MAP REDUCE SPRING 2016

Final Project: Build a Custom MapReduce System

Team Members:

Yogiraj Awati

Sarita Joshi

Ashish Kalbhor

Sharmodeep Sarkar (Sharmo)

Professor: Nat Tuck

TABLE OF CONTENTS

PRE-REQUISITE

- Oracle JDK version 1.8
- AWS CLI + Configuration
- Folder structure on local host
- Files to be included before running make file
- How folder structure is created on EC2 – master and slave nodes
- AWS configuration

TECHNOLOGIES

- Core JAVA
- Linux Bash scripts for cluster management and configuration
- TCP/IP communication across Master-Slave configuration
- SCP communication across different EC2 instances
- POM for checking the dependency and downloading the same

JAVA FILES

- Package → custom.mr
  Configuration.java
  FileInputFormat.java
  FileOutputFormat.java
  IntWritable.java
  Job.java
  LocalJobClient.java
  Mapper.java
  PartitionHelper.java
  Path.java

Reducer.java

ReducerDriver.java

SlaveDriver.java

Text.java

- Package → custom.mr.utils

  FileChunkLoader.java

  FileIO.java

  ProcessUtils.java

  RunShellScript.java

  TextSocket.java

BASH SCRIPT FILES

start-cluster

stop-cluster

my-mapreduce

makessh

installjava.sh

run-jar.sh

config

awsSetup.sh

exportTos3.sh

mergeMapperOutput

createReducerInput

startSlaves.sh

dataShipper.sh

splitFileList.sh

createPseudoReducerInput

splitFileList_pseudo.sh

DESIGN & LANDSCAPE ARCHITECTURE

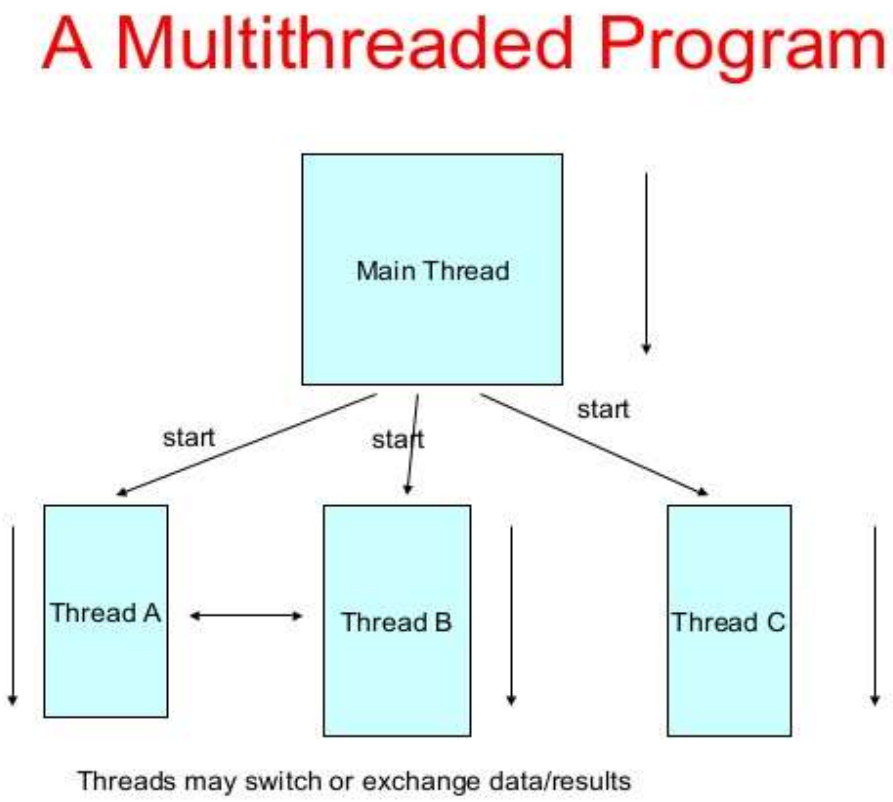
A. PSEUDO DISTRIBUTED MODE
B. AMAZON EC2

PSEUDO DISTRIBUTED MODE

Our custom map reduce implementation distinguishes the pseudo mode operation and EC2 mode based on the rules defined in Makefile. Accordingly the job task calls appropriate scripts and handles data distribution on different threads or EC2 instance respectively.

For Pseudo mode operation, we consider a constant chunk size of 10MB for splitting the data across multiple threads. The data distribution is handled by splittFileList_pseudo.sh

Based on the above split, we spawn those many these for map jobs to be executed in pseudo parallel mode.



Input Folder -> <localPath>/input/

Output Folder -> /tmp/part-*

This * depends on the number of reducer task defined as per Number of reducer set by the main class which is to be executed on our custom map reduce framework.

NOTE: For pseudo distributed mode, we provide local path our Hadoop framework. User must do the pre-processing on the data to retrieve the same via s3 bucket if required.

Output file is generated at /tmp which can then be synced to the user s3 bucket.

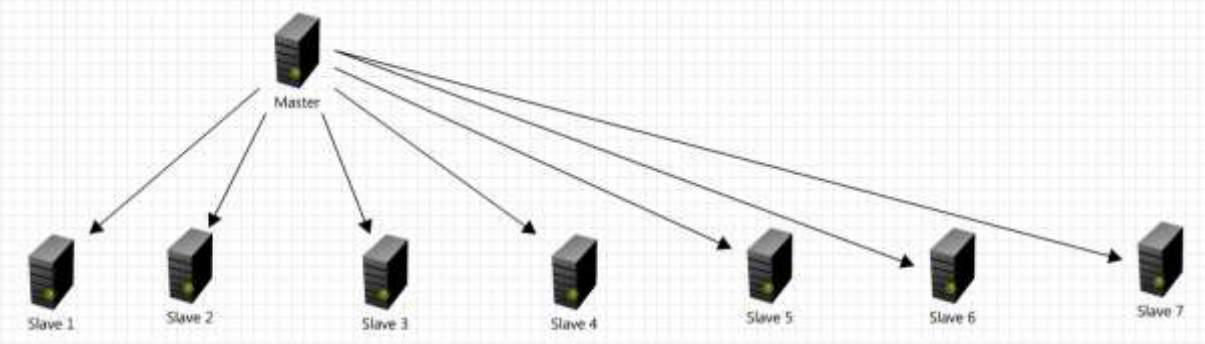
We follow the architecture same as that of the apache hadoop pseudo environment.

AMAZON EC2 MODE

CLUSTER MANAGEMENT

Our cluster management follows a Master-Slave configuration with the number of total EC2 instances determined by the start-cluster <arg>. This input argument determines the 1(Master) + (n-1) (Slaves)

Below is the representation for cluster configuration with input argument as 8:



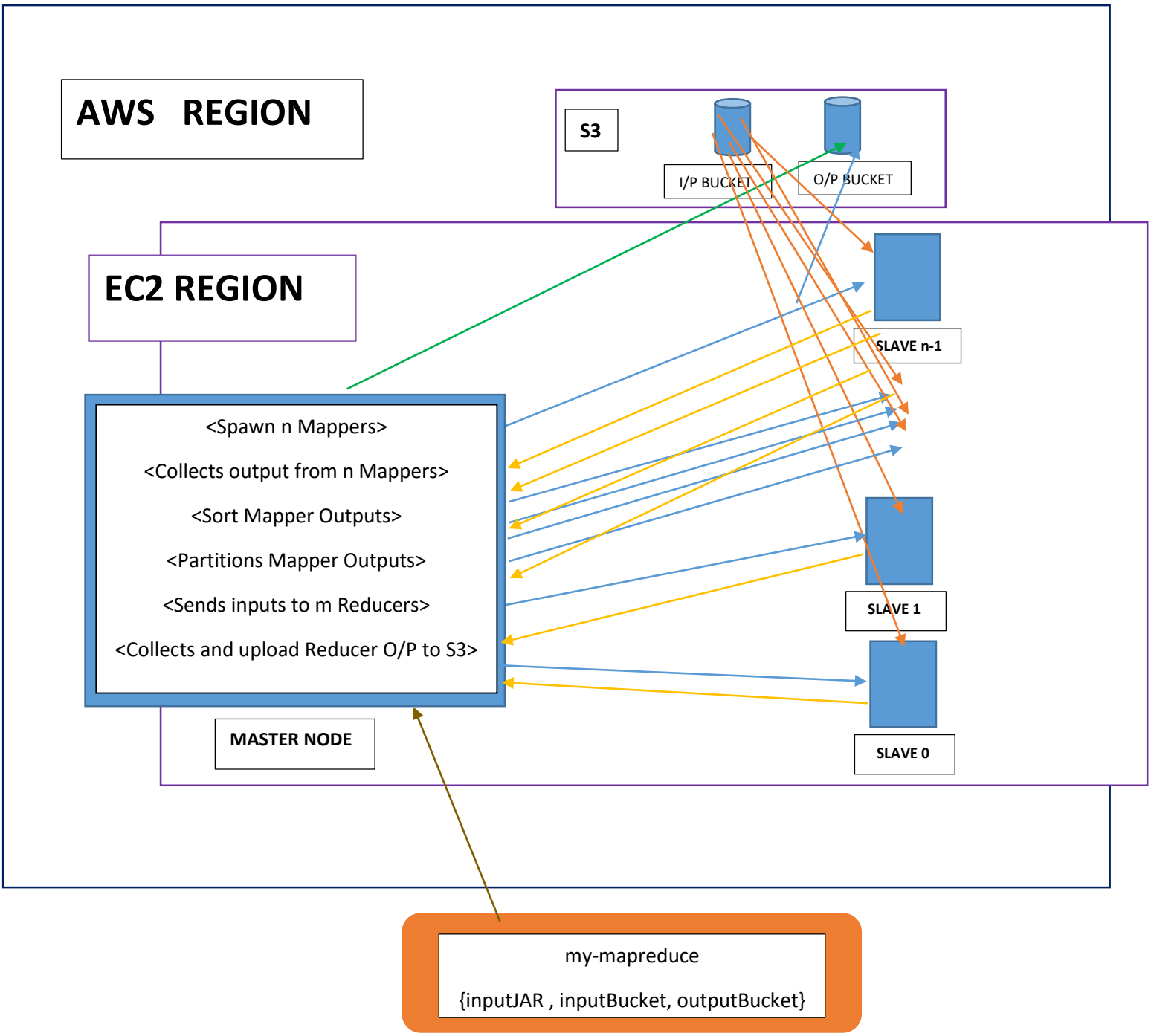
The master node stores the details of each EC2 instances in our current cluster and each slaves records the public DNS of the master node

Important Scripts-

Start-cluster (Starts N number of EC2 instances and configures SSH across)

Stop-cluster (Stops/terminates the running EC2 instances)

ARCHITECTURE DIAGRAM



Role of Master Node

Input jar (eg. WordCount.jar) runs on master node. The main class that implements functionality of Master Node is Job class. When the control comes to Job.waitForCompletion() , following process takes place:

- DataSplitting

  We make connection to S3 bucket which is given as input to the program. We process the data by creating list of files as an input to mapper which are of approximately equal size. The outcome of this script is 0.txt , 1.txt .. and depends on number of mappers set by the start cluster argument. So if ./start cluster 8 , total of 7 mappers would be spawned and the functionality attempts to make 7 list of files to be downloaded by the mapper function on SlaveDriver.

- Datashipping

  This script transfers the files generated by Datasplitting to respective mapper. Like 0.txt is send to mapper with cluster id 0.

- We run startSlave.sh that will start the Slaves in listening mode. Basically main function of SlaveDriver is called which makes slave nodes in listening mode
- We send a request message to all SlaveDriver using TCP socket. This message contains start mapper tag along with mapper and reducer class name. The server at SlaveDriver can identify this request using the tag and start the given mapper class.
- Master will now go in listening mode till it receives success acknowledgement from all mappers.
- Once all Mappers send their respective output to Master Node through SCP, the mapper outputs are accumulated in /tmp/allMapperOutput/ folder.
- Master Node sends a kill request to all SlaveDriver which terminates Mapper process on all EC2 instances. At this point we have intermediate output from all the Mappers on Master Node. We need to shuffle and sort this intermediate Mapper outputs. Following is the shuffle process
- Shuffling (Creating Reducer Input):

  In this phase we process Mapper outputs and create reducer input. Following is the process:

  - MergeMapperOutput
    Retrieves the key i.e the unique filename and the merges the data i.e prepares the data as per key against it's combined values to be passed to the default partitioner for preparing the input of the reducer.
  - CreatePartitions

    It takes summary and number of reducers as an input. Its partitions keys according to number of reducers using hash function. Accordingly, r0, r1 files are created which contains the keys list respective reducer0, reducer1…will get as an input.

  - CreateReducerInput

    r0 is moved to r0 Folder and ships to respective EC2 instance and then runs reducer driver against it. Basically we perform a shuffling based on the has function and prepares the reducers input as r0, r1

- Master Nodes goes into listening mode and waits for all Reducer success acknowledgements.
- For each acknowledgment received from Reducer, Master node receives part-* folder (reducer output) through SCP.
- export.sh exports all part folders received to S3 bucket received as an output path

ROLE OF SLAVE DRIVER (MAPPER FUNCTIONALITY)

- Slave Driver is instantiated by Master node by running runslave.sh script. Each instance has cluster id required for the processing.
- Downloading data from S3 bucket:
- FileChunkLoader reads the 0.txt or 1.txt etc files present at that EC2 instance. This text file contains list of files to be downloaded from S3 bucket. This function creates connection with S3 bucket and download files from S3 bucket to /tmp/input folder
- Starting Mapper using fetchMapperInput() :

  For each file entry in the /tmp/input folder , map() function is called. The implementation of the map() is given by Jar provided as input (WordCount.jar)

- context.write() :

  This function process each key . It replaces “[:*?/<>|]” with “” for each key. Creates file with key name at location /tmp/output(clusterId) and writes value associated with that key in that file

- Output of Mapper:

  At the end of all map() calls, we get output of the Mapper phase at location /tmp/output folder, which contains files with key name and values of that key inside respective key file. The output is sorted by key since we are creating files by key names.
- SlaveDriver than SCP its output to Master Node and goes into listening mode and waits for Kill message from Master node.
- Once SlaveDriver receives kill message from Master Node, Mapper terminates

ROLE OF REDUCER DRIVER (REDUCER FUNCTIONALITY)

This is instantiated by createReducerInput script which runs on Master Node. The input to Reducer is made available by Master node.

- reduce() of the provided jar (WordCount jar) is called on each file that is available in the input folder.
- Context.write() : Writes output to location /tmp/part-clusterid /ouput.txt
- Once all reduce calls are done on each key, Reducer Driver SCP its output to Master node and exits

ISSUES ENCOUNTERED and TROUBLESHOOTING

A. Acknowledgements: Acknowledgement was one of the major challenge we faced to make data transfer synchronous between EC2 Nodes. When Master node instantiates Mappers, Master node goes in the listening mode. It waits for Success Message send by Mappers once they finish their execution successfully. Such kind of acknowledgement helped us to successfully accumulate Mapper phase output at Master Node. Such kind of logic is also used in Reducer.
B. Multithreading: There was a case where we wanted Master node start Mapper nodes simultaneously to speed up the processing. So we made connections with Mapper instances simultaneously using thread.
C. Server/Client Implementation: Since there was only one JAVA program "SlaveDriver.java" running on each EC2 instance and the requirement was that the program should listen as well as send data through socket, we wanted to distinguish this feature. This was achieved by assigning cluster id to each node and cluster id =0 was made master. The logic of when to accept/receive data was manages accordingly.
D. Referring inner class of WordCount from a Mapper EC2 instance: Mapper instances needed names of inner class. So this was made possible by sending these name in the ACKs send from Master node to Mapper nodes.
E. When the number of Mappers required are less than number of Cluster instances instantiated: There was a situation where number of mappers required for the processing data was less than total number of EC2 instances instantiated. So we had to explicitly kill the SlaveDriver program running on these idle instances. We achieved this by broadcasting Kill message from Master node to all Slaves which enabled integrity of the process

STEPS TO EXECUTE

- make setup (This to do the required pre configuration for JAVA and AWS configuration)
- make jar (To create the required custom hadoop jar file as per test suite)
- make EMR (Runs the given jar in AWS EMR Mode)
- make pseudo (Runs the given jar in pseudo distributed mode using JAVA multithreading)

TEST-SUITE

- WordCount.jar: Runs in 1 mins in Hadoop & 50 secs in pseudo mode
  And less than a minute on our framework
- WordMedian.jar: Runs in 1min 20 secs in hadoop and 1 min in pseudo
  Mode less than a minute on our framework
- Assignment 2: Runs in 90 mins in hadoop & 60 mins in pseudo mode
  And 75 mins on our framework
- Assignment 5: Runs in 120 mins in Hadoop & 80 mins in pseudo mode and
  100 mins on our framework
- Assignment 4: Runs in 90 mins in Hadoop & 60 mins in pseudo mode and 75 mins on our framework

As discussed clearly in the conclusion of our interpretation, it works better than apache Hadoop environment. Yes, our code produce perfect output in less time.

TEAM CONTRIBUTION

1. Yogiraj Awati: Created architecture for Custom Map Reduce system and implemented it, Implemented Socket Programming for communication, defining hash function for partitioner (hash function tested on 18K Words)
2. Ashish Kalbhor: Handling deployment of EC2 instances, port settings for EC2 Node communication, system Implementation, Java reflection
3. Sarita Joshi: The design for our cluster landscape architecture E2E implementation. EC2 cluster automation, required environment setup on each instances and background job execution. Designed the flow of distributing the data based on size ratio to each of the instances. Involved in client server communication
4. Sharmodeep Sarkar: Implemented the Local client and integrated the scripts with JAVA. Designed the approach for distributing the data during mapper phase. Implemented the bash for data splitting across various dataset. Created the bash for receiving desired output from the EC2 instances to s3 bucket as well as to the local client system.

All team members equally contributed to system landscape and java configuration/thought processing/trouble shooting. This was a challenging yet interesting project.

CONCLUSION

The above describes our implementation of a Custom Map Reduce framework. This is a small implementation of the actual Hadoop Framework. For our implementation, we have taken some liberties and deviated from the actual Hadoop implementation for many of the functionalities. This has made our implementation extremely lucid and it also works faster than traditional Hadoop for our test cases. We can attribute our enhanced speed to the following factors:

- For load balancing amongst the various available slave clusters, we split the input data depending on the number of slaves and hence spawns equal number of mappers which executes in parallel. Traditional hadoop system load balances by reading the actual data from the files and then splitting them up into chunks of 128MB. So, for smaller inputs(less than 128MB), there is only one mapper that's spawned. But in our implementation, we split the data to the available slave clusters (mappers) based on the summary of the input file list. The actual file I/O is done in parallel by the slave clusters (mappers). This makes use of all the available clusters (started in start-cluster script) and hence adds on to our speed.

- Mapper and Reducer outputs are transferred across EC2 nodes using SCP which is relatively faster than sending data using TCP or similar protocols. This is based on our observation from our previous assignment (EC2 Sorting) where we had used TCP to transfer data across EC2 nodes.

- The Mapper output is created as files based on unique keys, ie. One file per key, whose name is a key and the values against that key is the content of the file. This method has proved to be extremely efficient during the merge and sort phase of the master node. Our implementation allowed us to Linux's inbuilt sort command. Otherwise we would have to read a huge file(s) (mapper output(s)) in-memory and then sort by keys. This would have been extremely slow during sorting and could also result in an "out of heap memory" error during the merge phase. The technique of writing the mapper output in files allowed us to use Bash

Scripts to merge files due to which there is no chance of our framework to get an "out of heap memory" error during this phase.

- The actual shuffling of the mapper outputs for the reducers (reducer inputs) are done by Bash Scripts. This greatly adds on to the speed of the framework, against an equivalent java implementation of the partitioning.

There are a few trade-offs of our implementation. One of them is generalization. Our system handles only the test cases as mentioned in the report above. As our mapper output consists of files with Mapper output key as filenames there may is a probability that we might miss a few keys if the keys contain characters forbidden in filenames (\ /: *? " < > |). But it is extremely rare that these 9 special characters would appear in the Mapper output keys hence the impact of this design decision proved to have more pros than cons. Overall this Custom Map Reduce implementation works quite well and efficiently for our specific test cases.