Shared-Memory Word Count

Problem: To count the occurrences of each word in a given set of text files (Large dataset (10GB) and Small dataset (1 MB)) using multithreaded approach to process large data concurrently. This experiment is supposed to be done on "single node" using Amazon EC2 on a “c3.large” instance. The time taken to execute word count program on the input data sets is also to be measured. The best performance achieved by varying the number of threads from 1 to 8 is to be recorded. In addition, a log file named as wordcount-python.txt is to be created.

Program Design: Following steps are taken in order to perform above experiment:

1. It’s not possible to perform operations on the large dataset due to the limitations of the allocated resources on large instance. For example: RAM of 3.75 GB is allocated so we can’t perform read write or other arithmetic operations on a 10GB file unless we split it into smaller files. Therefore, as a first step we are splitting this 10 GB file into smaller chunks by running a python program that splits the files in chunks for size 50KB to 100 MB depending upon the filesize
2. Next step is to read these files one by one and distribute it among threads so that they can execute the defined task concurrently. It gives better performance by ensuring full utilization of available cores.
3. Word Count Logic implementation starts by reading the file into a buffered array and storing the data as a "string" so that word splitting can be performed. Words are stored in a list and Hash Map is used to map key value pairs where key is the word and value is the number of occurrences of that word in that list.
4. After the word count the output data is stored in individual output files with the number of bytes where the file split.
5. In the end all the output files are merged by using a merge function (a custom function created in the program) and saved as a log file (wordcount-python.txt).
6. Thread start and Thread join functions are used in order to run and manage the threads.
7. Total time elapsed for executing the word count application is calculated.

Runtime environment settings:

Installed python interpreter Python 2.7

Steps for executing a python script file:

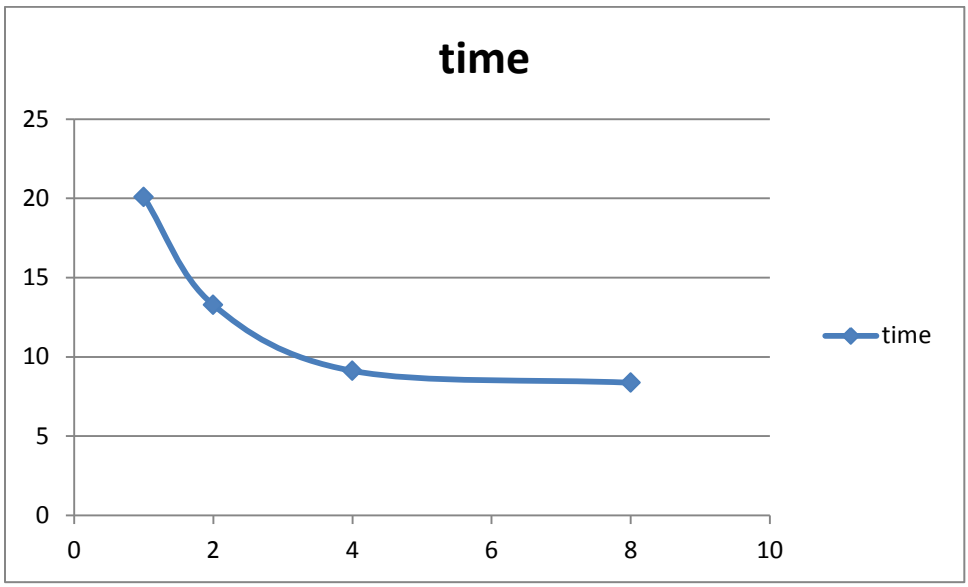
1. Installed python by using the apt-get install python.
2. Run the file using ./<file_name.py>

Installation steps to setup the virtual cluster (1-node):

1. Launch Instance and choose one of the instances as per requirement. In this project Amazon Linux AMI 2014.09.1 (HVM) has been chosen.
2. Select the instance.
3. After selecting the instance (c3.large) configure instance details.
4. Configure security group and storage as per requirement. I configured storage to 30 GB.
5. After completing the configuration, launch the instance and select an existing key pair or create a new key pair.
6. The private key file is to be downloaded and saved.
7. To connect to the cloud instance from unix system I used the command ssh –I <private key> username@hostIP

Configuration used:

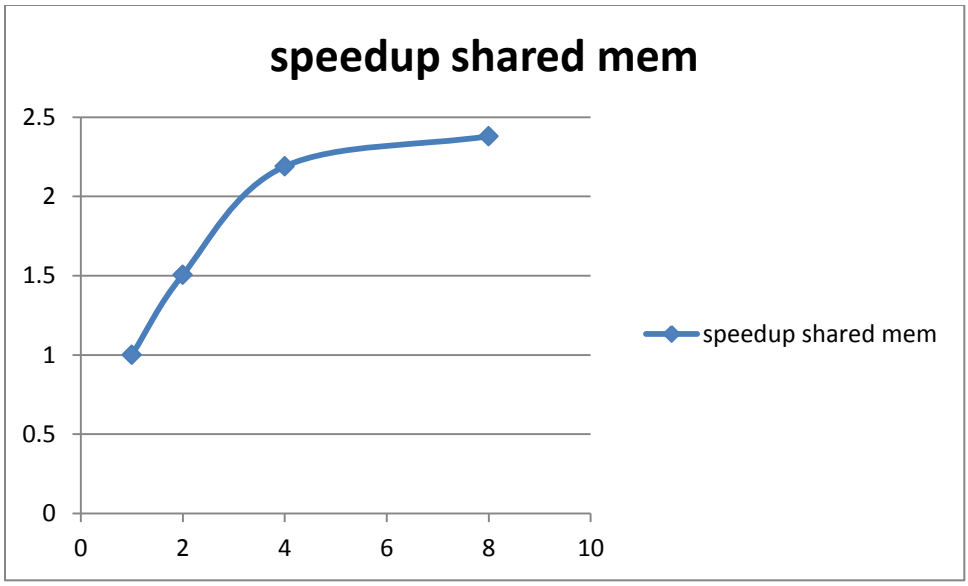
Performance:



X-Axis: Integer # of Nodes

Y-Axis: Time Elapsed (mins)

no of threads	time
1	20.08
2	13.28
4	9.12
8	8.37



Scale:

X-Axis: Integer # of Nodes

Y-Axis: Speed up

no of threads	speedup
1	1
2	1.506
4	2.19
8	2.38

Thus we calculate the speedup by using the formula

Speed up for "n" nodes = time for "1" node / time for "n" node

Explanation: In the above graph we can see that on increasing number of threads execution time is decreasing as threads are running concurrently. But as there are 2 virtual cores therefore we start achieving saturation after 4 threads.

Sort on Shared-Memory:

Problem: To implement and evaluate sorting in shared memory by using a python program. The sorting application should read a large file (10GB dataset) and sort it. Measure the performance of Shared-Memory sorting (including reading data from disk, sorting, and writing data to disk) and record the output in “sort1MB-sharedMemory.txt” for final submission.

Methodology: Following steps are taken in order to perform above experiment:

1. The given project requires performing operations on a large dataset which may put forth some limitations regarding the allocation of resources on a large instance. For example: with a RAM of 3.75 GB , the read, write or other arithmetic operations on a 10GB file could not be performed unless the large file is split in smaller files. Therefore, as a first step the 10 GB file is split in smaller chunks by running a python file file split program
2. Next step is to read these files one by one and distribute it among threads so that they can execute their tasks concurrently and a better performance is achieved by the full utilization of available cores.
3. Sort logic implementation starts by creating 26 output files (as there are 26 alphabets in English). Each file is supposed to store words which start with the same letter/character as the one shown in the file name.
4. The processing starts by reading the "Input" file. A "list" of words is created from the input file. Each word in the list is picked and the first character of that word is pulled out to identify the Output file name wherein this word will be sent for storage.
5. After collecting all the words starting with ‘a’ alphabets in one file, ‘b’ in other file and so on, there are 26 files in place. These data in each of these files is sorted internally inside the file. Next step is to merge these sorted files (or bucket of words) in an output log file.
6. Thread start and Thread join functions are used in order to run and manage the threads.
7. Total time elapsed for executing the word count application is calculated.

Runtime environment settings:

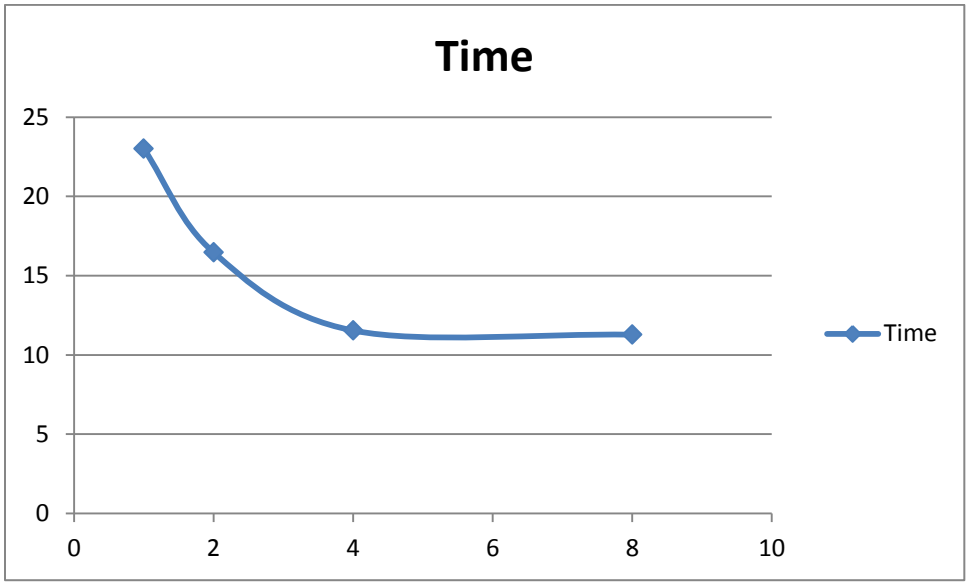
The runtime environment settings remain same as for the word count program described above.

Installation steps to setup the virtual cluster (1-node):

Sort Program is executed on the same virtual node as used for word count therefore no separate installation is required.

Configuration: The configuration settings remain same as for the word count program described above.

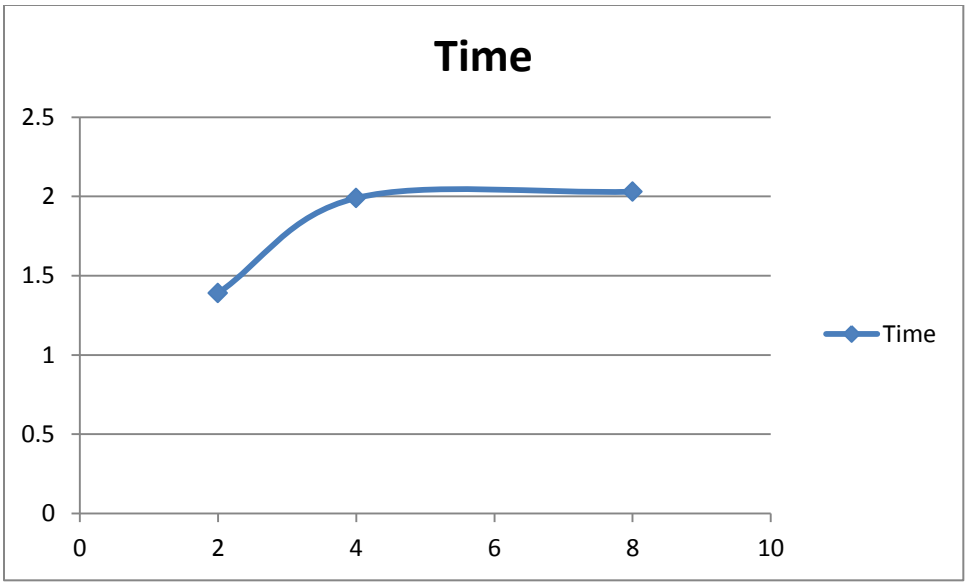
Performance:



X-Axis: Integer # of Nodes

Y-Axis: Time Elapsed (mins)

Threads	Time
1	23
2	16.47
4	11.54
8	11.28



Scale:

X-Axis: Integer # of Nodes

Y-Axis: Speed up

Threads	Time
2	1.39
4	1.99
8	2.03

Thus we calculate the speedup by using the formula

Speed up for "n" nodes = time for "1" node / time for "n" node

Explanation:

In the above graph we can see that on increasing number of threads execution time is decreasing as threads are running concurrently. But as there are 2 virtual cores therefore we start achieving saturation after 4 threads.

Hadoop

Installing Single Node Virtual Cluster

EC2 (Elastic Cloud Computing) is one of the services which Amazon Web Services (AWS) presents. We can use EC2 Service to launch virtual machines and virtual clusters. In EC2 each Virtual Machine is called Instance. Therefore, we should create instances. In this assignment we want to create 18 nodes to installed Hadoop. In this section we only create one instance. After the initial Hadoop configuration, we can create an AMI from our instance to create other 17 node automatically. There are a lot of types of instances which each of them has different types of CPU, Disk and etc. We Use "*c3.Large"* Instance for our clusters. Moreover, Ubuntu Server 14.04 LTS (HVM) 64-bit is installed as Operating System on these instances. The features of this instance are explained below:

In Addition to configure the network detail, we need to open SSH, IMCP and TCP protocols.

Hadoop Installation:

Hadoop Parts:

Hadoop has several different parts which each of them has specific responsibility. Each part is described briefly.

Job Tracker: the node that assigns map and reduce tasks to one Task Tracker (Slaves)

Name Node: the node which controls or manages the file system operations. Name Node has the responsibility to map these files to Data Nodes. Indeed, Name Node logged changes into a file, when the node is restarting, Name Node starts to read files and apply the changes. This process is relatively time consuming.

Secondary Name Node: this node reads the file system changes log and applies these changes. This process helps Name Node to start faster next time.

Data Node: they have the responsibility to read and write from file systems according to the instruction of master.

Task Tracker: Task Tracker executes tasks which master is assigned to them. Also, they have the responsibility to transfer data between Mapper and Reducer.

Installing Hadoop on Single Node:

In this part we used one "c3.large" Instance with above features. Also, "jdk-7" and "Hadoop 2.4.1" is installed. After we download and install "jdk-7", "SSH", we started to download Hadoop. The installation process contained a number of different configurations. At first we should add Hadoop environment variable to *".bashrc"* file. Indeed, we add JDK path and Hadoop path to this file. Secondly, we should modify some configuration files in Hadoop to declare each part of the Hadoop. Since in this section

there is only one node the IPs for configuration should be “localhost”. Configuration files are described in detail in next sections of this report.

One of the problems that we faced during the Hadoop installation is that the path of configuration files is changed in different versions of Hadoop. Therefore, we should search for different path in Hadoop folders to find the configuration files. But we can find the actual path in the apache site

http://hadoop.apache.org/docs/r2.4.1/hadoop-project-dist/hadoop-common/SingleCluster.html

Installing Hadoop on Multiple Node:

In this section, firstly we create one instance with features described previously. The steps of installation Hadoop are common for all type of nodes. There is a few configuration files which are different from Name Node and Data Nodes. Therefore, we installed Hadoop on one instance. Then, An AMI is created from that instance, finally, we used this AMI for creating other instances. At first, “JDK-7” and “Hadoop 2.4.1” is installed. Secondly we should change some configuration files.

1. Update “.bashrc” file
   In this file we should add some Hadoop Environment variable. Indeed, JDK and Hadoop path should be added on this file.
   During the installation we faced a problem with connecting password less *ssh*. Each time that we logout our instance the configuration of this progress is lost. Therefore, we add the configurations of password less progress on “.bashrc” in order to run each time that instance is logged in.

2. Update “hadoop-env.sh”
   In this file, the only task which should be done is to enter the Java home path of our machine instead of the path which was previously in this file by default.

3. Update “core-site.xml”
   In this file we specify the default file system manager for all slaves. Therefore, the IP of Name Node is used here.

4. Update “hdfs-site.xml”
   This file takes care of the Hadoop file system (HDFS) configurations. In this file we set the number of replications for our HDFS to 2. Also for simplicity we turned off permission checking on our HDFS.
   Moreover, we could modify the path for HDFS files in this configuration but we left it to use the default path for HDFS that uses /tmp partition.

5. Update “mapred.site.xml”
   In this file the information of job tracker is added. In our configuration Job Tracker is the Name Node. Therefore, we should write the IP address of Name Node.

6. Update “masters”
   In this file we should enter the addresses of all master nodes, including Name Node and Secondary Name Node. This file is one of the file which has different version for master nodes and slave nodes. In slave Nodes this file should be an empty files.

7. Update “slaves”
   The list of slaves addresses should be written here. However, in each slave node we should enter only the IP address of itself (localhost). But these change everytime we terminate the cluster or nodes. hence never stop the cluster and then the private ip will be the same and we don’t have to note the public ip only if needed to connect.

I used the steps from the site and also the configuration details from here

http://hadoop.apache.org/docs/r2.4.1/hadoop-project-dist/hadoop-common/ClusterSetup.html

http://disi.unitn.it/~lissandrini/notes/installing-hadoop-on-ubuntu-14.html

https://docs.google.com/document/d/1v-J19xwJn-Pw9F8OCgLn04dqKwYkGIOxyqRAIGpmHk0/edit?pli=1

Word Count and Running Process

In this section we are compile two types of “WordCount” program.

1. Example of “WordCount” which is in the Hadoop folder
2. The version which is written by ourselves

1. Running the “WordCount” Example of Hadoop

The following instructions are to run a MapReduce job .

1. Format the filesystem:

   ```
   $ bin/hdfs namenode -format
   ```

2. Start NameNode daemon and DataNode daemon:

$ sbin/start-dfs.sh

The hadoop daemon log output is written to the $HADOOP_LOG_DIR directory (defaults to $HADOOP_HOME/logs).

3. Browse the web interface for the NameNode; by default it is available at:
    - NameNode - http://localhost:50070/
4. Make the HDFS directories required to execute MapReduce jobs:
5. $ bin/hdfs dfs -mkdir /user
   $ bin/hdfs dfs -mkdir /user/<username>

6. Copy the input files into the distributed filesystem:

   $ bin/hdfs dfs -put etc/hadoop input

7. Run some of the examples provided:

   $ bin/hadoop jar share/hadoop/mapreduce/hadoop-mapreduce-examples-2.4.1.jar wordcount input output

8. Examine the output files:

   Copy the output files from the distributed filesystem to the local filesystem and examine them:

   $ bin/hdfs dfs -get output output
   $ cat output/*

   or

   View the output files on the distributed filesystem:

   $ bin/hdfs dfs -cat output/*

9. When you're done, stop the daemons with:

   $ sbin/stop-dfs.sh

The scalability of our Hadoop cluster is totally clear in this figure. As we increase the number of nodes in our cluster, the speed-up is also increasing proportionally. The below graph also shows that increasing the number of nodes in the cluster, decreases the running time for the wordcount application.

2. Running our version of "WordCount"

In this section we had 3 file (Mapper, Reducer, Driver). These file should be compile together in order to create one jar file.

Then we can run the "WordCount" program on our desire data set. First mapper start to run and after a while Reducer also start to run. It is mentioned that the time of mapper and reducer is overlapped to each other. There is an interval of time which mapper and reducer is running beside each other.

>$ ../hadoop/bin/hadoop jar /path/to/WordCount.jar org/myorg/WordCount input output

Sorting on Hadoop

In our implementation we used the large dataset (1mbsmalldataset) as our input file. In the map process we recognize each word and output them as the intermediate key/value pair. Reducer groups them into a single key/value and stores them in the output file sorted.

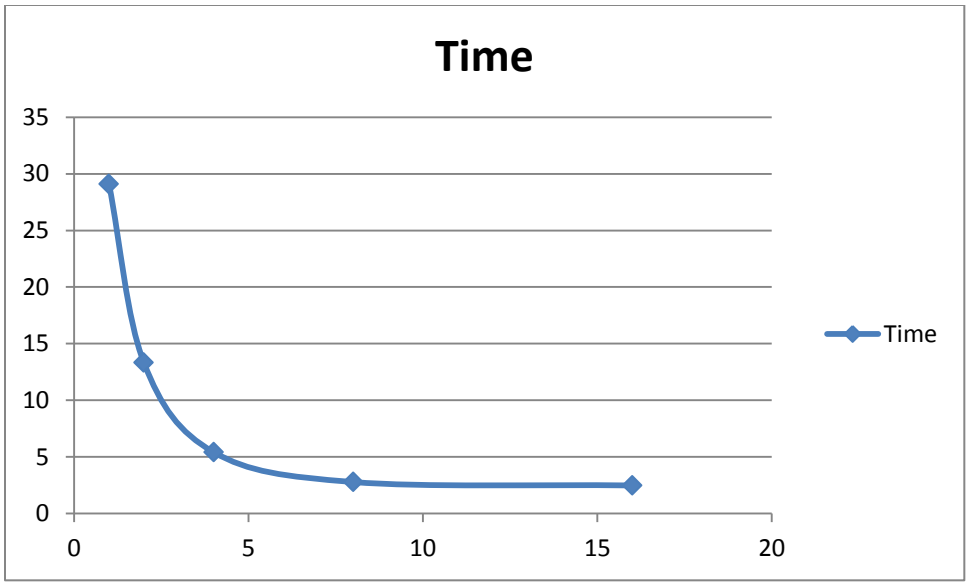
The process of compiling the code is similar to compiling the code we had for WordCount. We ran the code for both small and large datasets.

>$ ../hadoop/bin/hadoop jar /path/to/sort.jar org/myorg/sort input output

Performance in Hadoop:

We performed the hadoop calculations on 1,2,4,8,16 nodes and then noted down the time which the hadoop interface itself shows. And thus the wordcount program was executed with scaling no of nodes and then plotted the below graph.

As we can see



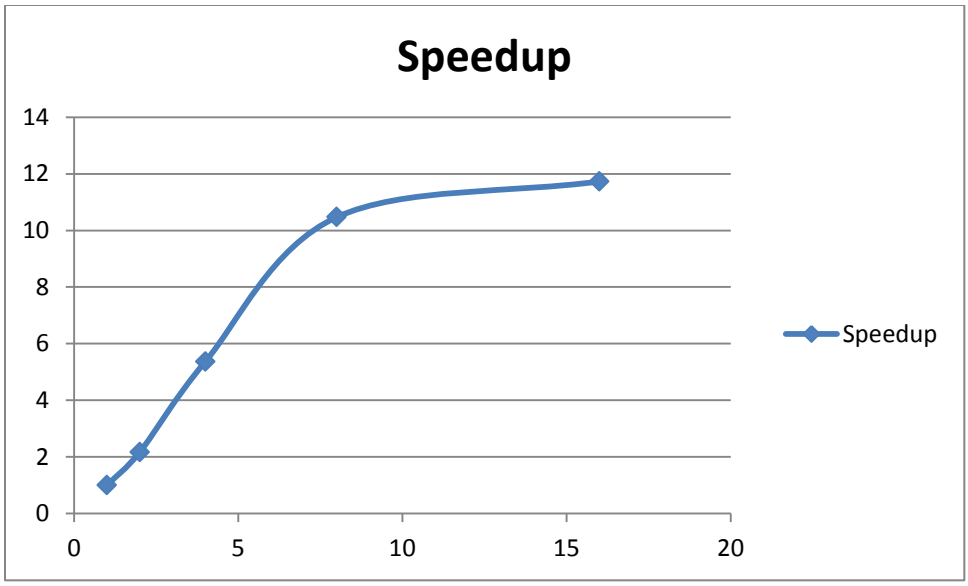
X-Axis: Integer # of Nodes

Y-Axis: Time Elapsed (mins)

Nodes	Time
1	29.11
2	13.33
4	5.42
8	2.78
16	2.48

Thus we calculate the speedup by using the formula

Speed up for “n” nodes = time for “1” node / time for “n” node



Scale:

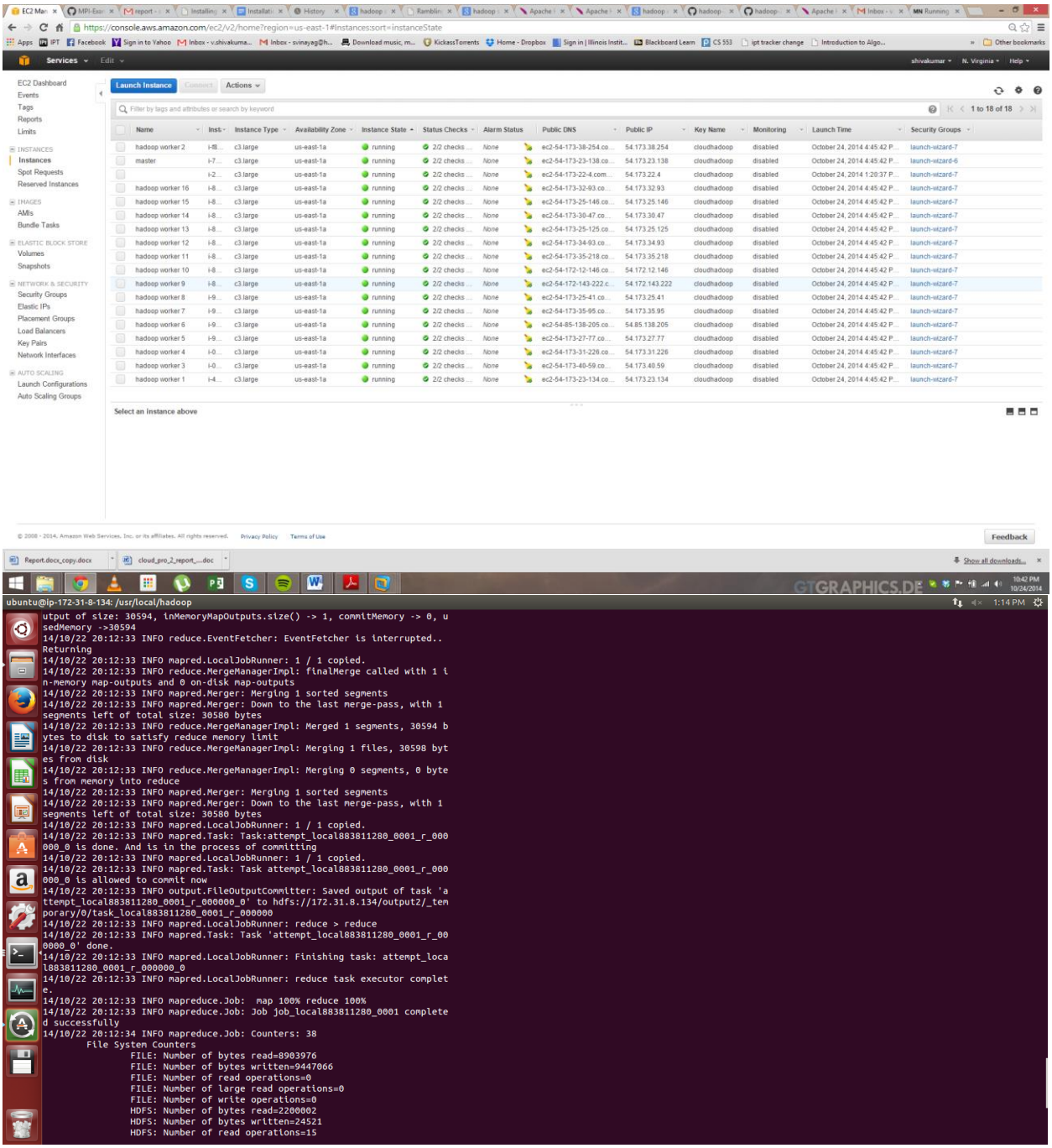
X-Axis: Integer # of Nodes

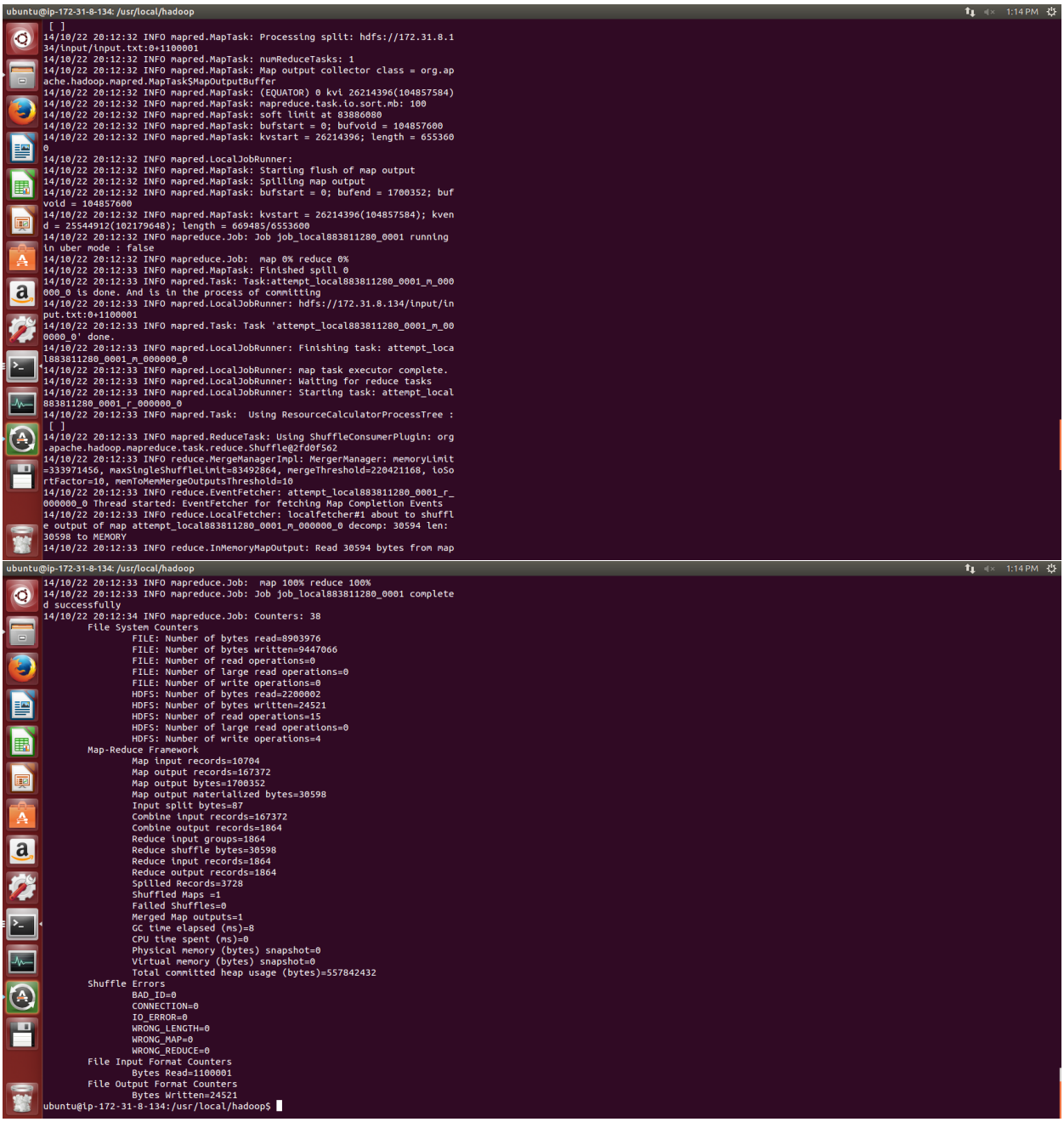
Y-Axis: Speed up

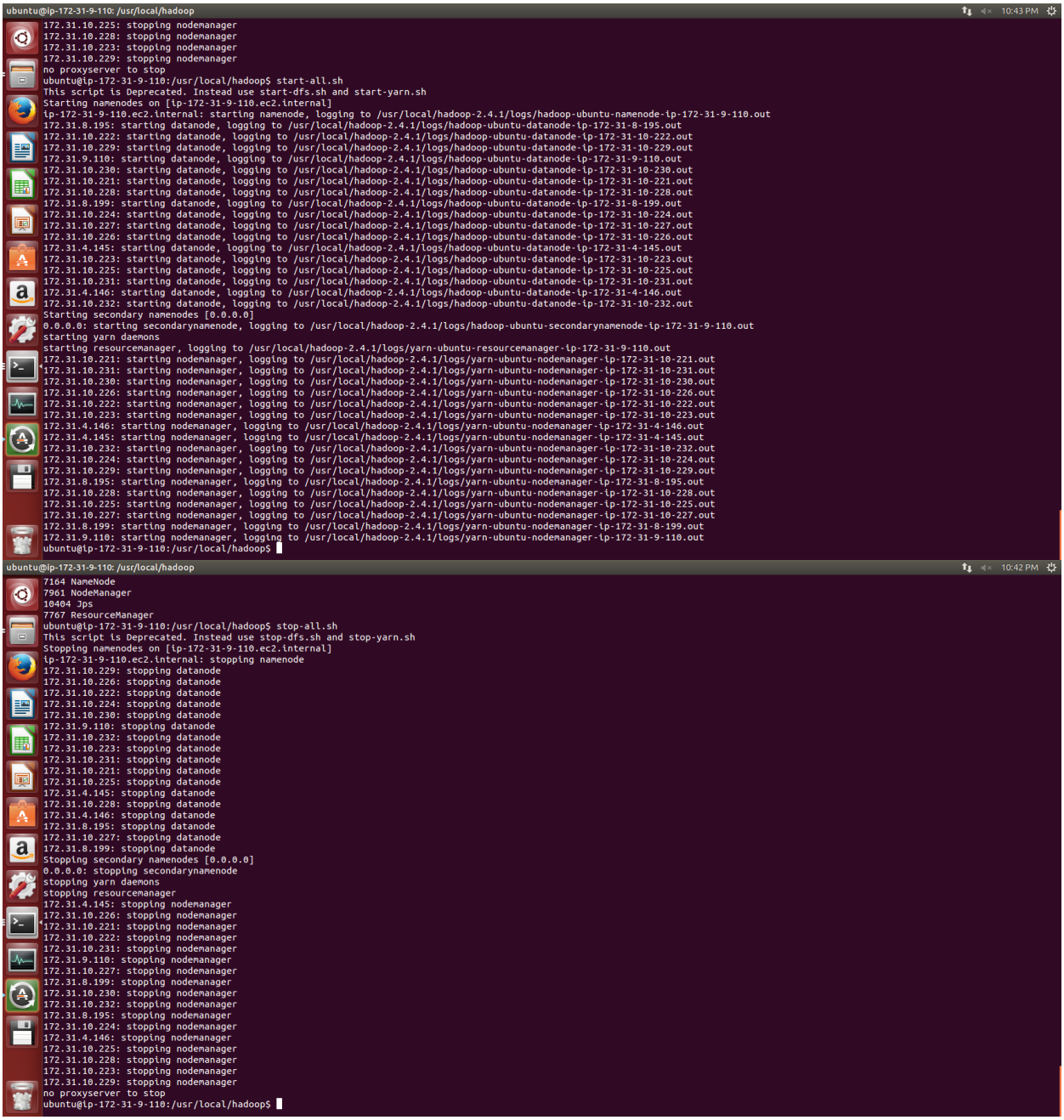
Nodes	Speedup
1	1
2	2.17
4	5.37
8	10.47
16	11.73

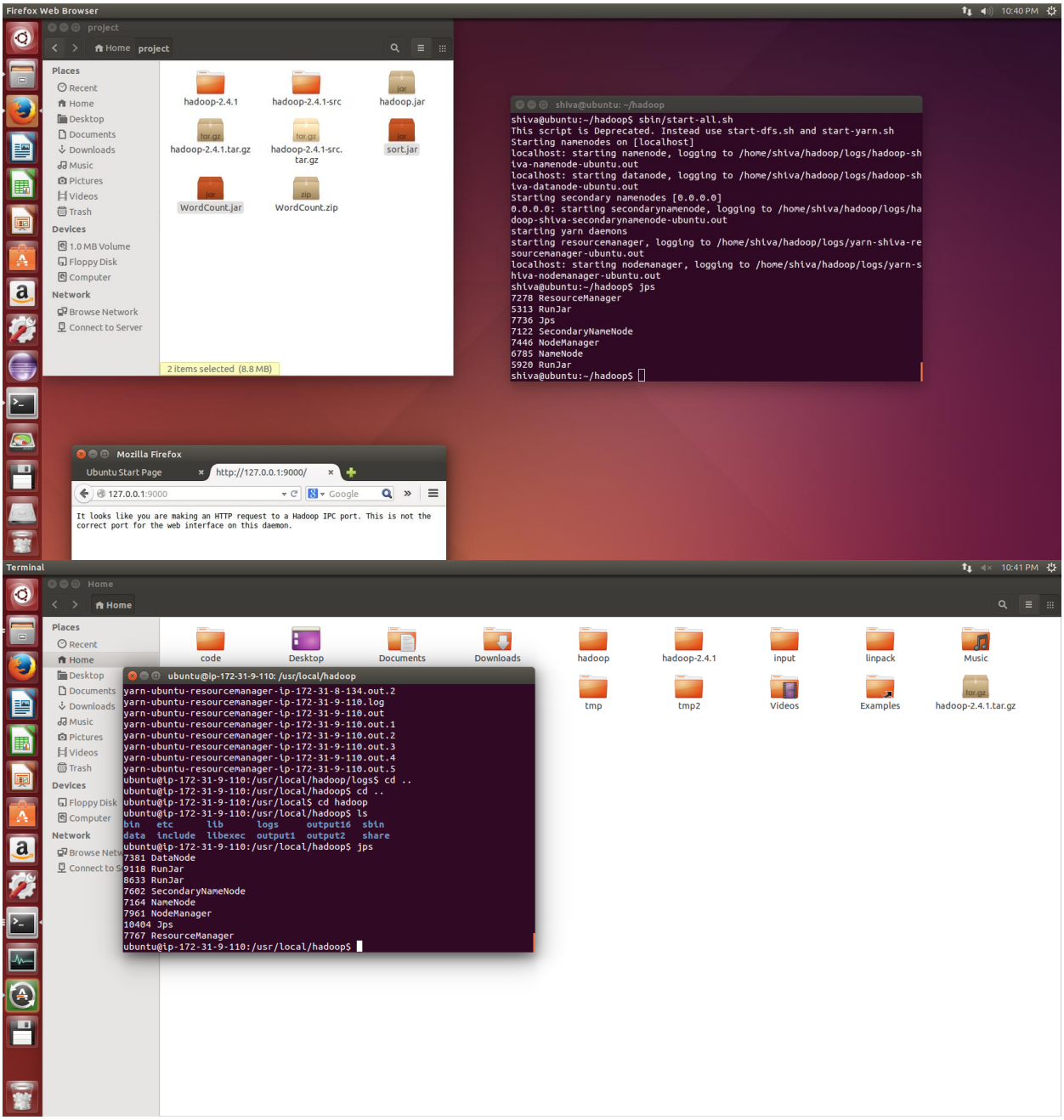
As we can see the performance flattens out at 8 nodes and it also is almost the same range for 16 nodes also.

Screenshots:




ubuntu@ip-172-31-8-134: /usr/local/hadoop
1:14 PM
[ ]
14/10/22 20:12:32 INFO mapred.MapTask: Processing split: hdfs://172.31.8.134/input/input.txt:0+1100001
14/10/22 20:12:32 INFO mapred.MapTask: numReduceTasks: 1
14/10/22 20:12:32 INFO mapred.MapTask: Map output collector class = org.apache.hadoop.mapred.MapTask$MapOutputBuffer
14/10/22 20:12:32 INFO mapred.MapTask: (EQUATOR) 0 kvi 26214396(104857584)
14/10/22 20:12:32 INFO mapred.MapTask: mapreduce.task.io.sort.mb: 100
14/10/22 20:12:32 INFO mapred.MapTask: soft limit at 83886080
14/10/22 20:12:32 INFO mapred.MapTask: bufstart = 0; bufvoid = 104857600
14/10/22 20:12:32 INFO mapred.MapTask: kvstart = 26214396; length = 6553600
14/10/22 20:12:32 INFO mapred.LocalJobRunner:
14/10/22 20:12:32 INFO mapred.MapTask: Starting flush of map output
14/10/22 20:12:32 INFO mapred.MapTask: Spilling map output
14/10/22 20:12:32 INFO mapred.MapTask: bufstart = 0; bufend = 1700352; bufvoid = 104857600
14/10/22 20:12:32 INFO mapred.MapTask: kvstart = 26214396(104857584); kvend = 25544912(102179648); length = 669485/6553600
14/10/22 20:12:32 INFO mapreduce.Job: Job job_local883811280_0001 running in uber mode : false
14/10/22 20:12:32 INFO mapreduce.Job: map 0% reduce 0%
14/10/22 20:12:33 INFO mapred.MapTask: Finished spill 0
14/10/22 20:12:33 INFO mapred.Task: Task:attempt_local883811280_0001_m_000000_0 is done. And is in the process of committing
14/10/22 20:12:33 INFO mapred.LocalJobRunner: hdfs://172.31.8.134/input/input.txt:0+1100001
14/10/22 20:12:33 INFO mapred.Task: Task 'attempt_local883811280_0001_m_000000_0' done.
14/10/22 20:12:33 INFO mapred.LocalJobRunner: Finishing task: attempt_local883811280_0001_m_000000_0
14/10/22 20:12:33 INFO mapred.LocalJobRunner: map task executor complete.
14/10/22 20:12:33 INFO mapred.LocalJobRunner: Waiting for reduce tasks
14/10/22 20:12:33 INFO mapred.LocalJobRunner: Starting task: attempt_local883811280_0001_r_000000_0
14/10/22 20:12:33 INFO mapred.Task: Using ResourceCalculatorProcessTree : [ ]
14/10/22 20:12:33 INFO mapred.ReduceTask: Using ShuffleConsumerPlugin: org.apache.hadoop.mapreduce.task.reduce.Shuffle@2fd0f562
14/10/22 20:12:33 INFO reduce.MergeManagerImpl: MergerManager: memoryLimit=333971456, maxSingleShuffleLimit=83492864, mergeThreshold=220421168, ioSortFactor=10, memToMemMergeOutputsThreshold=10
14/10/22 20:12:33 INFO reduce.EventFetcher: attempt_local883811280_0001_r_000000_0 Thread started: EventFetcher for fetching Map Completion Events
14/10/22 20:12:33 INFO reduce.LocalFetcher: localfetcher#1 about to shuffle output of map attempt_local883811280_0001_m_000000_0 decomp: 30594 len: 30598 to MEMORY
14/10/22 20:12:33 INFO reduce.InMemoryMapOutput: Read 30594 bytes from map
ubuntu@ip-172-31-8-134: /usr/local/hadoop
1:14 PM
14/10/22 20:12:33 INFO mapreduce.Job: map 100% reduce 100%
14/10/22 20:12:33 INFO mapreduce.Job: Job job_local883811280_0001 completed successfully
14/10/22 20:12:34 INFO mapreduce.Job: Counters: 38
File System Counters
FILE: Number of bytes read=8903976
FILE: Number of bytes written=9447066
FILE: Number of read operations=0
FILE: Number of large read operations=0
FILE: Number of write operations=0
HDFS: Number of bytes read=2200002
HDFS: Number of bytes written=24521
HDFS: Number of read operations=15
HDFS: Number of large read operations=0
HDFS: Number of write operations=4
Map-Reduce Framework
Map input records=10704
Map output records=167372
Map output bytes=1700352
Map output materialized bytes=30598
Input split bytes=87
Combine input records=167372
Combine output records=1864
Reduce input groups=1864
Reduce shuffle bytes=30598
Reduce input records=1864
Reduce output records=1864
Spilled Records=3728
Shuffled Maps =1
Failed Shuffles=0
Merged Map outputs=1
GC time elapsed (ms)=8
CPU time spent (ms)=0
Physical memory (bytes) snapshot=0
Virtual memory (bytes) snapshot=0
Total committed heap usage (bytes)=557842432
Shuffle Errors
BAD_ID=0
CONNECTION=0
IO_ERROR=0
WRONG_LENGTH=0
WRONG_MAP=0
WRONG_REDUCE=0
File Input Format Counters
Bytes Read=1100001
File Output Format Counters
Bytes Written=24521
ubuntu@ip-172-31-8-134:/usr/local/hadoop$


ubuntu@ip-172-31-9-110: /usr/local/hadoop
10:43 PM
172.31.10.225: stopping nodemanager
172.31.10.228: stopping nodemanager
172.31.10.223: stopping nodemanager
172.31.10.229: stopping nodemanager
no proxyserver to stop
ubuntu@ip-172-31-9-110:/usr/local/hadoop$ start-all.sh
This script is Deprecated. Instead use start-dfs.sh and start-yarn.sh
Starting namenodes on [ip-172-31-9-110.ec2.internal]
ip-172-31-9-110.ec2.internal: starting namenode, logging to /usr/local/hadoop-2.4.1/logs/hadoop-ubuntu-namenode-ip-172-31-9-110.out
172.31.8.195: starting datanode, logging to /usr/local/hadoop-2.4.1/logs/hadoop-ubuntu-datanode-ip-172-31-8-195.out
172.31.10.222: starting datanode, logging to /usr/local/hadoop-2.4.1/logs/hadoop-ubuntu-datanode-ip-172-31-10-222.out
172.31.10.229: starting datanode, logging to /usr/local/hadoop-2.4.1/logs/hadoop-ubuntu-datanode-ip-172-31-10-229.out
172.31.9.110: starting datanode, logging to /usr/local/hadoop-2.4.1/logs/hadoop-ubuntu-datanode-ip-172-31-9-110.out
172.31.10.230: starting datanode, logging to /usr/local/hadoop-2.4.1/logs/hadoop-ubuntu-datanode-ip-172-31-10-230.out
172.31.10.221: starting datanode, logging to /usr/local/hadoop-2.4.1/logs/hadoop-ubuntu-datanode-ip-172-31-10-221.out
172.31.10.228: starting datanode, logging to /usr/local/hadoop-2.4.1/logs/hadoop-ubuntu-datanode-ip-172-31-10-228.out
172.31.8.199: starting datanode, logging to /usr/local/hadoop-2.4.1/logs/hadoop-ubuntu-datanode-ip-172-31-8-199.out
172.31.10.224: starting datanode, logging to /usr/local/hadoop-2.4.1/logs/hadoop-ubuntu-datanode-ip-172-31-10-224.out
172.31.10.227: starting datanode, logging to /usr/local/hadoop-2.4.1/logs/hadoop-ubuntu-datanode-ip-172-31-10-227.out
172.31.10.226: starting datanode, logging to /usr/local/hadoop-2.4.1/logs/hadoop-ubuntu-datanode-ip-172-31-10-226.out
172.31.4.145: starting datanode, logging to /usr/local/hadoop-2.4.1/logs/hadoop-ubuntu-datanode-ip-172-31-4-145.out
172.31.10.223: starting datanode, logging to /usr/local/hadoop-2.4.1/logs/hadoop-ubuntu-datanode-ip-172-31-10-223.out
172.31.10.225: starting datanode, logging to /usr/local/hadoop-2.4.1/logs/hadoop-ubuntu-datanode-ip-172-31-10-225.out
172.31.10.231: starting datanode, logging to /usr/local/hadoop-2.4.1/logs/hadoop-ubuntu-datanode-ip-172-31-10-231.out
172.31.4.146: starting datanode, logging to /usr/local/hadoop-2.4.1/logs/hadoop-ubuntu-datanode-ip-172-31-4-146.out
172.31.10.232: starting datanode, logging to /usr/local/hadoop-2.4.1/logs/hadoop-ubuntu-datanode-ip-172-31-10-232.out
Starting secondary namenodes [0.0.0.0]
0.0.0.0: starting secondarynamenode, logging to /usr/local/hadoop-2.4.1/logs/hadoop-ubuntu-secondarynamenode-ip-172-31-9-110.out
starting yarn daemons
starting resourcemanager, logging to /usr/local/hadoop-2.4.1/logs/yarn-ubuntu-resourcemanager-ip-172-31-9-110.out
172.31.10.221: starting nodemanager, logging to /usr/local/hadoop-2.4.1/logs/yarn-ubuntu-nodemanager-ip-172-31-10-221.out
172.31.10.231: starting nodemanager, logging to /usr/local/hadoop-2.4.1/logs/yarn-ubuntu-nodemanager-ip-172-31-10-231.out
172.31.10.230: starting nodemanager, logging to /usr/local/hadoop-2.4.1/logs/yarn-ubuntu-nodemanager-ip-172-31-10-230.out
172.31.10.226: starting nodemanager, logging to /usr/local/hadoop-2.4.1/logs/yarn-ubuntu-nodemanager-ip-172-31-10-226.out
172.31.10.222: starting nodemanager, logging to /usr/local/hadoop-2.4.1/logs/yarn-ubuntu-nodemanager-ip-172-31-10-222.out
172.31.10.223: starting nodemanager, logging to /usr/local/hadoop-2.4.1/logs/yarn-ubuntu-nodemanager-ip-172-31-10-223.out
172.31.4.146: starting nodemanager, logging to /usr/local/hadoop-2.4.1/logs/yarn-ubuntu-nodemanager-ip-172-31-4-146.out
172.31.4.145: starting nodemanager, logging to /usr/local/hadoop-2.4.1/logs/yarn-ubuntu-nodemanager-ip-172-31-4-145.out
172.31.10.232: starting nodemanager, logging to /usr/local/hadoop-2.4.1/logs/yarn-ubuntu-nodemanager-ip-172-31-10-232.out
172.31.10.224: starting nodemanager, logging to /usr/local/hadoop-2.4.1/logs/yarn-ubuntu-nodemanager-ip-172-31-10-224.out
172.31.10.229: starting nodemanager, logging to /usr/local/hadoop-2.4.1/logs/yarn-ubuntu-nodemanager-ip-172-31-10-229.out
172.31.8.195: starting nodemanager, logging to /usr/local/hadoop-2.4.1/logs/yarn-ubuntu-nodemanager-ip-172-31-8-195.out
172.31.10.228: starting nodemanager, logging to /usr/local/hadoop-2.4.1/logs/yarn-ubuntu-nodemanager-ip-172-31-10-228.out
172.31.10.225: starting nodemanager, logging to /usr/local/hadoop-2.4.1/logs/yarn-ubuntu-nodemanager-ip-172-31-10-225.out
172.31.10.227: starting nodemanager, logging to /usr/local/hadoop-2.4.1/logs/yarn-ubuntu-nodemanager-ip-172-31-10-227.out
172.31.8.199: starting nodemanager, logging to /usr/local/hadoop-2.4.1/logs/yarn-ubuntu-nodemanager-ip-172-31-8-199.out
172.31.9.110: starting nodemanager, logging to /usr/local/hadoop-2.4.1/logs/yarn-ubuntu-nodemanager-ip-172-31-9-110.out
ubuntu@ip-172-31-9-110:/usr/local/hadoop$
ubuntu@ip-172-31-9-110: /usr/local/hadoop
10:42 PM
7164 NameNode
7961 NodeManager
10404 Jps
7767 ResourceManager
ubuntu@ip-172-31-9-110:/usr/local/hadoop$ stop-all.sh
This script is Deprecated. Instead use stop-dfs.sh and stop-yarn.sh
Stopping namenodes on [ip-172-31-9-110.ec2.internal]
ip-172-31-9-110.ec2.internal: stopping namenode
172.31.10.229: stopping datanode
172.31.10.226: stopping datanode
172.31.10.222: stopping datanode
172.31.10.224: stopping datanode
172.31.10.230: stopping datanode
172.31.9.110: stopping datanode
172.31.10.232: stopping datanode
172.31.10.223: stopping datanode
172.31.10.231: stopping datanode
172.31.10.221: stopping datanode
172.31.10.225: stopping datanode
172.31.4.145: stopping datanode
172.31.10.228: stopping datanode
172.31.4.146: stopping datanode
172.31.8.195: stopping datanode
172.31.10.227: stopping datanode
172.31.8.199: stopping datanode
Stopping secondary namenodes [0.0.0.0]
0.0.0.0: stopping secondarynamenode
stopping yarn daemons
stopping resourcemanager
172.31.4.145: stopping nodemanager
172.31.10.226: stopping nodemanager
172.31.10.221: stopping nodemanager
172.31.10.222: stopping nodemanager
172.31.10.231: stopping nodemanager
172.31.9.110: stopping nodemanager
172.31.10.227: stopping nodemanager
172.31.8.199: stopping nodemanager
172.31.10.230: stopping nodemanager
172.31.10.232: stopping nodemanager
172.31.8.195: stopping nodemanager
172.31.10.224: stopping nodemanager
172.31.4.146: stopping nodemanager
172.31.10.225: stopping nodemanager
172.31.10.228: stopping nodemanager
172.31.10.223: stopping nodemanager
172.31.10.229: stopping nodemanager
no proxyserver to stop
ubuntu@ip-172-31-9-110:/usr/local/hadoop$


Firefox Web Browser
10:40 PM
project
Home
project
Places
Recent
Home
Desktop
Documents
Downloads
Music
Pictures
Videos
Trash
Devices
1.0 MB Volume
Floppy Disk
Computer
Network
Browse Network
Connect to Server
hadoop-2.4.1
hadoop-2.4.1-src
hadoop.jar
hadoop-2.4.1.tar.gz
hadoop-2.4.1-src.tar.gz
sort.jar
WordCount.jar
WordCount.zip
2 items selected (8.8 MB)
shiva@ubuntu: ~/hadoop
shiva@ubuntu:~/hadoop$ sbin/start-all.sh
This script is Deprecated. Instead use start-dfs.sh and start-yarn.sh
Starting namenodes on [localhost]
localhost: starting namenode, logging to /home/shiva/hadoop/logs/hadoop-shiva-namenode-ubuntu.out
localhost: starting datanode, logging to /home/shiva/hadoop/logs/hadoop-shiva-datanode-ubuntu.out
Starting secondary namenodes [0.0.0.0]
0.0.0.0: starting secondarynamenode, logging to /home/shiva/hadoop/logs/hadoop-shiva-secondarynamenode-ubuntu.out
starting yarn daemons
starting resourcemanager, logging to /home/shiva/hadoop/logs/yarn-shiva-resourcemanager-ubuntu.out
localhost: starting nodemanager, logging to /home/shiva/hadoop/logs/yarn-shiva-nodemanager-ubuntu.out
shiva@ubuntu:~/hadoop$ jps
7278 ResourceManager
5313 RunJar
7736 Jps
7122 SecondaryNameNode
7446 NodeManager
6785 NameNode
5920 RunJar
shiva@ubuntu:~/hadoop$
Mozilla Firefox
Ubuntu Start Page
http://127.0.0.1:9000/
127.0.0.1:9000
Google
It looks like you are making an HTTP request to a Hadoop IPC port. This is not the correct port for the web interface on this daemon.
Terminal
10:41 PM
Home
Home
Places
Recent
Home
Desktop
Documents
Downloads
Music
Pictures
Videos
Trash
Devices
Floppy Disk
Computer
Network
code
Desktop
Documents
Downloads
hadoop
hadoop-2.4.1
input
linpack
Music
tmp
tmp2
Videos
Examples
hadoop-2.4.1.tar.gz
ubuntu@ip-172-31-9-110: /usr/local/hadoop
yarn-ubuntu-resourcemanager-ip-172-31-8-134.out.2
yarn-ubuntu-resourcemanager-ip-172-31-9-110.log
yarn-ubuntu-resourcemanager-ip-172-31-9-110.out
yarn-ubuntu-resourcemanager-ip-172-31-9-110.out.1
yarn-ubuntu-resourcemanager-ip-172-31-9-110.out.2
yarn-ubuntu-resourcemanager-ip-172-31-9-110.out.3
yarn-ubuntu-resourcemanager-ip-172-31-9-110.out.4
yarn-ubuntu-resourcemanager-ip-172-31-9-110.out.5
ubuntu@ip-172-31-9-110:/usr/local/hadoop/logs$ cd ..
ubuntu@ip-172-31-9-110:/usr/local/hadoop$ cd ..
ubuntu@ip-172-31-9-110:/usr/local$ cd hadoop
ubuntu@ip-172-31-9-110:/usr/local/hadoop$ ls
bin etc lib logs output16 sbin
data include libexec output1 output2 share
ubuntu@ip-172-31-9-110:/usr/local/hadoop$ jps
7381 DataNode
9118 RunJar
8633 RunJar
7602 SecondaryNameNode
7164 NameNode
7961 NodeManager
10404 Jps
7767 ResourceManager
ubuntu@ip-172-31-9-110:/usr/local/hadoop$

SWIFT

Our assignment deals with performing a word count application on ec2 cluster using *Swift*. The virtual cluster for this project includes 16+1(workers and head node) nodes, each node is C3.Large instance from Amazon ec2, promising a 64 bit architecture. We used Ubuntu Server 14.04 LTS along side windows , 3.75GB memory and a flexible storage space. The swift version used for this project is *Swift 0.94 ,* and python version 2.7.

Key-pair: vinodh-key-pair-

Credentials: credentials.csv

Region: US-WEST(Oregon)

AMI: (We created our own ami with size 30GB in order to run a 10GB dataset: the default ami had only 8GB size which dint allow us to perform word count on 10GB file and throwed us an error "no space in disk")

(i)After launching an instance of c3.large type from the amazon aws console, the first step is; we tried ssh-ing the instance and scp-ing keypair and credentials in terminal as follows:

(a)ssh -i /path/to/mykeypair.pem ubuntu@<PUBLIC_IP_OF_LAUNCHPAD>

(b)scp -i /path/to/mykeypair.pem /path/to/mykeypair.pem /path/to/credentials.csv ubuntu@<PUBLIC_IP_OF_LAUNCHPAD>

(ii)Now we tried setting up the Launchpad instance as follows:

Install python and libcloud library. Since we are using an Ubuntu instance:

```
sudo apt-get update
sudo apt-get install -y python python-pip git
sudo pip install apache-libcloud
```

Get the cloud-tutorials repository from git

```
git clone https://github.com/yadudoc/cloud-tutorials.git
cd cloud-tutorials/ec2
```

(iii)After this step we modified our configs file to our specification such as number of workers, key pair, credentials and paths etc, and saved it .Then we ran

source setup.sh

which started up the workers in the console and in the terminal as well.

Then we connected to the headnode with the “connect headnode” command

Now in the headnode we installed swift with the following command

```
$ wget http://swift-lang.org/packages/swift-trunk.tar.gz

$ tar xfz swift-trunk.tar.gz   # Extract the file

$ export PATH=/path/to/swift-trunk/bin:$PATH   # Add to PATH
```

(iv)After this we entered into /cloud-tutorials/swift-cloud-tutorial/part05. We copied our wordcount file and executed it which gave us the result we expected

We broke the 10GB data into chunks 50Kb-100MB(apprx 100-200 chunks depending upon the file size) and scripted in swift about the location of these files in the headnode. The wordcount program was written in python and integrated to swift.

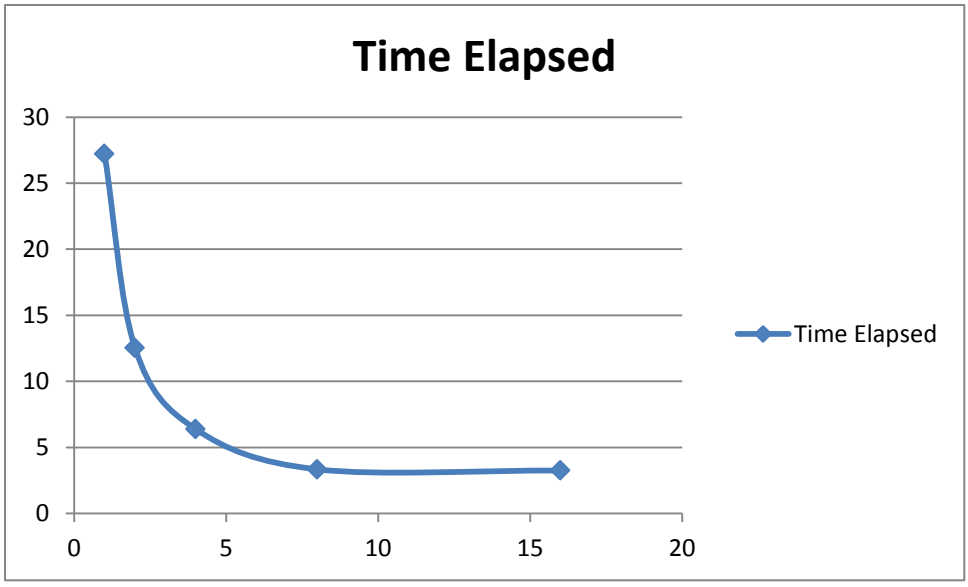
Going forward, we were getting accurate, constant word count values of the 10GB data set. The results of this wordcount were formulated in different files. So to create a single file result, we created another shell script to *merge* all these files together without duplication and ran it on the head node. A part of this assignment included sorting of a 10GB data set with swift.Initially we were working with t1.micro instance, when tried to run the 10GB data set on a T1 cluster, the performance was really low, say 3 times less than the C3, also we faced some memory overflow issues, which was because of the low memory offered in T1 group. Overall working with swift was really challenging and interesting. We explored some new concepts and boundaries in parallel programming.

SWIFT PERFORMANCE EVALUATION:

Our swift word count application includes, performing word count on 10GB dataset. Experiments are done on a 1,2,4,8 and 16 nodes of a ec2 cluster. Each experiment is done for three trails and values have been plotted for the average.

TIME UP GRAPH:

Nodes	Time Elapsed
1	27.21
2	12.55
4	6.38
8	3.34
16	3.25



Scale:

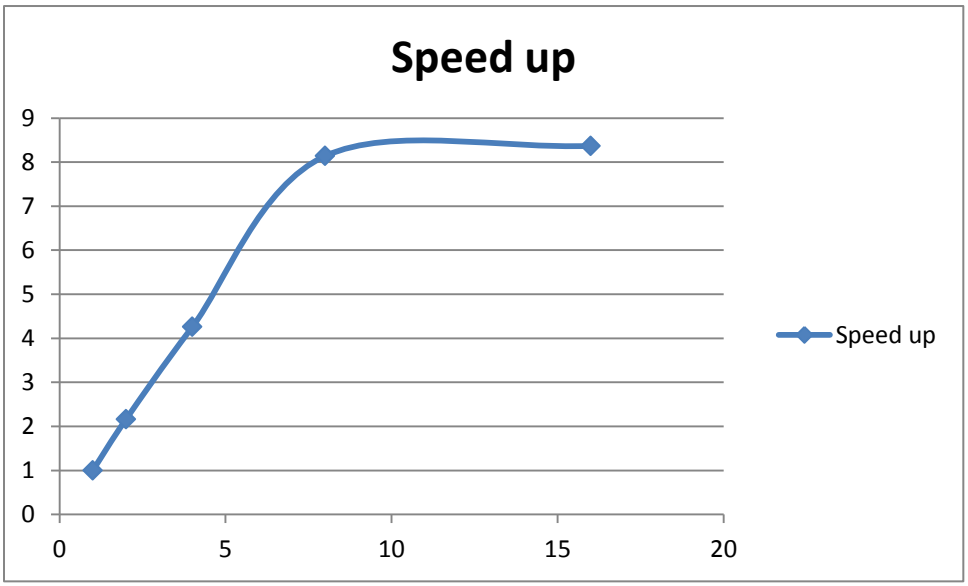
X-Axis: Integer # of Nodes

Y-Axis: Time Elapsed(minutes)

The graph shown above, the time-up graph, depicts the time taken to perform word count on a 10GB data set of Wikipedia. As expected the time difference between 1 node and 16 node is enormous. As we scale up from 1-16 nodes the time drops ideally.

SPEED GRAPH:

Nodes	Speed up
1	1
2	2.16
4	4.26
8	8.14
16	8.37



Scale:

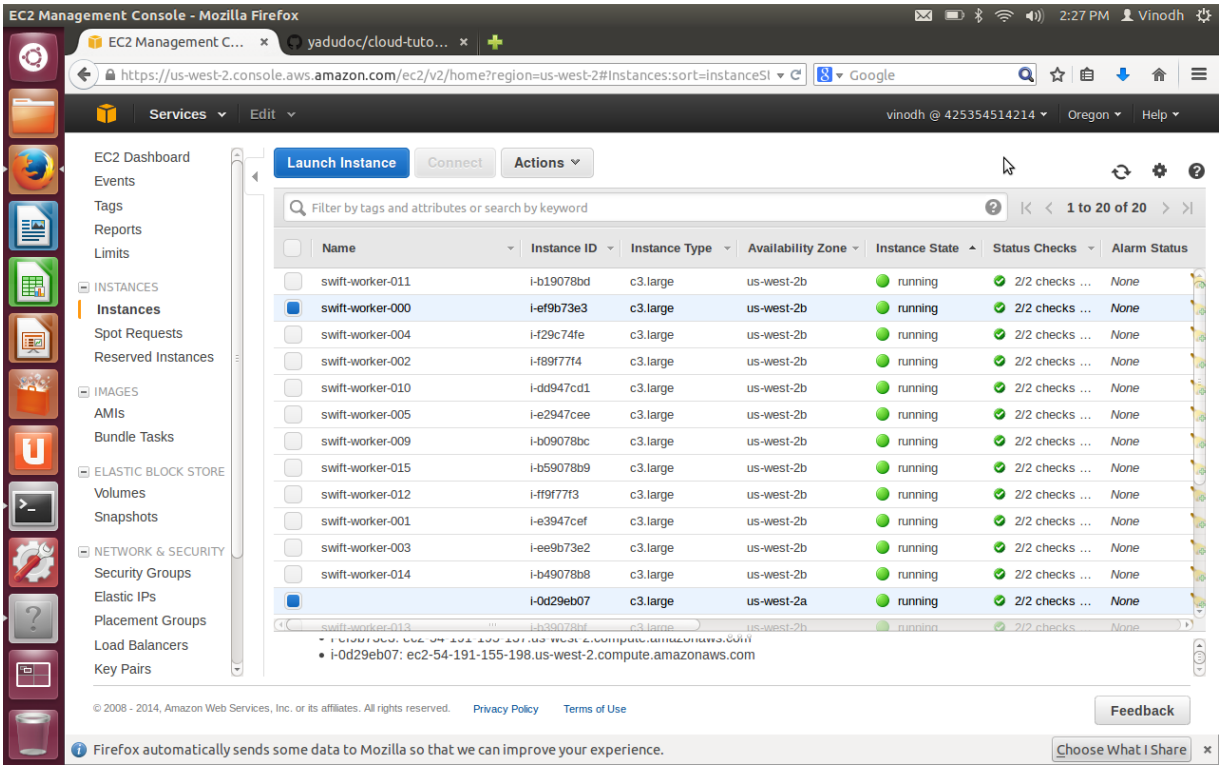
X-Axis: Integer # of Nodes

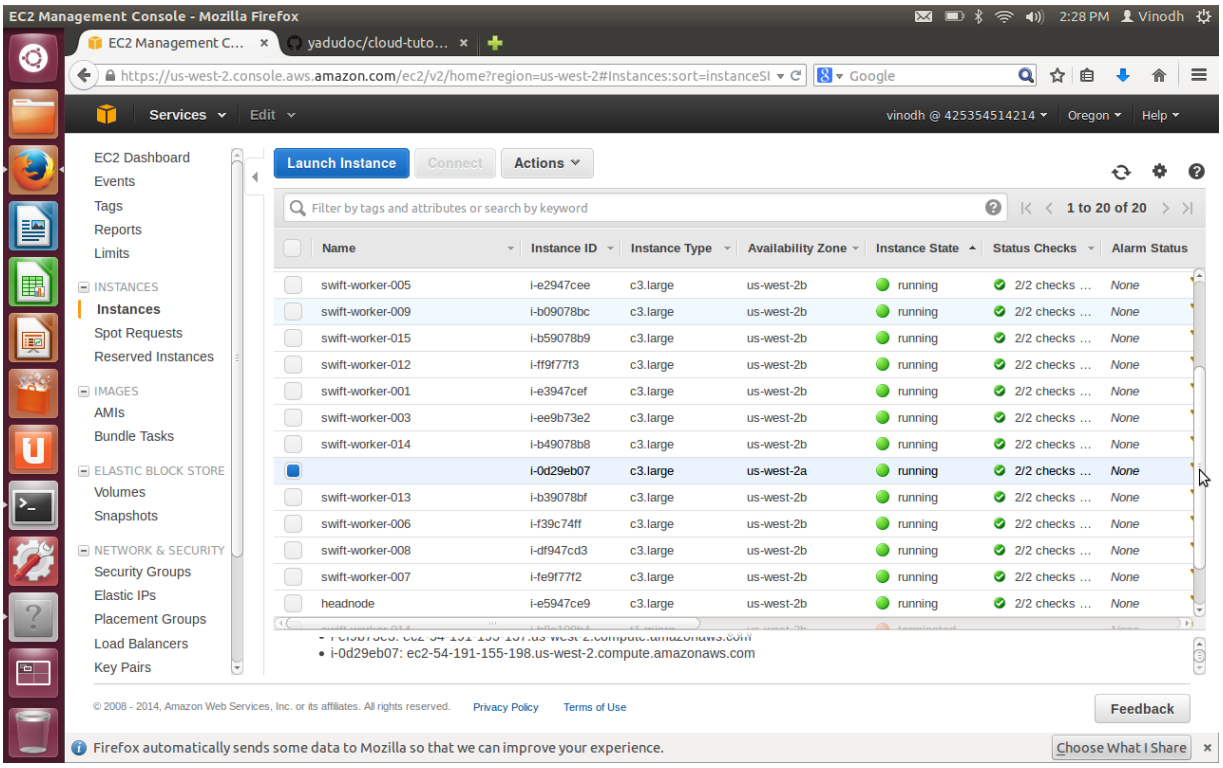
Y-Axis: Speed up(minutes)

The graph shown up above, the speed-up graph. Speed up is calculated by taking the time from single node as the *base value*.

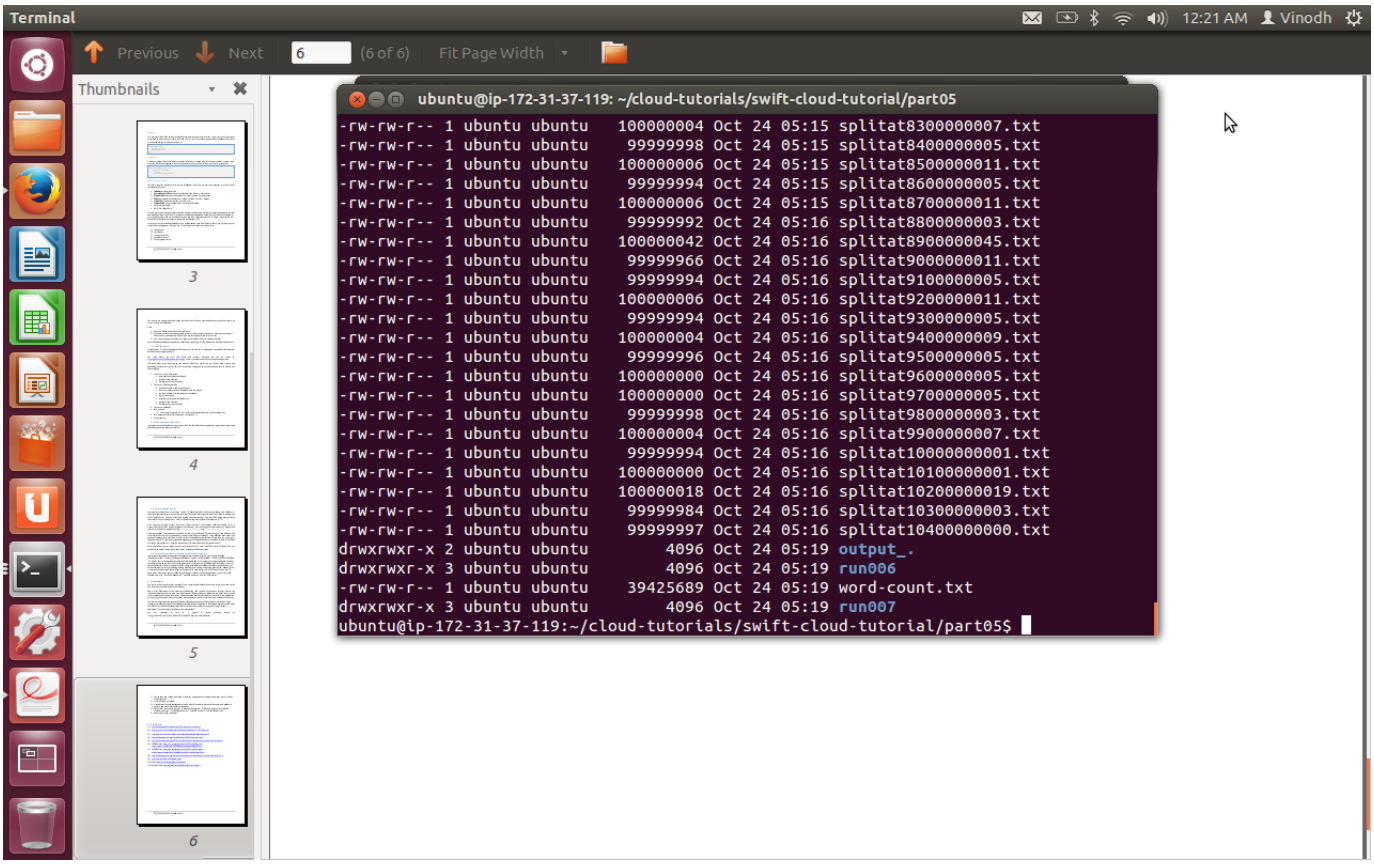
SCREEN SHOTS:

16 Instances Running

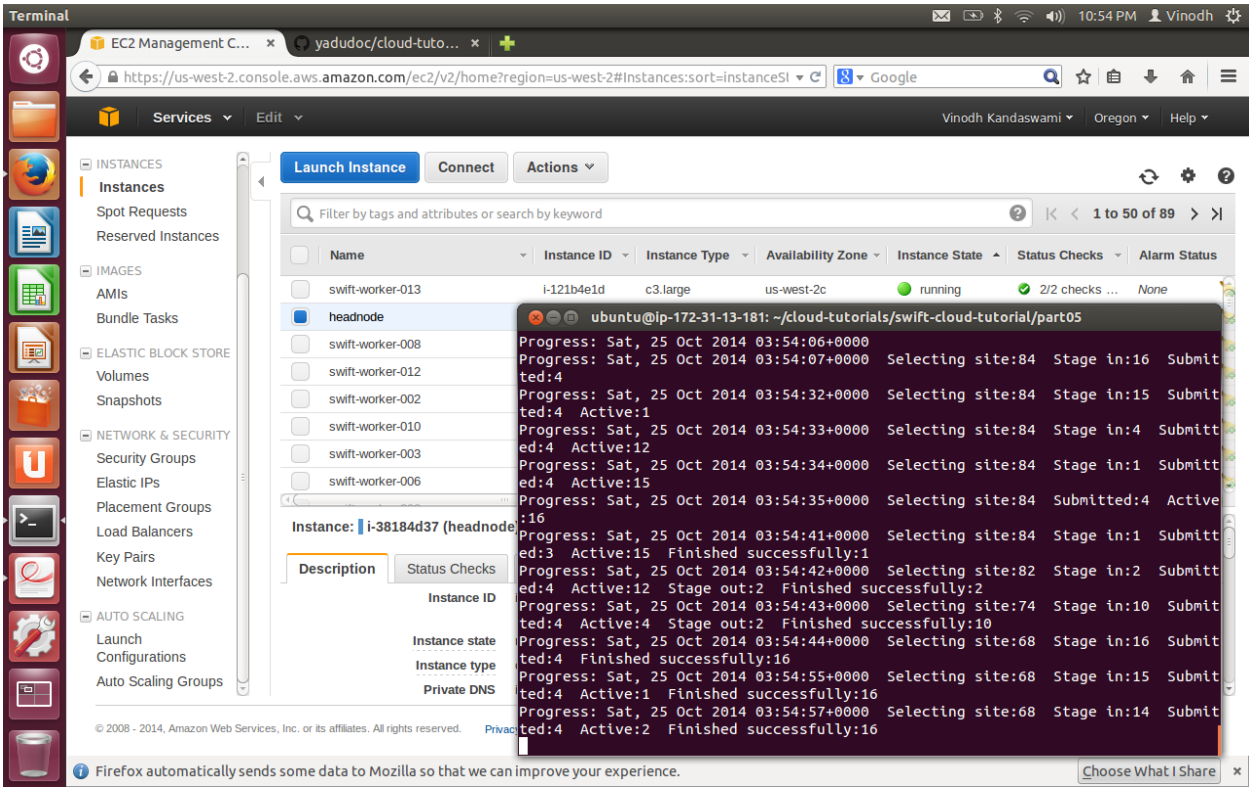



EC2 Management Console - Mozilla Firefox
EC2 Management C...
yadudoc/cloud-tuto...
https://us-west-2.console.aws.amazon.com/ec2/v2/home?region=us-west-2#Instances:sort=instanceSt
Services
Edit
vinodh @ 425354514214
Oregon
Help
EC2 Dashboard
Events
Tags
Reports
Limits
INSTANCES
Instances
Spot Requests
Reserved Instances
IMAGES
AMIs
Bundle Tasks
ELASTIC BLOCK STORE
Volumes
Snapshots
NETWORK & SECURITY
Security Groups
Elastic IPs
Placement Groups
Load Balancers
Key Pairs
Launch Instance
Connect
Actions
Filter by tags and attributes or search by keyword
1 to 20 of 20
Name
Instance ID
Instance Type
Availability Zone
Instance State
Status Checks
Alarm Status
swift-worker-005 i-e2947cee c3.large us-west-2b running 2/2 checks ... None
swift-worker-009 i-b09078bc c3.large us-west-2b running 2/2 checks ... None
swift-worker-015 i-b59078b9 c3.large us-west-2b running 2/2 checks ... None
swift-worker-012 i-ff9f77f3 c3.large us-west-2b running 2/2 checks ... None
swift-worker-001 i-e3947cef c3.large us-west-2b running 2/2 checks ... None
swift-worker-003 i-ee9b73e2 c3.large us-west-2b running 2/2 checks ... None
swift-worker-014 i-b49078b8 c3.large us-west-2b running 2/2 checks ... None
i-0d29eb07 c3.large us-west-2a running 2/2 checks ... None
swift-worker-013 i-b39078bf c3.large us-west-2b running 2/2 checks ... None
swift-worker-006 i-f39c74ff c3.large us-west-2b running 2/2 checks ... None
swift-worker-008 i-df947cd3 c3.large us-west-2b running 2/2 checks ... None
swift-worker-007 i-fe9f77f2 c3.large us-west-2b running 2/2 checks ... None
headnode i-e5947ce9 c3.large us-west-2b running 2/2 checks ... None
• i-0d29eb07: ec2-54-191-155-198.us-west-2.compute.amazonaws.com
© 2008 - 2014, Amazon Web Services, Inc. or its affiliates. All rights reserved.
Privacy Policy
Terms of Use
Feedback
Firefox automatically sends some data to Mozilla so that we can improve your experience.
Choose What I Share

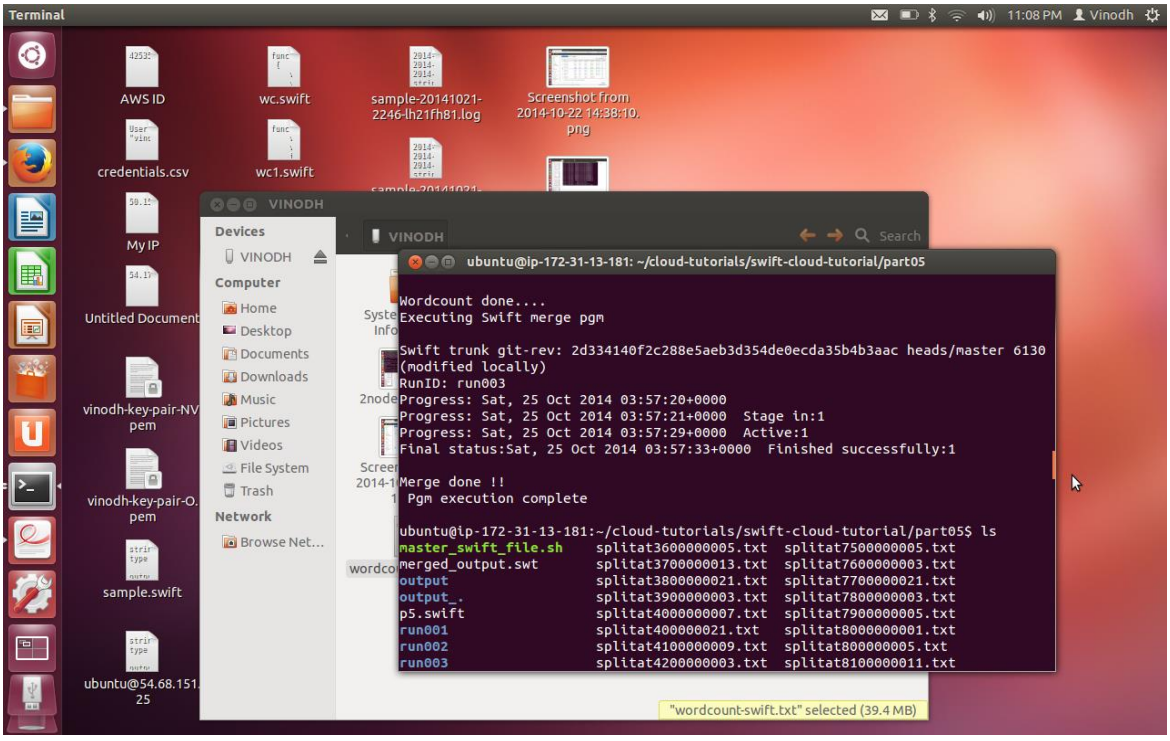
OUTPUT FILE:


Terminal
12:21 AM
Vinodh
Previous
Next
6
(6 of 6)
Fit Page Width
Thumbnails
ubuntu@ip-172-31-37-119: ~/cloud-tutorials/swift-cloud-tutorial/part05
-rw-rw-r-- 1 ubuntu ubuntu 100000004 Oct 24 05:15 splitat8300000007.txt
-rw-rw-r-- 1 ubuntu ubuntu 99999998 Oct 24 05:15 splitat8400000005.txt
-rw-rw-r-- 1 ubuntu ubuntu 100000006 Oct 24 05:15 splitat8500000011.txt
-rw-rw-r-- 1 ubuntu ubuntu 99999994 Oct 24 05:15 splitat8600000005.txt
-rw-rw-r-- 1 ubuntu ubuntu 100000006 Oct 24 05:15 splitat8700000011.txt
-rw-rw-r-- 1 ubuntu ubuntu 99999992 Oct 24 05:16 splitat8800000003.txt
-rw-rw-r-- 1 ubuntu ubuntu 100000042 Oct 24 05:16 splitat8900000045.txt
-rw-rw-r-- 1 ubuntu ubuntu 99999966 Oct 24 05:16 splitat9000000011.txt
-rw-rw-r-- 1 ubuntu ubuntu 99999994 Oct 24 05:16 splitat9100000005.txt
-rw-rw-r-- 1 ubuntu ubuntu 100000006 Oct 24 05:16 splitat9200000011.txt
-rw-rw-r-- 1 ubuntu ubuntu 99999994 Oct 24 05:16 splitat9300000005.txt
-rw-rw-r-- 1 ubuntu ubuntu 100000004 Oct 24 05:16 splitat9400000009.txt
-rw-rw-r-- 1 ubuntu ubuntu 99999996 Oct 24 05:16 splitat9500000005.txt
-rw-rw-r-- 1 ubuntu ubuntu 100000000 Oct 24 05:16 splitat9600000005.txt
-rw-rw-r-- 1 ubuntu ubuntu 100000000 Oct 24 05:16 splitat9700000005.txt
-rw-rw-r-- 1 ubuntu ubuntu 99999998 Oct 24 05:16 splitat9800000003.txt
-rw-rw-r-- 1 ubuntu ubuntu 100000004 Oct 24 05:16 splitat9900000007.txt
-rw-rw-r-- 1 ubuntu ubuntu 99999994 Oct 24 05:16 splitat10000000001.txt
-rw-rw-r-- 1 ubuntu ubuntu 100000000 Oct 24 05:16 splitat10100000001.txt
-rw-rw-r-- 1 ubuntu ubuntu 100000018 Oct 24 05:16 splitat10200000019.txt
-rw-rw-r-- 1 ubuntu ubuntu 99999984 Oct 24 05:16 splitat10300000003.txt
-rw-rw-r-- 1 ubuntu ubuntu 99999997 Oct 24 05:16 splitat10400000000.txt
drwxrwxr-x 2 ubuntu ubuntu 4096 Oct 24 05:19 output_.
drwxrwxr-x 3 ubuntu ubuntu 4096 Oct 24 05:19 run006
-rw-rw-r-- 1 ubuntu ubuntu 39425689 Oct 24 05:19 word-count.txt
drwxrwxr-x 3 ubuntu ubuntu 4096 Oct 24 05:19 run007
ubuntu@ip-172-31-37-119:~/cloud-tutorials/swift-cloud-tutorial/part05$

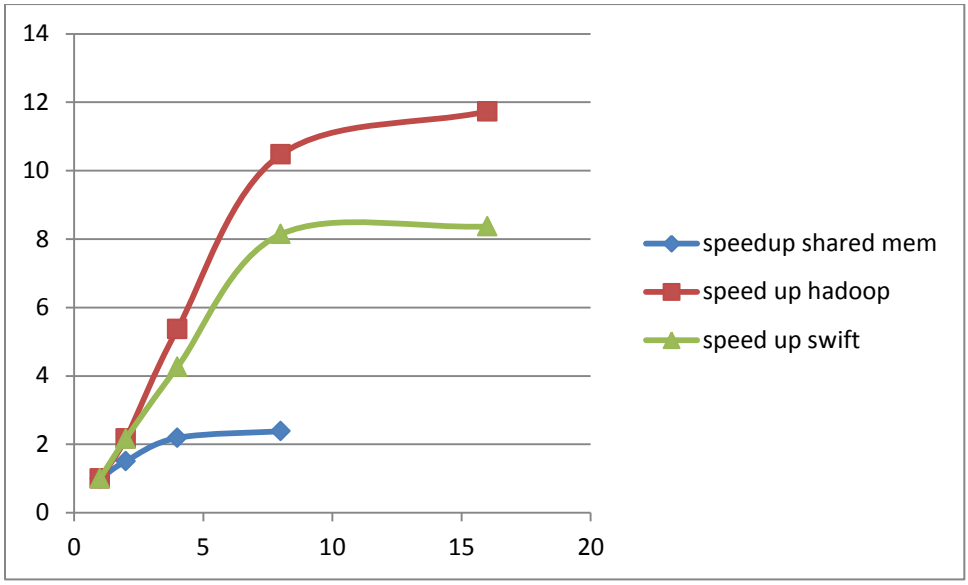
Start Time: (16 nodes) 03:54:06



End Time: (16nodes) 03:57:33



Comparison between shared memory ,hadoop and swift:



X-Axis: Integer # of Nodes

Y-Axis: Speed up

no of threads/nodes	speedup shared mem	speed up hadoop	speed up swift
1	1	1	1
2	1.506	2.17	2.16
4	2.19	5.37	4.26
8	2.38	10.47	8.14
16		11.73	8.37

Thus from the above graph it can be infered that for 100 nodes and 1000 nodes cluster hadoop gives the best performance.

PERFORMANCE COMPARISON BETWEEN SHARED MEMORY , HADOOP AND SWIFT

The difference in performance is due to

1.**shared memory** –we are using only one node so limited hardware whatever the number of threads be so performance is less than swift and hadoop.

2.**Hadoop** – Here it uses a mapper and reducer api which give it better performance as it does the splitting and node handling by itself . Also since we use a cluster which distributes the work among the slave nodes it give a better performance than shared memory and a almost equal performance with swift which uses a similar concept

3.**Swift-** Here swift utilizes both the instruction level parallelism and functional parallelism over the cluster of nodes it gives a better performance for lower number of nodes than hadoop . But as the number of nodes increase it almost becomes the same as hadoop.