MapReduce Red

Michael Hansen (mhansen1) & Michael Rosen (mrrosen)
April 14, 2013

MapReduce Design Overview:

Our mapReduce library is based around a basic client-server architecture, jobs can be added using the given JobMRR class, and they will be shipped first to the local participant, and then to the master with the next heartbeat. The master then splits the job into tasks according to how many processors it has access to and places the tasks into a global work queue. Each participant then pulls tasks from the queue according to how many processors it has access to and adds them to it's local queue. The participants follow a similar structure with worker threads pulling tasks from the participant queue and performing the mapreduce operations. We don't support explicit requests for just a map or just a reduce, but those can still be executed by passing an identity function for one of the steps.

The completed tasks are then written to intermediate files and the task id is sent back to the master. The master then removes the ids from its participant object and places the completed tasks in another global pool. This pool is searched through and adjacent tasks are merged and sent back out for reducing.

After all the tasks are complete (there are no running tasks, and only one completed task), the single remaining file is renamed as the output and the client is notified of the job's completion.

We do not write to a client-defined output, but we do provide methods to get the TreeMap we use out. We debated about going this route as opposed to appending to a user-defined file, and determined that this was a superior route, as the other solution demanded that the user write a function to convert the result pairs into whatever format they want it written in. This route gives the user more freedom, as they can take our structure and easily write it in their preferred format, or use it as-is in Java.

System Admin View:

MapReduce Red provides a clean, simple interface for running the MapReduce facility. Running the Server and Clients is straightforward and once these are running, the system can preform MapReduce jobs from any nodes.

RUNNING THE SERVER:

To run the server, compile and run the ServerMRR Java class on the master machine. This class takes only a single argument, the configuration file that will define the parameters of the system. Parameters take the form "[parameter]=[value]", with a single parameter per line. Below are the list of these parameters:

PARTICIPANTS :

A comma delimited list of participant nodes in the system, in "host:port". Only these nodes are accepted by the server, so any node that is not on this list can be a member of the mappers and reducers, nor can jobs be submitted from non-listed nodes.

USERNAME :

If REMOTE_START is enabled, a username is needed to initiate new nodes on the remote machines. This user must have adequate permissions to begin need JVMs on remote machines via ssh.

LISTEN_PORT :

The port on which the server will listen to incoming messages from the participants.

RETRIES :

The number of non-response status updates the server will accept from a participant before declaring the participant dead and reallocating the work from the participant elsewhere. Should a network failure occur for a short time, the participants may continue to work and reconnect with the server once the network is restored. This value should be kept relatively low as if a participant dies, reallocation of work will not occur until this count has expired.

LOCAL_PORT :

The port on which new jobs are submitted to the the system from applications. Applications should have prior knowledge of this port for the specific system (unless applications are allowed to read the configuration file).

REMOTE_START : (leave this off, we didn't have time to properly test it)

Whether the server will attempt to remotely start dead participants on the participant list (values either ON or OFF). This functionality requires an external script (ssh_work) which takes the following arguments:

username - The user to login to via ssh
hostname - The host to which to ssh to
path - The location of the ClientMRR class
master_host - The host of the server
master_port - The port on which the server listens for participant connections
local_listen_port - The port on which the participant will listen for job requests

This functionality is on by default.

TIMEOUT :

The number of seconds (+ 1) that the server will wait for communication from participants before declaring them dead (ie decrementing RETRIES).

Once initiated, the server will attempt to connect to the participants periodically for status updates and to distribute more work. The server also provides a basic interface for monitoring the status of the system. These are the commands for the user interface:

jobs

Prints a list of currently executing jobs.

stop [jobID]

Stops the job given by jobID and sends a ServerTerminationException back to the application.

power

Prints the current power of the system (total workers under all the participants).

quit

Terminates the server.

RUNNING THE CLIENT:

The clients(participants) are also easy to run, simply run the ClientMRR class with the arguments master_host, master_port and local_listen_port in that order. The master_host and master_port are the hostname and port on which the server is running and listening for participants. The local_listen_port is the port on which the participant shall listen for new job requests from applications.

Application Programmers View:

An application programmers writing for MapReduce Red is required to have prior knowledge of the port on which the system listens or access to a file provided by the system administrator which indicates this port. The system does not currently support any internal mechanism for getting this port (as the system administrator may wish to keep the internal files ie the configuration file, private). With this port, an application programmer needs to understand and utilize the two classes below to execute jobs on this system. Note the result of a MapReduce is a TreeMap structure read in by the provided methods. The application programmer is responsible to convert this into the desired output form if another structure is desired (or if a specific file output form is desired). Also note that the data records provided must be fixed length (in bytes) to allow for the proper partitioning and reading of records.

Below are the important classes:

JobMRR:

Description:

This class is used for job (map and reduce) execution and status. In order to successfully use this class, the MapReduce system must be running and a participant for this system must be running on the same node. To use this class to submit a job, a class extending the abstract ConfigurationMRR class must be prodived to configure the job. An option jobName may also be provided, but if left out, a default name will be given to the job based on the records and files on which the job runs. The needed fields and functions for the ConfigurationMRR class are provided below.

Parameters: (Note that these must be the same types as used in ConfigurationMRR!)

`MAPIN (Must be serializable)`

The class used to store read in records.

`REDKEY (Must be serializable)`

The key class for reduces.

`REDVAL (Must be serializable)`

The value class for reduces.

Constructors:

`public JobMRR(ConfigurationMRR<MAPIN, REDKEY, REDVAL> config)`

Constructs a new JobMRR class with the provided ConfigurationMRR object.

`public JobMRR(ConfigurationMRR<MAPIN, REDKEY, REDVAL> config, String name)`

Constructs a new JobMRR class with the provided ConfigurationMRR object and job name given by `name`.

Methods:

`public void submit()`

Submits a job to the MapReduce system with the configuration and name (if given) specified by the constructor. This method does not block and returns immediately, even if the job submission was unsuccessful. The `isComplete()`, `encountedException()` and `getException()` methods can be used to test if the job submission was successful.

`public boolean isComplete()`

Tests whether the job has been completed.

`public boolean encounteredException()`

Tests whether the job had an execption while executing. The specific exception can be accessed via the `getException()` method.

`public Exception getException()`

Returns the exception encountered by the job during execution. Returns `null` if no exceptions were encountered.

`public void waitOnJob() throws InterruptedException`

Blocks execution of the application until the job is complete. Throws `InterruptedException` if the wait is interrupted by another thread. This method can stall indefinitely if the MapReduce system crashes during execution of the job, so it is recommended an application use the timeout version of this method if indefinite stall upon uncompleted MapReduce is unacceptable.

`public void waitOnJob(long timeout) throws InterruptedException`

Blocks execution of the application until the job is complete for `timeout` milliseconds.. Throws `InterruptedException` if the wait is interrupted by another thread.

```
public TreeMap<REDKEY, REDVAL> readFile() throws IOException, ClassNotFoundException
```

Returns the final result of the MapReduce job has a TreeMap structure. Throws `IOException` if the file cannot be read. Throws `ClassNotFoundException` if the classes of the key, value or TreeMap cannot be resolved.

ConfigurationMRR:

Description:

This abstract class is used to configure a MapReduce job using a variety of fields and methods. This abstract class also have three type parameters, `MAPIN`, `REDKEY` and `REDVAL`, which are used to type the results of the maps and reduces. Note that the `REDKEY` must be a comparable class (implement interface `Comparable`) This class also includes unimplemented three abstract methods, described below. In order for a job to successfully execute, these methods and fields must be initialized and implemented.

Parameters:

`MAPIN`

The class used to store read in records.

`REDKEY`

The key class for reduces.

`REDVAL`

The value class for reduces.

Fields:

`int recordSize`

The size, in bytes, of an individual record.

`int start`

The record on which to begin the MapReduce, inclusive.

`int end`

The record on which to end the MapReduce, exclusive.

`String inFile`

The file from which to read records.

`String outFile`

The file which to write the resulting structure.

`int listenBackPort`

The port on which to listen for the signal that a job has been completed.

`int participantPort`

The port on which to contact the local participant to submit new jobs. This is provided by the system administrator.

Methods:

`abstract public MAPIN readRecord(byte[] record)`

This method translates a byte array representation of a record into a parameterized type `MAPIN` to be used by the `map` method.

`abstract public ArrayList<Pair<REDKEY, REDVAL>> map(MAPIN mapin)`

This method maps `MAPIN` to an array of key, value pairs in a `Pair` class structure. (The "Map" in MapReduce)

`abstract public REDVAL reduce(REDVAL val1, REDVAL val2)`

This method reduces `REDVAL` pairs from the same key into a single value. (The "Reduce" in MapReduce)

Pair:

Description:

This class forms a wrapper for key, value pairs used in the map step of the MapReduce.

Parameters:

`K`

The key class.

`V`

The value class

Fields:

`K key`

The key value.

`V value`

The value associated with the given key.

Constructors:

`public Pair(K key, V value)`

Constructs a new pair object with the key and associated value given.

Unimplemented Features // Limitations:

Remote start: we worked on a method to use shell scripts to remotely start participant processes, and also to bring them back in case of failure, but unfortunately it had some bugs, and by the time we ironed out all the other bugs in the project, we didn't have time to properly debug this feature. So for now participants must be started manually.

Sometimes the sockets seem to have race conditions, usually when the clients are started before the server has properly started its main loop, also it may just be my account, but my putty sessions occasionally froze. In both these situations, restarting the offending client solves the issue, and yes, our master is smart enough to not lose any tasks no matter what happens to the participants.

User error: Yes, we are kind of vulnerable to issues in the user-defined functions, we don't have a timeout, so infinite loops will kill us. And we intended to pass any caught exceptions back to the user, but I'm pretty sure that never quite got finished. Some exceptions do make it back, though.