Lab 3 of SP19-BL-ENGR-E599-30563

Goal

- Walk through of the K-means example in Harp
- Memory configuration of mapper
- From single mapper to multiple mappers
- Use different communication operations
- Learn how to debug the program

Deliverables

Submit the experiment results to Canvas.

Run K-means example (Points=1000, Centroids=10, Dimension=100, Iterations=100) and record the execution time in milli-seconds, excluding the data loading time.

- Thread number fixed to 24 and mapper number varying from 1 to 4 (with correct memory configuration).
- Thread number fixed to 24 and mapper number to 4 with different communication operations and record the time spent in *communication* for each communication operations (allreduce, regroupallgather, bcastreduce, pushpull).
    - The time evaluation is already done in allreduce model, use it as an example.
    - After modifying the codes, you shall re-compile the harp codes before launching test script.

Evaluation

Lab participation: credit for 1 point based upon a successful completion of the lab tasks

Download the Harp Codes for Lab Session

Git pull the codes from *labsession* branch (Please run *git clone* command on the login node)

```
cd ~
mkdir -p Labsession
cd Labsession
git clone -b labsession https://github.com/DSC-SPIDAL/harp.git
```

Compile the codes by Maven

```
cd ~/labsession/harp
mvn clean package -Phadoop-2.6.0
## copy compiled jars to Hadoop directory
cp core/harp-hadoop/target/harp-hadoop-0.1.0.jar $HADOOP_HOME/share/hadoop/mapreduce/
cp core/harp-collective/target/harp-collective-0.1.0.jar $HADOOP_HOME/share/hadoop/ma
preduce/
cp core/harp-daal-interface/target/harp-daal-interface-0.1.0.jar $HADOOP_HOME/share/h
adoop/mapreduce/
cp third_party/*.jar $HADOOP_HOME/share/hadoop/mapreduce/
```

The Harp API

Harp provides communication and computation modules for distributed implementations of data analytics algorithms. We first introduce fundamental data structures, the inter-mapper communication API. In the next Lab seesion, we shall cover the multi-threading schedulers.

Data Structure in Harp

Harp provides three levels of data structures: arrays and objects, partition, and table.

Primitive Data Type

Arrays and Serializable objects are the basic data structures, which include:

1. `ByteArray` : an array with byte-type elements
2. `ShortArray` : an array with short-type elements
3. `IntArray` : an array with int-type elements
4. `FloatArray` : an array with float-type elements
5. `LongArray` : an array with long-type elements

6. `DoubleArray` : an array with double-type elements

7. `Writable` : serializable object

Partition

`Partition` is a wrapper of the data structures shown above. Every partition has an ID. In collective communication, partitions from different processors with the same ID will be merged. The merge operation is defined by `PartitionCombiner` .

Table

`Table` is a container for partitions. It is a high-level data structure and the unit for collective communication.

An example of how to construct a table is:

```
Table<DoubleArray> table = new Table<>(0, new DoubleArrPlus());
for (int i = 0; i < numPartitions; i++) {
    DoubleArray array = DoubleArray.create(size, false);
    table.addPartition(new Partition<>(i, array));
}
```

Inter-Mapper Communication Operations

All of the communication APIs are defined at `core/harp-hadoop/src/main/java/org/apache/hadoop/mapred/CollectiveMapper.java` We select five operations used in the K-means. The rest of the APIs are introduced with details at our website tutorial page.

Allreduce

`allreduce` aims to first combine tables from other workers and then broadcast the accumulated table. All workers should run it concurrently. The definition of the method is:

```
/**
 * Allreduce partitions of the tables to all the
 * local tables.
 *
 * @param contextName
 *          the name of the operation context
 * @param operationName
 *          the name of the operation
 * @param table
 *          the table to hold the partitions
 * @return a boolean tells if the operation
 *         succeeds
 */
public <P extends Simple> boolean allreduce(
  String contextName, String operationName,
  Table<P> table) {
  boolean isSuccess =
    AllreduceCollective.allreduce(contextName,
      operationName, table, dataMap, workers);
  dataMap.cleanOperationData(contextName,
    operationName);
  return isSuccess;
}
```

```
//Example
allreduce("k-means", "allreduce_centroids", table);
```

Allgather

`allgather` aims to first collect tables from other workers and then broadcast the collection. All workers should run it concurrently. The definition of the method is:

```
/**
 * Allgather partitions of the tables to all the
 * local tables.
 *
 * @param contextName
 *          the name of the operation context
 * @param operationName
 *          the name of the operation
 * @param table
 *          the table to hold the partitions
 * @return a boolean tells if the operations
 *         succeeds
 */
public <P extends Simple> boolean allgather(
  String contextName, String operationName,
  Table<P> table) {
  boolean isSuccess =
    AllgatherCollective.allgather(contextName,
      operationName, table, dataMap, workers);
  dataMap.cleanOperationData(contextName,
    operationName);
  return isSuccess;
}
```

```
//example
allgather("k-means", "allgather_centroids", table);
```

Broadcast

`broadcast` aims to share a table in one worker with others. All workers should run it concurrently. The definition of the method is:

```
/**
 * Broadcast the partitions of the table on a
 * worker to other workers.
 *
 * @param contextName
 *          the name of the operation context
 * @param operationName
 *          the name of the operation
 * @param table
 *          the table used to hold the
 *          partitions
 * @param bcastWorkerID
 *          the worker ID of broadcasting data
 * @param useMSTBcast
 *          if minimum-spanning tree algorithm
 *          is used
 * @return a boolean tells if the operation
 *         succeeds
 */
public <P extends Simple> boolean broadcast(
  String contextName, String operationName,
  Table<P> table, int bcastWorkerID,
  boolean useMSTBcast) {
  boolean isSucess =
    BcastCollective.broadcast(contextName,
      operationName, table, bcastWorkerID,
      useMSTBcast, dataMap, workers);
  dataMap.cleanOperationData(contextName,
    operationName);
  return isSucess;
}
```

```
// example to bcast data from mapper 0 and to use minimum-spanning tree
broadcast("k-means", "bcast_centroids", table, 0, true);
```

Push

`push` aims to collect all workers' partitions and store them in global. All workers should run it concurrently. The definition of the method is:

```java
/**
 * Push the partitions of local tables to the
 * global table.
 *
 * @param contextName
 *          the name of the operation context
 * @param operationName
 *          the name of operation
 * @param localTable
 *          the local tables
 * @param globalTable
 *          the global table which acts like a
 *          distributed dataset, each partition
 *          in this table is unique
 * @param partitioner
 *          when a partiitoner is used, the
 *          local partitions are sent to the
 *          global table even without partiiton
 *          ID association
 * @return a boolean tells if the operation
 *         succeeds
 */
public <P extends Simple, PT extends Partitioner>
  boolean push(String contextName,
    String operationName, Table<P> localTable,
    Table<P> globalTable, PT partitioner) {
  boolean isSuccess =
    LocalGlobalSyncCollective.push(contextName,
      operationName, localTable, globalTable,
      partitioner, dataMap, workers);
  dataMap.cleanOperationData(contextName,
    operationName);
  return isSuccess;
}
```

```java
//example
push("k-means", "push_local_centroids", table_local, table_global, new Partitioner(this.getNumWorkers()));
```

Pull

`pull` aims to distribute the partitions stored in global to all workers. All workers should run it concurrently. The definition of the method is:

```
/**
 * Pull partitions in the global table to the
 * local tables. If any partition ID conflicts
 * with the partition ID in the local table,
 * combines them.
 *
 * @param contextName
 *          the name of the operation context
 * @param operationName
 *          the name of the operation
 * @param localTable
 *          the local tables
 * @param globalTable
 *          the global table, acts like a
 *          distributed dataset
 * @param useBcast
 *          if using broadcast in scattering the
 *          partitions
 * @return a boolean tells if the operation
 *         succeeds
 */
public <P extends Simple> boolean pull(
  String contextName, String operationName,
  Table<P> localTable, Table<P> globalTable,
  boolean useBcast) {
  boolean isSuccess =
    LocalGlobalSyncCollective.pull(contextName,
      operationName, localTable, globalTable,
      useBcast, dataMap, workers);
  dataMap.cleanOperationData(contextName,
    operationName);
  return isSuccess;
}
```

```
// example
pull("k-means", "global-local", table_local, table_global, true);
```

A Walk Through of K-means in Harp

The file of K-means example is at the following path

```
cd harp/contrib/src/main/java/edu/iu/kmeans/
```

The `common` folder contains the entrance of the K-means program, where it configures the Harp mapper function and launch the Hadoop job. The other folders refer to different distributed implementations of K-means. The file *common/KmeansMapCollective.java* configures and launches a harp job upon Hadoop system.

```
 private void runKMeans(int numOfDataPoints,
   int numCentroids, int vectorSize,
   int numIterations, int JobID, int numMapTasks,
   Configuration configuration, Path workDirPath,
   Path dataDir, Path cDir, Path outDir,
   String operation)
   throws IOException, URISyntaxException,
   InterruptedException, ClassNotFoundException {

...

do {
// ----------------------------------------------------------------------

...

 // configure kmeans job
   Job kmeansJob = configureKMeansJob(
     numOfDataPoints, numCentroids, vectorSize,
     numMapTasks, configuration, workDirPath,
     dataDir, cDir, outDir, JobID,
     numIterations, operation);

...

 // run kmeans job
   jobSuccess =
     kmeansJob.waitForCompletion(true);

...

 // check whether job is successful
     if (!jobSuccess) {
       System.out.println(
         "KMeans Job failed. Job ID:" + JobID);
       jobRetryCount++;
       if (jobRetryCount == 3) {
         break;
       }
     } else {
       break;
     }
   } while (true);
 }
```

Job Configuration

Line 226 to 245 configures the mapper function we will use in K-means. Each mapper corresponds to a type of synchronization pattern in distributed system, namely the computation model. Find more details in our website documentation

```
226     // use different kinds of mappers
227     if (operation.equalsIgnoreCase("allreduce")) {
228       job.setMapperClass(
229         edu.iu.kmeans.allreduce.KmeansMapper.class);
230     } else if (operation
231       .equalsIgnoreCase("regroup-allgather")) {
232       job.setMapperClass(
233         edu.iu.kmeans.regroupallgather.KmeansMapper.class);
234     } else if (operation
235       .equalsIgnoreCase("broadcast-reduce")) {
236       job.setMapperClass(
237         edu.iu.kmeans.bcastreduce.KmeansMapper.class);
238     } else if (operation
239       .equalsIgnoreCase("push-pull")) {
240       job.setMapperClass(
241         edu.iu.kmeans.pushpull.KmeansMapper.class);
242     } else {// by default, allreduce
243       job.setMapperClass(
244         edu.iu.kmeans.allreduce.KmeansMapper.class);
245     }
```

Secondly, we must setup the memory allocated to each mapper in Hadoop and the ratio of JVM heap memory from Line 265 to 274,

```
265     jobConf.setInt(
266       "mapreduce.map.collective.memory.mb", this.mem_per_mapper);
267     // mapreduce.map.collective.java.opts
268     int xmx = (int) Math.ceil((this.mem_per_mapper)*0.8);
269     int xmn = (int) Math.ceil(0.25 * xmx);
270     jobConf.set(
271       "mapreduce.map.collective.java.opts",
272       "-Xmx" + xmx + "m -Xms" + xmx + "m"
273         + " -Xmn" + xmn + "m");
274     return job;
```

where *mapreduce.map.collective.memory.mb* is the memory allocated to each mapper in MB, which overrides the Hadoop mapper configuration xml files *mapred-site.xml*. Here the default value is 2GB, and make sure the value memory per mapper times the number of mappers does not exceed the total memory allocated to a node specified by *yarn.nodemanager.resource.memory-mb* in *yarn-site.xml*

The *xmx* and *xmn* refers to the maximum and minimum value of memory available to the JVM heap. By default, it is 80%.

Job launch

The invocation of function *waitForCompletion* will launch the mapper function defined in the job configuration step and wait for its return.

```
jobSuccess =
  kmeansJob.waitForCompletion(true);
```

Here we do not setup the reducer function in the job configuration step at line 254, because Harp uses in-memory collective `java 254 job.setNumReduceTasks(0);` communication operation such as *allreduce* to execute the similar task of a *Reducer* in the standard MapReduce framework.

Whenever a job is launched, the control flow is taken by the mapper function defined in `allreduce/KmeansMapper.java`, which shall execute steps as follows

- Data Loading (All the mappers)
- Local Computation (All the mappers)
- Collective Communication (Among mappers)
- Output Result (Master mapper)

Except for data loading and output of result, the other steps are iteratively executed in a for loop.

Data Loading

K-means has two types of data. 1) The training data (data points) is loaded from HDFS , 2) the model data (centroids) is loaded by master mapper and broadcasted to the slave mappers.

```
// load data
ArrayList<DoubleArray> dataPoints =
  loadData(fileNames, vectorSize, conf);
numPoints = dataPoints.size();
```

```
...

    Table<DoubleArray> cenTable =
      new Table<>(0, new DoubleArrPlus());
    if (this.isMaster()) {
      loadCentroids(cenTable, vectorSize,
        conf.get(KMeansConstants.CFILE), conf);
    }

...

     // broadcast centroids
     broadcastCentroids(cenTable);
...
```

Local Computation

Each node holds a portion of training data and a complete copy of the centroids data. The local computation computes the MSE value (mean square error) and updates the centroids table by the partial results.

```
     // iterations
     for (int iter = 0; iter < iteration; iter++) {
       previousCenTable = cenTable;
       cenTable =
         new Table<>(0, new DoubleArrPlus());

...

     // compute new partial centroid table using
     // previousCenTable and data points
       MSE = computation(cenTable, previousCenTable,
         dataPoints);
...

}
```

Inter-mapper Communication

The inter-mapper communication at each iteration plays the same role as the *reducer* in MapReduce.

```
    // iterations
    for (int iter = 0; iter < iteration; iter++) {

...

    // AllReduce;
      allreduce("main", "allreduce_" + iter,
        cenTable);

...

      calculateCentroids(cenTable);

...

}
```

allreduce is a communication operation API defined by Harp, which first *reduce* the partial results to Mapper 0 and then broadcast the results to all of the other mappers. *allreduce* adopts the minimum spanning tree algorithm in reducing the data, which makes it fast when compared to standard *Reduce* operation.

After the *allreduce* communication, the mapper needs to re-calculate the centroids table, which could also be done within the *allreduce* communication operation via the user-defined *allreduce* operation.

Output Results

When all of the iterations are completed, calculate the final MSE value and write back the centroids table to HDFS. This step is only executed at the Master mapper.

```
    // evaluate the results
    calcMSE(conf);

    if (this.isMaster()) {
      outputCentroids(cenTable, conf, context);
    }

    //save evaluation info
    if (this.isMaster()){

      FileSystem fs = FileSystem.get(conf);
      Path path = new Path(conf.get(KMeansConstants.WORK_DIR) + "/evaluation");
      FSDataOutputStream output =
        fs.create(path, true);
      BufferedWriter writer = new BufferedWriter(
        new OutputStreamWriter(output));
      writer.write("MSE : " + this.finalMSE + "\n");
      writer.write("Compute Time : " + this.compute_time + "\n");
      writer.write("Comm Time : " + this.comm_time + "\n");
      writer.close();

    }
```

Run K-means on Multiple Mappers

We provide a bash script to run the K-means example. Please check the file

```
## to edit the script
vim contrib/test_scripts/k-means.sh
## to run the script
contrib/test_scripts/k-means.sh
```

The first part of the script file defines a variety of paths related to our execution.

```
#!/bin/bash

#
# test script for KMeans, using random generated dataset
#

#get the startup directory
startdir=$(dirname $0)
harproot=$(readlink -m $startdir/../../)
bin=$harproot/contrib/target/contrib-0.1.0.jar
hdfsroot=/harp-test
hdfsoutput=$hdfsroot/km/

if [ ! -f $bin ] ; then
    echo "harp contrib app not found at "$bin
    exit -1
fi
if [ -z ${HADOOP_HOME+x} ];then
    echo "HADOOP not setup"
    exit
fi

hdfs dfsadmin -safemode get | grep -q "ON"
if [[ "$?" = "0" ]]; then
    hdfs dfsadmin -safemode leave
fi

#workdir
workdir=test_km

mkdir -p $workdir
cd $workdir
```

The second part of the script defines a function to call the program execution. Here $1, $2, ... represents the arguments fed to the function. At the end of the function, the output files are downloaded from HDFS to your local folder

```
runtest()
{
        # <numOfDataPoints>: the number of data points you want to generate randomly
        # <num of centriods>: the number of centroids you want to clustering the data to
        # <size of vector>: the number of dimension of the data
        # <number of map tasks>: number of map tasks
        # <number of iteration>: the number of iterations to run
        # <work dir>: the root directory for this running in HDFS
        # <local dir>: the harp kmeans will firstly generate files which contain data points to local directory. Set this argu
        # <communication operation> includes:
        #    [allreduce]: use allreduce operation to synchronize centroids
        #    [regroup-allgather]: use regroup and allgather operation to synchronize centroids
        #    [broadcast-reduce]: use broadcast and reduce operation to synchronize centroids
        #    [push-pull]: use push and pull operation to synchronize centroids
        # <mem per mapper>
    hadoop jar $bin edu.iu.kmeans.common.KmeansMapCollective $1 $2 $3 $4 $5 $hdfsoutput/$6 /tmp/kmeans $6 $7

    if [ $? -ne 0 ]; then
        echo "run km failure"
        exit -1
    fi

    ## retrieve the results from HDFS
    if [ -d ./$6 ];then
     rm -rf ./$6/*
    else
     mkdir -p ./$6
    fi

    hdfs dfs -get $hdfsoutput/$6/evaluation ./$6
    hdfs dfs -get $hdfsoutput/$6/centroids ./$6
}
```

The third part of the script runs the experiment with 1000 data points, 10 centroids, feature dimension of 100, 2 mappers, 100 iterations, allreduce model, and 2000MB memory per mapper.

```
#run test
runtest 1000 10 100 2 100 allreduce 2000
```

The output result is at `./test_km/allreduce/evaluatation`

Debugging of the Harp workload

Harp workload is running within the Hadoop runtime. When a program fails, we must first collect any useful log information, and secondly insert print out statements into the source codes to identify and isolate the problematic code sections.

Collect Log Information

There are two types of log files related to the Harp workload.

The Hadoop system logs are located at `$HADOOP_HOME/logs` folder by default, where each of your node keeps a log file that traces all of the events, including any exceptions (e.g., fail to start the node). For instance, a file named `hadoop-lc37-datanode-j-070.log` refers to the log information of datanode *j-070* owned by user *lc37*. Similarly, there are log files for Namenode, NodeManager, and ResourceManager.

1. hadoop-${user}-namenode-${node}.log
2. hadoop-${user}-secondarynameNode-${node}.log
3. hadoop-${user}-datanode-${node}.log
4. yarn-${user}-nodemanager-${node}.log
5. yarn-${user}-resourcemanager-${node}.log

The Application specific log files are defined at file `yarn-site.xml`

```
<property>
     <name>yarn.nodemanager.log-dirs</name>
     <value>/local-path/userlogs</value>
</property>
```

The log files named by the Application ID (Job ID). For instance, a folder named `application_1524272601886_0164` contains all the log files for Application 1524272601886_0164. It includes three subfolders

1. *container152427260188601640100000 1* records the log information of the AppMaster container process.
2. *container1524272601886016401000002* records the log information of the first YarnChild container process.
3. *container1524272601886016401000003* records the log information of the second YarnChild container process.

There are two YarnChild containers because job `application_1524272601886_0164` launches two mappers. By checking the contents of these subfolders (e.g., the syslog file), you may figure out what causes the crash of the mapper functions.

Print out Intermediate Values

A most straightforward yet effective way of debugging parallel and distributed program is to print out any intermediate values of variables or setup check points. In Harp, you may use the `LOG` class to record variable status and check them in the *syslog* file mentioned above.

```
LOG.info("Start collective mapper.");
```

For example, setup a check point at the beginning of the mapper function execution. If the program crashes, and you are not able to find the message *Start collective mapper.* in syslog file of a YarnChild, then the error occurs even before the launch of the mapper function.

Also, you could use normal Java printing function such as the `System.out.println` to record information, and these information could be found at `stdout` and `stderr` files alongside the `syslog` within the Application specific log folder.

```
System.out.println("Errors: " + err);
```