**15-440 Map Reduce**

**Tyler Healy – Justin Greet**

**System Administration**

**Starting the Map Reduce System**

Our Map Reduce package contains many Java files that work together to allow problems following the Map-Reduce paradigm to be run in parallel across multiple systems. However, in order for a machine to be a participant in the computation of one of these problems, it must have the package installed on a machine. The package can be installed by copying the files of the package into a directory on the participating system.

Once the package is installed, a system can be prepared for participation by doing the following:

**15-440 Map Reduce**

**Tyler Healy – Justin Greet**

**Reference Manual**

**Examples**

**Colorcount**

Before we get into the specific interfaces that need implemented, let us walkthrough an example of a program that successfully completes a Map-Reduce job.

The idea of Colorcount is to take an image and count the number of pixels of each color: black, blue, aqua, purple, red, green, yellow, and white. The mappers will map the pixels to colors, and the reducers will combine the results to count the number of pixels of each color.

Take a moment to examine the code contained in the following three files include with the Map-Reduce package:

* ColorcountMap.java
* ColorcountReduce.java
* ColorcountConfig.java

The ColorcountMap class implements the Mapper interface, and defines the map behavior for a given input key and value. Here, the input key is a long, which is not used in our calculation, and the input value is a pixel, or set of three bytes. The map function examines the pixel and determines what color that pixel is closest to. That result is then collected.

The ColorcountReduce class implements the Reducer interface, and defines the reduce (and combine) behavior for a given intermediate key and a set of associated values. Here, the intermediate key is the name of the color, and the associated values are an array containing Integers that represent the number of occurrences. The reduce function takes a key and sums together all the associated values into one final Integer value. That result is then collected.

Lastly, the ColorcountConfig class, which implements the Configuration interface, provides all the necessary information for the job to run. You will have to change result of getMapperClassPath, getCombinerClassPath, and getReducerClassPath to the location of the .class files for each of these classes, but an example of how the String should be formatted is included.

Once the Configuration file is correctly adjusted for the system you are running on, you can change individual details of the job, such as the job name and the number of workers and reducers. The values of getMasterLocation and getWorkerLocations also need to be adjusted to include hosts on your system that will be running the Map-Reduce program. More information about setting the values of these two methods is given below in “Configuration.” Lastly, you will need to set the location of the input image for this job, as well as the desired output location. This is done in the getFileInputPath and getFileOutputPath respectively.

Details such as getInputFormat, getRecordLength, getOutputKeyClass and getOutputValueClass are set to work with this job and should not be adjusted for Colorcount.

Once the ColorcountConfig.java file is correctly adjusted for your system and has been compiled, it is ready to be passed to a Map-Reduce system to complete a job. This can be done by entering the following command in the shell of a system running the Map-Reduce program (more information for how to prepare a system to accept a job is given in System Administration):

*start /path/to/config/file.class*

The system then attempts to run your job by dividing map work amongst the workers (provided by the Configuration file). The mappers, each with a distinct subset of the input data, complete the map (and combine in the case of Colorcount) operations on the input data. This data is passed to the master worker (provided by the Configuration file), which concatenates the data together and partitions it into subsets to be given to the reducers. The reducers then run the reduce operation on its data and outputs it to file as specified in the Configuration file. There will be one output file for each reducer, and each one will contain a subset of the final results of your job.

As an example, try using this image:

<http://2.bp.blogspot.com/-EZHl3fasivY/TWa-Sh8dGdI/AAAAAAAAACY/SjWdzc-25fA/s1600/yankees+win2.jpg>

The output produced by Map-Reduce should be as follows:

<aqua,128303>

<black,204852>

<blue,119589>

<green,59>

<purple,1277>

<red,32086>

<white,244100>

<yellow,18534>

**Baseball Slugging Percentage**

In baseball, a statistic called a slugging percentage is an indicator for the quality of a batter. It is a weighted average of at-bats, with each type of hit being weighted by their strength (a single is given value 1, a double value 2, a triple value 3, and a home run value 4). The mapper takes a string representation of an at-bat for a player and maps it to an integer. The reduce then calculates the slugging percentage of all players by taking the data from the map and calculating the average.

Take a moment to examine the code contained in the following three files include with the Map-Reduce package:

* BaseballMap.java
* BaseballReduce.java
* BaseballConfig.java
* baseballOutputGenerator.java

The BaseballMap class implements the Mapper interface, and defines the map behavior for a given input key and value. Here, the input key is a long, which is not used in our calculation, and the input value is a word, structured as follows: $playerName-$atBatOutcome. The possible values ofr $atBatOutcome are “single”, “double”, “triple”, and “homer”. The map function examines the at bat outcome and converts it to an Integer.

The BaseballReduce class implements the Reducer interface, and defines the reduce behavior for a given intermediate key and a set of associated values. Here, the intermediate key is the name of the player, and the associated values are an array containing Integers that represent different at-bat outcomes. The reduce function takes a key and uses the associated at-bat outcomes to calculate the slugging percentage for the given player.

Lastly, the BaseballConfig class, which implements the Configuration interface, provides all the necessary information for the job to run. You will have to change result of getMapperClassPath, getCombinerClassPath, and getReducerClassPath to the location of the .class files for each of these classes, but an example of how the String should be formatted is included.

To generate new values on which to test, simply open up baseballOutputGenerator.java, change the variable numberOfAtBats however many trials you would like to see, change destinationPath to whatever you set the input file path in the configuration file to, then run baseballOutputGenerator.

This example is run much in the same way as Colorcount, so refer to the information above for the basics regarding setting up the configuration file and executing the job. **User Interfaces**

**Configuration**

The Configuration file contains all the necessary information for the Map-Reduce package to complete a job.

A user can create a configuration file by implementing the Configuration interface included in the Map-Reduce package, as well as all the functions required by the interface. A description of each function is given below.

*getJobName()*

Defines the name of the job. Used in monitoring the jobs currently running.

*getMapperClassPath()*

Defines the full path of the .class file containing the Map class as defined by the user. This class must implement the Mapper interface. This should be given as a String.

*getCombinerClassPath()*

Defines the full path of the .class file containing the Combiner class as defined by the user. This class must implement the Reducer interface. This should be given as a String. If the job does not have a combiner, *null* should be returned.

*getReducerClassPath()*

Defines the full path of the .class file containing the Reduce class as defined by the user. This class must implement the Reducer interface. This should be given as a String.

*getMasterLocation()*

Defines the machine (host and port) the will be declared the master node. The master is responsible for splitting the input, distributing work to the mappers, partitioning the intermediate outputs, and distributing work to the reducers. This method should return a String given in the following format:

*“hostname:port”*

*getWorkerLocations()*

Defines the machines (host and ports) that are available to do work (either mapping or reducing). Each one of these machines may be assigned to do mapping or reducing work during the job. For each host, two ports must be provided. One is for obtaining work from the master, while the other is used to let the master know that it is alive. It is important to note that if the machine is not running the Map-Reduce application configured to the proper host, it will not be available to do work. More information on preparing a system to do work is given in the “System Administration” manual. This method should return a String given in the following format:

*“hostname:port:port”*

*getFileInputPath()*

Defines the file that contains the raw input to the Map-Reduce job. This should be given as a String representing the full path of the location of the file.

*getFileOutputPath()*

Defines the file to which the Map-Reduce job will write the output. This should be given as a String representing the full path of the location of the file. If the file does not exist, it will be created. If it does exists, it will be overwritten.

*getNumOfMappers()*

Defines the number of map workers (and thus the number of input splits) for the Map-Reduce job. This method should return an int.

*getNumOfReducers()*

Defines the number of reduce workers (and thus the number of partitions of the intermediate output) for the Map-Reduce job. This method should return an int.

*getInputFormat()*

Defines the class of the InputFormat that is used in this job. The InputFormat describes the input, the types of the input keys and values, and the methods to read the input. More information this interface is given below.

*getOutputKeyClass()*

Defines the Class of the keys that will be given as output. This method should return a Class. Our Map-Reduce package allows for the following Output keys:

* String

*getOutputValueClass()*

Defines the Class of the values that will be given as output. This method should return a Class. Our Map-Reduce package allows for the following Output keys:

* String
* Integer

*getRecordLength()*

Defines the length of the record. This method should return an int that is greater than the length of the longest record that could be a result of the Map-Reduce job.

**Mapper and Reducer**

A Map-Reduce program must implement these two interfaces: Mapper and Reducer. These form the core of the Map-Reduce job to be completed.

**Mapper**

The mapper takes a set of input key/value pairs and produces a set of intermediate key/value pairs.

The number of mappers to be used for a given job is defined in the configuration file. The input is split amongst the mappers, and each mapper takes a partition of the input and calls the map function on each input record.

The map reduce takes the following inputs:

* Input key
* Input value
* OutputCollecter

The outputs of the map operation are collected in the provided OutputCollecter, which is explained later.

If applicable, the intermediate outputs are passed through a combiner. More information about a combiner is given later.

The intermediate outputs are written to a file. The Output Path, as set in the configuration file, determines where the results are written. The locations of these files are then passed to the master worker, which concatenates the results together to be reduced.

**Reducer**

The reducer takes the intermediate key/value pairs as produced by the mappers, and combines them into the final result of the Map-Reduce job. This is done by grouping all values with equal keys together, and running the reduce method on the key and its associated set of values.

The number of reducers to be used for a given job is defined in the configuration file. The intermediate key/value pairs are split amongst the reducers by hashing the keys into regions. The reducer then collects the data to be reduced, collects all values with equal keys together, and calls the reduce method on each key and its associated set of values.

The reduce method takes the following inputs:

* Intermediate key
* Set of Intermediate values
* Output Collecter

The outputs of the map operation are collected in the provided OutputCollecter, which is explained later.

The intermediate outputs are written to a file. The Output Path, as set in the configuration file, determines where the results are written.

**Job Input**

**InputFormat440**

The InputFormat describes the input that is being given to the Map-Reduce job. The InputFormat verifies the input, splits into into logical InputSplit instances, and provides the RecordReader implementation that turns the raw input into actual data that can be mapped.

The following InputFormat implementations are include with out Map-Reduce package:

* TextInputFormat440
* ImageInputFormat440

**InputSplit440**

The InputSplit represents the data that an individual mapper is to read and map. The InputSplit is a logical split. This means the result sent to a mapper is not the data itself, but directions on which data to read from the input file.

The following InputSplit implementations are included with our Map-Reduce package:

* TextSplit440
* ImageSplit440

**RecordReader440**

The RecordReader takes an InputSplit and converts the raw input data into key/value pairs (records) that can be used by the mapper. It converts the logical split of data into actual records that the map method can be called on.

The following RecordReader implementations are included with our Map-Reduce package:

* LineRecordReader440
* WordRecrdReader440
* ImageRecordReader440

**15-440 Map Reduce**

**Tyler Healy – Justin Greet**

**Design Report**

**Requirements Met**

* Our system is operable on the unix.andrew and ghcXX.andrew hosts, and should port well to other similar systems
* Configuration files are easily human-readable and human-editable. A user has to simply implement our Configuration interface.
* The system is designed to minimize dispatch latency by preserving processes at participants
* Once the system has been set up, a job can be started from any participating node
* Maps and reduces are scheduled and dispatched to maximize parallelism and performance
* System is able to recover from worker failure
* Each participating node has a simple I/O facility to input operations and receive outputted feedback
* Tools are provided to enable the starting and stopping of jobs, as well as monitoring jobs
* Documentation is provide by way of a System Administration Manual and a User Reference Manual
* Two working examples are included, Colorcount and Baseball

**Requirements Failed to Be Met**

* System easily and correctly runs multiple jobs
* System handles worker failure on reduce nodes (though map failure is accounted for)

**Capabilities and Limitations**

We have developed a system capable of taking a user-designed job that fits the Map-Reduce paradigm, and easily and efficiently computing it across a series of machines over a shared network and distributed file system. This system is easy to use, as the application user just has to follow our System Administration directions to prepare the system, and implement a few interfaces to have a job available to be run. These interfaces are easy to understand, are well documented in the User Reference, and can be implemented without much effort.

Once prepared, our system can start a job from any participating node in the system. This allows flexibility in the system. The system is also capable of splitting up the map and reduce work into an extremely large amount of partitions and sending these parts to multiple systems for map and reduce work. This makes the system very efficient, as it well utilizes the parallel nature of the Map-Reduce problems.

Our system can also load map and reduce classes, as well as input files, from anywhere on the distributed file system that our system is running on. This means that a user does need to have their implementations and input saved and compiled to the disc local to the participating node they are starting the job from. This further increases the flexibility of our Map-Reduce system.

Our system is not without some basic limitations, however. A user needs to correctly implement the map, reduce, and configuration interfaces for the job to be successful. Also, the system only handles certain types of input and output in its current state. This includes text and image inputs, which allow most Map-Reduce problems involving large text data or image processing to be computed through our system. Currently, the only output formats that are available for use are String, Integer, Long, and Double. If given more time, work would be put in to either support additional classes, or, in the best case, make the package general enough to support any arbitrary class.

Another limitation on the system is that behavior is undefined when a second job is started before the first job finishes. We understand that this severely limits the usefulness of the package and this is the first error that would be fixed if we had more time.

**Important Design Decisions**

**Partitioning**

One aspect of the Map-Reduce process is the partitioning of intermediate keys and values into separate buckets to split up the work amongst reducers. In many Map-Reduce implementations, such as Hadoop, this involves some sort of sort and shuffle algorithm.

In our system, a mapper prints their output to file. The master worker then concatenates the files together to group all the intermediate data in one location. The master worker then reads each key, and hashes it into a bucket. There are as many buckets as there are reducers as provided by the Configuration file (*getNumOfReducers*). Each one of these buckets is provided to a reducer in the form of a logical split. The reducer then uses the logical split to read in its information, group all values with like keys, and perform the reduce operation on each pair consisting of a key and a set of associated values.

This is very efficient, as the master worker has to do no work other than the concatenation of the files and the logical split. There is no sort or shuffle necessary for this implementation. The reducer then takes care of reading the data into memory and computing the result.

**Bad Records**

One of the highlights of our system is the handling of improper input. It’ll sometimes be the case, especially with a large amount of data, that some of the data will be malformed and will cause improper records. However, we wanted to design our system in a way that the job could still produce meaningful results if this was the case. Often times, there gigabytes upon gigabytes of data, and we didn’t want the possibility of one improper record crashing the entire job.

We implemented the following method for detecting improper records. After the input splits are created, a mapper is sent a split. If it is successful, it outputs the results to file and lets the master know that it was successful. However, if a failure occurs, the master will detect it in its communications with the worker. If this happens, the master attempts to send the split to a different mapper. If it was not a bad record that caused the failure, this map should be successful. However, if a bad record was behind the failure, it’ll also fail at this node. Therefore, we have decided that after two failures, the master should discard that split.

Unfortunately, this could cause a large amount of correct data being lost. However, the system is designed to take large amounts of data that can be mapped across many many machines. Losing one split due to a bad record will more than likely cause only a small amount of data to be lost.