CARNEGIE MELLON UNIVERSITY

15640: Project 3 Report

Building a Map-Reduce facility

Yang Sun (Andrew ID: yksun) | Yuan Gu (Andrew ID: yuangu)

4/12/2013

Table of Contents

Requirement Checklist 2
Design and Implementation 3
Register 3
Start 3
Stop 3
Monitor 4
Submit 4
Status Checker 4
Map-Reduce 4
Input and Output 4
Map and Reduce 5
Intermediate Results Collecting and Sorting 5
Examples 5
Calculate Sum (StringInputMapper, StringInputReducer) 5
Grep Odd Numbers (OddNumberGrepMapper, OddNumberGrepReducer) 5
Future Works 6

Requirement Checklist

We spent a lot of time to make sure our implementation meets the project requirements. In general, our Map-Reduce facility is able to handle the following conditions/requirements:

- We have already deployed our facility on Andrew clusters. For details about how to start and run the programs, please refer to the documentation for system administrators and programmers.
- As suggested, the system reads a configuration file to finish up the deployment. It is purely human readable and editable, implemented by using Java Property file. For the configuration parameter, please refer to the documentation for system administrators.
- The master controller is in charge of dispatching the map tasks and reduces tasks. The participants have nothing to do except executing the tasks and receiving user input commands. The output file of tasks will write to AFS directly without passing via network. Under such model, the dispatch latency and overhead can be minimized.
- User can issue any commands or submit jobs from any participants in the facility, including the master controller. However, master controller will not get involved to execute map tasks or reduce tasks as we think it is already busy enough to do the dispatching. Under crazy busy mode, too many tasks running on master controller would reduce the efficiency.
- The facility is able to execute multiple MapReduce tasks concurrently without any problem. We have already done tricks (e.g delay the map tasks so that multiples running concurrently and then release results to master controller) to evaluate the performance of concurrency and the experiments were succeeding without any synchronization issues.
- When the master controller dispatches the tasks, it will look for the clients with least burden. That means, the clients which have least running tasks will be selected to execute the next task. In such a way, the task can be run efficiently and the maximum performance gain can be achieved.
- Our facility is able to recover the map and reduce tasks from a failure/disconnected client. Basically, we implemented two levels of protection. First of all, if the client is dropped at the time or before the time we dispatch the map/reduce task, an exception will be raised while dispatching and the master can detect it and redistribute the task to another client instead. If no remaining clients are available to accept this task, an error message will be prompted to the user. Secondly, if the client is dropped after we dispatch the map/reduce task. In other words, the map/reduce task is already running on target client, but the connectivity has some problem. The master status checker service (details can be found in the next section) will detect it and similarly, the master will remove the client from the job tracking table and redistribute all its running tasks to other free clients. Therefore, any failure of clients can be treated appropriately.
- Since the facility support fixed length input file only, we provide a tool OutputRecordWriter to help programmers preprocess the input file. After running the

MapReduce task, another tool InputRecordReader can be used to convert the reduce output to human-readable output. Note that we can definitely integrate these steps into our facility, however, the application programmer may use original MapReduce input and output under some circumstances. Therefore, we leave this option to the programmer to decide what to do with these files. An illustration example is already included in the source code, under Examples/StringOutputWrite.java and Examples/StringInputRead.java.

- We have provided the management tools, such as start, stop and monitor. The user can issue these commands directly from the command line in any participants, including the master controller. Note that, when the master and clients start, the background MapReduce facility has not been turned on yet. The user has to issue a "start" command before submitting any MapReduce jobs.
- Documentation for System Administrators has been included in the same directory as this report.
- Documentation for Programmers has been included in the same directory as this report.
- Two working examples, including the source code and generated jar file, have been included in the submission package. Details can be found in the next few sections.

Design and Implementation

Register

When a client is brought online, it will send a registration message to the master controller to confirm the healthy status. The master controller will then look for the client's IP address and port from the property file according to its client ID and add it to the client job tracker. The reason we design like this but not depend on the property file only is that once a client is dropped or crashed and then brought online again, it can join the facility again immediately. The master controller dynamically manipulates the job tracker and checks the status of all participants indicated in the property file, even if one is temporarily offline.

Start

When the master and clients start, the background MapReduce facility has not been turned on yet. The user has to issue a "start" command before submitting any MapReduce jobs. Before the master controller receives the "start" command, the facility can do nothing at all. The user can start the facility in any participants, including the master controller. No matter where the user issues the command, the command message will be sent to the master controller and the master controller will turn on the background job tracker and status checker.

Stop

When a stop command is issued, the master will notify the status checker and facility to shut down. That means, after a successful stop, the facility will deny any job submission. However, already running jobs will continue till finish. Although we can choose to aggressively interrupt

these processes, we think keeping them running is the proper way we should go for. Again, this command can be issued in any participants, including the master controller.

Monitor

This is a simple command that just prints out the number of tasks running on the current participant. Since master controller doesn't get involved to execute map/reduce tasks, a monitor command issued from master controller will always return 0. Any other participants will query the number of running tasks to the job tracker on the master controller. Similarly, we can choose to maintain a task list on each client. However, we think it would be better to have a centralized model, which is the job tracker, to handle all job-related cases.

Submit

The submit interface is explained in the documentation for programmer. Here we will focuses on the methodology. The command can be issued from any participants, including the master controller. After several error checking processes, the master controller will choose n clients with least burden to run the map tasks, where n is the number of maps programmer defined in the Map Class. Once the map task is done, the client will send a message notifying the master controller. The master controller will again choose m clients with least burden to run the reduce tasks after getting notified for completion from all map tasks involvers, where m is the number of reduce programmer defined in the Reduce Class. Similarly, master controller will be notified if a client finishes the reduce task. Finally, the job submission issuer will be prompted the result after all reduce tasks are done.

Status Checker

Recover the tasks running on failed or a disconnected client is difficult without the help of status checker. Because the job tracker will close the socket connection after the task is sent to corresponding client successfully and assumes its running status unless it gets notified. To check the running status, we implemented the status checker running on master controller, which will frequently (every 5 sec) query the connectivity and running status to each client. If no response is received or connection failed, we are sure that the client is dropped even if it is possible that the task is still running normally. But how can we collect the result? Therefore, the master will then choose several replacements to take over all the tasks running on this client to ensure all of them can be finished correctly. The failed client can join the facility again via registration step explained above.

Map-Reduce

Input and Output

For processing of the input records, we support fixed-length input records, the length of which could be specified by developers. Besides, with the help of Java generics, we design and implement a generic input record reader which allows developers to define their own input record classes. In this way, we provide the maximum flexibility for processing input records.

The same thing happens to the processing of output records. Developers could define their own output records which they could utilize in the reducing phase to generate the final output.

Map and Reduce

This is one of our proud points: we implement exactly the same interfaces as Hadoop 0.2x (old version APIs) for mappers and reducers. We use a lot of Java generics and inheritances to implement this. Mappers should receive an input record and a <K, V> pair collector to collect the <K, V> pairs generated. Reducer should receive a Key, followed by a list of corresponding Values, and then use an output record collector to collect the outputs.

User could define their Key and Value classes. And for the convenience of them, we also provide an example implementation of string-based Key and Value classes, which is the same as the TextWritable class of Hadoop.

Intermediate Results Collecting and Sorting

Our intermediate result collector will keep collecting intermediate <K, V> pairs from mapper, and spill intermediate <K, V> pairs to disk if the memory cannot hold them all. After the mapping phase is over, our system will automatically sort all the spills and partition them (user could also define their own partition algorithms here). Before the reducing phase starts, reducers will collect all partitions belonging to them and sort the partitions.

We simulate the sorting process of Hadoop by applying merge sorting algorithms to the intermediate results. Our implementation will iteratively do 2-way merge sort until all the chunks of intermediate results are sorted.

Examples

We provide the following two examples for the reference.

Calculate Sum (StringInputMapper, StringInputReducer)

The program simply calculates the total sum of the numbers from 1 to 999 appearing in the input file. The expected result would be (999 + 1) * 499 + 500 = 499500. To run this example, try to issue **"submit ../Jar/15640.jar Examples.StringInputMapper test.input Examples.StringInputReducer reduce-output"**.

Grep Odd Numbers (OddNumberGrepMapper, OddNumberGrepReducer)

Similarly, the program greps odd numbers from 1 to 999 appearing in the input file and count the total number of odds. The expected result would be 500. To run this example, try to issue **"submit ../Jar/15640.jar Examples.OddNumberGrepMapper test.input Examples.OddNumberGrepReducer reduce-output"**.

Future Works

Because of time limitation, we haven't had enough time to do more robustness tests. In addition, we implemented the String Input Type only for the Map and Reduce. Although it is sufficient for normal usage, it would be better to have more types as provided in Hadoop MapReduce framework. Similarly, our system only supports fixed-length inputs. We would like to extend it to support at least line-based text inputs, just like what most Hadoop programmers expect. As sorting phase has significant influence on the performance of our system, we would like to improve the sorting algorithm. For example, we might choose to implement a k-way merge soring instead of the current 2-way merge soring. Besides, many tricks could be applied here: for example, instead of merging all chunks into one and the partition them. We could apply the partition algorithm while the last two chunks are merging.