"YXCG" SIMPLIFIED MAPREDUCE PROGRAMMING MODEL

Yaxin Huang: huang.yax@husky.neu.edu
Xin Fang: fang.x@husky.neu.edu
Chenjin Hou: cheerhou@gmail.com
Gang Liu: beyougang@gmail.com

Abstract

This "YXCG" project aims to implement basic MapReduce programming model. It is implemented in Java. Users can specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key. This MapReduce program is composed of a Map() procedure that performs filtering and a Reduce() procedure that performs a summary operation.

Since "YXCG" is implemented in plain Java and the framework will only use the filesystem provided by the OS, it outperforms Hadoop MapReduce in speed, but it is fault-tolerantless and the users need to do some more configuration before actually run the job.

In this paper, you will find all the information about the project including introduction, our API, implementation overview, evaluation, possible improvements and conclusion.

1. Preparation and Introduction

MapReduce is a framework for processing parallelizable problems across huge datasets using a large number of computers (nodes), collectively referred to as a cluster (if all nodes are on the same local network and use similar hardware) or a grid (if the nodes are shared across geographically and administratively distributed systems, and use more heterogeneous hardware).

To implement our own MapReduce Programming Model, a good understanding of Hadoop MapReduce framework is necessary[3]. The first important thing we need to know is the Hadoop MapReduce framework consists of a single JobTracker and one TaskTracker per cluster-node. The JobTracker is responsible for scheduling the jobs' component tasks on the slaves, monitoring them and re-executing the failed tasks. The slaves execute the tasks as directed by the master.

Second, the Hadoop MapReduce framework operates exclusively on <key, value> pairs, that is, the framework views the input to the job as a set of <key, value> pairs and produces a set of <key, value> pairs as the output of the job, conceivably of different types. The key and value classes have to be serializable by the framework and hence the user needs to use the Writables provided by Hadoop MapReduce or implement their own Writable. Additionally, the key classes have to implement the WritableComparable interface to facilitate sorting by the framework. Input and Output types of a MapReduce job:

(input) <k1, v1> → map → <k2, v2> → combine → <k2, v2> → reduce → <k3, v3> (output)

Third, applications specify the input/output locations and supply map and reduce functions via implementations of appropriate interfaces and/or abstract-classes. These inputs, and together with other job parameters, comprise the job configuration. The Hadoop job client then submits the job (jar/executable etc.) and configuration to the JobTracker which then takes the responsibility of distributing the configuration to the slaves, scheduling tasks and monitoring them, providing status and diagnostic information to the job-client.

By understanding the Hadoop MapReduce framework, we implemented our own MapReduce system in a much simplified way. Our "YXCG" framework consists of a single master and a couple of slaves specified in the configuration file. The master is the JVM created when the user runs the job by invoking the user-created jar file. The master is responsible for splitting the input file, sending the map task requests to slaves, collecting the map results and doing the reduce. A slave is the listener procedure running on machines specified in the configuration file and it is responsible for receiving request from the master, doing the map task and sending back the results to the master.

Plus, we also figure out a way to communicate between two machines by using Socket, a way to transfer file safely by using SCP and a way to load user-specified Mapper and Reducer by using URLClassloader. More details can be found in the Overview of Implementation section.

In conclusion, the features of "YXCG" framework are:

- Parallel computation which views the input as line-based text.
- Restricted output <key, value> type. The key type can only be String, and the value type can only be among String, Double, Integer and Long.
- User self-configuration required. The users needs to write the configuration file and copy necessary files to all the machines on their own.
- No fault-tolerance. The job will fail, output error message, throw exception and terminate directly when the job meets error or some node fails.
- Single master versus multiple slaves model. The master is the machine where the user-specified jar file is run. The slaves are the machines who help to do the map task by doing the map on a split assigned to each one of them. There will only be one reducer (the master) in the framework.

2. API

This project's API provides methods for creating a job, doing map and reduce. Clients can write their own mapper by extends Mapper class. For example:

```
// User needs to specify the key and
// value types.
public class MyMapper extends Mapper<String,
String, String, Double>
{
@Override
public void map(String key, String value,
Context<String, Double> context) {
// Do something
context.wirte(your_key, your_value);
}}
```

The user needs to override the map method. The input key is the line number of the value. The value is a line in the input file. The context is an object from which the user can invoke the write() method to output <key, value> pairs. The output key can only be `java.lang.String`. The output value can only be among `java.lang.String`, `java.lang.Integer`, `java.lang.Long` and `java.lang.Double`.

In the Reducer, the reduce() method takes a key and a list of values corresponding to this key and the Context object. The usage of the Context object is the same as the one in Mapper. The key and corresponding list of values are fetched from the map results. An example of user-specified Reducer is as followed:

```
public class MyReducer extends Reducer<String,
Double, String, Double> {
@Override
public void reduce(String key, List<Double>
values, Context<String, Double> context) {
// Do something
context.write(your_key, your_reduced_value);
}}
```

The user needs to create an instance of the Job class and use the submit() method to start the MapReduce job. Before submitting the job, the user has to configure the input file path, the output directory path, the path to the jar file that the user is using, the output key class and the output value class (it is assumed that the key type is the same both for Mapper and Reducer, and the same for value as well). Here is an example on how to create and submit a job by using this API:

```
//Create a new job
Job job = new Job (new Configuration(args[0]));
job.setJar(args[3]);
//Specify various job-specific parameters
job.setMapperClass(MyMapper.class);
job.setReducerClass(MyReducer.class);
job.setOutputDirPath(args[2]);
job.setInputFilePath(args[1]);
job.setOutputKeyClass(String.class);
job.setOutputValueClass(Double.class);
//Submit the job,
job.submit();
System.out.println("Job complete? " +
job.isComplete());
System.out.println("Job success? " +
job.isSuccessful());
```

3. Implementation

3.1 Overview

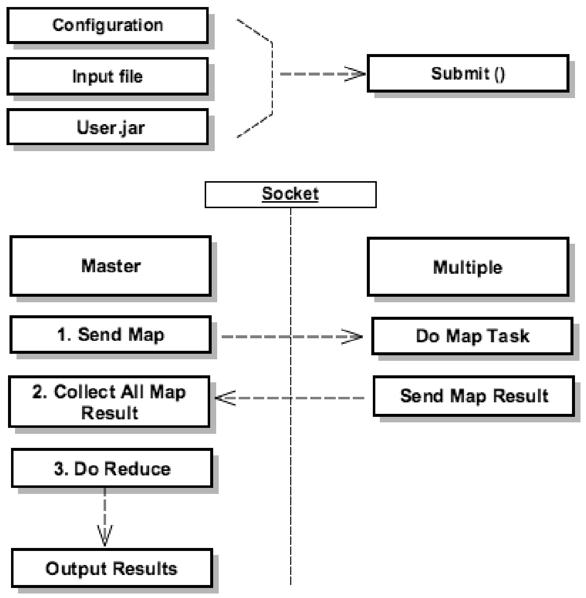

Figure 3-1 Overall Structure of “YXCG” System

Figure 3-1 displays the overall structure of our system. The user needs to have a configuration file, where each line is tab-separated and contains the machine information such as: host-name (the master machine should use "MASTER" as an identifier), ip-address, port number for socket communication, private key path in master machine, login user name. The machine where the users submit the job will become the master. An example of the configuration file is:

```
MASTER 52.4.23.157 4444 key.pem ec2-user
Slave01 52.1.234.132 4444 key1.pem ec2-user
Slave02 52.1.234.198 4444 key2.pem ec2-user
```

After the users submit the job, the master will first split the input file into several splits, where number of splits equals to the number of slave machines. This is not good, because it means the user cannot specify the split size like what they will do in Hadoop MapReduce. But since this makes the implementation simpler, we adopt this way of splitting a file in our system.

Each split will be copied to a slave machine through SCP. The private key for slave to scp files to master and the user jar file will also be copied to slaves in this way.

After the required files for the slaves to do the map task and send results back are transferred to slaves, the master sends a request to each slave to tell them to start the map task. The request will include all the information that the slaves need to locate the user-defined Mapper and how to return the map task result.

After the master receives all map task results from slaves, it will combine the results by reading the contents of those files to a HashMap (`java.util.HashMap<K,V>`). In this way the system accomplish the same result done by the shuffle phase of Hadoop MapReduce.

At last the master will invoke the reduce process similar to how slaves invoke the map process. The results will be output to user-entered output directory.

3.2 Socket-based Communication

A socket[4] is an endpoint for communication between two machines. In our system, the server-client mode is used. Because the system is majorly implemented in Java, the socket API In java.net is utilized to accomplish communications between two JVMs, even in the case when they reside in different machines.

The two key classes from the java.net package used in creation of server and client programs are:
* `java.net.ServerSocket`
* `java.net.Socket`

A server program creates a specific type of socket that is used to wait for client requests, which is called server socket. In the case of a connection request, the program creates a new socket through which it will exchange data with the client using input and output streams. The socket abstraction is very similar to the file concept: developers have to open a socket, perform I/O, and close it.
In this project, the class Listener uses the ServerSocket. The steps for creating a simple server program are:

1. Open the Server Socket:
```
ServerSocket server = new
ServerSocket( PORT ); // Provide the port number
```
2. Wait for the Client Request:
```
Socket client = server.accept();
```
3. Create I/O streams for communicating to the client:
```
DataInputStream is = new
```

```
DataInputStream(client.getInputStream());
PrintStream os = new
PrintStream(client.getOutputStream());
```

4. Perform communication with client

```
//Receive from client:
String line = is.readLine();
//Send to client:
os.println("Hello");
```

5. Close socket:

```
client.close();
```

The implementation of the master procedure is in the Job class. After the job is submitted, the master serves as a client in the server-client model and send requests to all the slaves, who serve as the server, at the same time. The steps for creating a simple client socket program are:

1. Create a Socket Object:

```
Socket client = new Socket(server, port_id);
```

2. Create I/O streams for communicating with the server.

```
is = new
DataInputStream(client.getInputStream());
os = new
DataOutputStream(client.getOutputStream());
```

3. Perform I/O or communication with the server:

```
//Receive data from the server:
String line = is.readLine();
```

4. Close the socket when done:

```
client.close();
```

3.4 SCP

Secure copy or SCP[6] is a means of securely transferring computer files between a local host and a remote host or between two remote hosts. It is based on the Secure Shell (SSH) protocol. Because sometimes the input split needed to be transferred between machines is larger than the memory size, using socket connection to transfer such a large file (precisely, java.lang.String) might cause some issues, and the user jar file cannot be serialize and deserialize in a simple way, in our implementation, we choose to use SCP when we need to copy files from one machine to another.

By using SCP, the private key, IP address, login user name and the destination should all be provided. That information is supposed to be included in the configuration file. However, this is the shortcoming of the system, because it requires user to update the configuration from time to time. For example when they reboot the machines, their IP addresses will probably be different from the previous ones.

3.5 File splits

User's input file will be split according to the size of each split we set. The inputs are users file path, size of each split and the output path. Our initial design goal was to let the user set the split size, but because the time limitation for our project, when we call this API we will set the split size as a number which can let the number of splits equal to number of slaves.

So there are two main things we need to consider about. The First one is how to read the file. In our project, we use RandomAccessFile() to read file.

The RandomAccessFile class in the Java IO API allows you to move around a file and read from it or write to it as you please. You can replace existing parts of a file too. This is not possible with the FileInputStream or FileOutputStream. A random access file behaves like a large array of bytes stored in the file system. There is a kind of cursor, or index into the implied array, called the file pointer; input operations read bytes starting at the file pointer and advance the file pointer past the bytes read.

Then we need to consider about setting Buffer Size. It is best to use buffer sizes that are multiples of 1024 bytes. That works best with most built-in buffering in hard disks. We use 8kb as the maximum read buffer size.

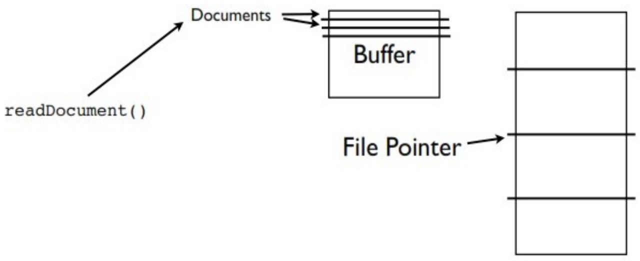

Figure 3-2 How RandomAccessFile Reads a File[5]

Figure 3-2 shows how RandomAccessFile reads a file. Users file will be split into several chunks that have byte need to read. If the bytes need to read is larger than 8KB, then it will be separated according to the size. Since RandomAccessFile has a kind of cursor, so it will remember the position last read.

3.5 Multi-Threaded Map Request

The master sent the map task request to the slaves in a multi-threaded style, allowing it to send all request out and receive the response in different threads. Thus, the master does not need to wait for a slave to finish all the tasks to move on to another. It can just send all the files and requests out, wait until all map tasks results are collected.

4. Evaluation

We evaluate the efficiency of "YXCG" system by comparing its performance with the Hadoop MapReduce. Because the "YXCG" system implementes very similar API to the Hadoop MapReduce API, we can write almost the same codes to do the comparison.
In our test, we use the purchase information as input, which is a line-based text document, and each line is tab-separated. Our demo user program would like to compute the median of sale prices of each category. The user program will print the elapsed time of the whole job and in this way we can measure the performance.

To measure the performance of our "YXCG" system, we choose the AWS EC2 as the environment we will use, since EC2 allows us to use private keys to login or scp files between machines. Our input file is a 33,167,806-line text file, which is about 1.77 GB. In the first test phase we fix the input size and vary the number of nodes in the cluster from 2 (one master and one slave) to 5 (one master and four slaves). In the second phase we fix the number of nodes and vary the lines of input, from a thousand lines to more than 30 million lines. We ran each single test for three times and pick the best to display in the charts.

We use the AWS EMR to run the user program implemented with Hadoop MapReduce API. The job elapsed time is measured by EMR and the precision vary from seconds to minutes. In both experiments, the machines we use are Amazon Linux AMI 2015.03 (HVM), SSD Volume Type, m3.xlarge, which is of CPU 4GHz, 30GB Memory and 2 * 80 GB SSD.

4.1 Varying the Number of Nodes

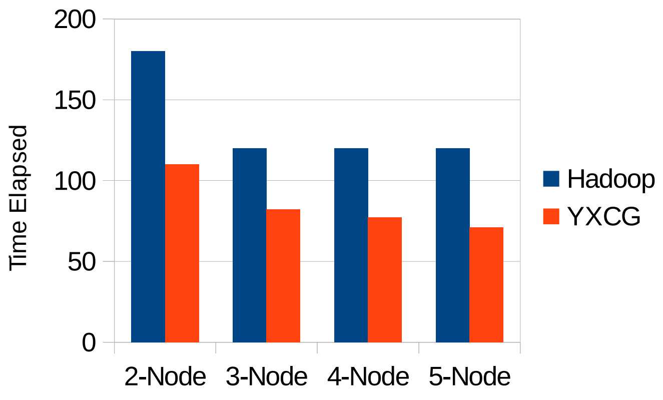

Figure 4-1 YXCP compared with Hadoop MapReduce when Input Size Fixed

Figure 4-1 displays the performance comparison between YXCG and Hadoop MapReduce. From the chart we may infer that the YXCG outperforms Hadoop MapReduce by 1/3, and because YXCG is implemented in plain Java and it incurs less JVM overhead, it may still outperform Hadoop MapReduce when the number of nodes in the cluster is increased. Moreover, YXCG does not implement the heartbeat between TaskTrackers and JobTracker as Hadoop does, and therefore it has less time consumption in the communication. But when the size of input is much larger and the number of nodes required is much more, Hadoop MapReduce will do better since it allows more than one reducer in the cluster.

Table 4-1 2-Node Comparison with Varied Input

Number of Lines	YXCG 2-Node	Hadoop 2-Node
1K	0	41
5K	0	42
10K	0	42
50K	1	43
1M	3	48
10M	28	60
30M	110	180

4.2 Varying the Size of Input

Table 4-1 shows the comparison between YXCG and Hadoop MapReduce when the number of nodes in the cluster is 2. As the number of lines increased, both of them took more time but YXCG starts with less time, because Hadoop MapReduce always cost time to build the filesystem and create a bunch of JVMs.

But as the size of the input increase, the time taken by YXCG increased dramatically. This is possibly because it does not allow the cluster has more than one reducer, and all the work needs to be sent back to master to do the reduce task.

Table 4-2 5-Node Comparison with Varied Input

Number of Lines	YXCG 5-Node	Hadoop 5-Node
1K	45	0
5K	45	0
10K	44	0
50K	46	0
1M	48	2
10M	60	17
30M	120	71

Table 4-2 shows the performance comparison when the number of nodes is set to 5. The number here also tells how YXCG's running time increases dramatically as the size of input increases. Thus, changing the number of reducers in the cluster according to the size of the input file and the size of the cluster, is very important.

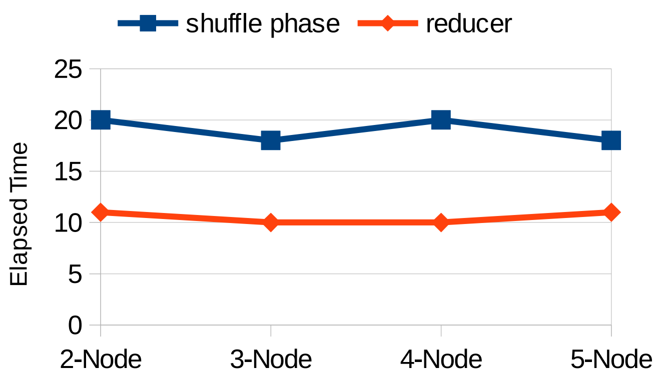

Figure 4-2 YXCG Shuffle Time and Reduce Time of Different Number of Nodes

Figure 4-2 illustrates how much time it takes for YXCG to do the shuffle and reduce when the input file is 30M lines (1.7GB). From the diagram, we can know that actually the shuffle phase and reducer takes almost constant time, though the number of nodes varies. Thus, the reason for YXCG less competitive than Hadoop MapReduce when size of input increases to large enough, might be the map result scp time. In YXCG, every slave need to send back its map result to master and they contend for the disk-write resource.

5. Discussion

There are a lot of improvement can be done to our submitted version, since it is a really simple implementation of MapReduce. Here we discuss several possible improvements to our work. First we can allow the cluster to have more than one reducer. This can be achieved by adding some methods to the Listener program and allows it to receive reduce request as well. The Job class can be rewritten to let the user set the split size. This feature can allow user to test the best parameter for the specified job. For better user experience, we can simplify the configuration process probably by using other applications to build the cluster.

6. Conclusion

In our paper we discussed the features of YXCG and how we implemented the components of it. We also compared its performance with Hadoop MapReduce with same specification and the experiment showed YXCG's strengths and shortcomings. The YXCG framework might be very simple and will be excelled by Hadoop MapReduce at some point, but it is still a good option when the job is relatively light-weighted.

7. Achknowledgement

[1] Prof. Jan Vitek, for his help in clearing doubts about how to locate a class by name in a Jar file and how to use reflection to create an instance of the class.
[2] Jue Yang, for her discussion of the possible overall structure of the simplified program.
[3] Apache Official Document, The MapReduce Tutorial:
https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html

[4] Chapter 13, Socket Programming, Object Oriented Programming with Java: Essentials and Applications:
http://www.buyya.com/java/Chapter13.pdf
[5] RandomAccess File Digram:
https://cruisingcrux.
[6] SCP Basic Concepts:
http://en.wikipedia.org/wiki/
[7] Java Tutorial, Defining and Starting a Thread:
https://docs.oracle.com/javase/tutorial/essential/concurrency/runthread.html