上海交大—巴黎高科卓越工程师学院
SJTU-ParisTech Elite Institute of Technology

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

Project2: Hadoop MapReduce "from scratch"

褚子豪 **516260910011**
邓若凡 **516261910008**
王甫 **516261910014**

Contents

1 System diagram **3**

2 Functions description **3**

2.1 *BASIC.java* 3

2.2 *MASTER.java* 4

2.3 *SLAVE.java* 4

3 How to build/run **4**

3.1 Preamble 4

3.2 Build and Run 5

4 Experiment **5**

4.1 Comparison of two methods 5

4.2 Improvements 7

1 System diagram

The overall MapReduce word count process is almost the same as presented in the Project2 presentation slides, but there are still some differences. We have used docker to realize the slaves. The system diagram of our program design is as following:

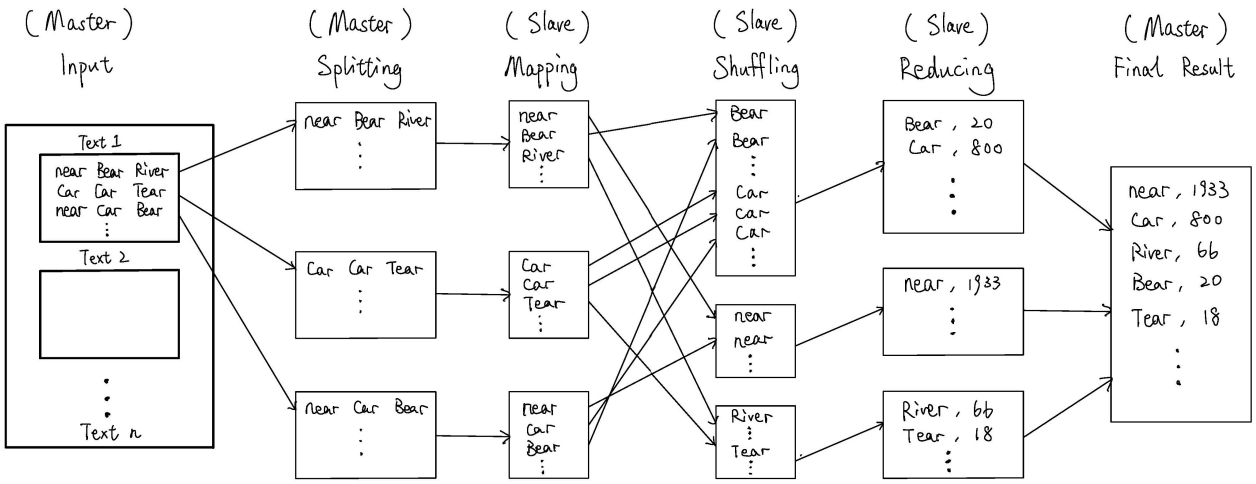

Figure 1: System diagram of the program

2 Functions description

2.1 *BASIC.java*

The aim of the file *BASIC.java* realizes the trivial method of word counting on one processor. Generally speaking, it reads files in a folder, splits the words contained, counts the number of occurence of the words and realizes a quick sort according to frequency and letter priority.

- Function *readFile* is designed to read all the files in a folder and split the words contained. Then use a HashMap to store the number of occurence attached to each word appeared.

- Function *wordCmp* is used for comparing priority of two words. The first criteria is the number of occurence while the second one is the name of the word. If data[i] has higher priority compared with data[j], then return true.

- Function *partition* is a mono-direction partition from left to right which put the words with a higher priority to the left side of the pivot and those with a lower priority to the right side.

- Function *sort* realizes a quick sort without recursion by using stack. Function *wordCmp* and Function *partition* are also called to realize the quick sort algorithm.

- Function *output* writes all the words and corresponding frequencies into a file to store them.

2.2 *MASTER.java*

The file *MASTER.java* defines the actions in the master program. In our design, master is called twice, one for the first step: splitting and distributing while the other for the last step: gathering and final sorting.

- Function *distribute* is designed to read all the files in the folder and divide them into three files (Sx1, Sx2, Sx3) then distribute to the three slaves.
- Function *collect* is used to gather the files already reduced by the slaves (RM1, RM2, RM3) and store the results for sorting.
- Functions *partition* , *wordCmp* and *output* are the same as those in BASIC.java.
- Function *sort* realizes a quick sort using the files gathered by Function *collect* and obtain the final results *result.txt*.

2.3 *SLAVE.java*

The file *SLAVE.java* defines the actions in the slave program. We have created three slaves using docker. Each one is responsible for Mapping, Shuffling and Reducing work with the files it receives. In our design, each slave is called twice, first time for mapping and transfering while the second time for reducing.

- Function *map* is designed to allocate the words by the beginning lettre. We use the first letter "R" and "i" to divide the words on a slave into three parts and store them respectively into three files.
- Function *transferSM* sends the files obtained by Function *map* to the corresponding slave. For example, on slave 1, we have the original file received Sx1. We divide it into SM11, SM21, SM31 in the previous function, then send SM11 to itself, SM21 to slave2 and SM31 to slave3.
- Function *reduceSM* is used for reducing the files obtained on a slave after mapping. For example, for slave1, it receives the files SM11, SM12, SM13 from itself, slave2 and slave3. Then it count the frequency of the words contained in these files (reducing) and store them in the file RM1.

3 How to build/run

3.1 Preamble

Our program should be run on Linux. Docker and Java environment are needed. There are also some preparation works:

• Use docker to create three containers with names: worker1, worker2 and worker3. Don't need to run them in advance, all the implementations are written in *deploy.sh*.

• Create a virtual network and confirm that all the three containers are connected.

• Try to transfer files between containers using scp service (ex. Hello.txt). Just to avoid the confirmation of transmission when we run the program.

• Type a command in the shell using sudo and put in the password. Just to avoid entering the password later.

3.2 Build and Run

All the instructions have been written in the file *deploy.sh*, including the start of containers, implementation of master and slaves, file transmission between the master and slaves, and timing. We firstly compile the three Java files then test them using the following commands:

- Use "*sh deploy.sh compile*" to compile the three .java files into runnable jars.
- Use "*sh deploy.sh basic #foldername*" to test the trivial method. The result is stored in *result.txt*.
- Use "*sh deploy.sh master #foldername*" to test the MapReduce method. The result is stored in *result.txt*.

```
wangfu@ubuntu:~/java/CNS_pr2/Java_homework/CNS2$ sh deploy.sh basic corpus
2019/12/27-16:03:39:746
basic: finish reading files
basic: finish sorting
2019/12/27-16:03:54:087
14489 ms
wangfu@ubuntu:~/java/CNS_pr2/Java_homework/CNS2$ sh deploy.sh master corpus
Docker version 19.03.5, build 633a0ea838
2019/12/27-16:04:18:067
master: finish file arrangement
worker3
worker1
worker2
worker1: finish mappingworker2: finish mapping
worker3: finish mapping

worker2: finish transferring SM
worker1: finish transferring SM
worker3: finish transferring SM
worker1: finish reducing
worker2: finish reducing
worker3: finish reducing
2019/12/27-16:04:58:795
40916 ms
```

Figure 2: Demo of build and run

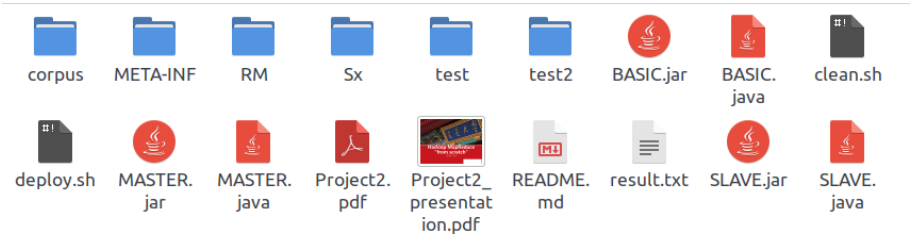

Figure 3: Files used and generated by the program

4 Experiment

4.1 Comparison of two methods

We have done the experiment for three files with a size of 117B, 8.2MB and 193.8MB respectively. The test results are shown as follows:

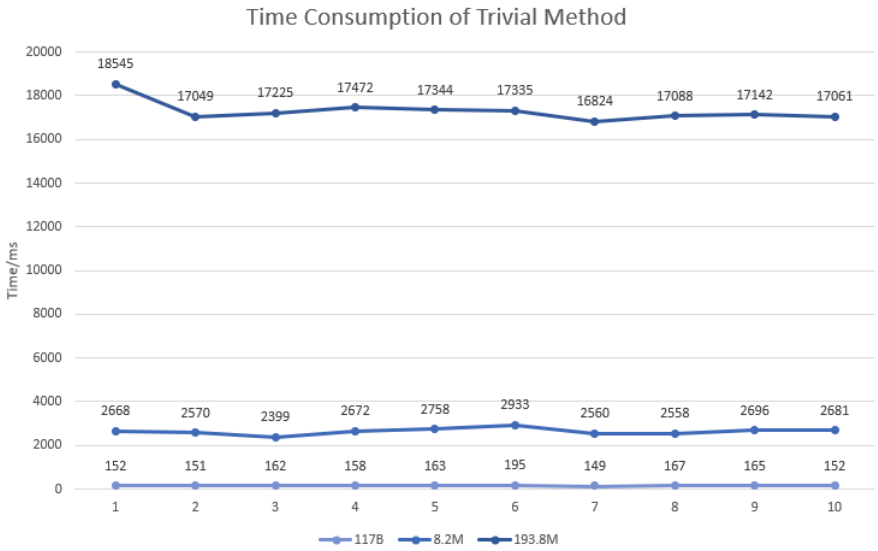

Figure 4: Time consumption of Trivial method

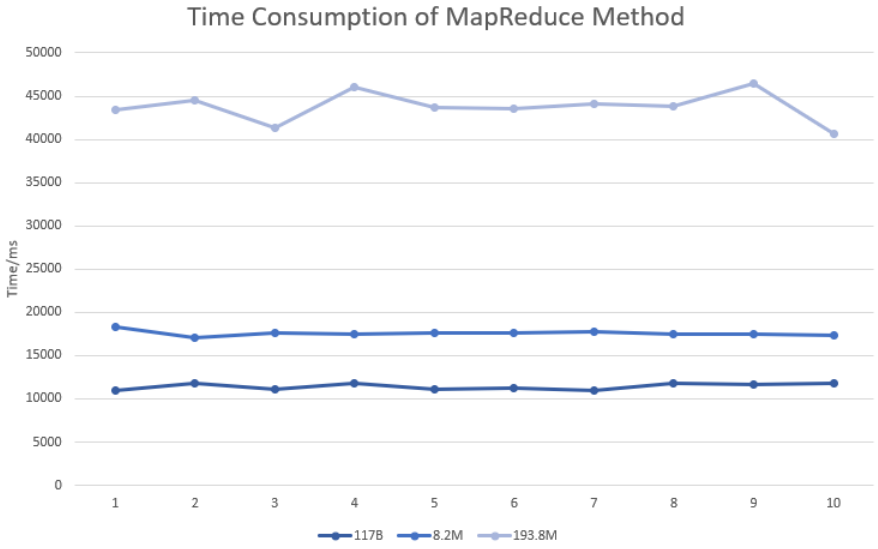

Figure 5: Time consumption of MapReduce method

The three curves of time consumption show that both of the two methods are stable for files with different sizes. As we can observe in the chart of performance, the bigger the file is, the more time these two methods consume. Theoretically, the MapReduce method should consume less time than the trivial method. However, in the experiment, we obtain the opposite result. After analyze and discussion, we give the following reasons:

- The reading, creation and transimission of the files cost too much time for MapReduce method. We can see that even with the smallest file, MapReduce method still needs more than 11 seconds in average.

- We have done the work with docker on one computer where all the processes share the resources of one processor. If we use three computers to execute the slave programs, the result may be better.

- The input file is still not big enough to show the power of the MapReduce method. We can see that, with the increase of the file size, the time consumption of trivial method augments faster than that of

MapReduce method. Hence, if the test file is large enough, for example, several GB, it is certain that the MapReduce method will cost less time.

- The operation system may use more than one cores or threads of the processor to run the trivial method. But as we generated three slaves on one computer, each slave has at most one core to use (the processor only has four cores). The available threads for each slave is less than that for the trivial process.

4.2 Improvements

To improve the performance of our MapReduce method, we can have some improvements in the algorithm as follows:

- We can avoid reading the input file line by line and dividing it into three files. In reverse, it is possible to distribute the whole text files one by one, which can economize the time of reading files.

- In our program, we create three slaves. It is possible that with more slaves, the performance of MapReduce could be better.

- For the mapping step, we divide the words by the first lettre "R" and "i". Maybe it is not the best choice, some of the slaves receive much more words than the others. We can try other choices to make the mapping more equal.

- The last thing is to run the slave program on different computers and test with larger files.

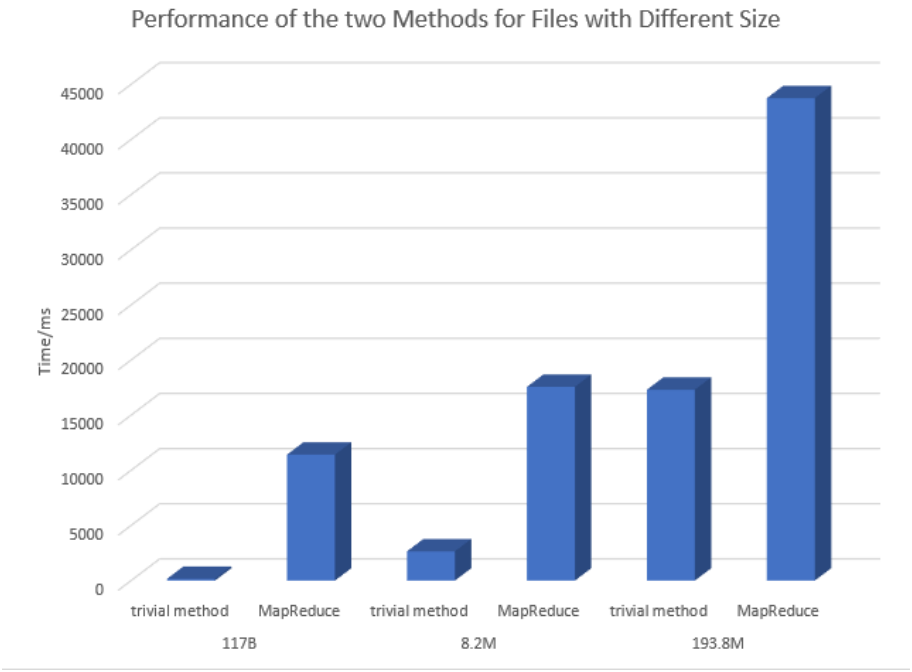

Figure 6: Performance of two methods for files with different sizes