**CSS 534  
Program 3: Writing an Inversed Indexing Program with MapReduce**

Name: Yifeng Zhang

Date: Nov 11, 2019

Contents

1. Implementation Summary …………………………………………………………………. 2

2. Source code …………………………………………………………………………………. 2

3. Execution Output …………………………………………………………………………… 3

3.1 Comparison of performance ………………………………………………………. 3

3.2 Output ……………………………………………………………………………… 3

4. Discussions …………………………………………………………………………………. 3

5. Lab ………………………………………………………………………………………….. 5

*Note: please read readme.txt to know the files detain in my submission folder*

**1. Implementation Summary**

In this programming assignment, we write and execute an inversed-indexing program with Task Parallelism - MapReduce. The goal of inversed indexing program is to create a list of documents for each keyword we input.

1. The main function receives keywords or terms in args.
2. We use the JobConfObject to pass these keywords to the map() function. In map class we can retrieve all the keywords from JobConf and put them into a hashSet – keyWords set. Map invoked for each line of a file and we using fileSplit files with each line and get filename. When we tokenizer each line that is same as retrieve each file. We compare each word we tokenizer with keyWords set to collect them with Text-filename.
3. After map function, we pass pairs of a keyWords and a filename(docID) to Reduce. In reduce class, we prepare a hashmap to maintain all filename\_count (filename: table key and count : table value) as an Object. When retrieve all values for a key, we count the occurrence of filename by increasing the value of fileaname in the hashtable. Once retrieve all the values, sort the Object – filename\_count by its count number. Create a Text by append all the filename\_count in sorted order. Print it with output.collect(key, text).

This implementation does not need combiner since we already count them in reduce function according to filename. The objects with different filename we passes are unique. If use combiner method, reduce function will collect the object - filename\_count as count as 1.

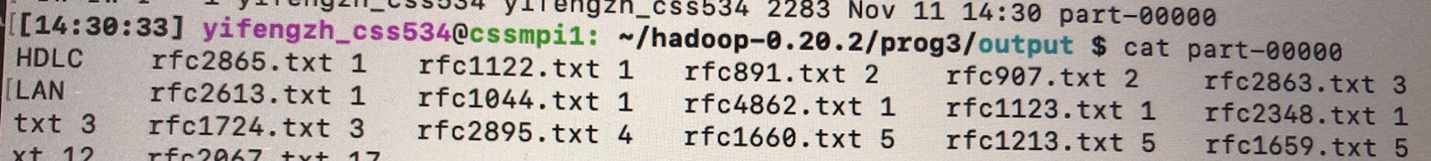


Figure 1: result with using combiner method

**2. Source code**

The source code (InvertedIndexes.java) is under src folder in my submission report. Use compile.sh to run my code.

*Note: I added sort documents into my inverted indexes program.*

**3. Execution Output**

**3.1 Comparison of performance**

Compare the performance between single-node and four-node executions. The following shows my program execution performance. The snapshots in my submission folder.

Table: Comparison of performance of sequential and parallel version

|  |  |  |
| --- | --- | --- |
|  | Sequential (1 computing node) | Parallel (3 computing node) |
| Computing Time | 256682 ms | 80215 ms |

**Performance improvement: 256682/80215 = 3.20**

**3.2 output: part-00000**

I cannot use diff command to compare the output with single-node and four-node execution. But I checked them with eyes, and they are same. The belowing is the snapshot of part-00000.

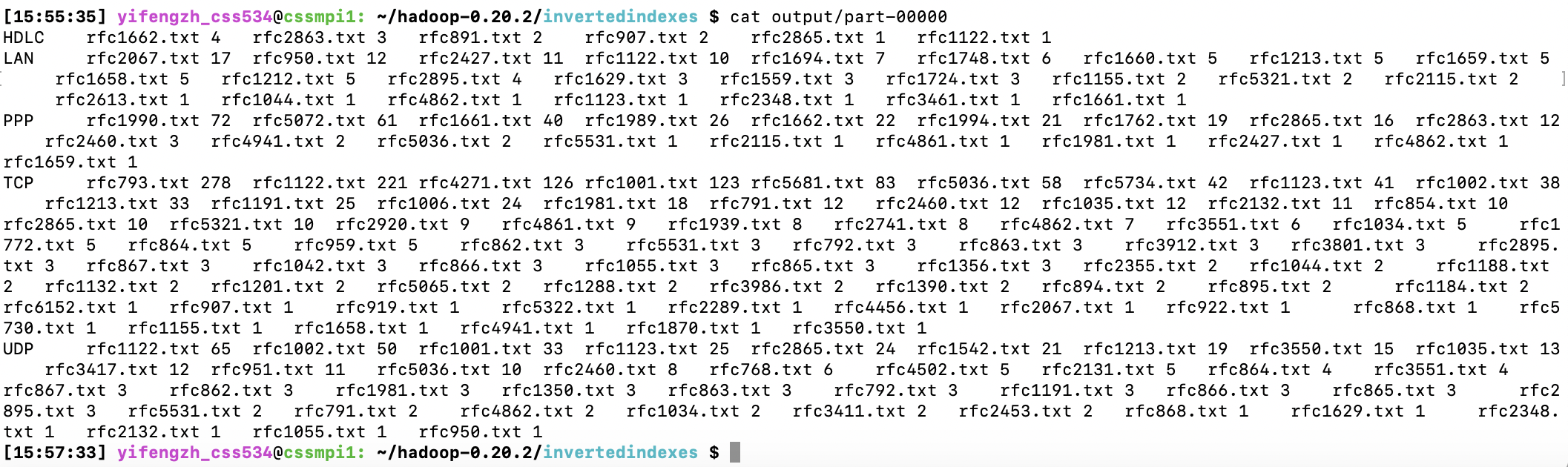


Figure 2: output part-00000

Note: All the other results show in my snapshot folder.

**4. Discussions**

If I wrote inverted indexing program with MPI, I will retrieve the files by files at the all ranks together (at same time) by stripe the file data to each rank.

1. Use result = new List<List<Object<filename, count>>> () to store all the result. One HashMap for keyword(key) and index of result(value) to store the keyword and pass this to each rank.
2. Allocate an int array [numbers of keyword] to each rank, store the retrieve sub-result of one file into array. Clean the array each time before retrieve a file data.
3. Retrieve the stripe of file data in each rank and store the count into array. Send the array back to master rank with MPI\_reduce, and it will sum up all the count of each keyword while we know the filename already in master rank.
4. In master rank, append those sub-result with filename into result.
5. After retrieving all files, create new output = List<String>. Sort the result for each keyword with filename-count and store it into output.
6. Finally, print the output with key words.

**Compare MapReduce and MPI in:**

1. File distribution over a cluster system

Mapreduce: Name node (at master) maintains a name space and metadata, and Data nodes (at slaves) maintain copies of each data block.

MPI: Allocate and copy the file data with file striping to each rank. Master node maintains filename for inverted indexing program. Master node send stripe of file data to worker node, while worker node receive the stripe of file data.

1. Collective/Reductive operation to create inverted indexing

MapReduce: Since in MapReduce, it shuffles and sorts the data after map, collect/reduce the data into each rank according to key words.

MPI: collect/reduce the data for every key word into master node.

1. Amount of boilerplate code

MapReduce: Only need Map and Reduce. (2 boilerplate code)

MPI: use MPI\_send, MPI\_receive, MPI\_reduce.

MPI use more boilerplate code than MapReduce.

1. Anticipated execution performance

For anticipated execution performance, MapReduce will be much slower, because it saves all intermediate results into disk while MPI uses all memory. For fault tolerance, while you mentioned about a potential checkpoint features in MPI, it must be implemented by each user but not at the system level.

1. Fault tolerance; recovery from a crash

MapReduce (Hadoop) has its built-in fault tolerance and fault compensation capabilities. Every data block has a copy that is stored on the servers. And it also generates logs during the execution process. It’s fault tolerant.

MPI use checkpoint and restart to support both data and process fault tolerance. It’s not stable and data may be lost. Less support fault tolerance than MapReduce.

Potential checkpoint features in MPI, it must be implemented by each user but not at the system level.

**5. Lab**

After successfully installing Hadoop-0.20.2, run WordCount.java with some input files(I create 2 files which are different from files01.txt and files02.txt).

1. Using 2 input files (rfc01.txt and rfc02.txt) in input folder and put in my Hadoop account. A snapshot of input files:

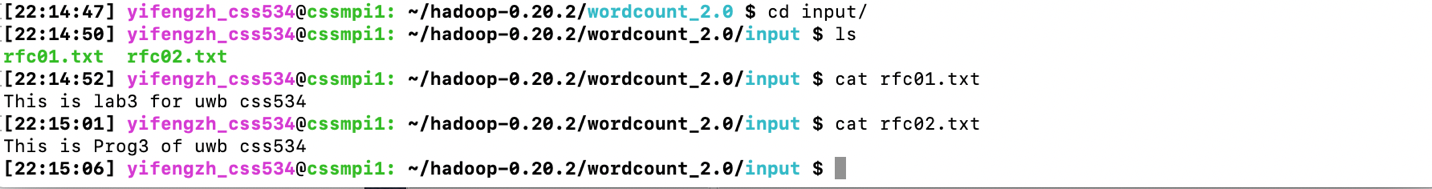


Figure 3: input files

1. After compiling the WordCount.java source code, run it with Hadoop. A snapshot of a MapReduce execution:

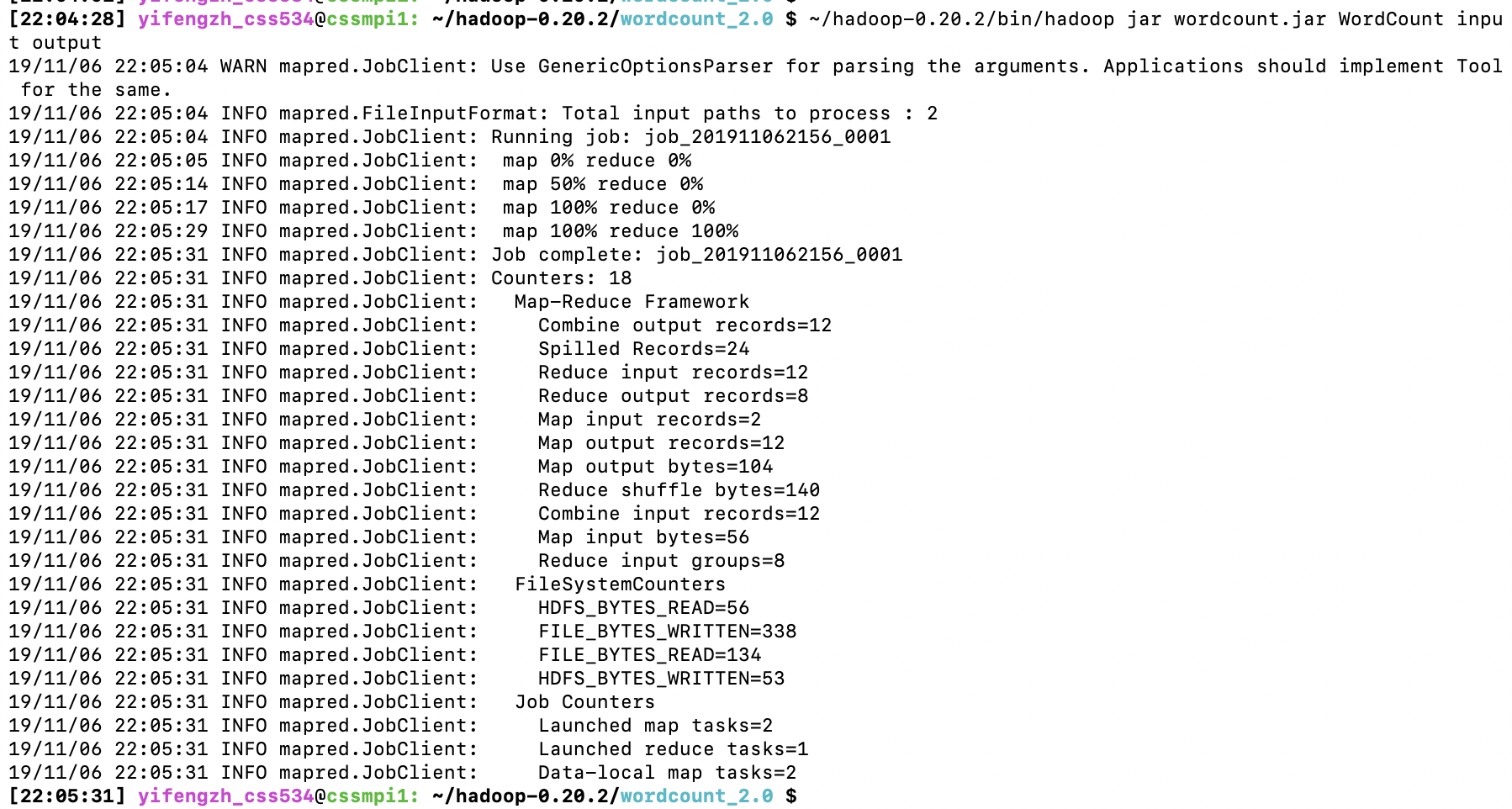


Figure 4: MapReduce execution

1. A snapshot of an output result in my Hadoop account:

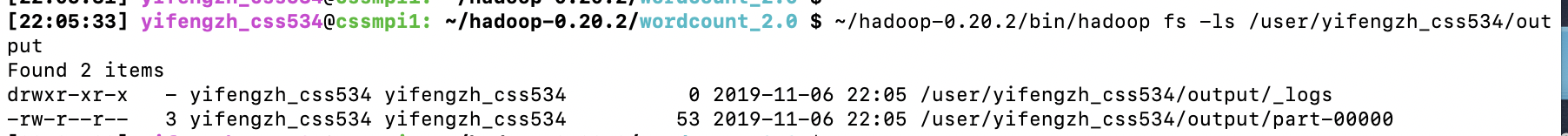


Figure 5: output after running mapreduce

1. A snapshot of my result file - part-00000: comparing with my input files, it is correct.

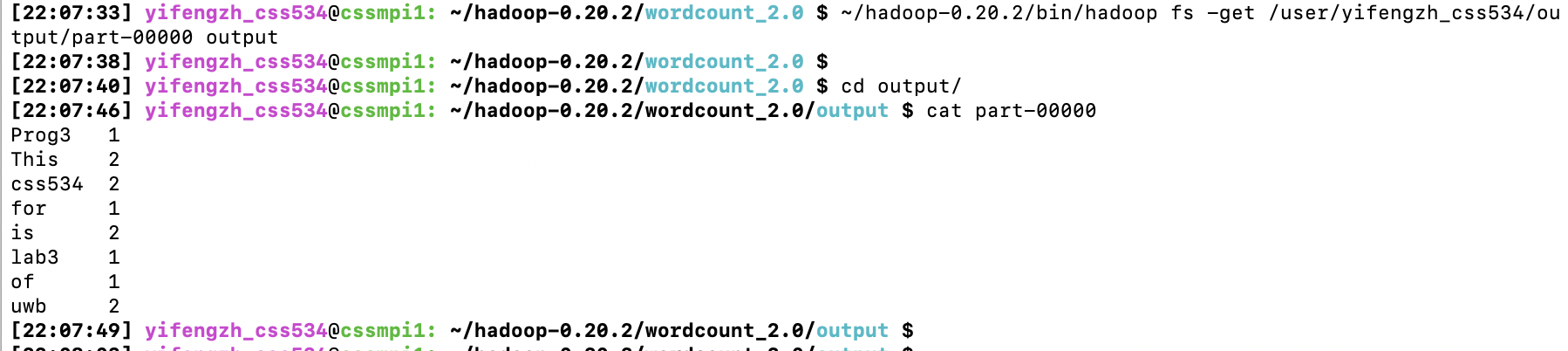


Figure 6: part-00000

*Note: all the lab files in wordcount\_2.0 folder under my submission folder.*