A MapReduce Implementation in SCS

Sand L. Corrêa, Renato F. G. Cerqueira,
Departamento de Informática
Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio)
scorrea@inf.puc-rio.br,rcerq@inf.puc-rio.br

1 MapReduce

MapReduce is a library to support parallel computation over large data set on unreliable clusters of computers. MapReduce is based on two concepts from functional languages to express data-intensive algorithms: map and reduce functions. The Map function processes the input data and generates a set of intermediate <*key,value*> pairs. The Reduce function, then, merges the intermediate pairs that have the same key. An overall MapReduce application's dataflow is shown in Figure 1. In the map phase, the user input data is split into smaller pieces and assigned to workers (copies that execute on a cluster of machines). These workers, then, execute user-defined map functions to produce the intermediate <*key,value*> pairs. Also during this phase, partitions and sort functions are executed to, respectively, distribute the pairs around the key and sort them, so that occurrences with the same key are kept together. In the reduce phase, workers iterates over the sorted data executing user-defined reduce functions. Those functions produce one or more outputs, which can be merged to produce a single output.

This document describes a MapReduce implementation using the SCS system. The description is organized in five sections. Section 2 presents the basic MapReduce interfaces. Section 3 presents the job schedule interfaces. Section 4 discurss how to build MapReduce Applications using our framework. Section 5 outline some steps to run a demo application. For more information on MapReduce operations please refers to [1].

2 MapReduce API Description

In this section, we describe the interfaces we use to implement the MapReduce framework.

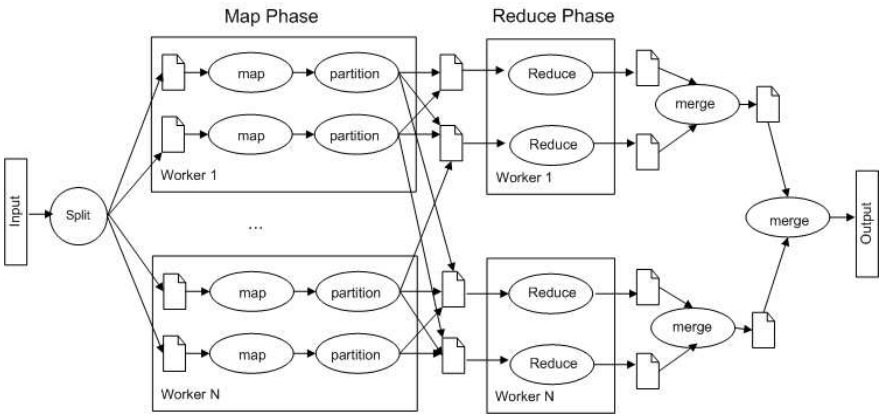

Figure 1: A MapReduce application dataflow

`TaskStatus`

This is an enumeration that describes the possibles status for a MapReduce task, that is: `MAP`, `REDUCE`, `ERROR` or `END`.

`Reporter`

`Reporter` is a facility for MapReduce applications to report progress and set application-level status messages. The `Reporter` interface defines the following methods:

- `boolean open()`: this method opens the report file
- `void report (in long level, in string message)`: this method is used to write progress or report messages. The arguments `level` and `message` denote the level message ("error", "warning", etc) and the message itself, respectively.
- `void close()`: this method is used to close the report file.

`FileSplit`

The `FileSplit` interface represents a split or partition of the input file. It defines the following methods:

- `string getPath()`: this method returns the complete path of the file associated to the split.
- `long long getLength()`: this method returns the length of the file associated to the split.

RecordReader

The RecordReader interface defines methods to read $< key, value >$ pairs. It reads an unformatted file (represented by a `FileSplit` object) and returns its content as a $< key, value >$ pair. Typically the `RecordReader` converts the byte-oriented view of the input, provided by the `FileSplit` object, and presents a record-oriented view for processing. RecordReader thus assumes the responsibility of processing record boundaries and presents the tasks with keys and values. It defines the following methods:

- `void open(in string confFileName, in FileSplit fileSplit, in Reporter reporter)`: opens the input file from which the content will be read. It take as input the name of the configuration file (where some important properties are defined), the `FileSplit` associated to the input file and the `Reporter` object to report progress or errors.
- `void close()`: closes the input file.
- `boolean next (out any key, out any value)`: returns the next $< key, value >$ pair from the input file.
- `FileSplit getFileSplit()`: returns the FileSplit associated to the `RecordReader`.

RecordWriter

The `RecordWriter` interface defines method to write $< key, value >$ pairs to an output file. Typically, RecordWriter implementations write the outputs to the FileSystem. It defines the following methods:

- `void open(in string confFileName, in FileSplit fileSplit, in Reporter reporter)`: opens the output file to which the pairs will be written to. It take as input the name of the configuration file, the `FileSplit` associated to the output file and the `Reporter` object to report progress or errors.
- `boolean write (in any key, in any value)`: writes the $< key, value >$ pair to the output file. It take as input the key and value values.
- `void close()` : closes the output file.
- `FileSplit getFileSplit()`: returns the `FileSplit` associated to the `RecordWriter`.

IOFormat

The `IOFormat` interface defines operations to describes the input and output specification for a MapReduce process.

- `RecordReader getRecordReader(in TaskStatus status)`: returns a `RecordReader` object to be used in a map or reduce operation. It takes as input the operation status (`MAP` or `REDUCE`)
- `RecordWriter getRecordWriter(in TaskStatus status)`: returns the `RecordWriter` object to be used in a map or reduce operation. It takes as input the operation status (`MAP` or `REDUCE`).
- `FileSplits getSplits(in string confFileName, in Reporter reporter)`: splits the original input file for the MapReduce process. Returns the splits.

OutputCollector

The `OutputCollector` is a facility provided by the MapReduce framework to collect data output by the Mapper or the Reducer (either the intermediate outputs or the output of the job). It defines the following methods:

- `void collect (in any key, in any value)`: collects a $< key, value >$ pair.
- `void flush()`: flushes the output buffer where the pairs are stored.
- `void close()`: close the output buffer.

Mapper

The `Mapper` interface maps input $< key, value >$ pairs to a set of intermediate $< key, value >$ pairs. It defines the following method:

- `void map(in any key, in any value, in OutputCollector collector, in Reporter reporter)`: maps a single input $< key, value >$ pair into an intermediate $< key, value >$ pair. It takes as input the `key` and `value` values, the `OutputCollector` object to collect mapped keys and value and the `Reporter` object.

`Reducer`

The `Reducer` interface reduces a set of intermediate values which share a key to a smaller set of values. It defines the following method:

- `void reduce (in any key, in Iterator values, in OutputCollector collector, in Reporter reporter)`: reduces values for a given key. It takes as input the `key` values, `Iterator` - a set of `values` to be reduced - the `OutputCollector` object to collect reduced key and combined values and the `Reporter` object.

`Partitioner`

The `Partitioner` partitions the key space. It defines the following method:

- `long getPartition(in any key, in any value, in long numPartitions)`: gets the partition number for a given key (hence record) given the total number of partitions i.e. partitions the key space.

2.1 `Task`

The `Task` interface represents a task in the MapReduce process. Each task is identified by an id, receives an input and produces an output. It defines de following methods:

- `long getId()`: returns the id of the task.
- `void setStatus(in TaskStatus status)`: sets the status of the task.;
- `TaskStatus getStatus()`: returns the status of the task.
- `FileSplits getInput()`: returns the inputs of the task.
- `FileSplits getOutput()`: return the outputs of the task.
- `void run()`: executes the task.

;

3 Job Schedule API Description

Figure 2 shows our implementation of a MapReduce framework using SCS components. Broadly the framework comprises two types of component: master and worker. The master is a special component that controls the entire dataflow, picking idle workers and assigning them a map or reduce task. The master executes within its own component container, while worker components execute user-defined map or reduce functions and are instantiated in different execution nodes, in individual containers. Master assigns tasks to workers, invoking an *execute* method (defined in the *Worker* interface) and passing as parameters the task and a SCS event channel. The later is used by the worker to notify the master that the task has been completed. Next, we describe the mater and worker interfaces.

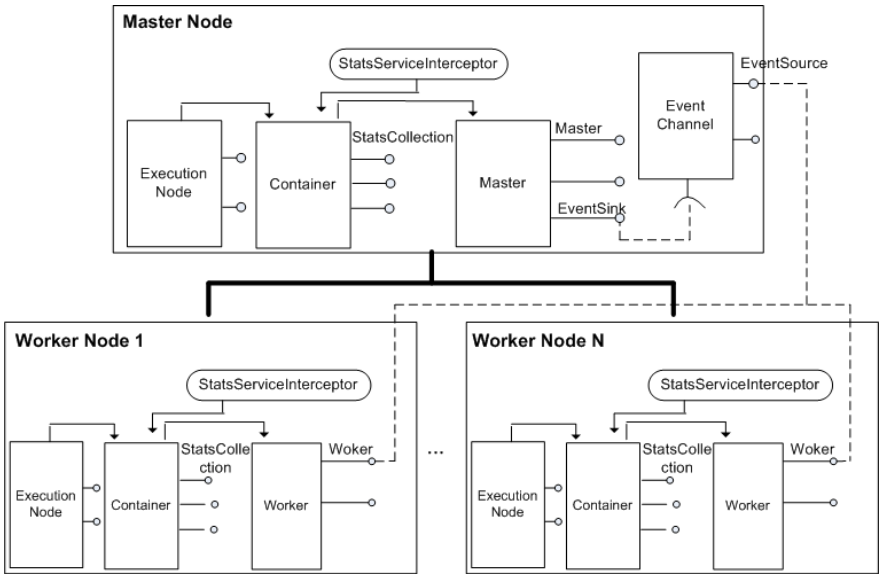

Figure 2: MapReduce using SCS components

`Master`

Defines the following method:

- `void start(in string confFileName, in Reporter reporter)`: initializes the MapReduce process. It takes as input the configuration file and the `Reporter` object.

`Worker`

Defines the following methods:

- `boolean ping()`: used to test if a worker is running.
- `void execute (in core::IComponent channel, in Task task)`: executes the task passed as argument. Uses the channel argument -an event channel - to notify the status of the execution.

4 Implementing a MapReduce Application

Our MapReduce framework provides implementation for most of the interfaces defined above. In particular, the programmer must implement the following interfaces:

- `RecordReader`
- `RecordWriter`
- `Mapper`
- `Reducer`

In addition to implementing these interfaces, the programmer must also extend the `IOFormatServant` class. This class is provided as an abstract class that provides implementation for some methods from the IOFormat interface. In particular, the following methods must be implemented:

- `RecordReader doGetRecordReader(TaskStatus status)`: this method should return the `RecordReader` for a given task.
- `RecordWriter doGetRecordWriter(TaskStatus status)`: this method should return the `RecordWriter` for a given task.

5 Configuration File

In order to run a MapReduce application, it is necessary to provide a configuration file containing some important properties to the execution. In the SCS package, the configuration file is found at directory scripts/execute and is denoted by mapReduce.properties. In the script, all properties are properly commented.

The SCS package also provides a MapReduce demo, a Word Counter Application. In order to run this application follows the steps:

1. edit file mapReduce.properties and set the configuration of the execution (providing the nodes where master and worker will run, the input file, number of mappers and reducers required, etc.)

2. run the Execution Node application in each node required in the execution

3. run the application. Considering the project directory (src/java), the command is:

   - `java scs.demos.mapreduce.app.MapReduceApp ../../scripts/execute/mapReduce.properties`

4. the progress of the application can be watched listing src/java/reporter.debug

5. the results of the MapReduce process is written to directory src/java/scs/demos/mapreduce/database. Those results are labeled with the pattern "mapped" or "reduced"

References

[1] J. Dean and S. Ghemawat, “Mapreduce: Simplified data processing on large clusters,” in *Proceedings of OSDI 2004*, (Berkeley, USA), pp. 137–150, USENIX Association, 2004.