Running Examples

HelloWorld Example

- In order to run these examples, you should have a running Kubernetes or a Mesos cluster, configured as stated on Twister2 website.
- You can submit jobs to chosen resource scheduler by using Twister2 executable;
  - ./bin/twister2
- When submitting jobs to Kubernetes clusters, you need to specify the cluster name as "kubernetes". Likewise this parameter should be “mesos” if submitting to a Mesos cluster.
- You can submit HelloWorld job in examples package with 8 workers as;
  - ./bin/twister2 submit kubernetes jar examples/libexamples-java.jar edu.iu.dsc.tws.examples.basic.HelloWorld 8
  - If you are using our testbed cluster “Echo”,
    - login to your account
    - change you directory to twister2/twister2/bazel-bin/scripts/package/twister2-0.1.0/
    - then run the command above.

HelloWorld Example

- You can check the status of the submitted job through the dashboard provided by the resource scheduler. For our Echo Cluster the addresses are;
  - Kubernetes ---> http://149.165.150.81:8080/#/jobs
  - Mesos---> http://149.165.150.81:5050
- HelloWorld job runs for 1 minute. After 1 minute, the job becomes COMPLETED if everything is fine.
- Lets check the code; In the main method;
  - First set the number of workers by using the arguments passed by the command line. If there is no arguments then the default value is 4
  - Then we load the configurations from command line and config files(yaml files) and put it in a JobConfig object.
    - `Config config = ResourceAllocator.loadConfig(new HashMap<>());`
    - `JobConfig jobConfig = new JobConfig();`
    - `jobConfig.put("hello-key", "Twister2-Hello");`

HelloWorld Example

- We then create the Twister2 job object by the following code.
  - `Twister2Job twister2Job = Twister2Job.newBuilder()`
  - `.setJobName("hello-world-job")`
  - `.setWorkerClass(HelloWorld.class)`
  - `.addComputeResource(2, 1024, numberOfWorkers)`
  - `.setConfig(jobConfig)`
  - `.build();`
- LAst thing we do is submitting the job
  - `Twister2Submitter.submitJob(twister2Job, config);`
- After the submission execute method of this HelloWorld class will be called on each worker. So that means the code in the execute method will be run on every worker
- `public void execute(Config config, int workerID, IWorkerController workerController,`
- `IPersistentVolume persistentVolume, IVolatileVolume volatileVolume)`
-

HelloWorld Example

- In our case the rest of the code writes hello message from each worker including their IDs, total number of workers and the message and then sleeps for one minute

```
// lets retrieve the configuration set in the job config
String helloKeyValue = config.getStringValue("hello-key");
// lets do a log to indicate we are running
LOG.log(Level.INFO, String.format("Hello World from Worker %d; there are %d total workers "
      + "and I got a message: %s", workerID,   workerController.getNumberOfWorkers(), helloKeyValue));
List<JobMasterAPI.WorkerInfo> workerList = null;
try {
 workerList = workerController.getAllWorkers();
} catch (TimeoutException timeoutException) {
 LOG.log(Level.SEVERE, timeoutException.getMessage(), timeoutException);
 return;
}
String workersStr = WorkerInfoUtils.workerListAsString(workerList);
LOG.info("All workers have joined the job. Worker list: \n" + workersStr);
try {
 LOG.info("I am sleeping for 1 minute and then exiting.");
 Thread.sleep(60 * 1000);
 LOG.info("I am done sleeping. Exiting.");
} catch (InterruptedException e) {
 LOG.severe("Thread sleep interrupted.");
}
```

Kmeans Example

- Process of clustering numerous datasets containing large numbers of records with high dimensions calls for innovative methods.
- Traditional sequential clustering algorithms are unable to handle it. They are not scalable in relation to larger sizes of data sets, and they are most often computationally expensive in memory space and time complexities.
- The parallelization of data clustering algorithms is paramount when dealing with big data.
- K-Means clustering is an iterative algorithm hence, it requires a large number of iterative steps to find an optimal solution, and this procedure increases the processing time of clustering.
- Twister2 provides a “dataflow task graph” based approach to distribute the tasks in a parallel manner and aggregate the results which reduces the processing time of K-Means Clustering process.

Kmeans Example

- The following command generates and writes the datapoints and centroids in the local file system and submits the Kmeans job to Kubernetes cluster.
  - *./bin/twister2 submit kubernetes jar examples/libexamples-java.jar edu.iu.dsc.tws.examples.batch.kmeans.KMeansJobMain -workers 4 -iter 2 -dim 2 -clusters 4 -fname /tmp/output.txt -pointsfile /tmp/kinput.txt -centersfile /tmp/kcentroid.txt -points 100 -filesys local -pseedvalue 100 -cseedvalue 500 -input generate -parallelism 4*
- Sample output of this example can be seen from the worker logs and it would look like this;
  - [2018-10-05 10:44:18 -0400] [INFO] edu.iu.dsc.tws.examples.batch.kmeans.KMeansJob: %%% Final Centroid Values Received: %%%[[0.6476253753699173, 0.8468354813977953], [0.2687721020384673, 0.5083954227865372], [0.7860664115708306, 0.5381449347446825], [0.6675069260759725, 0.17798022472253153]]

Kmeans Example implementation details

- KMeansConstants
  - public static final String ARGS_WORKERS = "workers";
    public static final String ARGS_ITR = "iter";
    public static final String ARGS_FNAME = "fname";
    public static final String ARGS_POINTS = "pointsfile";
    public static final String ARGS_CENTERS = "centersfile";
    public static final String ARGS_DIMENSIONS = "dim";
    public static final String ARGS_CLUSTERS = "clusters";
    public static final String ARGS_NUMBER_OF_POINTS = "points";
    public static final String ARGS_FILESYSTEM = "filesys"; // "local" or "hdfs"
    public static final String ARGS_POINTS_SEED_VALUE = "pseedvalue"; //range for random data points generation
    public static final String ARGS_CENTERS_SEED_VALUE = "cseedvalue"; //range for centroids generation
    public static final String ARGS_DATA_INPUT = "input"; //"generate" or "read"
    public static final String ARGS_PARALLELISM_VALUE = "parallelism";

Kmeans Example implementation details

- The entry point for the K-Means clustering algorithm is implemented in KMeansJobMain
  - It retrieves and parses the command line parameters submitted
  - Sets the submitted variables in the Configuration object
  - Put the object into the JobConfig and submit it to KMeansJob class
- KMeansJob
  - main class for the K-Means clustering which has the following classes namely KMeansSource, KMeansAllReduceTask, and CentroidAggregator
  - execute method in KMeansJob invokes the KMeansDataGenerator to generate the datapoints file and centroid file, if the user has specified the option ARGS_DATA_INPUT as "generate"
  - it will invoke the KMeansDataGenerator class and store the generated datapoints and centroids in each worker (locally) or in the distributed file system which is based on the option ARGS_FILESYSTEM as "local" or "hdfs"
  - it will invoke the KMeansFileReader to read the input datafile/centroid file either from locally or HDFS

Kmeans Example implementation details

- KMeansJob (cont.)
  - the datapoints are stored in DataSet (0th object) and centroids are stored in DataSet (1st object) and call the executor as given below;
    - taskExecutor.addInput(graph, plan, "source", "points", datapoints);
    - taskExecutor.addInput(graph, plan, "source", "centroids", centroids);
    - taskExecutor.execute(graph, plan);
  - This process repeats for ‘N’ number of iterations as specified in the KMeansConstants
  - For every iteration, the new centroid value is calculated and the calculated value is distributed across all the task instances.
    - DataSet<Object> dataSet = taskExecutor.getOutput(graph, plan, "sink");
      Set<Object> values = dataSet.getData();
      for (Object value : values) {
       KMeansCenters kMeansCenters = (KMeansCenters) value;
       centroid = kMeansCenters.getCenters();
      }
  - At the end of every iteration, the centroid value is updated and the iteration continues with the new centroid value.
    - datapoints.addPartition(0, dataPoint);
      centroids.addPartition(1, centroid);

Kmeans Example implementation details

- KMeansSourceTask
  - retrieve the input data file and centroid file name,
  - first calculate the start index and end index which is based on the total data points and the parallelism value as given below:
    - int startIndex = context.taskIndex() * datapoints.length / context.getParallelism();
      int endIndex = startIndex + datapoints.length / context.getParallelism();
  - Then it calls the KMeansCalculator class to calculate and get the centroid value for the task instance.
    - kMeansCalculator = new KMeansCalculator(datapoints, centroid,
      context.taskIndex(), 2, startIndex, endIndex);
      KMeansCenters kMeansCenters = kMeansCalculator.calculate();
  - Finally, each task instance write their calculated centroids value as given below:
    - context.writeEnd("all-reduce", kMeansCenters);

Kmeans Example implementation details

- KMeansAllReduce Task
  - retrieves the calculated centroid value in the execute method
    - public boolean execute(IMessage message) {
      centroids = ((KMeansCenters) message.getContent()).getCenters();
      }
  - write the calculated centroid value without the number of datapoints fall into the particular cluster as given below:
    - @Override
      public Partition<Object> get() {
      return new Partition<>(context.taskIndex(), new KMeansCenters().setCenters(newCentroids));
      }
- CentroidAggregator
  - implements the IFunction and the function OnMessage which accepts two objects as an argument.
    - public Object onMessage(Object object1, Object object2)
  - It sums the corresponding centroid values and return the same.
    - ret.setCenters(newCentroids);