MapReduce Framework Writeup
Nick Zukoski (nmz)

1) Design:

Configuration:

All configuration constants are located in Configurations/Configurations.java. This means that to reconfigure, the framework must be recompiled. Included in this are parameters about the number of records to put to a map or reduce, the number of tasks to have running on a node at a given time, the address and port of all of participants and of the master as well as timeout information.

Master/Participant Setup:

My MapReduce framework is designed with a single Master coordinating process that handles all allocation of work. A new job can be requested on any node but if it is requested on a participant, it is simply bundled up and sent to the master. Once the master has a new job, it divides it into map tasks and begins distributing them to the participants.

Load handling is done by the Master process by only doling out a certain number of tasks (*MAX_TASKS_PER_NODE* in the configuration) to each node and waiting to hear back that they have finished one before sending another out. In this way, the need for migration is limited only to node failure.

Mapping/Reducing and records:

My MapReduce framework deals exclusively with fixed length records. Both keys and values in records are stored and passed into client code as Strings. The record size is set by the job and includes space for both the key and the value (seperated by the string "::"). When a new job is requested, the framework splits the mapping into Map tasks that each deal with *RECORDS_PER_MAP* records. On the participants, these records are read from their partition, mapped, and then written out to a temporary file.

Once all of the maps have completed, a round of reduces is drawn up by the master. Each reduce is assigned whole map output files until its number of records is over *RECORDS_PER_REDUCE*. The tasks are then sent to the participants. On the participants, the input files are all read in, the values are all sorted and then reduced using the jobs reduce function and the results are written back out to another temporary file.

If a reduce round finishes and there is only one file left, then the job is done and that file is copied to the desired output file and then all of the temporary files are cleand up.

Failure:

The Master node keeps track of which nodes seem to have failed and when a new Node fails, it is added to that list. Also, any jobs that were on that node at that time are added back to the task queue to be sent out to other participants. Currently the only failure mode that is checked is when the master is unsuccessful in sending a task or request to a participant. This means that it is possible for a node to fail and if the master does not send another task to that node, the master will wait forever for a response.

2) Application Interface

To write a job that can be executed using this framework, the application programmer must create a class that conforms to the MapReduceJob interface (located in Util). This interface specifies 4 important functions and a few getters and setters for keeping track of input and output files at runtime. The Map function takes in one key and one value and returns a List of key value pairs that are the output of the map. The reduce identity function returns a string representing the reduce identity that is passed in at the start of a reduce. The reduce function takes in a single key, a list of values corresponding to that key, and an initial value. The initial value is either the Reduce identity or the result of a previous reduce on the same key. The Reduce function must return a single string value corresponding to the key passed in. Finally, MapReduceJob specifes that the application programmer must decide how big each record is with the getRecordSize() function. If the programmer underestimates, then records will be cut short at the given size.

3) System Administrator Interface

To get the framework up and going, first the sysadmin should put in appropriate values to Configuration/Configuration.java. Once well configured, on each of the participants as specified in the config, Slave/SlaveController should be run and on the single master as specified in the config, Master/MasterController should be run. Now, to start a job use the command start from any node:

start (Job class name) (output file name) (input files...)

To stop the whole system, simply use the command

quit

To list all of the jobs and their information, use

list

from the master node. If list is used on a participant, it will show information about the currently running tasks on that participant.

6) Examples

Two examples have been provided in the Examples folder.
WordCount: which takes in a large corpus of text and returns the number of times each word occurred in the corpus.
LargestWord: which takes in a large corpus of text and returns the single largest word in it.

A helper class has also been provided to encode and decode records files.
WordCountFileCreator takes in 3 arguments, (encode/decode) (outputFile) (inputFile)
if encode is given, it encodes the input file into a record file that either WordCount or

LargestWord can use (though it draws its record size only from WordCount, so if LargestWord's record size is changed, it will be incompatible).

if decode is given, the input file is output as a file containing (key):(value) lines.

Provided are 4 files containing excerpts from Jane Austin's Pride and Prejudice and James Joyce's Ulysses both of which were obtained from Project Gutenberg (http://www.gutenberg.org/)

The text of these files are in largeWC_1.txt, largeWC_2.txt, largeText_1.txt, and largeText_2.txt and all of them have been put into record files with a .rec extension as well for convenience.

An example of how to run WordCount is shown:

start Examples.WordCount fullWordCount.rec Examples/largeText_1.rec

This takes in largeText_1.rec and outputs the result to fullWordCount.rec

To convert this .rec file back to a human readable file, use the command

java WordCountFileCreator decode fullWordCount.txt fullWordCount.rec

5) Things that would have been nice

- A better admin system to allow for stopping and restarting of jobs as well as getting information from any node.

-A better failure detection system to notice silent node failures.

-A record size that involves both a fixed key and value record size (or variable size) such that no special characters are needed and as such don't have to be excluded from keys.

-A better failure recovery system that is able to recover from node failures more consistently

-More creativity in my example jobs

P.S.
This was a really cool project, I had a lot of fun making and thinking about it!