Map Reduce - 1

Abhiram R, Smit Mehta, Abhik Mondal, Goutam Rajiv

April 29th, 2013

Abstract

The current implementation of mapreduce in hadoop assumes homogenity in the configuration of the computers in the cluster. The goal of this project is to make a node-performance-aware implementation of the mapper in Hadoop. The aim of this project is to create an interface to invoke the mappers of other implementations of mapreduce (for example, Phoenix++ and GPMR) so that various capabilities of the nodes in the cluster can be exploited (Multi-processor capabilities, in case of Phoenix, and GPU capabilities in case of GPMR). In this project, we have come up with an interface that allows communication between Hadoop and Phoenix code.

Contents

1 The MapReduce Paradigm

The MapReduce is a programming model that combines higher order functions in functional programming languages, map and reduce. Programmers define the map and reduce functions in their application. The input to the map is a set of records, and each invocation to the map function outputs its own independent key/value pair. All the pairs with the same key value are passed to a common reducer, and there are many such reducers. The reduce function outputs a new sequence of output values. The MapReduce model was invented at Google, by Jeffrey Dean , and Sanjay Ghemawat.

2 Hadoop

Hadoop provides a distributed file system and a framework for the analysis and transformation of very large data sets using the MapReduce paradigm. An important characteristic of Hadoop is the partitioning of data and computation across many (thousands) of hosts, and executing application computations in parallel close to their data .HDFS is the file system component of Hadoop. While the interface to HDFS is patterned after the UNIX file system, faithfulness to standards was sacrificed in favor of improved performance for the applications at hand. HDFS stores file system metadata and application data separately.

2.1 Architecture

2.1.1 NameNode

The HDFS namespace is a hierarchy of files and directories. Files and directories are represented on the NameNode by inodes, which record attributes like permissions, modification and access times, namespace and disk space quotas. The file content is split into large blocks (typically 128 megabytes, but user selectable file-by-file) and each block of the file is independently replicated at multiple DataNodes (typically three, but user selectable file-by-file). The NameNode maintains the namespace tree and the mapping of file blocks to DataNodes(the physical location of file data).

2.1.2 DataNode

Each block replica on a DataNode is represented by two files in the local host's native file system. The first file contains the data itself and the second file is block's metadata including checksums for the block data and the block's generation stamp. The size of the data file equals the actual length of the block and does not require extra space to round it up to the nominal block size as in traditional file systems. Thus, if a block is half full it needs only half of the space of the full block on the local drive.

2.1.3 HDFS Client

User applications access the file system using the HDFS client, a code library that exports the HDFS file system interface.When an application reads a file, the HDFS client first asks the NameNode for the list of DataNodes that host replicas of the blocks of the file. It then contacts a DataNode directly and requests the transfer of the desired block. When a client writes, it first asks the NameNode to choose DataNodes to host replicas of the first block of the file.

3 Compiling Hadoop

We compiled the hadoop source code using maven.

3.1 Maven

Maven is a build automation tool used primarily for Java projects. Maven serves a similar purpose to the Apache Ant tool.Maven dynamically downloads Java libraries and Maven plug-ins from one or more repositories such as the Maven 2 Central Repository.

4 Benchmarking

Benchmarks make good tests, as you also get numbers that you can compare with other clusters as a sanity check on whether your new cluster is performing roughly as expected. We can tune a cluster using benchmark results to get the best performance out of it.Hence performing benchmark tests for CPU, RAM and the hard disk is useful.

4.1 TestDFSIO

The TestDFSIO benchmark is a read and write test for HDFS. It is helpful for tasks such as stress testing HDFS, to discover performance bottlenecks in your network, to shake out the hardware, OS and Hadoop setup of your cluster machines (particularly the NameNode and the DataNodes) and to give you a first impression of how fast your cluster is in terms of I/O.

Benchmarking the DOS Lab computers and running jobs in that cluster. We got access to the DOS Lab computers where we ran the TestDFSIO (Read and Write benchmarks).

The results received will be used to classify the node .

Here were the Write results on two of the systems we ran it on:

IP Addr:10.6.9.44

```
13/04/09 14:36:55 INFO mapred.FileInputFormat: TestDFSIO : write
13/04/09 14:36:55 INFO mapred.FileInputFormat: Date & time: Tue Apr 09 14:36:55 IST
2013
13/04/09 14:36:55 INFO mapred.FileInputFormat: Number of files: 10
13/04/09 14:36:55 INFO mapred.FileInputFormat: Total MBytes processed: 1000
13/04/09 14:36:55 INFO mapred.FileInputFormat: Throughput mb/sec: 244.140625
13/04/09 14:36:55 INFO mapred.FileInputFormat: Average IO rate mb/sec: 249.97665405273438
13/04/09 14:36:55 INFO mapred.FileInputFormat: IO rate std deviation: 31.63254243812152
13/04/09 14:36:55 INFO mapred.FileInputFormat: Test exec time sec: 10.816
```

IP Addr:10.6.9.50

```
13/04/09 15:20:09 INFO mapred.FileInputFormat: TestDFSIO : write
13/04/09 15:20:09 INFO mapred.FileInputFormat: Date & time: Tue Apr 09 15:20:09 IST
2013
13/04/09 15:20:09 INFO mapred.FileInputFormat: Number of files: 10
13/04/09 15:20:09 INFO mapred.FileInputFormat: Total MBytes processed: 1000
13/04/09 15:20:09 INFO mapred.FileInputFormat: Throughput mb/sec: 90.3995660820828
```

```
13/04/09 15:20:09 INFO mapred.FileInputFormat: Average IO rate mb/sec: 91.90361785888672
13/04/09 15:20:09 INFO mapred.FileInputFormat: IO rate std deviation: 11.60454454665146
13/04/09 15:20:09 INFO mapred.FileInputFormat: Test exec time sec: 13.497
```

4.2 TeraSort

Basically, the goal of TeraSort is to sort 1TB of data (or any other amount of data you want) as fast as possible. It is a benchmark that combines testing the HDFS and MapReduce layers of an Hadoop cluster.Typical areas where TeraSort is helpful is to determine whether your map and reduce slot assignments are sound (as they depend on the variables such as the number of cores per TaskTracker node and the available RAM), whether other MapReduce-related parameters such as io.sort.mb and mapred.child.java.opts are set to proper values, or whether the FairScheduler configuration you came up with really behaves as expected.

5 Phoenix (Literature Review)

The Phoenix project (Stanford University) is the application of MapReduce models on shared-memory systems. Most MapReduce implementations are limited primarily by Disk and Network I/O. Shared memory implementations are void of these bottlenecks, and hence their performance is determined primarily by workload-influenced details, like intermediate key/value data layout, memory allocation pressure, and framework overhead.

The MapReduce pipeline adopted by Phoenix is a static one, which is similar to implementations which are cluster based.The current Phoenix implementation provides an application-programmer interface (API) for C and C++. However, similar APIs can be defined for languages like Java or C#. The API includes two sets of functions . The first set is provided by Phoenix and is used by the programmer's application code to initialize the system and emit output pairs . The second set includes the functions that the programmer defines . Apart from the Map and Reduce functions, the user provides functions that partition the data before each step and a function that implements key comparison. Note that the API is quite small compared to other models. The API is type agnostic. The function arguments are declared as void pointers wherever possible to provide flexibility in their declaration and fast use without conversion overhead.

5.1 Functions

5.1.1 Functions provided in runtime

int phoenix scheduler (scheduler args t * args):

Initializes the runtime system. The scheduler args t struct provides the needed function & data pointers .

void emit intermediate(void *key, void *val, int key size):

Used in Map to emit an intermediate output <key,value> pair. Required if the Reduce is defined .

5.1.2 Functions provided by user

int (*splitter t)(void *, int, map args t *):

Splits the input data across Map tasks. The arguments are the input data pointer, the unit size for each task, and the input buffer pointer for each Map task

void (*map t)(map args t*):

The Map function. Each Map task executes this function on its input

int (*key cmp t)(const void *, const void*):

Function that compares two keys

5.2 Basic Control flow

The runtime is controlled by the scheduler, which is initiated by user code. The scheduler creates and manages the threads that run all Map and Reduce tasks. It also manages the buffers used for task communication. The programmer provides the scheduler with all the required data and function pointers through the scheduler args t structure. After initialization, the scheduler determines the number of cores to use for this computation. For each core, it spawns a worker thread that is dynamically assigned some number of Map and Reduce tasks.

5.3 Buffer Management

Two types of temporary buffers are necessary to store data between the various stages. All buffers are allocated in shared memory but are accessed in a well specified way by a few functions. Whenever we have to re-arrange buffers (e.g., split across tasks), we manipulate pointers instead of the actual pairs, which may be large in size. The intermedi- ate buffers are not directly visible to user code.

6 GPMR (Literature Review)

GPMR is a stand alone implementation of MapReduce for GPU clusters, a library in C++, that leverages the power of GPU clusters for large-scale computing. By combining large amounts of maps and reduce items into chunks, and using partial reductions and accumulation, this MapReduce model better utilizes the GPU capabilities of the nodes in the cluster.

There are several challenges in this approach. Multiple GPUs cannot communicate via a network because they cannot source or sink Network input/output. Also, GPU has no support for out of core operations and virtual memory. A naive implementation of mapreduce for GPUs wastes the inherent capabilities of the GPUs, mainly its multi-core nature. Finally, the GPU architecture is not well supported by the naive model.

GPMR tackles the above challenges - movement of data, out-of-core management, while maintaining full GPU access. Since it is stand alone (not sitting on top of Hadoop etc), it does not handle fault tolerance, and it does not provide a distributed file system. (The paper uses the term storage agnostic). The four stages of the Mapper in GPMR are : the Map, Accumulation, Partial Reduction, Combination, and Partition . These substages can be activated by the user. Map processes an input chunk and outputs a set of key/value pairs. The entire chunk is copied to the GPU via a user-supplied function at once and processed by one or more user-supplied kernels.

This allows the GPU to be used maintaining the mapreduce portions of the other independent data elements. GPMR assumes the chunk sizes and Map works on one chunk at a time, and its output will consume most of the GPU memory. This enables an efficient out-of-core technique to add to GPU MapReduce. Chunks used are that which are a fraction of the size of available memory, allowing the system to Map or Reduce a chunk while simultaneously streaming another chunk to or from the GPU. One particular facet of the Map stage (and the Reduce) is the need for load balancing. GPMR tracks the per-GPU work in a dynamic queue. If one GPU finishes its work in its local queue and other GPUs have much more work to do, we shift chunks between the local queues. The chunks must hence implement a serialization method.

Thus, GPMR is a good candidate for our project, in terms of exploiting the GPU capabilities of nodes in the cluster, by creating an interface from Hadoop.

7 Implementation

The ***Mapper*** class in *org.apache.mapreduce* is the part of the code that maps the input key/value pairs to a set of intermediate key/value pairs. Maps are the individual tasks that which transform input pairs into intermediate pairs, although the intermediate record format need not be the same as the input record format.

An input record (or pair) can map to multiple output key/value pairs. The Hadoop MapReduce framework spawns one map task for each InputSplit which is generated by the Inputformat for the particular job. All Mapper Implementations (including our PhoenixMapper) can access the configuration for the job using the context. More specifically, using the command , *jobContext.getConfiguration().*

The functioning of the Mapper can be divided as follows:

- Setup Phase - calls *setup(org.apache.hadoop.mapreduce.Mapper.Context)*
- Map Phase - calls *map(Object, Object, Context)* for each key/value pair in the input split.
- CleanUp Phase - *cleanUp (context)*

After this, all intermediate outputs generated are passed to the *Reducer.* Our part of the project need not deal with the reducer aspects, so details about the reducer part have been omitted from this report. One thing to note is that the Mapper outputs are partitioned per Reducer. Users are given the power to control which key goes to which reducer by implementing a custom *Partitioner.*

With the Mapper functionality understood, our next step is to understand which part of the code to modify to call Phoenix's mapper for certain key/value pairs. In order to exert greater control on map processing, we need to override the *run(context)* method. Within the run() function, we can make a call (using *JNI* or system call based on the way we want to pass the key/value pairs) from Hadoop to Phoenix++ for the jobs that we want. But various parts of the code are affected by this modification. TheChess following section explains the technicalities and the way that we tackled the issues.

We extend the *Task* class to make our own implementation of a map task which we call, *PhoenixMapTask.* The *Mapper* class is the one that implements the *run()* function. But in the *LocalJobRunner*, the *MapTask* class is used currently. Now for some nodes, we need to use *MapTask*, and for some nodes we need to *PhoenixMapTask*. Within *MapTask*, the *Mapper* class is used, and its *run()* function is called. Instead, whenever we want to run Phoenix, we need to create some interface to another mapper that we want to use, say *PhoenixMapper* somehow. So we first create a *PhoenixMapper* which extends *Mapper* and overrides the *run()* function.

The *LocalJobRunner* class is an important part of Hadoop's Mapreduce. It implements MapReduce locally, in process, for debugging. It also has a private class called *MapTaskRunnable*, which is a Runnable which handles a map task to be run by the *executor*.

The *run()* function of the *LocalJobRunner* is responsible for setting up a new map task (*PhoenixMapTask*, in our case), setting the configurations required, the user information, and the output files. It also maintains a *LocalJobRunnerMetric*, which keeps track of various important parameters about the mapping process. Once the required configurations have been setup, the *run()* function of our *MapTask (PhoenixMapTask),* is called, and the corresponding task (indexed by a mapId) is completed.

A snippet of our new code is shown below:

```
while (read.ready()) {
    line = read.readLine();
//  System.out.println(line);
    context.write((KEYOUT)new
    Text(line.split("\\s+")[0]),
    (VALUEOUT)new IntWritable(Integer.parseInt(line.split("\\s+")[1])))
}
```

7.1 The Job class

The *Job* class allows the user to configure a job, submit it, control the execution, and view its state. Basically, there are bunch of *set()* methods that need to be called by the user, and then once the job is submitted, the user can wait for the completion of the job while viewing the state at any point of time. From the user point of view, he must

- Create the application defining the *map()* and *reduce()* tasks.
- Describe various aspects of the job via *Job.set()* commands.
- Monitor its progress.

An example is shown :

```
Configuration conf = new Configuration();
Job job = new Job(conf, "Average");

job.setOutputKeyClass(Text.class);
job.setOutputValueClass(Text.class);

job.setMapperClass(Map.class);
job.setReducerClass(Reduce.class);

job.setInputFormatClass(TextInputFormat.class);
job.setOutputFormatClass(TextOutputFormat.class);

FileInputFormat.addInputPath(job, new Path(args[0]));
FileOutputFormat.setOutputPath(job, new Path(args[1]));

job.waitForCompletion(true);
```

8 WordCount

We wrote our implementations of wordcount in phoenix as well as hadoop. Here is a detailed explanation of the same:

8.1 In Hadoop

WordCount example reads text files and counts how often words occur. The input is text files and the output is text files, each line of which contains a word and the count of how often it occured, separated by a tab.

Each mapper takes a line as input and breaks it into words. It then emits a key/value pair of the word and 1. Each reducer sums the counts for each word and emits a single key/value with the word and sum.

As an optimization, the reducer is also used as a combiner on the map outputs. This reduces the amount of data sent across the network by combining each word into a single record.

```
public class WordCountMapper extends
    Mapper<LongWritable, Text, Text, IntWritable> {

    //hadoop supported data types
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    //map method that performs the tokenizer job
    //framing the initial key value pairs
    public void map(LongWritable key, Text value,
       OutputCollector<Text, IntWritable> output, Reporter reporter)
       throws IOException {

    //taking one line at a time and tokenizing the same
        String line = value.toString();
        StringTokenizer tokenizer = new StringTokenizer(line);

    //iterating through all the words available in that line
    //forming the key value pair
        while (tokenizer.hasMoreTokens())  {
            word.set(tokenizer.nextToken());

        //sending to output collector which inturn
        //passes the same to reducer
        output.collect(word, one);
        }
    }
}

public class WordCountReducer extends
     Reducer<Text, IntWritable, Text, IntWritable> {
     //reduce method accepts the Key Value pairs from mappers,
     //do the aggregation based on keys and produce the final out put
     public void reduce(Text key, Iterator<IntWritable> values,
           OutputCollector<Text, IntWritable> output, Reporter reporter)
               throws IOException       {
              int sum = 0;

     /*iterates through all the values available with a key
```

```
          and add them together and give the
           final result as the key and sum of its values*/
     while (values.hasNext())   {
             sum += values.next().get();
     }
     output.collect(key, new IntWritable(sum));
     }
}
```

8.2 In Phoenix

We used an implementation of wordcount and modified to suit to our requirements. We modified 'map_reduce.h' to block the reduce and merge functions and print the output of mapper only. This output is then feeded back to the PhoenixMapper.java which appropriately writes the key value in context. This context is subsequently passed to the reduce function.

```
/*  // Run reduce tasks and get final values
    get_time (begin);
    run_reduce();
    print_time_elapsed("reduce phase", begin);
    get_time (begin);
    run_merge();
    print_time_elapsed("merge phase", begin);
    result.swap(*this->final_vals);
    // Delete structures
    delete [] this->final_vals;
    print_time_elapsed("run time", run_begin); */
```

The above code is the code that was commented out from Phoenix's mapreduce.h file. Basically, the idea is to stop the program at the mapper phase, and obtain the output from there itself. Some amount of code for debugging was also added in the emit_intermediate function.

```
void emit_intermediate(typename container_type::input_type& i,
            key_type const& k, value_type const& v)
const {
    i[k].add(v);
    printf("%s\t%d\n", k, v);
}
```

8.3 Results

We ran the word count program twice, once using only Hadoop's inbuilt Mapper, and once with our newly created PhoenixMapper (only). This was done for the sake of benchmarking. The results are as follows

	Hadoop Mapper	**PhoenixMapper**
Execution Time(s)	3.081	1.991

9 Conclusion

This project gave us a good insight into the detailed world of large scale software development. We learnt the intricacies involved in adding features (even small ones) to an existing software base. We were able to contribute to the Hadoop source code and provide a custom Mapper implementation, to support shared memory mapreduce implementations. We thank the TAs for their continuous support, and also Professor Janakiram for giving us this project.

References

[1] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[2] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE, 2010.

[3] Jeff A Stuart and John D Owens. Multi-gpu mapreduce on gpu clusters. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 1068–1079. IEEE, 2011.

[4] Justin Talbot, Richard M Yoo, and Christos Kozyrakis. Phoenix++: modular mapreduce for shared-memory systems. In *Proceedings of the second international workshop on MapReduce and its applications*, pages 9–16. ACM, 2011.

[5] Richard M Yoo, Anthony Romano, and Christos Kozyrakis. Phoenix rebirth: Scalable mapreduce on a large-scale shared-memory system. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 198–207. IEEE, 2009.