REPAST SIMPHONY BATCH RUNS USING HADOOP

MICHAEL HENRY AND DON BENNETT - MITRE, NICK COLLIER - REPAST DEVELOPMENT TEAM

0. Before We Get Started

Before we can do anything with Repast Simphony, we need to make sure that we have a proper installation of the latest version. See the Repast Requirements Web Page for instructions on downloading and installing Repast Simphony and Java.[1]

This document assumes working knowledge of the Repast Simphony batch run mechanism as well as knowledge of Hadoop. More info on the batch run mechanism can be found in the Repast Batch Run Getting Started Guide.[2] More information on Hadoop can be found at the Hadoop Website.[3] The example code and the documentation on which this documents is based were contributed to Repast Simphony by Mitre under a BSD style license and we thank them for their contribution. The project was developed by the MITRE Corporation under FAA Mission-Oriented Investigation and Experimentation.

1. Repast Simphony Batch Runs Using Hadoop

The purpose of this document is to explain how to enable Repast Simphony simulations to run on a Hadoop cluster as a map / reduce job. The simulations themselves are run as mappers and the output produced by each simulation is passed to the reduce via a write on a Hadoop Context. The companion code to this document is an example of the Java Zombies model adapted to work with Hadoop. That Repast Simphony project should be contained in the zip file that accompanies this document. The example source is divided into two packages: zombies and hadoop. The zombies package includes the model as well as any model specific hadoop code (e.g. the JZombies specific Hadoop reducer). The hadoop package includes code that can be used for any model and adapts the archive created by Repast Simphony batch for use with Hadoop.

The process of configuring a Repast Simphony model to run on Hadoop is two-fold:

(1) Adapting your model to work with Hadoop by converting data output to key value pairs and writing a Hadoop Reducer to process that data in some appropriate way.
(2) Adapting the archive produced by Repast Simphony's batch run mechanism to work with Hadoop.

[1] https://repast.github.io/requirements.html
[2] https://repast.github.io/docs/RepastBatchRunsGettingStarted.pdf
[3] https://hadoop.apache.org

1.1. **Adapting Your Model to Work with Hadoop.** The first step to performing batch runs on Hadoop is to adapt your code to work with Hadoop.

(1) Create a custom `DataSink` and register it with Repast's Data Set recording mechanism. This `DataSink` will transform your output into key / value pairs and pass those to Hadoop. You register a `DataSink` using a `ModelInitializer`. The `HadoopDataSink` in the companion code is an example of such a `ModelInitilializer`. The `ModelInitializer` must also be specified in the scenario.xml in your model's scenario folder in order for it to be run when the scenario is loaded. See Listing 1 for an example from the Java Zombies scenario.

```xml
<Scenario>
...
<model.initializer class="hadoop.HadoopDataSink" />
</Scenario>
```

Listing 1. Specifying the ModelInitializer in scenario.xml

A `ModelInitilializer` runs when a scenario is loaded. Its initialize method typically adds a controller action to the scenario that will be executed whenever a run or batch is initialized. When a DataSink is to be added, this is done in the batchInitialize method. See Listing 2. The `DataSink` itself is added by finding by name the DataSetBuilder for the data set that will produce the data you want to send to the reduce. You then add a `DataSink` implementation to that builder. The example `DataSink` in `HadoopDataSink` example code should work for any project. There, the data added via the append method method is appended to a StringBuilder as key / value pairs. The key and value are delimited by ':' and each pair by a ','. The collect method takes the String produced by the StringBuilder, extracts the "run" key / value pair and writes a new key value to Hadoop where the key is the run number and the value is all the key value pairs (see Listing 3).

```
@Override
public void initialize(Scenario scenario, RunEnvironmentBuilder builder) {
scenario.addMasterControllerAction(new NullAbstractControllerAction() {
   @Override
   public void batchInitialize(RunState runState, Object contextId) {
      DataSetRegistry registry =
         (DataSetRegistry)runState.
         getFromRegistry(DataConstants.REGISTRY_KEY);
      DataSetManager manager = registry.getDataSetManager(contextId);
      DataSetBuilder<?> builder = manager.getDataSetBuilder("Agent Counts");

      builder.addDataSink(new DataSink() {
         ...
```

Listing 2. Adding the Controller Action with a ModelInitializer

```
new DataSink() {

   private Context collector = COLLECTOR;
   private StringBuilder entry;

   @Override
   public void rowStarted() {
      if(entry == null){
         entry = new StringBuilder();
      }else{
         collect(entry.toString());
         entry = new StringBuilder();
      }
   }

   @Override
   public void rowEnded() {
      entry.replace(entry.length() - 1, entry.length(), "");

   }
   ...

   @Override
   public void append(String key, Object value) {
      entry.append(key + ":" + value + ",");
   }

   // collects run number -> output
   @SuppressWarnings("unchecked")
   private void collect(String key){
      String[] fields = key.split("\\,");
      for(String field : fields){
         if(field.startsWith("run")){
            int run = (int) Double.parseDouble(field.split("\\:")[1]);
            try {
               collector.write(new Text(run + ""), new Text(key));
            } catch (Exception e) {
               e.printStackTrace();
            }
         }
      }
   }
}
```

Listing 3. Collecting the Data with a Custom DataSink

(2) The next step is to add a Reducer class that extends Hadoop's Reducer<Text, Text, Text, Text>. This class will be responsible for recording all output from the simulation from the fields collected in the DataSink. An example is provided for the JZombies project in `HadoopZombies`. Exactly what happens in the reducer will be model dependent, but it will typically iterate over the values passed from the DataSink, splitting them into their constituent key value pairs and do something with those key value pairs. In the example case, the reducer iterates through all the key values pairs to determine the last tick in which a Human was alive and writes out that value.

(3) The final step is to create a class with main method that calls `MapReduceJob.run()`, passing that the number of mappers to use (or 0 for automatic config), the name of the job on the cluster and the name of the reducer class. In the JZombies example, the main method is in `HadoopZombies` where the reducer is also defined but it need not be. The `MapReduceJob` class is part of the hadoop package in the example code and performs the typical Hadoop job configuration as well as providing a Mapper that will run Repast Simphony simulations in batch mode.

1.2. **Adapting Repast Simphony's Batch Archive to Work with Hadoop.** Once you've modified your code to work with Hadoop as described in the steps above, you need to adapt the jar archive created by Repast Simphony's batch run mechanism so that it can perform the batch runs as mappers on the Hadoop cluster.

(1) Generate the complete_model.jar etc. and as described in the Repast Batch Run Getting Started Guide.

(2) Run `HadoopUtils.makeHadoopJar`, passing it the main class from step 3 in the previous section, the path to the complete_model.jar from step 1 of this section and the path of a jar file that will be created. This new jar file can then be used to perform the actual model runs on a Hadoop cluster. The `HadoopUtils` class is in the hadoop package in the sample code.

1.3. **Running the Hadoop Job.**

(1) Transfer the jar created in step 2 of the previous section to a machine that has access to the hadoop commands (e.g. hadoop -fs etc.)

(2) Run this jar as a normal java processes (i.e. java -jar MyHadoopJar.jar)

(3) After the process completes, there will be a folder named "output" within the jar that was executed. This folder will contain the Repast Simulation output from the data sinks.