MapReduce Report

**Index**

1 A short introduction 2

2 What the system does 2

3 Architecture design 3

job\_master.sh test 4

4 Advanced scenario and analysis 4

4.1 Check other fields 4

4.2 impact of distributing the data 4

5 Challenges and solutions 5

5.1 Input checking and initialization 5

5.2 Code backup and creating pipes 6

5.3 Semaphores 6

5.4 Fully distributed version 6

6 conclusion 7

Xinyue Wang

17200583

# A short introduction

MapReduce engine is a simple system that can take some formatted resource files and calculate the amount of a certain element.

I started my work with the map function, testing it little by little. Then I built my job master second, and tested the connection between these two. Semaphores were a challenge. It took me a while to figure out all the critical sections and how to protect them. After job master and map were finished I start to write my reduce.sh.

The basic idea behind MapReduce is quite charming. Instead of sending data to the function, it tries to send simple and basic functions to the large amount of data, responding to simple returns quickly. By building up simple, fast functions, and synchronizing their work, the MapReduce is highly efficient.

# What the system does

In single node MapReduce, this is how things work:

1. Initialize the system, wipe everything in map\_result folder and keys file. If any lock left over from last abnormal stop, it will be taken care of too. If any pipe doesn’t exist, it will be created.
2. The job master counts the files in a certain folder and starts the same amount map functions in the background. Each map function will handle a file.
3. Map functions cut the key from each line, return the key with a value 1 after it to a file named after the key. If this file doesn’t exist, one is created. If it does, it’s appended in it. In the meantime, map will send the key it meets to job master. Job master will check if this key already exists in the keys file. If so, the key will be ignored, if not, the key will be appended to the keys file.
4. After map is finished parsing a file, it will return ‘map finished’ to job master. Job master will stop listening to the pipe after it reads same amount of ‘map finished’ as running map functions, which corresponds to the number of files in first\_example directory.
5. After all map processes have completed, job master will count the keys in keys file, and initialize same amount of reduce functions, each one dealing with one key (one file in map\_result).
6. After a reduce is done its job, it will send the key and the count to job master, job master will echo it to the terminal. If job master receives all the result from reduces, it will stop listening on pipe and print ‘I’m done’.

The advanced scenario just tweaked a little bit about input folder and other things, basic logic is the same.

# Architecture design

To prepare the system, I wrote several scripts to make sure everything ran smoothly in my job\_master.sh.

First was to wipe all the things inside result folder. I used rm function to do that just with a wildcard \* - meaning “everything” inside the folder.

Second thing was to wipe clean the keys file. There are several ways to do it, I choose to send a true to keys file to wipe everything inside.

Also, I wanted to clean up if there are any undesirable lock files left over if last execution didn’t finish properly. If simply try running rm \*-lock here, this will give error if there’s no file name like that. Therefore, I use an if condition to check a ls result before running rm \*-lock. If there are any files named with a -lock, they will get wiped before any serious work starts.

I tried to figure out how to kill all the processes named with a .sh which were still running as part of the cleanup job. Unfortunately, I didn’t know how to separate my job\_master.sh from other .sh processes to avoid this command killing my job\_master.sh.

I wrote an if condition to check if the pipes exist and if not create them.

I checked the first parameter as a directory address, if it wasn’t, an error message was sent to my errorLog file and program exited.

I listed all the files in $1 (parameter one of job\_master.sh), passed the result to word count function with a pipe. The file number is line number minus one, because the last line will be a total.

I passed all the files in $1 to map.sh and ran them parallel in background.

The map scripts started to read lines from the files given to them, cut the elements needed. I used semaphores to protect a critical section where the key/value pair is sent to the output file.

I protected another critical section sending the key through map\_pipe to job master.

After parsing, I protected the last critical section sending ‘map finished’ line as an ending signal through map\_pipe to job master.

Back in the job master, map counter is the counter for how many ending signals have been sent to job master. I initialize it to zero. Every time I receive an ending signal through pipe, the counter was incremented. If the message through pipe is not an ending signal, the line is sent to a grep function to check if it exists in the keys file. If it does this signal will be ignored else, it will be appended to the keys file.

At this point the map scripts have done their jobs. The information has been sorted into different files is ready for further processing. Time to reduce.

Job master counts the keys in keys file and initializes a reduce counter to zero, which will keep track of how many reduce functions have finish their job and sent feedback.

Then the job master reads through the keys file and for every key creates a reduce function to handle it.

Reduce function is simple, it counts lines in the file and return the key and sum number to job master through reduce\_pipe. Writing to the Reduce\_pipe is a protected critical section.

Job master will keep track of the messages received from reduce functions and echos it to the terminal. If all the reduce functions have sent their feedback, job master will echo a finish line and exit normally.

### job\_master.sh test

job\_master.sh is the main part of the whole project. it’s the hardest to code and hardest to test. I did the test and build up work piece by piece.

Some main tests which were run are as follows:

* Counting files in first\_example, echo the result in terminal.
* Start same amount of map.sh in background, check by ps.
* Build connection between job\_master.sh and map.sh by sending certain information and echo on terminal.
* Collect keys and save them in the keys file. cat the keys file to see does it contain the right keys.
* Test the while loop which oversees the collecting of finish signals from map.sh by echoing count numbers on the terminal.

# Advanced scenario and analysis

## Check other fields

I tried to modify map.sh to check and group country information. Little things need to be changed. All that was needed was to change the cut field in map.sh to the country (which is from -f2 to -f8).

reduce.sh didn’t need to be modified as it collects the key from keys file.

If you wish to test this yourself, please feel free to run job\_masteri.sh.

## impact of distributing the data

By running the same workload on the same machine using time function I got the following results:

|  |  |  |
| --- | --- | --- |
|  | Run 5 different files | Run a single csv file |
| Real | 0m14.283s | 0m12.346s |
| User | 0m4.001s | 0m4.049s |
| Sys | 0m4.436s | 0m4.398s |

The result shows the single map takes less time to finish than five maps, this is unexpected and I think the reasons could be:

1. Content switch is time consuming as my computer only has 2 cores instead of 5.
2. Sleep 1 in p.sh could be triggered and turn into some waste of time while only one file don’t really have critical section problem, so no time waste while waiting for resource to unlock.

# Challenges and solutions

## Input checking and initialization

The input check and initialization are on my mind since I always forget to pass a parameter to job\_master.sh and this can results in some nonsense and leave a huge bunch of undesired processes and locks in my system.

First, I needed to put an input check to give me error message and exit the script abnormally. I remembered I used to do this in one of the labs about semaphores. It has an input check for the number of parameters, there should be at least one parameter passed in.

So, I adjusted the code to require one and only one parameter. If there’s any more or less parameters being passed in, it will return an error message on terminal.

Other than making sure the user only gives one parameter, I need to ensure the parameter being passed in is a directory/folder that’s valid. Therefore, I got a directory checker to make sure the parameter is a valid directory. If it’s not, it will return error message and exit abnormally.

After done this, I still wanted to improve it by both showing error messages on screen but also saving the error messages in an error log, so I could check my errors later. I did some research on stackoverflow, someone had a whole table of standard output code snippets. I learned that to both output to a terminal and write to a file, you need to pipe the result to a function called ‘tee’ with argument of my error file, and use a ‘-a’ to append. The input checker is almost done at this point. One more thing missing is the time stamp.

The code to get a time stamp is quite simple, $(date). It’s a global variable that you can call any time you want and will return you the time in a standard format. I fed this to echo and everything about input checking is now complete.

The second thing is that the result is being written into the same files again and again. So, if the file is never being wiped clean before running again, the file will grow larger and the result will be incorrect. Cleaning out the files is a tiring job to do by hand, so I wrote some lines as an initializer to do all the clean up job before any code is run again. Minimum requirement is to remove prior results in results folder and remove the content in the keys file. Wildcard is helpful here, so I tried rm function in cml first and then coded it in my job\_master.sh.

I really need to mention that rm is a useful but dangerous function, I accidently rm some files unintentionally, I’m lucky they are not important files. Still, after this lesson, I backup my whole work in my Ubuntu VM and csserver from time to time. Moreover, I always touch some sample files in a solitary folder and test my command first before I do some real potentially damaging work.

After these two lines, another thing to come to mind, there are always undesired lock and processes left over in my system after any unsuccessful run. I needed to run ‘rm \*-lock’ and ‘killall \*.sh’ in terminal. I wanted to automate this step. So first I simply put ‘rm \*-lock’ in my job\_master.sh. It worked fine until a situation where there were no more undesired left over files from last (successful) run. In that situation it gave an error about being unable to find any matching file in the directory. Even though this did not affect the correctness of the output, I didn’t want this error! Therefore, I put this line in an ‘if’ condition to check whether there is a lock using ls as test condition. The ls result was redirected to /dev/null so it will be silent in the terminal.

This is the good, here comes the bad. I didn’t figure out how to run a ‘killall \*.sh’ and avoid job master from killing itself. I went through a lot of research and finally gave up because the time spend on trying to fix that problem outweigh the time I need to type ‘killall \*.sh’ every time after abnormal exit of job master.

## Code backup and create pipes

As previously mentioned, I got frightened by the rm function and decided to backup my work. After research I realize there are two ways to do the job of sending files, but I don’t really understand the difference between these two commands. It’s rsync and scp. Both used with a -r to recursively traverse through the folders. I chose scp simply because its name is shorter. I backed up from my Ubuntu VM to csserver using the command scp -r, follow by the address of the file I wanted to sync, and the server to try to send my files to.

After backup, I felt safer and more daring when running rm. Something was missing though. My pipes are gone!

Google told me that it’s because a pipe is not regular file and cannot be sent by both rsync and scp.

So, I needed to write some code in job master to have a check and create the pipes if needed before every new run.

The logic is quite simple, check if there is a certain named pipe, if not, create one.

## Semaphores

Semaphores are easy to talk about in theory but hard to code and check. The appearance of synchronization is based on content switch so basically, it depends on luck and large amount of test to check if it works.

Without a doubt any area of code where more than one process might try to get into at the same time is a critical section. When job master starts five parallelly running map.sh, they might try to write to the same file together.

That’s why you need to protect ‘writing to file’ and only one map.sh can be in the critical section at a time.

Other than that, the map.sh could try to send message through pipe at the same time too. Which means pipe should be protected too.

p.sh and v.sh are quite like the ones we’ve learnt. The main idea is to create a hard link between the file which needs to be protect and a symbolic lock file.

Sleep 1 in p.sh is time consuming. This is apparent while running, if you output the keys to the terminal, sometimes you can see a stop caused by sleep 1.

## Fully distributed version

Even though I tried to build a link between VM and csserver, the result is just too disappointing. I spent a whole week searching through internet, talked to my classmates and tried all the potential possibilities. There was simply no way I could get the VM IP address and set it into listen mode and receive message from server.

But I still got a thought through about how to fulfil this task in my mind.

master.sh can have a condition checking for pipes. If the pipes don’t exist, create them.

I would built a ‘job\_masteri.sh’ on my server side and let it run in the background, listening to the network pipe all the time. Whenever it receives an instruction from the local host, it will check if it is an available file. If it is, start a map.sh to run it.

The master.sh would have network pipe to communicate with server-side job master and local named pipe to communicate with local job master. While their map.sh are mapping, they will send the keys received from map.sh to master.sh. The master.sh should be in critical section since two job masters could try to send keys to it at the same time. Then master.sh will write the whole keys into a global\_keys file. After both job masters finish their jobs, they will send an end signal to master. If master receives two end signals, it will stop knowing the maps are finished and will start the reduces now.

Then the master will count the lines in global\_keys and cut the lines into half. Once the list is determined, the first thing that will need to be done is that the files will have to be swapped between the two nodes. Say on local side we have map result p1, p2, p3, and on server side we have p1’, p2’, p3’. We decide to let local job master handle p1 and p2, server handle p3. We then need to send p3 to append after p3’ and p1’, p2’ to append after p1, p2.

Then use the reduce functions to count the lines in each appended file and print the result.

# conclusion

After finishing this project, I gained some more self-belief in coding bash. Before it, I always felt quite unsure of myself when coding in bash, not only because of the strict syntax, but also the logic, especially in synchronization, is difficult to comprehend. After weeks of trying and seeing how my code now runs smoothly, I feel it’s all worth it.

Still I feel disappointed that I could not handle the fully distributed MapReduce. If I’ll have time later, I’ll give it another try.

Thanks for Anthony, it’s really a pity we cannot have him for more fun lectures.

Lastly thanks for Ernestas, all the TAs of OS (They are the best TA group ever!) and other helpful classmates.