Parallel Data Processing with MapReduce in Python

Arjun Sahilkumar Shah
Computer Science Department
San Jose State University
San Jose, CA 95192
408-854-0614
arjunshah2511@gmail.com

ABSTRACT

A prominent parallel data processing tool MapReduce is gaining significant momentum from both industry and academia as the volume of data to analyze grows in leaps and bounds. The MapReduce framework has emerged as one of the most widely used parallel computing platforms for processing data on large data scales like terabytes and petabytes. This study intends to assist the database and open source communities in understanding technical aspects of MapReduce framework. We propose a model of efficient computation using the MapReduce paradigm. Although Python is not a functional programming language, it has built-in support for both of these concepts of Map and Reduce. The paper will characterize the MapReduce and discuss its inherent pros and cons. It also discusses the open issues and challenges raised on parallel data analysis with Map Reduce.

1. INTRODUCTION

In this age of data explosion, parallel processing is essential to processing a massive volume of data in a timely manner. MapReduce is a software framework created and patented by Google to handle the massive computational requirements for their search engine and associated analysis processes. Google's goal was to create a system where all that computational horsepower could be easily utilized for a wide variety of tasks that needed to be distributed across many, many computers. It is a scalable and fault-tolerant data processing tool that enables to process a massive volume of data in parallel with many low-end computing nodes. The strength of these distributed computation techniques is in their scalability, their ability to handle extremely large and growing computations fast and easily. However, MapReduce has inherent limitations on its performance and efficiency. Therefore, many studies have attempted to overcome these limitations.

The goal of this paper is to give an overview of major approaches of MapReduce framework and classify them based on their strategies. Section 2 reviews the design and the important concepts of MapReduce. Section 3 discusses the advantages and disadvantages of MapReduce. Section 4 gives the simple implementation of MapReduce. Section 5 gives detailed explanation of multiprocessing module in Python.. Finally, Section 6 concludes this paper.

2. DESIGN

MapReduce is a programming model and an associated implementation for processing and generating large data sets with a parallel, distributed algorithm. MapReduce framework consists of two phases: **map** and **reduce**. A map function processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function merges all the intermediate values that are associated with the same intermediate key. The Mapper reads data in the form of key/value pairs. If the Mapper writes anything out, the output must be in the form of key/value pairs. After the map phase is over, all intermediate values for a given intermediate key are combined together into a list. This list is given to a Reducer. There may be a single Reducer, or multiple Reducers. All values associated with a particular intermediate key are guaranteed to go to the same Reducer. The intermediate keys, and their value lists, are passed to the Reducer in sorted key order. This step is known as the shuffle and sort. The Reducer outputs zero or more final key/value pairs. In practice, the Reducer usually emits a single key/value pair for each input key.

MapReduce utilizes the Google File System(GFS) as an underlying storage layer to read input and store output. Hadoop is an open-source project overseen by the Apache Software Foundation. It is originally based on the papers published by Google in 2003 and 2004. Since GFS is a proprietary distributed file system developed by Google for its own use we will see Hadoop in more detail. Hadoop consists of two core components: The Hadoop Distributed File System (HDFS) and MapReduce. HDFS stores the data on the cluster, while MapReduce processes the data on the cluster. In HDFS, data is split into blocks and distributed across multiple nodes in the cluster. Each block is typically 64 MB or 128 MB in size. Different blocks from the same file will be stored on different machines. This provides for efficient MapReduce processing. Blocks are replicated across multiple machines, known as **DataNodes**. The default replication is a three-fold which means that each block exists on three different machines. This is followed in order to guarantee fault-tolerance. A master node called the **NameNode** keeps track of which blocks make up a file, and where those blocks are located. This is also know as the metadata, i.e. NameNode stores the metadata.

MapReduce can be broken down into two separate processes, the "Map" stage and the "Reduce" stage. In python, the data is presented to MapReduce as a dictionary of key. Figure 1 illustrates an overview of the Map Reduce structure in Hadoop architecture. Figure 2 shows a more detailed process of the MapReduce framework.

2.1 Map Stage

Every key/value pair in the input data dictionary is sent to a different instance of the same map function. These map functions can be spread over multiple machines or processors. In real, the role of the key in the input dictionary is to identify the data (the associated value) to be processed by a single map instance. The map function is defined as outputting zero or more key/value pairs where the output key does not necessarily correspond to the input key. There are possibilities where different map instances may output the same key values. One of the key things is to recognize that outputting values when processes are spread over various computers spread all over the world is not as simple as returning a value as its done in a Python function. A special output function must be handed to the map function in order to output the values. This also resolves the issue of outputting multiple key/value pairs while in the middle of the map process. The resulting key/value pairs are stored in an intermediate data dictionary storage for its use in the reduce stage. Since multiple map function instances may result into values for the same key value in the intermediate data dictionary, those multiple values are stored as a list of values for each key.

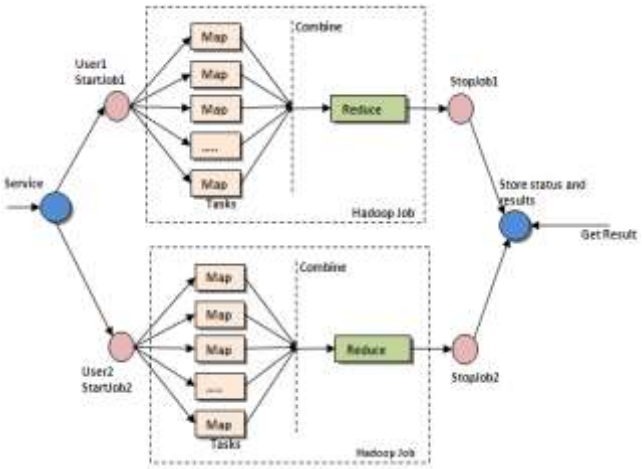

Figure 1. MapReduce in Hadoop architecture

2.2 Reduce Stage

Surprisingly, the reduce stage process is theoretically identical to the map stage process. Instead of using the original input data dictionary, the reduce stage uses the intermediate data dictionary which is produced by the map stage. Similar to the map stage, multiple identical reduce function occurrences are spread across various machines and each reduce function receives one of the key/value pairs from the intermediate data dictionary. The reduce function also outputs the key/value pairs. However, the role of the reduce function is to reduce/decrease the large list of values associated with the key to a single or just a few resultant values. Usually, the output key from the reduce function is same as its input key.

The MapReduce framework follows the runtime scheduling scheme. It means that it does not have a specific execution plan that states which tasks will run on which nodes before the execution. A plan for executions in MapReduce is determined entirely at runtime. MapReduce achieves fault tolerance by detecting failures and reassigning tasks of failed nodes to other healthy nodes with the help of run time scheduling. The nodes which are already done with their assigned tasks are reassigned another input block. In this way, the faster nodes will process more input blocks and the slower nodes process less inputs managing load balancing. MapReduce makes use of a speculative execution. For example if one node has a slow disk controller, then it may be reading its input at only 10% the speed of all the other nodes. So when 99 map tasks are already complete, the system is still waiting for the final map task to check in, which takes much longer than all the other nodes.

By forcing tasks to run in isolation from one another, individual tasks do not know the origin of the input. The same input can be processed multiple times in parallel, to exploit differences in machine capabilities. When majority of the tasks in a job are about to finish, the Hadoop platform will schedule redundant copies of the remaining tasks across several nodes which do not have other work and are idle. This process is known as speculative execution.

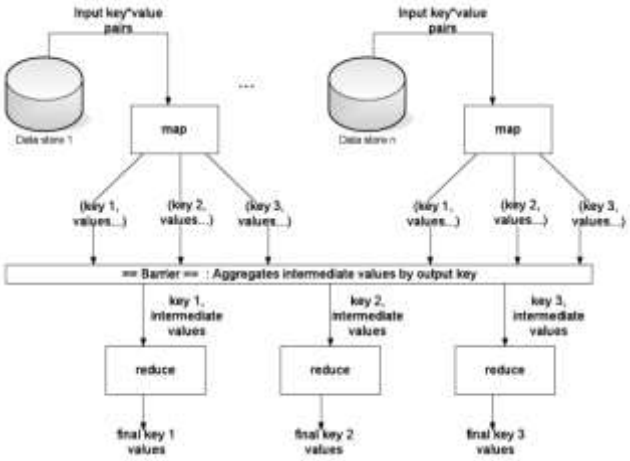

Figure 2. MapReduce computation

Hadoop's scheduler makes several implicit assumptions: Nodes can perform work at roughly the same rate. Tasks progress at a constant rate throughout the time. There is no cost to launching a speculative task on a node that would otherwise have an idle slot. A task's progress score is representative of fraction of its total work that it has done. Specifically, in a reduce task, the copy, sort and reduce phases each take about 1/3 of the total time. Tasks tend to finish in waves, so a task with a low progress score is likely to be considered as a staggler. Tasks in the same category require roughly the same amount of work.

3. ADVANTAGES AND DISADVANTAGES

3.1 Pros

MapReduce is a massively scalable, parallel processing framework that works in tandem with HDFS. MapReduce

processes exceedingly large amounts of data without being affected by traditional bottlenecks like network bandwidth by taking advantage of the data proximity.

Simplicity: Applications can be written by the developers in the language of their choice, such as Java, C++ or Python.. MapReduce jobs are easy to run.

Scalability: MapReduce is capable of processing petabytes of data that in stored in HDFS on one cluster. It can also add servers to increase the processing power.

Speed: Hadoop reduces the time to solve problems by a significant amount from days to hours or minutes.

Flexibility: MapReduce not depend on any schema or data model. A programmer can deal with irregular or unstructured data more easily than they do with DBMS.

Resiliency and High Availability: Multiple job and task trackers ensure that jobs which fail halt the process and restart automatically.

Independent of the storage: MapReduce is independent from underlying storage layers. Hence it can work with different storage layers.

Security and Authentication: MapReduce works with HDFS and HBase security in order to make sure that only authorized users can access and work against the data in the system.

3.2 Cons

MapReduce lacks some of the features and there are certain cases where mapreduce is not a suitable choice. Some of them are:

When you are dealing with Real-time processing. When the immediate processes need to talk to each other. When the processing requires a lot of data to be shuffled over the network.
When streaming of data needs to be handled. When desired result can be achieved with a standalone system. It is obviously less painstaking to configure and manage a standalone system as compared to a distributed system. When Online Transaction Processing(OLTP) is needed. MapReduce is not a suitable choice for a large number of short online transactions.

4. MAPREDUCE IMPLEMENTATION

At the highest level, a MapReduce system takes in 3 inputs: a data dictionary, a map function and a reduce function. The system inturn divides the data across multiple instances of the map and reduce functions and then returns a dictionary of results. The system basically breaks down into 3 pieces:
FuncThread class: This class is for creating specialized Thread objects that hold the necessary entities for either a map or reduce operation, which map or reduce function to use, the key/value pair to process and the output function to use.
runThreads() function: This function uses the abstraction for the map and reduce processes so that the same code can be used for them. This function creates a FuncThread object for every key/value pair in the given dictionary and starts each one of those FuncThread objects, causing the given function to run on its own thread. runThreads() provides an output function to the given map/reduce function that combines all the outputs into a dictionary whose values are lists of the outputted values. runThreads() prints out the number of map and reduce instances(threads) it is using at once.
mapReduce() function: This function is actually used to run the MapReduce process but it also calls the runThreads() twice, once with the input data dictionary and the map function and other with the intermediate data dictionary and the reduce function. The dictionary that is returned from the reduce process is the resultant dictionary.

The above functions does serve its purpose of correctly executing map and reduce functions on a given dataset but Python handling threads means that the map and reduce function instances do not execute in parallel. This is due to the global interpreter lock used in the Python interpreter. The GIL handicaps parallelism in the Python interpreter, by serializing execution of user-threads for various complicated reasons. In order to overcome this, Python multiprocessing module is used which creates separate processes to execute independently

5. MULTI-THREADING VS MULTI-PROCESSING

There are two approaches in parallel programming to run code depending on the application. One is via threads and other through multiple processes. Submitting jobs to different threads can be pictured as sub-tasks of a single process and those threads will usually have the access to the same memory, i.e. shared memory. In case of improper synchronization this can lead to conflicts.
A better approach is to submit multiple processes to completely separate memory locations, i.e. distributed memory. In this case each process will run completely independent of each other.

5.1 Multiprocessing Module

Python's Standard Library contains the multiprocessing module which has lot of powerful features. This paper mentions two different approaches from the multiprocessing module.

5.1.1 The Process class

The most basic approach is to use the process class from the multiprocessing module. Processes are spawned by creating a Process object and then calling its start() method. Process follows the API of threading.Thread. Depending on the platform, multiprocessing supports three ways to start a process which are:
spawn- The parent process starts a new python interpreter process. The child process will inherit only those resources necessary to run the process objects run() method. Using this method to start a process is slow compared to fork or forkserver.
fork- The parent process uses os.fork() to fork the Python interpreter. When the child process begins, it is effectively identical to the parent process. All resources of the parent are inherited by the child process.
forkserver- When the program starts and selects the forkserver start method, a server process is started. So, whenever a new process is needed, the parent process connects to the server and requests that it fork a new process.
To select a start method one can use the set_start_method(). One can also use the get_context() method to obtain a context object.

Context objects have the same API as the multiprocessing module, and allow one to use multiple start methods in the same program. Multiprocessing supports two types of communication channel between processes: Queues – They are thread and process safe. Pipes – The pipe() function returns a pair of connection objects connected by a pipe which by default is duplex(two-way). Multiprocessing contains equivalents of all the synchronization primitives from threading. For example, one can use a lock to make sure that only one process prints to standard output at a time.

5.1.2 The Pool class

Another and more convenient way for simple parallel-processing tasks is provided by the Pool class. Python provides a easy-to-use Pool class to manage our own worker process pool. Probably the best thing about the Pool class is that it provides a map function of its own which automatically partitions and distributes input to a user-specified function pool of worker processes. There are four methods that can be useful in the Pool class and they are : **Pool.apply**, **Pool.map**, **Pool.apply_async** and **Pool.map_async**.

The detailed description of each functions is as follows: **apply**(func[, args[, kwds]])- Calls the func with arguments kwds. It blocks until the result is ready. When the blocks are given, apply_async() works better compared to this function. Also, func is only executed in one of the workers of the pool.

apply_async(func[, args[, kwds[, callback[, error_callback]]]])- This is a variant of apply() method which returns a result object. If callback is specified then it should be a callable which accepts a single argument. Callback is applied when the result is ready otherwise error_callback is applied when the call fails. If error_callback is specified then it should be a callable which accepts a single argument. If the target function fails, then the error_callback is called with the exception instance. Callback should be completed as soon as possible otherwise the thread which handles the result will get blocked.

map(func, iterable[, chunksize]) – This is equivalent of the in-built map function. It blocks until the result is ready. This method divides the iterable into a number of chunks and then submits to the process pool as separate tasks. The size of these chunks can be specified by setting the chunksize to a positive integer.

map_async(func, iterable[, chunksize[, callback[, error_callback]]])- This is a variant of the map() method which returns a result object. If callback is specified then it should be a callable which accepts a single argument. Callback is applied when the result is ready otherwise error_callback is applied when the call fails. If error_callback is specified then it should be a callable which accepts a single argument. If the target function fails, then the error_callback is called with the exception instance. Callback should be completed as soon as possible otherwise the thread which handles the result will get blocked.

The Pool.apply and Pool.map methods are equivalent to Pythons in-built apply and map functions. For example, if we set the number of processes to 4, it means that the Pool class will allow only 4 processes running at the same time. Consider a simple square function that returns a square of an input number.

```
def square(x):
    return x**2
pool = mp.Pool(processes=4)
results = [pool.apply(square,args=(x,)) for x in range(1,7)]
print results
Ans. [1,4,9,16,25,36]

pool = mp.Pool(processes=4)
results = pool.map(square,range(1,7))
print results
Ans. [1,4,9,16,25,36]
```

The Pool.map and Pool.apply will lock the main program until a process has finished and this feature is useful when we want to get the results in a particular order. In contrast, the async variants will submit all processes at once and retrieve the results as soon as they are finished. For example, the resulting sequence of squares could have been [4,1,16,9,25,36] if the square processes finished in that order. Other difference is that we need to make use of the get() method after the apply_async() call in order to obtain the return values of the finished processes. Considering the same square example, the code snippet would look something like this:

```
pool = mp.Pool(processes = 4)
results = [pool.apply_async(square,args=(x,)) for x in range(1,7)]
output = [p.get() for p in results]
print output
Ans. [1,4,9,16,25,36]
```

The MapReduce discussion is a great simplication of actual MapReduce system employed by Google and other available implementations. The actual system is far more sophisticated and optimized with additional functionality, particularly in the areas of process management and shared data access.

6. CONCLUSION

MapReduce is simple but provides good scalability and fault-tolerance for massive data processing. We discussed the pros and cons of MapReduce and its simple implementation. Making use of the multiprocessing module instead of multithreading in Python is a better way of MapReduce framework. MapReduce shows that many problems can be solved in the model at scale unprecedented before. Due to runtime scheduling with speculative execution, MapReduce reveals low efficiency. Python multiprocessing code is very easy to use. We donot need to organize our problem as a MapReduce problem in order to properly use the process pool class.

7. REFERENCES

[1] J.Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In Communications of the ACM, 51 (1): 107-113, 2008.

[2] J.Dean. Designs, lessons and advice from building large distributed systems. Keynote from LADIS, 2009.

[3] C. Ranger et al. Evaluating mapreduce for multi-core and multiprocessor systems. In Proceedings of the 2007 IEEE HPCA, pages 13-24, 2007.

[4] Multiprocessing- Process based parallelism from Python 3.5, https://docs.python.org/dev/library/multiprocessing.html.

[5] S.Ghemawat et al. The google file system. ACM SIGOPS Operating Systems Review, 37(5):29–43, 2003.

[6] J. Dean et al. MapReduce: Simplified data processing on large clusters. Communication of the ACM, 51(1):107-113, 2008.