Documentation

This program is an inversed-indexing Hadoop Java program where the input is text files and keywords and the output is each keyword and a list of text file's names and the number of times the keyword appears in this text file. The algorithm uses the concept of MapReduce and implementation uses Hadoop distributed file system.

The **main()** function configures the job to be done by setting the job's name ("BBBBB"), Text as output format for input key and values, TextInputFormat and TextOutputFormat for output keys and values respectively. Main function also extracts keyword1, keyword2... keywordn and put into the job configuration. The current time before the job starts running(startTime) and the current time after the job ended (endTime) is recorded in miliseconds for performance analysis.

First, **map()** function retrieves and recorded all keywords from JobConf object into a String array called keywordArr[]s. It also creates an int array keywordCount[] to keep track of number of keyword appearances. For example, keyword at keywordArr[i] appears keywordCount[i] times. The function takes in a filename and tokenize each word in the file with a space. For each word that matches a particular keywordi in the keywordArr[], the count for keywordi is incremented by 1. After all words are processed, map() generate and pass pairs of a Text keyword and a Text filename\_count to Reduce. Examples are (keyword1, filename\_count), (keyword2, filename\_count). The passing is done by calling the OutputCollector output.collect(keyword, filename\_count).

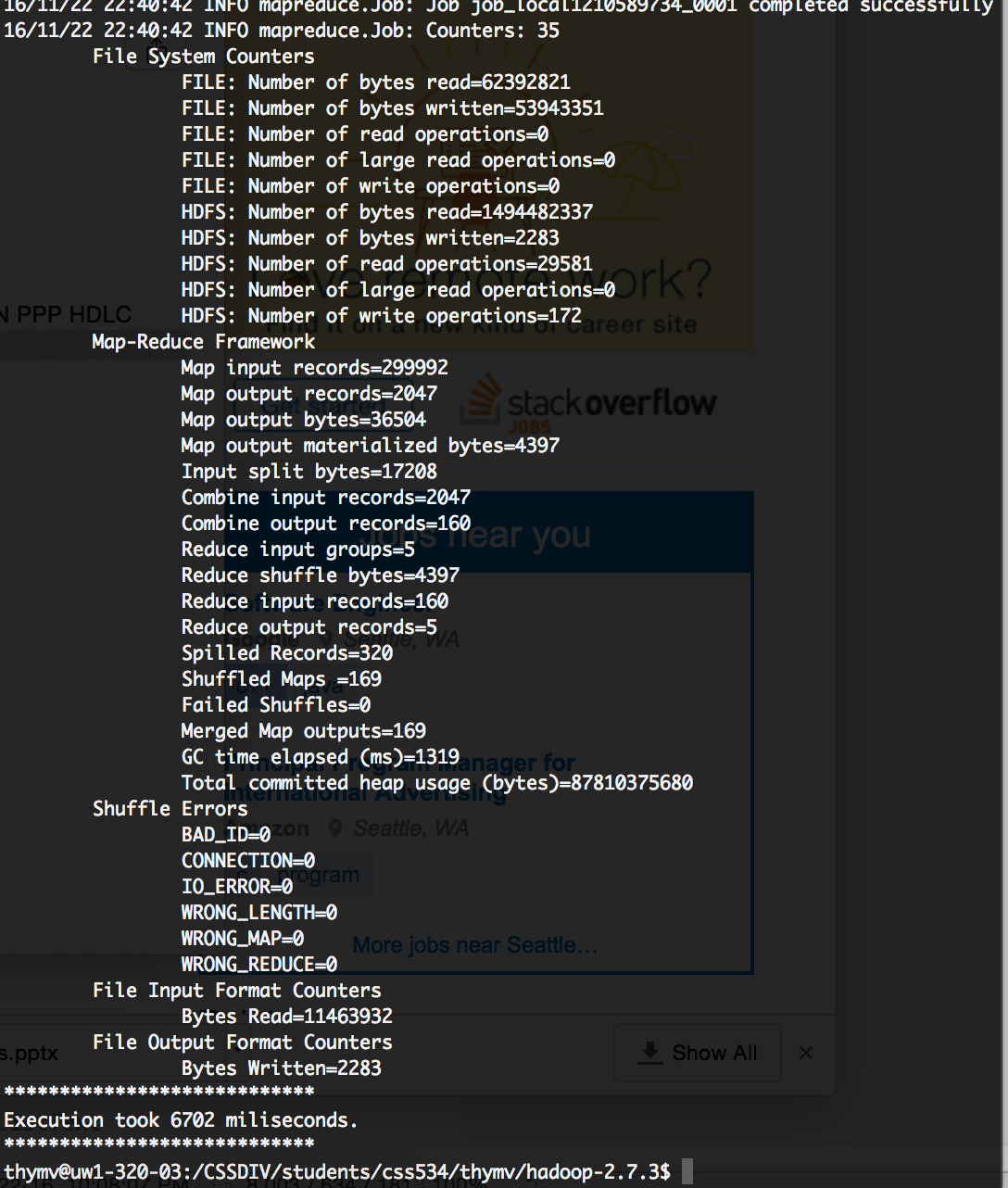
The **reduce()** function processes a keyword and all filename\_counts for that keyword. First 2 LinkedLists are created: LinkedList<String> filenameList contain unique filenames and LinkedList<Integer> countList keeps track of the counts associated with filenames. For example, the ith Integer in countList reflects the number of times the keyword appears in the ith filename on filenameList. In order to accomplish that, reduce() split every filename\_count to filename and checks if the filename already exists on filenameList. If filenameList does not have this file, this filename and count are added to filenameList and countList respectively. Otherwise, filenameList already has the file, so we remove the old count on countList, sum it up with the new count, and put the result back to countList at the same position.

After all filename\_counts are processed, I added the additional feature of sorting the filenames based on the counts. LinkedList<String> orderedFilename contains filenames in oder from most count to least count, and LinkedList<Integer> orderedCount contains the counts to corresponding orderedFilename's filename of the same index. Each filename and count from filenameList and countList are put on orderedFilename and orderedCount on the ith index position so that the order (most to least count) is preserved.

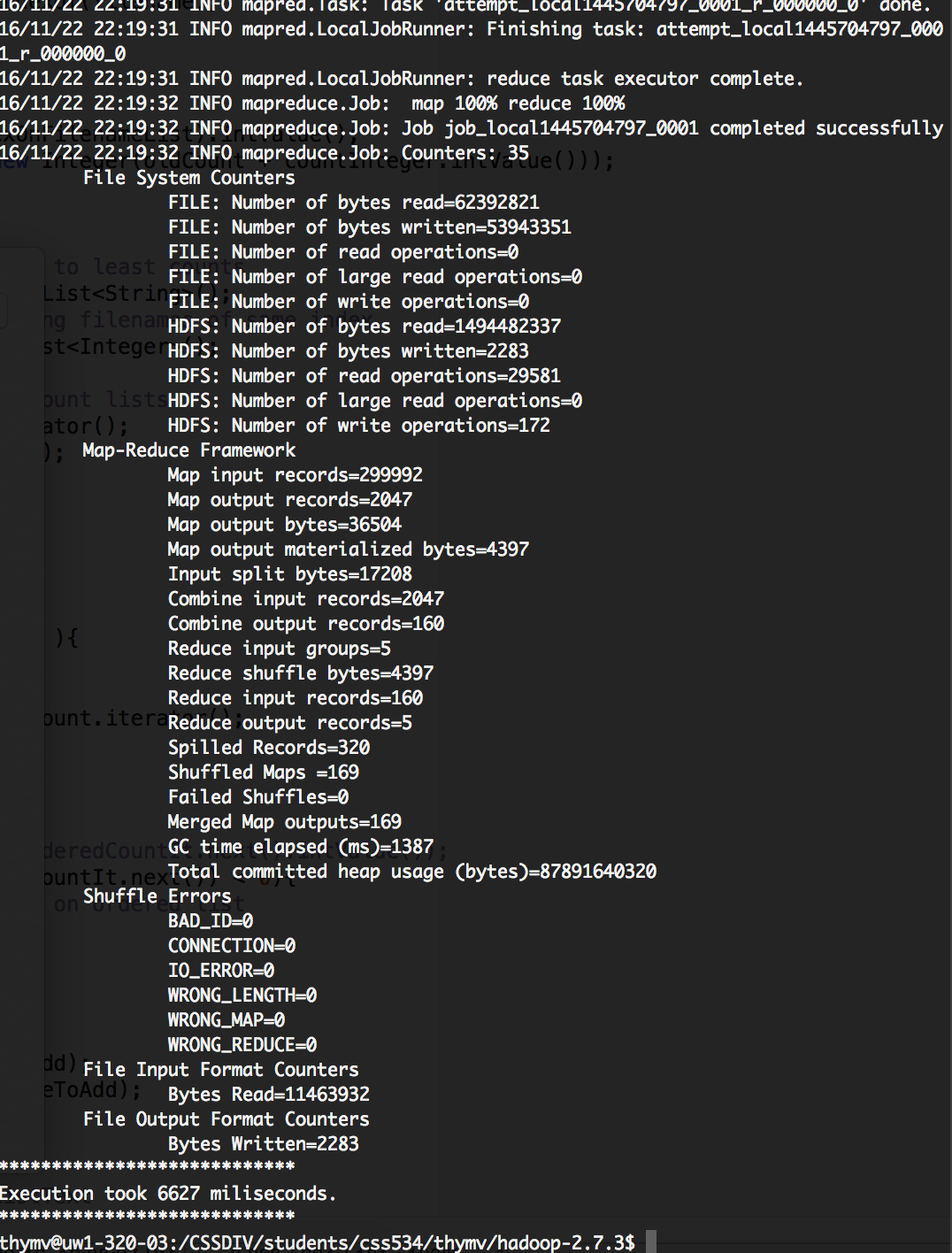
Finally, all filename\_count is concatenated (separated by a space character) into String docListText, and OutputCollector output collects the keyword as key and docListText as output value.

Execution output

**Output for 1 node (uw1-320-03):**



**Output for 4 nodes (uw1-320-03, 02, 04, 09):**



**Performance improvement**

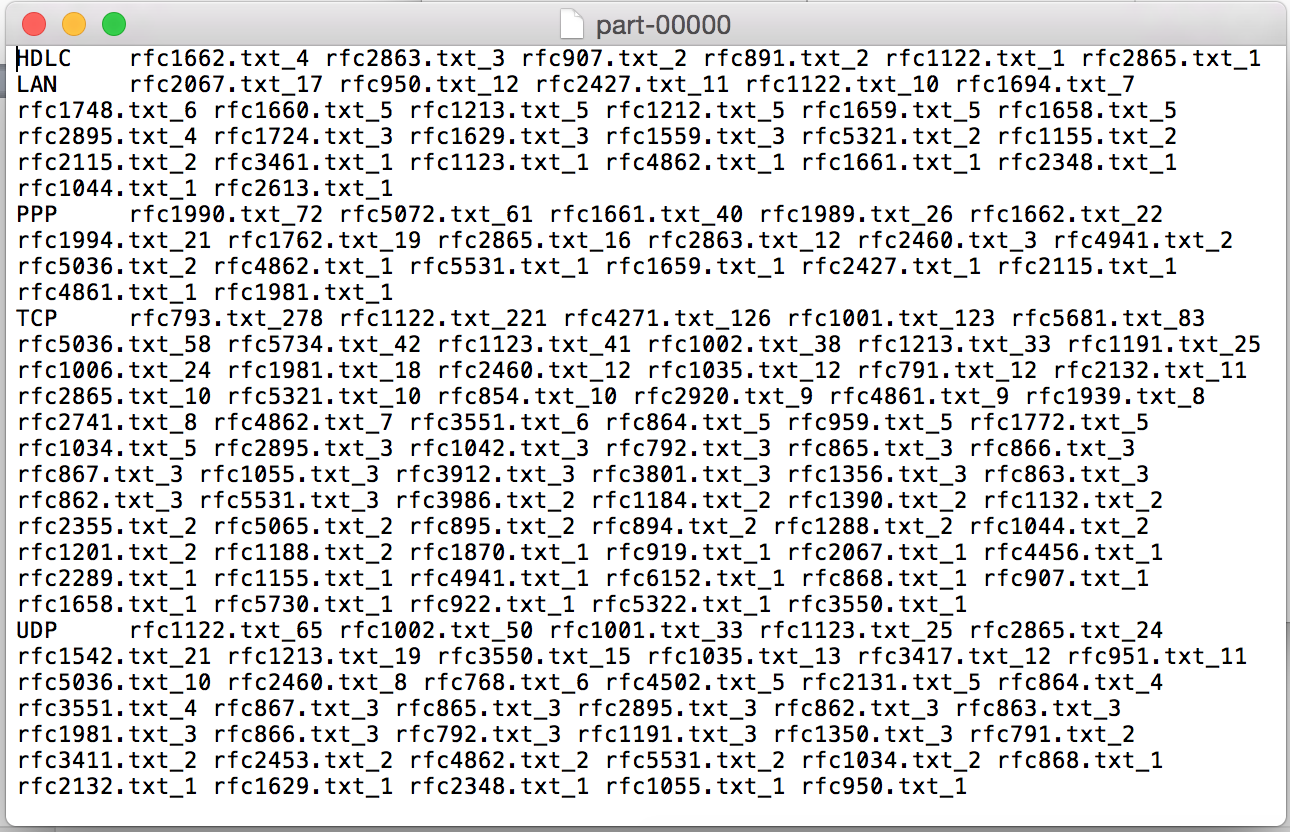
= (1 computing node)/(4 computing nodes)

= 6702 miliseconds / 6627 miliseconds

= 1.011

**Additional work:**

For each keyword, the filenames and the associated count (number of keyword appearances) appear from most counts to least counts.



(continues next page)

Discussion

**Programmability:**

This program can be written using MPI but it would be more cumbersome and there are less advantages than using Hadoop MapReduce. First of all, in MPI case, the distribution of text files to all computing nodes would have to be manually coded. In Hadoop MapReduce case, this is automatic, so the programmer does not have to worry about it. Second, in MPI case, the master node would take the responsibility of broadcasting all the text files, and this is causes a bottleneck. Third, scalability would be an issue for MPI but less so for Hadoop MapReduce. If the size of all the text files are too big, the (MPI) master node may not have enough space, whereas in Hadoop case, distributed file system allows massive storage and is much more scalable. However, the advantage of using MPI is that the setup is much more simple than Hadoop and that it is free.

**Usability:**

MapReduce pros:

- Massively large unstructured data can be stored

- Partitioning of data is automatic, so the programmer can focus on solving the problem and less on data transferring to nodes.

- Built-in fault-tolerance to take care of problems like network bottleneck, node crashes

MapReduce cons:

- No shared memory between tasks, so it is not suitable when communication between tasks is required.

- Because of the algorithm concept (mapping and reducing), there is limitation on the type of problems that can be solved.

- Not suitable for transactional data and not the optimal solution for structured data.

**Possible applications:**

A specific application that can use MapReduce is analyzing and predicting user behavior in different scenarios on big online shops such as Amazon, Rakuten, and Alibaba. The amount of data generated is massive and should be stored in a distributed file system.

Another application is DNA sequence processing. Each human genome is about 3 billion DNA nucleotides, which can be represented as characters. MapReduce allows storage for massive DNA sequence data from the population and can be used to do genotyping (and other types of analysis).

**Performance considerations:**

Reconfiguration for yarn and other set-ups are required to make use of more than 1 node. The current set-up cannot show much performance improvement when switching from 1-node to 4-nodes execution.

The sorting filenames by their associated counts in my program is simple and satisfies the requirement, but it may not be efficient if the list of filename\_counts is significantly large. This is because the current implementation does not consider sorting performance as an issue. If the goal is scalability in terms of sorting a much greater number of filename, a more efficient sorting algorithm should be used (merge sort, heap sort, radix sort, etc.).