Distributed Systems.
Project 3: Map-Reduce Framework

Ankit Chheda - achheda
Vinay Balaji - vbalaji

Design:

Configuration:

All configuration constants, with exception of location of workers, can be defined in properties.txt. This means that to reconfigure, the framework need not be recompiled. Included in this are parameters about the the address and port of the master. It holds the information about the file system configuration like the number of splits/file and the replication factor. The locations of the workers are stored in worker-config.txt.

Map/Reduce Workflow:

Client -> worker/master->maps->combines->Reduces

Whenever a client submits a job on any of the nodes, it automatically goes to the master node. The master node starts the maps on the workers and waits for all the maps on a node to be finished. When all the maps for a particular job on a specific worker are done, that worker goes through a combine phase. The combine phase reads in the outputs from all maps and merge sorts them into a file. We then perform a local reduce on this file and hash the output into chunks for each global reduce task. The outputs of the map and combine operations are stored locally on each worker.

The global reduce(s) reads in its assigned chunks from each worker and then merge sorts them into a single file. This data is then reduced using the job programmer's reduce method. The output of each reduce contains a unique key-value pair which is not present on any other reducer. The reducer writes this key-value output pair to the DFS.

Once the job is done, we delete all the intermediate files that are associated with the job from the local file system, hence keeping it clean.

The application programmer can then read the output files from the DFS.

System Administrator Interface

To get the framework up and going, first the sysadmin should put in appropriate values to properties.txt and worker-config.txt. As a feature of our project, the scripts that start our processes automatically detect any outdated files and freshly compiles the framework on its own, so no manual compilation / recompilation is necessary. The Makefile itself does not need to be updated either, as this is also updated automatically.

Workers:

To startup, all workers have to be started on the worker machines as:

./scripts/run.sh Worker <port-number>

This does the following:
- creates the Distributed File System directory.
- initializes the local work monitor.
- initializes the listener which listens to the master for commands.
- starts the heartbeat monitor.

Master:

After starting all the workers, we move on to starting the master. The master has to be started on the machine specified in the configuration file. The master can be started using:

./scripts/run.sh Master

Starting the master leads to the following:
- Finds the workers specified in the config file and connects to them.
- Start the bootstrap process which distributes the files from the directory specified in properties.txt.
- Starts the heartbeat.
- Starts the scheduler.
- Starts listener for Worker tasks.
- Starts the listener for the client.
- Starts the shell for Job Management.

When the master has started and detected the worker, it automatically bootstraps and distributes the files on the DFS as per the parameters specified in the properties.txt. On receiving a job from the client, it processes it and distributes work amongst workers to get the job done!

Client:

Once all the workers and the master are up, you can run a job by invoking a client from any node. Client can be started by,

./scripts/run.sh <name of client main class>

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. Whenever a map fails, the master has the scheduler re-schedule the job. If a reduce fails, the master will schedule another reduce on a worker node.

API:

Clients for individual map reduce tasks have to be coded by the Application programmer defining 2 classes which implement the interfaces MapTask and ReduceTask and Serializable.

MAP:

For the Map, the programmer is intended to create a Map class implementing the MapTask and Serializable interfaces. The map class will override the map()

map() takes in one record line as the input and gives out a list of key-value pair as its output. The input string can be specified to be parsed by the programmer and the delimiter can be specified while defining the MapReduce object which will be covered later.

public List<Pair<String, String>> map(String line);
Input- String line
Output- List<Pair<String key, String value>>

Reduce:

For the Reduce, the programmer is intended to create a Reduce class implementing the ReduceTask and Serializable Interfaces. The reduce class will override the reduce()

reduce() takes in a String and an Iterator as its input and produces a Pair<String, String> at the output. The input string is a Key and the input Iterator is an iterator for all the values corresponding to that key.

public Pair<String, String> reduce(String key, Iterator<String> values);
Input - String key, Iterator<String values>
Output Pair<String Key, String Value>

Herein, the application programmer defines the map() and reduce() methods in their respective classes.

When these methods have been appropriately defined, the application programmer should create a map-reduce object in the following manner:

MapReduce <varname> = new MapReduce (new MapTaskImpl(), new ReduceTaskImpl(), int suggestedNumberOfReducers, List<File> inputFiles, String delim, String resultName, String jobName);

As the object is created, the last thing that needs to be done is to call the mapreduce on this object, as:

varname.mapreduce();

Example Programs:
Two examples have been provided in the submission.
WordCount: Takes in the list of files and gives the number of occurrences of a word in that file.
WordLength: Takes in a list of files, and groups the words on the basis of the word length. The output has a list of words of the same length.

What Works:

Bootstrapping (initial file distribution)
Management tool (command prompt on Master node)
Submission of tasks to the Master (not any other nodes)
Execution of Map, Combine, and Reduce phases
Cleanup of working directory for each worker

What could have been better:

Writing results to DFS
Handling multiple jobs concurrently with no issues
Using RMI instead of bare Socket programming
Spending more time on fully designing project before implementing
Resolving concurrency issues related to state-keeping tasks on master / workers