.1875,11.3340 ,11.1310 ,11.1138,1
1.0158 ,10.9424,10.8668 ,11.1562,11.1513 ,11.8337,10.7437, 23.8685 ,36,13)
exec <- data.frame(time = t, x = x)
barplot(exec$time,main="Mean execution times for k = 3 and big-corpus",xlab="prog
ram",ylab="time(min)",names.arg=x)
```



Future Work

Create Multiple reducers to make reducing phase parallel. Also explore various optimization techniques of MapReduce