Hadoop Submissions via SGE on Pythagoras HPC

Panagiotis Nastou, PhD

Department of Mathematics

School of Sciences

University of the Aegean

Introduction

Pythagoras is a traditional HPC system. HPC environments typically support batch job submissions using resource management systems such as TORQUE (or PBS) and SGE. The SGE resource management system is used in Pythagoras HPC.

Hadoop has its own scheduling, and manages its own job and task submissions and tracking (YARN: Yet Another Resource Negotiator). Unfortunately, both Hadoop and SGE compete for the complete control over the resources. The challenge is how to enable users to run Hadoop jobs in Pythagoras HPC using SGE. The target of this integration is to allow researchers to continue to use their MPI-based applications (using C/C++, Python, R, Java and Fortran) and to exploit Hadoop to address new data-intensive problems at the same time.

The adopted approach is based on the concept of configuring an N-node Hadoop cluster on demand by requesting N-nodes with multiple cores via SGE. The user actually requests a number of slots for his/her application the SGE allocates the requested slots equally balanced among the available HPC nodes and finally a hadoop cluster is built using the allocated nodes. Other users still have the ability to run other scientific calculations. We actually have the ability to run heterogeneous applications over the same HPC. In summary, the concept of this approach consists of the following steps:

1. Request a number of slots (cores) via SGE using the parallel environment hadoop.
2. The SGE allocates the slots equally balanced among the available HPC nodes and provides to hadoop system the allocated nodes and their description.
3. Upon the reception of the nodes, the Haddop configurations and environments are set up on this set of nodes. The Hadoop Distributed File System (HDFS) can be configured in a transient or persistent mode. In the transient mode, the HDFS is set up to use local storage. This means that before the completion of the Hadoop run, the user should transfer the produced data from the local storage to the front-end storage where the home directory of the user is located. To the contrary, in the persistent mode, the HDFS is set to symbolically link to a directory of the user home directory in the front-end storage. This means that no data transfer action is needed by the user before the completion of the Hadoop job. The default HDFS mode is the transient mode. Currently is the only mode supported on Pythagoras.

A Map-Reduce Paradigm

In this section, I will explain the development phase of a Map-reduce example. I will use a very popular example from apache hadoop site, the WordCount paradigm. Moreover, it is assumed that the developer should always check its application in a machine of least 8 physical cores, 16GB main memory and a disk of at least 500GB before submitting to a cluster. As for software, it assumed that the developer has any Linux distribution and has already installed JDK 8 (either openJDK or Oracle) and the IDE Eclipse for Java developers.

At first we download hadoop-2.6.0.tar.gz and let we are under the directory ~/softwareInstallation/tmp. The installation procedure is as follows,

```
$tar xzvf hadoop-2.6.0.tar.gz

$cd hadoop-2.6.0

$mkdir tmp

$mkdir hdfs

$mkdir logs

$chmod 777 tmp

$chmod 777 hdfs

$chmod 777 logs

$su

#mv hadoop-2.6.0 /usr/local

#chown -R root.root hadoop-2.6.0

#ln -sfn  /usr/local/hadoop-2.6.0 /usr/local/hadoop

#cd /usr/local/hadoop
```

Now we copy the *core-site.xml, hdfs-site.xml, mapred-site.xml and yarn-site.xml* as they appear in Appendix B. The following lines must be added in .bashrc and .bash_profile:

```
export JAVA_HOME=/usr/lib/jvm/java

export HADOOP_HOME=/usr/local/hadoop

export PATH=$PATH:$HADOOP_HOME/bin:$HADOOP_HOME/sbin

export HADOOP_MAPRED_HOME=$HADOOP_HOME

export HADOOP_COMMON_HOME=$HADOOP_HOME

export HADOOP_HDFS_HOME=$HADOOP_HOME
```

export YARN_HOME=$HADOOP_HOME

export JAVA_LIBRARY_PATH=$JAVA_LIBRARY_PATH:$HADOOP_HOME/lib/native

Now, we are ready to initiate our cluster in pseudo-distributed mode on our single node:

$hdfs namenode -format

$start-dfs.sh

$start-yarn.sh

The above procedure was successfully tested on Fedora 31. We continue now with the development phase.

At the end of the day the user should stop the hadoop cluster. Then the following procedure should be followed:

$stop-yarn.sh

$stop-dfs.sh

$su

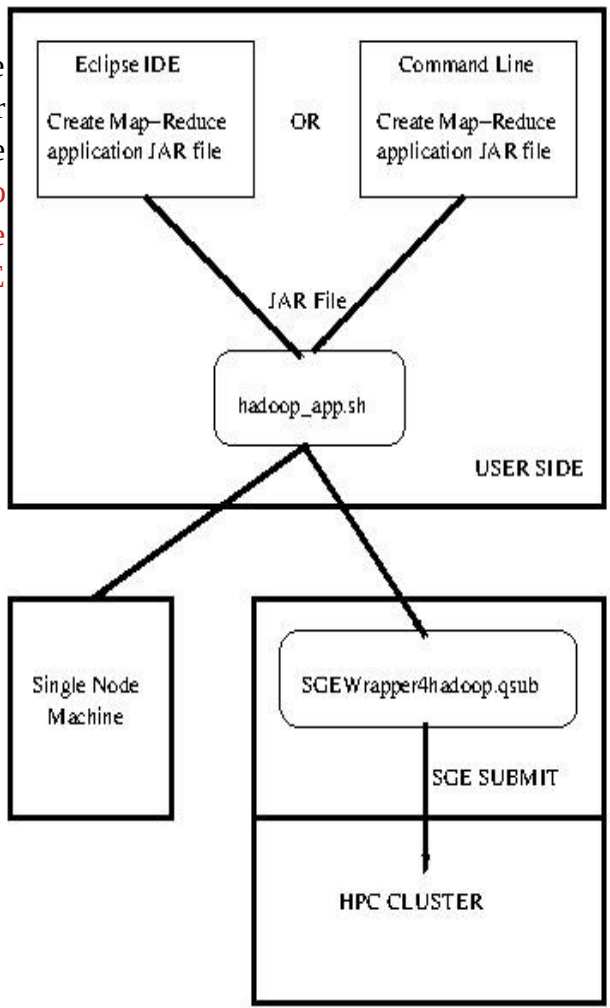
#cd /*usr*/local/hadoop/tmp

#rm -rf *

#cd ../hdfs

#rm -rf *

In the Figure on the right side, the phases of the development of a Map-Reduce application and their inter-dependencies are presented. I can try to describe each phase separately in the next sections. It is worth to note that the user side and the machine side should have the same java and hadoop versions (currently JAVA SE JDK 8 and hadoop 2.6.0).



A. Creating the Jar using the Eclipse IDE

Now we will use the IDE Eclipse in order to develop our map-reduce application. Our target is to create the jar file that will be executed by our hadoop platform. The steps are as follows

1. Launch Eclipse
2. New Java Project

3. Then Right-click on src folder under the name of the project. Select New Class giving the name WordCount.java.
4. Copy the code WordCount from Appendix B inside this class file.
5. Right Click on Project Name and Properties>Java Build Path>Add External Jars. We select all jars from the following directories:
    - /*usr*/local/hadoop-2.6.0/share/hadoop/mapreduce,
    - /*usr*/local/hadoop-2.6.0/share/hadoop/hdfs,
    - /*usr*/local/hadoop-2.6.0/share/hadoop/common,
    - /*usr*/local/hadoop-2.6.0/share/hadoop/common/lib
6. Right Click on Project Name and Export>Java>Jar File. Press Next. Deselect .classpath, .project. We give the name WordCount.Jar under the correct directory. Then Next. In the JAR MANIFEST SPECIFICATION put in the field labeled MAIN CLASS the java class that contains the main function and then click Next, Finish and our jar file is ready to run.

B. Creating the Jar file from command line

The jar file of our application can be created without the use of IDE Eclipse. Assuming we are under the directory ~/workspace/MapReduceExample/src where the file WordCount.java that contains our code exists. Then we write the following commands,

1. $javac -cp .:$(/usr/local/hadoop/bin/hadoop classpath) ./WordCount.java
2. $jar cvfe WordCount.jar WordCount *.class

and the jar file is ready for execution.

C. Running Our Map-Reduce example on our single machine

With editor nano or vi we create a text file that contains a number of words. Let testSample.txt its name under directory ./wordInput. Now, we are ready to create our application script. Let hadoop_app.sh its name.

```
---------hadoop_app.sh-------------
#!/usr/bin/bash
if [ $# != 3 ]
then echo "Usage: hadoop_app <jar_file> <LocalInputDirectory> <LocalOutputDirectory>"
fi
#Input Data preparation
hdfs dfs -mkdir /user
```

```
hdfs dfs -mkdir /user/$USER

hdfs dfs -put $2 input

#Starting our application and getting output data

hadoop jar $1 input output

hdfs dfs -get output/* $3/

hdfs dfs -cat output/*

---------end of hadoop_app.sh----------
```

Now we are ready to run it in our single node machine.

```
$./hadoop_app.sh WordCount.jar ./wordInput /output
```

where wordInput is the local directory where our input files exist and output is the directory under current working directory where the output files of our application will be stored.

D. Running Our Map-Reduce example on Pythagoras

A user creates a directory for his application under his home directory. Let myApp be the name of this directory. In this directory, we create our jar application file, namely WordCount.jar, the directory wordInput where the text file textSample.txt is stored and the directory output where the output files will be stored by our application jar. We create the following script

----------hadoopApp_Core.sh----------

```
#!/bin/bash

echo "Usage: hadoopApp_Core <jar_file> <localInputFile> <localOutputDirectory>"


#Input Data preparation

hdfs dfs -put -f $2 /input

#Starting our application

hadoop jar $1 /input output

#Getting Output Data

hdfs dfs -get output/* $3/
```

---------End of hadoopApp_Core.sh----------

It is easy to see that the above script has three arguments, the jar file, the directory where the jar input files are located and the output directory. In order to run our application we have to use the scripts hadoopClusterBuild.sh, hadoopClusterRelease.sh and hadoopApp_Wrapper.sh. The first is system scripts while the second one is a user script. The content of hadoopClusterBuild.sh, hadoopClusterRelease.sh appears in Appendix B while the content of hadoopApp_Wrapper.sh is as follows,

----------hadoopApp_Wrapper.sh----------

```
#!/bin/bash

#################################################################################

#  sge.qsub - A sample submit script for SGE that spins up a Hadoop cluster

#  for a map/reduce programming paradigm using myHadoop

#  Panos Nastou for Pythagoras Cluster FINAL VERSION   March 2020

#################################################################################

#SGE resource management system options

#$ -S /bin/bash

#$ -N Jefferson
```

#$ -V

#$ -cwd

#$ -o ./Jefferson.out.$JOB_ID

#$ -e ./Jefferson.err.$JOB_ID

#$ -q icsdQueue.q

#$ -pe hadoop 12

source ./hadoopClusterBuild.sh

source ./hadoopApp_Core.sh ./WordCount.jar ./wordInput ./output

source ./hadoopClusterRelease.sh

----------End of hadoopApp_Core.sh ----------

Now we are ready to run our application by simply giving

$qsub hadoopApp_Wrapper.sh

After the execution, under the directory named output the user will find the file part-r-00000 which is the file that contains the resulted files of the execution of WordCount.jar. Moreover, there is another directory under the current working directory named Jefferson_hadoop_conf.${JOB_ID} (Jefferson is the job name) where the configuration files and the log files are stored. Finally, as the preamble of the script determines two files are created where standard output and error streams of our script are stored.

APPENDIX B

-------**WordCount Code**----------

```
import java.io.IOException;
import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount {
  public static class TokenizerMapper
       extends Mapper<Object, Text, Text, IntWritable>{
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(Object key, Text value, Context context) throws IOException, InterruptedException
{
      StringTokenizer itr = new StringTokenizer(value.toString());
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);
      }
    }
```

```
  }

  public static class IntSumReducer
       extends Reducer<Text,IntWritable,Text,IntWritable> {
    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values, Context context) throws IOException,
InterruptedException {
      int sum = 0;
      for (IntWritable val : values) {
        sum += val.get();
      }
      result.set(sum);
      context.write(key, result);
    }
  }

  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
```

```
  }
}
----------------------hadoopClusterBuild.sh----------------------------
#!/bin/bash
################################################################################
#  hadoopClusterBuild.sh: Creates the hadoop cluster for our application
#  It is called immediately after SGE parameters inside our hadoop Wrapper.
#  Panos Nastou for Pythagoras Cluster FINAL VERSION   April 2020
################################################################################

### Hadoop, myHadoop and Spark Configuration Variables
### If these aren't already in your environment (e.g., .bashrc), we must define
### them.  We assume hadoop and myHadoop were installed in $HOME/hadoop-stack
#The variable MH_IPOIB_TRANSFORM must be set properly ince our network does not operate
#with IP over Infiniband protocols. It operates with IP over 10G Ethernet protocols.
#The variable MH_PERSIST_DIR must be empty (default).

SCRATCH=/state/partition1/hadoopScratch/${USER}/${JOB_NAME}.${JOB_ID}
export HADOOP_INSTALLATION_DIR=/opt/hadoop
export HADOOP_HOME=${HADOOP_INSTALLATION_DIR}/2.6.0
export MH_HOME=${HADOOP_INSTALLATION_DIR}/contrib/myHadoop
export MH_WORKDIR=${SGE_O_WORKDIR}
export MH_JOBID=${JOB_ID}
#export MH_IPOIB_TRANSFORM=""
export MH_IPOIB_TRANSFORM="s/\\([^.]*\\).*$/\\1.local/"
export MH_SCRATCH_DIR=${SCRATCH}
export SPARK_HOME=/opt/spark
export PATH=${HADOOP_HOME}/sbin:${HADOOP_HOME}/bin:${SPARK_HOME}/bin:$
{SPARK_HOME}/sbin:${MH_HOME}/bin:${PATH}
export WRKDIR=$(pwd)
```

```
export HADOOP_CONF_DIR=${WRKDIR}/${JOB_NAME}_hadoop_conf.${JOB_ID}
export HADOOP_COMMON_LIB_NATIVE_DIR=lib/native
export SPARK_CONF_DIR=${WRKDIR}/${JOB_NAME}_spark_conf.${JOB_ID}
export JAVA_HOME=/usr/java/latest
export JAVA_LIBRARY_PATH=${JAVA_LIBRARY_PATH}:{HADOOP_HOME}/lib/native

#We load the hadoop module: In Pythagoras the default is hadoop 2.6.0
module load hadoop

#We configure our cluster on the set of nodes returned by SGE
#Particularly the HDFS is configured so as to provide the storage appropriately.
#The following command configures the HDFS in transient mode
myhadoop-configure.sh
#We are ready now to start all Hadoop daemons
start-dfs.sh
start-yarn.sh
----------End hadoopClusterBuild.sh----------
----------hadoopClusterBuild.sh----------
#!/bin/bash
##############################################################################
#  hadoopClusterRelease.sh: It is called immediately after the completion of our
#  application script. It is responsible for the release of every allocated storage.
#  Panos Nastou for Pythagoras Cluster FINAL VERSION   April 2020
##############################################################################
#Stop all the Hadoop daemons
stop-yarn.sh
stop-dfs.sh
#Clean up after executing our code
myhadoop-cleanup.sh
----------End hadoopClusterRelease.sh----------
```