Homework 3 Report

Mayank Mohta (mmohta)
Yuchen Tian (yuchent)

NOTE: This document describes the high-level architecture of our MapReduce implementation. For how to launch and test our program, please see the **README** file. For how to write an application under our framework, please see the **api-guide.pdf**.

Capabilities, assumptions and limitations of our system

Following are assumptions made by our system:
1. Input file should be a plain text file (ASCII characters).
2. Each line is submitted as an input to the map function. This line can contain any number of characters (for example a document in word count, a pair of vertices denoting an edge in the indegree example).
3. The user can specify number of worker hosts and how many map slot and reduce slots are present on each worker (for example, in a 4 core machine, we can have 4 slots (processes), and an example valid assignment is 3 map slots and 1 reduce slot).
4. Task tracker failures can be handled, but job tracker failure is not handled.
5. We assume that each map task will produce output which fits in memory buffer. Hadoop framework uses more completed spilling mechanism to handle when the in memory buffer fills up.
6. We have tested our system with some failure scenarios. However, were not able to test the scalability and robustness very thoroughly.

System Design

Introduction: We have tried to model our Map Reduce implementation based on the Hadoop's implementation (with many simplifying assumptions). Figure 1 borrowed from the online tutorial [http://answers.oreilly.com/topic/2141-how-mapreduce-works-with-hadoop/] outlines the basic high level architecture of our design with some modifications as explained.

Structure: Our Map Reduce implementation contains four packages that covers different aspects of system:
1.Framework: This package contains all the components responsible for running the whole framework, including JobTracker, TaskTracker, Map and Reduce workers and the communicates between them.
2.IO: This package works as an intermediate layer for the framework components to interact with the underlying file system, including reading and writing records.
3.mapred: This package is provided for the user to write mapreduce application. It contains the abstract implementations of Mapper and Reducer.
4.example: This package contains two example Map Reduce program: 1. Word Count, 2. InDegree Count.

Cluster Setup: Before running any map reduce tasks, the cluster needs to setup. The python script name “setup.py” sets up the cluster which starts the following processes

1. JobTracker: This is the “global master” process for the cluster and it runs on the configured host. It has two working threads. One is responsible for assigning tasks to worker hosts, coordination, retries and monitoring the progress of the entire job. The other is responsible for sending the current progress to the client side.
2. TaskTracker: This is “local master” for the host and one instance of it runs per worker host. It coordinates with the worker processes running on the local host, receives tasks from JobTracker and assigns work to them. It sends the progress of current running tasks to the jobtracker through the heartbeat.

The number of map worker processes (Map slots) and reduce processes (Reduce Slots) on a host is configurable. For example, if the host is a quad core machine, then we might configure 3 map slots and 1 reduce slot to take advantage of the multi-core machines.

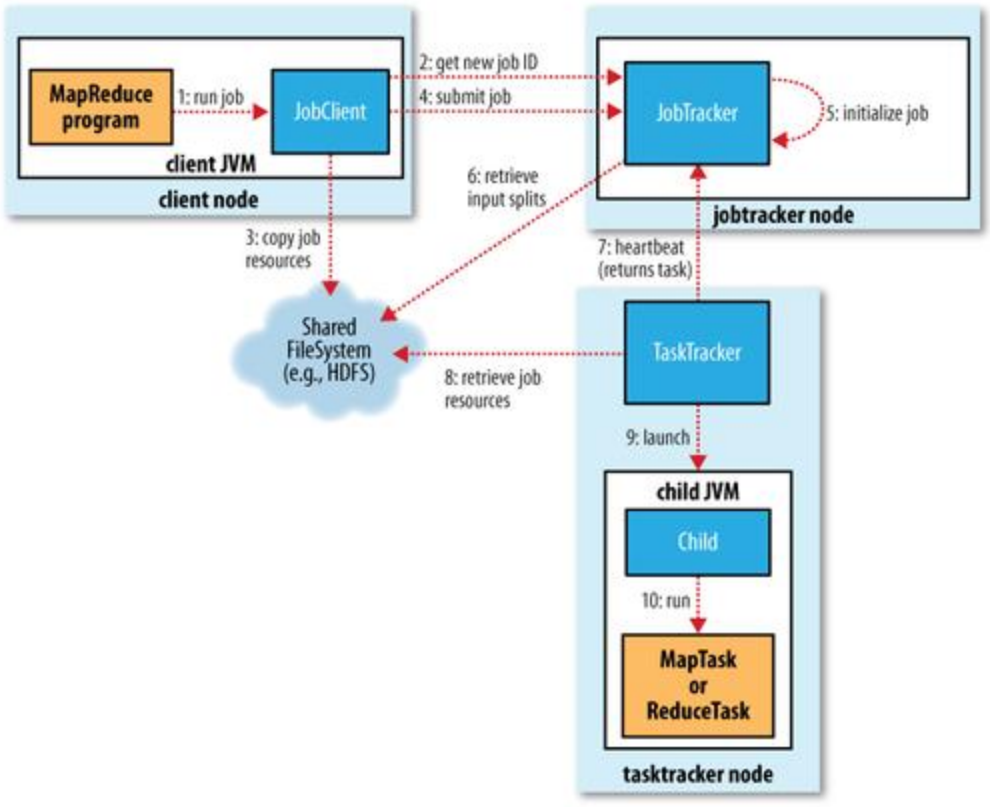

Figure 1

Life cycle of a Job

The complete story of what happens from start to end of the job.

1. The client program sets up a “Job” object. This object contains details like input directory (AFS location), output directory(AFS location), name of map class, name of reduce class, number of reducers, location of jar on AFS and the splits of input files. All these information is required to

setup and run the job on the map reduce cluster.
2. Once the Job object is setup, the client JVM uses the API call implemented in JobClient class to submit the job for execution. Under the hood, client creates a socket to communicate with the JobTracker process (could be running on a separate host).
3. The client api then keeps polling for periodic updates from the job tracker about the progress of the job. This is used to report progress to user.
4. When the job tracker receives a new job, it assigns it a new Id. An actual split is created by using a fixed size chunk of the file in bytes (which can be configured by the user). Each split corresponds to a separate “Map Task”. For simplicity, currently it is assumed that the input file can only be in text format and if end of line (“\n”) is used to split then it does not lead to corruption of input. If the split boundary happens to be in middle of a line, then the first split gets that line and the second split starts from the next line (Figure 2). Note that, since the files are present on AFS, a “Split” object just stores the start and the end of the file (RandomAccessFile java library is used to access the correct part of the file)

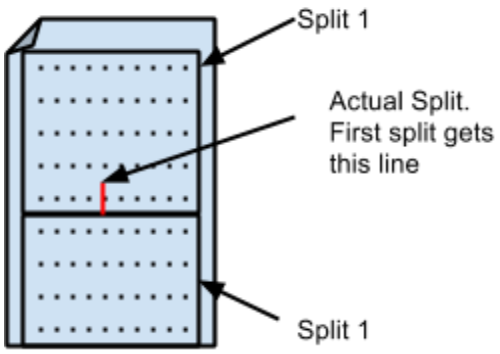

Figure 2

5. Once all the splits are created, map tasks are created (each with one split) and added to the pendingMapTasks queue in the JobTracker.
6. Each TaskTracker from respective worker host sends periodic heartbeats to JobTracker. This heartbeat contains number of free map and reduce slots. It also reports the status of currently tasks running on it.
7. Job Tracker responds to these heart beats by assigning new pending map tasks to be run on the worker having free slots
8. Once a request to run a new task is received by the task tracker, it is send to the free mapper process to run (Note, map and reduce run in separate JVMs than the task tracker. The reason for this is to isolate failures and prevent task tracker from crashing when a map or a reduce task fails). As required, we reuse the JVM across different tasks. Once a worker process is started (using ProcessBuilder api) it keeps communicating with the task tracker which assigns it new tasks whenever available.
9. The worker process uses reflection to instantiate a Map object and starts running the map task.
10. Once a map task is finished, task tracker is notified, which sends the update of completion of this map task to job tracker and this map slot is free to run other map tasks.

11. The Job tracker keeps track of whether all the map tasks have finished. If this is the case, then it starts submitting the reduce tasks to TaskTrackers.
12. Once all reduce tasks are finished, JobTracker marks this job as done and notifies the client about the success.

Map and Reduce executions

When a map task is running, it gets each record from the split assigned to it through TextRecordReader and runs the map function on each record. The output of the map task is stored in an in-memory buffer. Once, all records of the split are completed, the in-memory buffer is sorted per reduce partition. A unique sorted file is created for each of the reduce partitions (equal to number of reducers set when to Job was setup by the application code) . Hence, at the end of the Map phase, there will be files equal to the number of map tasks for each reduce partition. These files are on AFS. Hence, when a reducer starts (each reducer is responsible for one unique partition), it merges all the files written by different map tasks for its reduce partition. In our implementation, we employ the K-way merge algorithm and wrap it as an iterator. In this way, we don't create the single merged file. This iterator emits a ReduceUnit each time, which is basically a wrapper class contains a key and all the values associated with it. Then it will be consumed by the user defined reduce function.

Failure Detection and Retries

Our systems can retry each failed map task or reduce task a configurable number of times. It is also robust to failure of a complete worker node (task tracker). In this case, the JobTracker (master) realizes that it has not received heartbeat from the failed task tracker for some amount of time and assumes that it might have failed. All the tasks which were running on the task tracker are put in the pending queue and assigned to other active task trackers.
The challenge here was to avoid double counting in case of failure. This is done by adding "incomplete" prefix to output file name when the task is running. Once the task has completed successfully, it renames its file to the correct name. All the incomplete files are ignored. Thus, only one correct output file can be created for any task (map or reduce)

Config File

We provide a simple config file which has key-value pairs for different configurations. A detailed description of all the fields are provided as comments in the "cluster_config.txt" file provided with the code.

Examples Codes

We provide 2 simple examples along with the code. They are located in "cmu/cs/distsystems/hw3/examples" directory in the source code. ExampleProg1 is the standard wordcount example. The second example is counting the indegrees of a graph given in the edgelist format in a file.

What we can improve?

There are few things we intended to implement at first, but give up due to various reasons.

Mapper:
In Hadoop, the mapper has circular memory buffer, and using a background thread to sort and write the output. After the task is done, all the file will be put into a single sorted file and the partitions are given to the reducer.
In our implementation, we don't set limit to memory buffer, and all the sorting is in memory. Our implementation, when comparing to the Hadoop implementation is much simpler, but I have some flaws:
1.Not robust: as cluster machines are generally very cheap and have limited memory, a memory limit is necessary.
2.Fragmentation: following our implementation, a lot of tiny files will be created. This will be a problem in a distributed file system, where the block size is usually very big. The efficiency is suffered under our current design

Reducer:
In Hadoop, the reducer starts will the mappers at the same time. After starting, the partition files are beginning to copy to reducer and beginning to merge, so after all the mapper are finished, reducer can begin to work immediately.
But in our implementation, the reduce tasks are launched only after all the map tasks finished. We make the decision following the two reasons:
1.AFS is not the same as HDFS. Locality is not so important in our scenario. So we don't need to copy file on the fly to save the IO time.
2. If we follow Hadoop's implementation, it will greatly complex our message passing system, because we have to design a way for reducer to talk to mappers.

Type System:
In our current implementation, all the keys and values are required to be strings. Recall that in Hadoop, we can use specifies the input and output types ourselves. We intended to implement this at first, but later we found that we are limited by how Java handles the generic type. In Java, it is very hard to get type information dynamically. Although there are some tricks out there to achieve this, I believe it is beyond the scope of this assignment. I think this is why Hadoop employs its own type system rather than uses the native Java type system to bypass such limitation.