

1. Introduction

The purpose of this project is to create a MapReduce engine in bash. MapReduce is a big data processing system, and like other big data processing systems it makes use of parallel processing and distributed systems to speed up the processing of large volumes of data. MapReduce uses two simple functions, map and reduce which process data and produce results in key/value pairs. The map scripts will process the raw data and produce an intermediary result which is processed by the reduce scripts to produce a final result. These map and reduce tasks are managed by a job master program which manages map and reduce tasks on individual machines. In the case of distributed systems a further program, called the master node, is required to manage individual job master programs. For the project the first step was to create a centralised version of the MapReduce engine in bash, then to test advanced scenarios, such as what effect parallel processing has, and finally to create a distributed version of the engine across two machines.

2.1 Requirements part (i)

The first part of this project requires that we make a centralised (single node) MapReduce engine.

This single node engine requires a map.sh script a reduce.sh script and a job_master.sh script which will manage these map.sh and reduce.sh tasks.

For the purpose of this task we were given a list of transactions to process. This data was saved in two formats: either one single file containing 100 transactions, called SalesJan2009.csv or in 5 separate files containing 200 transactions each which are called SalesJan2009aa SalesJan2009ab SalesJan2009ac SalesJan2009ad SalesJan2009ae.

The user should run the job_master.sh script with a subdirectory containing either SalesJan2009.csv or the five smaller files as argument.

When the job_master.sh is run it will start one map.sh script per file.

For each distinct value in field 2 which contains the product purchased (either Product1, Product2 and Product3) the map.sh will create a file for each of these values. It will then go through the file line by line and save a key value pair containing the product name and the value one to the appropriate file i.e. the one with the same name as the product name.

When the map.sh processes are finished they should be able to communicate this to the job_master.sh. This will be done using a named pipe.

The job_master.sh will then start a reduce.sh script for each of the files containing key/value pairs that was saved down (3 in this case).

The reduce.sh scripts will count each of the key/value pairs in each file and reduce this to one key value pair containing the product name and the count e.g. for Product 1 this will be Product1 847. This will then be communicated to the job_master.sh which will print it to the screen and then terminate.

The processes should be able to run concurrently without any synchronisation issues. i.e. the correct output should be produced every time.

2.2 Requirements part (ii)

The second part of the project is an extension of the first. The additional requirements for this section are to check other fields other than the product ids as well as to analyse the impact of distributing the data.

2.3 Requirements part (iii)

In part three the requirement is to create a fully distributed version of the single node MapReduce scenario in part 1. This means that we should be able to run this scenario across 2 machines.

Achieving the above requires a further script, called master.sh. This master node will manage a job_master.sh script on the local machine by communicating using named pipes and it will manage another job_master.sh on a distant machine using a network pipe.

Once the job_masters have started their map.sh scripts as usual and these map.sh scripts have finished, the master node will need to select half of the keys to be processed by the reducers on one machine and the other half will be processed by reducers on the other machine.

The above requires that the key/value pairs that have been saved down by the map.sh scripts are “swapped” so that each machine has the key/value pairs that its reducers will process. This requires sending files over the network pipe.

Once all key/value pairs have been processed by the reducers the final count will be displayed on the screen.

3.1 Architecture/Design part (i)

The first thing that was done was to create the job_master.sh, map.sh and reduce.sh scripts. This section has one subsection per script as well as a section on synchronisation explaining why I did not need to use semaphores for the project.

For the best comprehension, reading along through the code alongside these sections is recommended.

3.1.1 job_master.sh

- check if the user has provided an argument, if not, exit with exit code 1 and prompt user to provide subdirectory as argument
- check if the argument provided by user is not a directory, exit with exit code 2 and tell the user that an invalid directory was specified
- check if a named pipe called map_pipe exists, and if not, create one (this assumes there are no non named pipe files called map_pipe)
- check if a named pipe called reduce_pipe exists, and if not, create one (this assumes there are no non named pipe files called reduce_pipe)
- Iterate over all of the files in the directory given as argument, and start one map.sh script per file, giving the file name as argument to the map.sh script.
- count how many map.sh scripts were started
- check if a file called keys exists, if not create one.
- Use a while loop to read keys sent from the map_pipe (see section 3.1.2 for details on information sent from map.sh scripts) while there is still at least one map.sh script running. This is done using a check variable which will be incremented when each map.sh finishes.
- Unique key values read in the above step will be appended to the keys file

- Read the keys file and iterate over each value in the keys file, start one reduce.sh script per value, giving the value as argument to the reduce.sh script.
- count how many reduce.sh scripts were started
- Similarly to what was done with the map.sh scripts, use a while loop to read key/value pairs sent from the reduce_pipe (see section 3.1.3 for details on information sent from reduce.sh scripts) while there is still at least one reduce.sh script running. This is done using a check variable which will be incremented when each map.sh finishes.
- The input read in the above step will be printed to the screen showing all of the correct counts of the products and the job_master.sh script will terminate.

3.1.2 map.sh

- Use the cut command to read the second field of the file given as argument. This field contains the products.
- Iterate over the values returned by the above step.
- The map.sh will check for each value if a file with the same name as the value exists, if not it will create it then append the value to the file and echo the unique key to the map_pipe to be read by the job_master.sh script.
- If the file already exists it will simply append the value to the file
- After the map.sh script has iterated over all values it will send a message indicating that it has finished to the job_master.sh. Once this message has been read it will terminate.

3.1.3 reduce.sh

- Read the file that was given as argument and iterate over all values in the file
- for each value in the file reduce.sh will increment a count variable
- It will then echo the key value pair of the file name (one of the products) and the count to the job_mastersh via the reduce_pipe.
- It will then send a message indicating it has finished to the job_master.sh

3.1.4 Synchronisation

As mentioned previously I did not include semaphores in this project. The reason for this is that the only time that concurrent processes are trying to edit a shared file the operations they are using are atomic.

The map.sh scripts append key/values to files, but append operations are atomic in bash up to 4KB <https://www.notthewizard.com/2014/06/17/are-files-appends-really-atomic/> and we are only interested in the count of keys not their order, so there is no need to protect this section with semaphores.

The other situation where processes are writing concurrently is when they are writing to pipes. But writes to pipes are also atomic up to a minimum of 512 bytes <https://unix.stackexchange.com/questions/68146/what-are-guarantees-for-concurrent-writes-into-a-named-pipe>, so there was no need to protect these with semaphores either.

3.2 Architecture/Design part (ii)

Analysing different fields required amendments to the job_master.sh and the map.sh. This section is split into one subsection for modifications to the job_master.sh, one for the map.sh and one for the analysis of distributing the data.

When running this using a different field, I used field 4, the credit card names, and I got the following results when using the SalesJa2009.csv file:

Mastercard, 277
Amex, 110
Visa, 522
Diners, 89

Interestingly when using the five smaller files, I got the following results:

000", 1
Mastercard, 277
Amex, 110
Visa, 521
Diners, 89

The 000" anomaly was because there was a comma in the number 13,000 in field 3 of SalesJan2009ac and this was interpreted as a field separator. Note: this only showed up in gedit or nano, not LibreOffice.

3.2.1 job_master.sh

- The job_master needed to be modified to take an additional argument.
- This additional argument is the field to be checked (a number) and will be passed to the map.sh script when it is run.

3.2.2 map.sh

- The map.sh also needed to be modified to take an additional argument, this was also the field to be checked.

3.2.3 Analysis

1 file	5 files
.120 s	.109 s
.127 s	.105 s
.139 s	.104 s
.118 s	.106 s
.124 s	.106 s
.112 s	.104 s
.131 s	.105 s
.129 s	.110 s
.129 s	.102 s
.120 s	.107 s

Average time for 1 map process: .125 s

Average time for 5 map processes: .106 s

Although 5 processes were faster I was expecting a bigger speed up than 15% given that my pc has 2 cores. I thought that this might be due to the fact that these files (and times) are quite small and the speed up from using multiple processes is being overshadowed by the cost of spawning the additional processes. I decided to test on files 200 times bigger to see if this made a difference.

1 file	5 files
17.562 s	11.982 s
17.537 s	12.000 s
17.499 s	12.046 s
17.528 s	11.738 s
17.775 s	10.710 s
17.652 s	11.307 s
17.819 s	11.608 s
17.672 s	11.972 s
18.185 s	12.380 s
18.456 s	12.249 s

Average time for 1 process: 17.769 s

Average time for 2 processes: 11.799 s

Using larger files the speed up achieved was over 33% which was much closer to what expected.

3.3 Architecture/Design part(iii)

The first thing that was done was to create master.sh script, modify the job_masters.sh scripts and create two new scripts, read.sh and echo.sh to allow communication over the network pipe. This section has one subsection per script and one for the networking pipe.

3.3.1 master.sh

- The master.sh process first sets up a connection with the distant machine using netcat and the echo.sh and read.sh scripts (these are detailed in sections 3.3.4)
- The master.sh script will need 3 named pipes, in_pipe for incoming messages from the distant machine, out_pipe for outgoing messages to the distant machine and a local pipe for communicating with the local job_master.sh. If these pipes do not already exist the master.sh script will create them.
- It will then send directory names to local and distant job_masters.sh so they know which files to process
- It will then wait for a list of keys from the distant and local machines and will save these in a global_keys file
- Once a message has been received that all the keys have been sent the master.sh will create two new files, new_keys and send_keys. It will save half of the global_keys in new_keys to be processed by the local job_master.sh and the other half in send_keys to be processed by the distant job_master.sh.
- Now the master.sh needs to transfer files between machines. Since some map.sh scripts processed the keys in new_keys on the distant machine, these files will need to be transferred to the local machine
- Similarly some map.sh scripts processed the keys in send_keys on the local machine, and these files will need to be transferred to the distant machine.
- Once a message has been sent/received that all files have been transferred, the job_masters will run their reducers whose output will be sent back to and then displayed by the master.sh

3.3.2 job_master.sh (local)

- The only modifications to the local job_master.sh are that it was started by the master.sh and communicates with it via the local_pipe.
- certain portions of the script will only be executed once a message has been received from the master.sh

3.3.3 job_master.sh (distant)

- the distant job_master.sh will open a connection with the other machine using netcat, echo.sh and read.sh similar to the master.sh
- certain portions of the script will only be executed once a message has been received from the master.sh

3.3.4 Networking pipe

I used two scripts echo.sh, read.sh and netcat to set up a networking pipe between the two machines

echo.sh

echo.sh continuously listens reads from the out_pipe on the same machine and will echo what it reads to the netcat process running on the same machine. This will then be sent over the network pipe to the other machine

read.sh

read.sh continuously reads what is being output by the netcat process on the same machine and echoes what it reads to the in_pipe which will be read by processes on the local machine

Netcat

On the local machine netcat would listen on port 2012 and what it received from the distant machine would be piped to read.sh. What was written to it by echo.sh would be sent to the distant machine.

Netcat would work in the same way on the other machine, where it was completing the other side of the connection.

When running the distributed version it took approximately 2.7 seconds to process two 1000 line files vs .22 seconds for the centralised version. Given the small times seen here the overhead of using messages to communicate is making the centralised version seem far faster. However, I suspect that with larger files the distributed version would outperform the centralised version.

To measure the overhead generated by communicating over the network pipe I would suggest tracking the time between when each message is sent and when it is received and summing these times up over the course of the program.

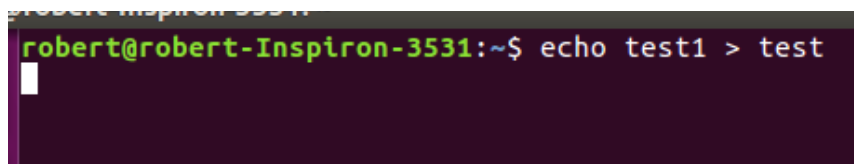
4. Challenges

4.1 Using named pipes to facilitate interprocess communication.

When I first used named pipes in this project I ran into a number of concurrency issues, which was very surprising because, as I mentioned above in section 3.1.4 there should be no concurrency issues when using pipes.

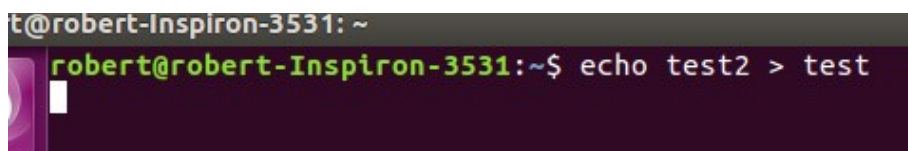
I believe the main issue I was having using pipes was the way I was reading from them. I was using the read command to read from pipes and this was resulting in only one value being read. The way I tested this is shown below.

First of all created a named pipe called test and echoed test1 to it



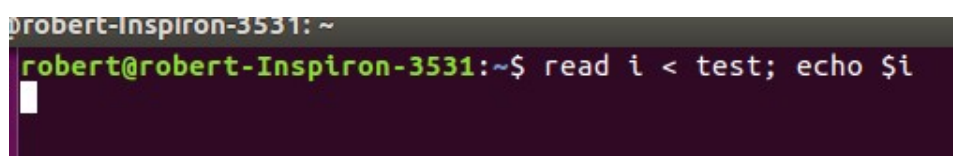
```
robert@robert-Inspiron-3531:~$ echo test1 > test
```

Then I echoed test2 to it



```
t@robert-Inspiron-3531: ~
robert@robert-Inspiron-3531:~$ echo test2 > test
```

Then I ran the below command twice (the idea being to read both writes to the pipe)



```
robert-Inspiron-3531: ~
robert@robert-Inspiron-3531:~$ read i < test; echo $i
```

The behaviour I observed is shown below and wasn't what I was expecting at all.

```
robert@robert-Inspiron-3531: ~
robert@robert-Inspiron-3531:~$ read i < test; echo $i
test2
robert@robert-Inspiron-3531:~$ read i < test; echo $i
█

robert@robert-Inspiron-3531:~$ echo test1 > test
robert@robert-Inspiron-3531:~$ █

robert@robert-Inspiron-3531:~$ echo test2 > test
robert@robert-Inspiron-3531:~$ █
```

After I tried to read the second value the command was blocked, which implied that there was nothing to read. Also, even though I had echoed test1 to the pipe first it was test2 that was read.

When I was investigating this behaviour I came across the following website: <https://wpollock.com/ShScript/Fifos.htm>. Here the author says that “Once the reader process is connected to the named pipe, all sender processes wake up. However the kernel **does not** run processes in any particular order, so there is no guarantee of FIFO behavior with multiple senders.”

Based on this, my understanding of what was hapening was that, my two writing processes had been blocked initially and were woken up when the reader connected, but the read only read one value and the other was lost.

I noticed that the author used used cat instead of read to read from pipes, and all writes to the pipe were being (although not necessarily in FIFO order) so I decided to use cat.

My format for reading from pipes changed from:

Reader

```
read i < pipe
echo $i
./V.sh #remove lock
```

to:

Reader

```
variable=$(cat < pipe)
for i in $variable; do
    echo $i
done
```

Writer

```
./P.sh #only allow one writer at once
echo message > pipe
```

Writer

```
echo message > pipe
```


The other big issue I faced when doing this project was when sending files over the network pipe in part 3.

At first the connection would always break when I tried sending over one value at a time. I instead tried concatenating all the values into a single string and sending this across the pipe in one go. I did this instead of sending the entire file, because for bigger files it may be necessary to split them into smaller chunks to be sent over the network pipe.

5. Conclusion

In conclusion I feel happy that I have met most of the targets I set out to achieve for this project. I found this project extremely engaging and I feel like I have come away with a level of knowledge of bash that I would never have expected to have at the start of the course. My knowledge of MapReduce has also been greatly improved and I'm looking forward to using the skills I've learned here in the future.