Available Online at www.ijcsmc.com

International Journal of Computer Science and Mobile Computing

A Monthly Journal of Computer Science and Information Technology

ISSN 2320–088X

IJCSMC, Vol. 4, Issue. 11, November 2015, pg.181 – 186

SURVEY ARTICLE

A Survey on MapReduce Performance and Hadoop Acceleration

Ms. Deshmukh M.R, Mrs. Raut P.Y
Department of Computer Engineering & Pune University
Pravara Rural Engineering College, Loni, Pune, Maharashtra, India
monalidshmkh@gmail.com
getdiya2008@gmail.com

***Abstract*— MapReduce is implementation for generating large data sets with a parallel, distributed algorithm on a cluster. Hadoop is open source implementation of the MapReduce programming data-model used for large-scale parallel applications such as web indexing, data mining, and scientific simulation. Hadoop-A framework is able to levitate Hadoop acceleration and give significant performance compared to existing Hadoop framework. A new scheduling algorithm, Longest Approximate Time to End (LATE), that is highly robust to heterogeneity. There are factors which will make Hadoop more reliable scalable and efficient. With extensive survey we have gone through various papers and studied mechanism for improving performance of MapReduce architecture.**

Keywords—component; formatting; style; styling; insert (key words)

I. INTRODUCTION

Apache Hadoop is an open-source software framework used for distributed processing and distributed storage of big data sets on cluster computers. Hadoop achieves high performance using a divide-and-conquer approach that distributes the application processing across servers [1]. Parallel processing technique gives benefit for applications that sequentially process large data files. In Hadoop all the modules are designed with a fundamental assumption that hardware failures should be automatically handled in software by the framework. Hadoop distributions consist of Hadoop Distributed File System (HDFS) and MapReduce programming model. Hadoop may have various distributions one of these are Pig and Hive these components provide high level interfaces to HDFS.

The MapReduce model popularized by Google is very attractive for ad-hoc parallel processing of arbitrary data. MapReduce breaks a computation into small tasks that run in parallel on multiple machines, and scales easily to very large clusters of inexpensive commodity computers. A key benefit of MapReduce is that it automatically handles failures, hiding the complexity of fault-tolerance from the programmer. If a node crashes, MapReduce reruns its tasks on a different machine. Hadoop implements MapReduce framework with two categories of components: a Job Tracker and many Task- Trackers. The Job Tracker commands Task Trackers (a.k.a. slaves) to process data in parallel through two main functions: map and reduce. In this process, the Job Tracker is in charge of scheduling map tasks (Map Tasks) and reduce tasks (Reduce Tasks) to Task Trackers. It also monitors their progress, collects runtime execution statistics, and handles possible faults and errors through task re-execution. Between the two phases, a Reduce Task needs to fetch a part of the intermediate output from all finished MapTasks. Globally, this leads to the shuffling of intermediate data (in segments) from all Map Tasks to all Reduce Tasks.

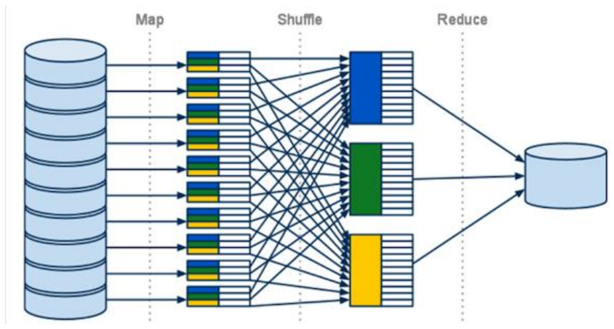

Fig. 1. MapReduce Operations.[2]

Fig. 1 shows general operation happening inside MapReduce operations. After map operation shuffling is carried out and then Reduce operation takes place. After a brief initialization period, a pool of concurrent MapTasks starts the map function on the first set of data splits. As soon as Map Output Files (MOFs) are generated from these splits, a pool of ReduceTasks starts to fetch partitions from these MOFs. At each ReduceTask, when the number of segments is larger than a threshold, or when their total data size is more than a memory threshold, the smallest segments are merged.

In following section we have gone through different research papers on MapReduce and Hadoop performance issues.

II. RELATED WORK

Weikuan Yu et al. [3] reveal that the original architecture faces a number of challenging issues to exploit the best performance from the underlying system. To ensure the correctness of MapReduce, no ReduceTasks can start reducing data until all intermediate data have been merged together. This results in a serialization barrier that significantly delays the reduce operation of ReduceTasks. More importantly, the current merge algorithm in Hadoop merges intermediate data segments from MapTasks when the number of available segments (including those that are already merged) goes over a threshold. These segments are spilled to local disk storage when their total size is bigger than the available memory. This algorithm causes data segments to be merged repetitively and, therefore, multiple rounds of disk accesses of the same data.

To address these critical issues for Hadoop MapReduce framework, Hadoop-A is designed, a portable acceleration framework that can take advantage of plug-in components for performance enhancement and protocol optimizations. Several enhancements are introduced:

1) A novel algorithm that enables ReduceTasks to perform data merging without repetitive merges and extra disk accesses;

2) A full pipeline is designed to overlap the shuffle, merge, and reduce phases for ReduceTasks; and

3) A portable implementation of Hadoop-A that can support both TCP/IP and remote direct memory access (RDMA).

Since ReduceTasks are able to merge data by staying above local disks, Authors refer to this new algorithm as network-levitated merge (NLM). An extensive set of experiments to evaluate the performance of Hadoop-A and evaluation demonstrates that the network-levitated merge algorithm is able to remove the serialization barrier and effectively overlap data merge and reduce operations for Hadoop ReduceTasks. Overall, Hadoop-A is able to double the throughput of Hadoop data processing.

Characterization and analysis reveal a number of issues, including

1) Serialization between Hadoop shuffle/merge and reduces phases,

2) Repetitive merges and disk access,

3) Lack of portability to different interconnects.

A.Network-Leviated Pipeline Of Shuffle, Merge, and Reduce Phases

To address the first two issues in Hadoop a network-levitated merge algorithm that avoids repeated merges and then detail the construction of a new pipeline to eliminate the serialization barrier.

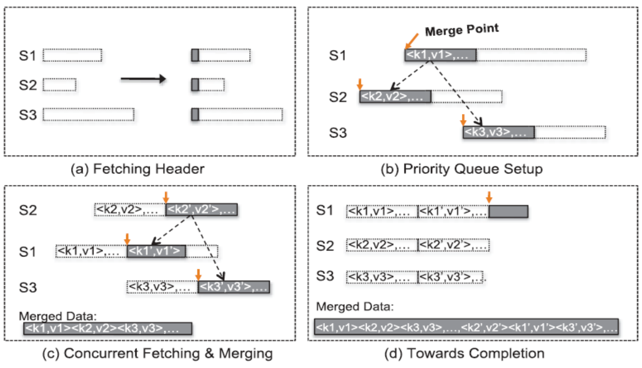
S1
S2
S3
(a) Fetching Header
Merge Point
<k1,v1>,...
<k2,v2>,...
<k3,v3>,...
(b) Priority Queue Setup
<k2,v2>,... <k2',v2'>,...
<k1,v1>,... <k1',v1'>
<k3,v3>,... <k3',v3'>,...
Merged Data:
<k1,v1><k2,v2><k3,v3>,...
(c) Concurrent Fetching & Merging
<k1,v1>,... <k1',v1'>,...
<k2,v2>,... <k2',v2'>,...
<k3,v3>,... <k3',v3'>,..
Merged Data:
<k1,v1><k2,v2><k3,v3>,...,<k2',v2'><k1',v1'><k3',v3'>,...
(d) Towards Completion

Fig 2. Network-Levitated Merge Algorithm.[3]

1 Network-Levitated Merge

Hadoop resorts to repetitive merges because of limited memory compared to the size of data. For each remotely completed MOF, each ReduceTask invokes an HTTP GET request to query the partition length, pull the entire data, and store locally in memory or on disk. This incurs many memory loads/stores and/or disk I/O operations. We design an algorithm that can merge all data partitions exactly once and, at the same time, stay levitated above local disks. Fig. 2 shows our network-levitated merge algorithm.

The key idea is to leave data on remote disks until it is time to merge the intended data records. As shown in Fig. 2a, three remote segments S1, S2, and S3 are to be fetched and merged. Instead of fetching them to local disks, our new algorithm only fetches a small header from each segment. Each header is especially constructed to contain partition length, offset, and the first pair of <key,val>. These <key,val> pairs are sufficient to construct a priority queue (PQ) to organize these segments. More records after the first <key,val> pair can be fetched as allowed by the available memory. Because it fetches only a small amount of data per segment, this algorithm does not have to store or merge segments onto local disks.

Instead of merging segments when the number of segments is over a threshold, keep building up the PQ until all headers arrive and are integrated. As soon as the PQ has been set up, the merge phase starts. The leading <key, Val> pair will be the beginning point of merge operations for individual segments, i.e., the merge point. This is shown in Fig. 2b. Our algorithm merges the available <key, Val> pairs in the same way as is done in Hadoop. When the PQ is completely established, the root of the PQ is the first <key, Val> pair among all segments. Extract the root pair as the first <key, Val> in the final merged data. Then update the order of PQ based on the first <key, Val> pairs of all segments. The next root will be the first <key, Val> among all remaining segments. It will be extracted again and stored to the final merged data.

When the available data records in a segment are depleted, our algorithm can fetch the next set of records to resume the merge operation. In fact, our algorithm always ensures that the fetching of upcoming records happens concurrently with the merging of available records. As shown in Fig. 2c, the headers of all three segments are safely merged; more data records are fetched, and the merge points are relocated accordingly. Concurrent data fetching and merging continues until all records are merged. All <key,val> records are merged exactly once and stored as part of the merged results. Fig. 2d shows a possible state of the three segments when their merge completes. Since the merge data have the final order for all records, we can safely deliver the available data to the Java-side ReduceTask where it is then consumed by the reduce function.

2 Pipelined Shuffle, Merge, and Reduce

Besides avoiding repetitive merges, algorithm removes the serialization barrier between merge and reduce. The merged data have <key, Val> pairs ordered in their final order and can be delivered to the Java-side ReduceTask as soon as they are available. Thus, the reduce phase no longer has to wait until the end of the merge phase.

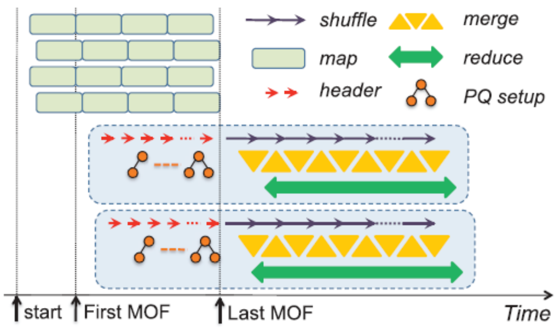

Fig 3. Pipelined shuffle, merge, and reduce.[3]

In view of the possibility to closely couple the shuffle, merge, and reduce phases, they can form a full pipeline as shown in Fig. 3. In this pipeline, MapTasks map data split as soon as they can. When the first MOF is available, ReduceTasks fetch the headers and build up the PQ. These activities are pipelined. Header fetching and PQ setup are pipelined and overlapped with the map function, but they are very lightweight, compared to shuffle and merge operations.

As soon as the last MOF is available, completed PQs are constructed. The full pipeline of shuffle, merges, and reduces then starts. One may notice that there is still a serialization between the availability of the last MOF and the beginning of this pipeline. This is inevitable in order for Hadoop to conform to the correctness of the MapReduce programming model. Simply stated, before all <key, Val> pairs are available, it is erroneous to send any <key, Val> pair to the reduce function (for final results because its relative order with future <key, Val> pairs is yet to be decided. Therefore, our pipeline is able to shuffle, merge, and reduce data records as soon as all MOFs are available. This eliminates the previous serialization barrier in Hadoop and allows intermediate results to be reduced as soon a possible for final results.

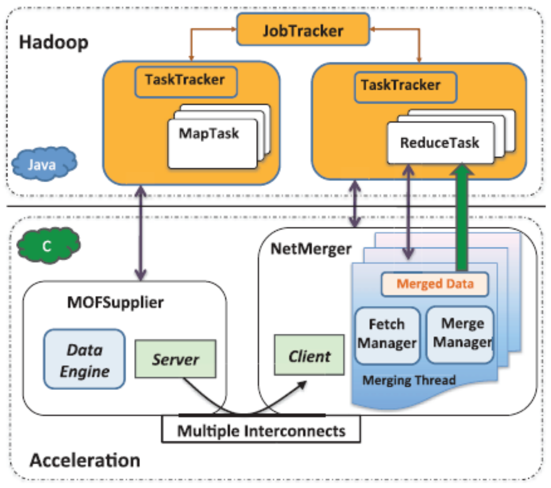

Fig 4. Software architecture of Hadoop-A.[3]

Fig. 4 shows the architecture of Hadoop-A. Two new user configurable plug-in components, MOF Supplier and Net-Merger, are introduced to leverage RDMA-capable interconnects and enable alternative data merge algorithms. Both MO Supplier and Net Merger are threaded C implementations. The choice of C over Java is to avoid the overhead of the Java Virtual Machine (JVM) in data processing and allow flexible choice of new connection mechanisms such as RDMA, which is not yet available in Java.

A primary requirement of Hadoop-A is to maintain the same programming and control interfaces for users. To this end, we design the MOF Supplier and Net Merger plugins as native C programs that can be launched by Task Trackers. A user can choose to enable or disable the acceleration, which is controlled by a parameter in the configuration file. Hadoop programs can run without any change when the Hadoop-A plug-in is activated.

Dawei Jiang et al [4], considered the factors affecting MapReduce performance which consist of impact of the architectural design of MapReduce, including programming model, storage-independent design and scheduling. In particular, authors identified factors that affect the performance of MapReduce: I/O mode, indexing, data parsing, group in schemes and block-level scheduling. Authors also considered negative impact of these factors by proper implementation, and observed what will be the performance improvement.

Authors conducted a performance profile for MapReduce. Started analysis by reviewing the MapReduce programming model and the execution flow of a MapReduce job. Then, from an architectural perspective, Following are the design factors may hurt the performance.

a. MapReduce programming model: The programming model itself does not specify how intermediate pairs produced by map() functions are grouped for reduce() functions to process.

b. Storage independence: storage independent design is considered to be beneficial for heterogeneous systems since it enables MapReduce to analyze data stored in different storage systems [5].

c. Scheduling: the runtime scheduling strategy enables MapReduce to over elastic scalability, namely the ability of dynamically adjusting resources during job execution.

Different possible combinations of the above factors result in a huge search space so limit the performance evaluation of MapReduce on two kinds of storage systems, a distributed file system (namely HDFS) and a database system (namely Postgre SQL). With certain number of benchmarking authors have measured the performance. benchmarking entails the evaluation of seven tasks which are carried out under Amazon EC[6] are as follows Scheduling Micro Benchmark, Comparison of Different I/O Modes, Record Parsing MicroBenchmark, Grep Task, Selection Task, Aggregation Task, Join Task

Hadoop's scheduler starts speculative tasks based on a simple heuristic comparing each task's progress to the average progress. Although this heuristic works well in homogeneous environments where stragglers are obvious, but it can lead to severe performance degradation when its underlying assumptions are broken. An especially compelling environment where Hadoop's scheduler is inadequate is a virtualized data centre. Virtualized "utility computing" environments, such as Amazon's Elastic Compute Cloud (EC2) [6], are becoming an important tool for organizations that must process large amounts of data, because large numbers of virtual machines can be rented by the hour at lower costs than operating a data centre year-round. So there is necessity to design an improved scheduling algorithm that reduces Hadoop's response time.

Matei Zaharia et al [7], designed a simple algorithm for speculative execution that is robust to heterogeneity and highly effective in practice. The algorithm is called LATE for Longest Approximate Time to End. LATE is based on three principles: prioritizing tasks to speculate, selecting fast nodes to run on, and capping speculative tasks to prevent thrashing. LATE can improve the response time of MapReduce jobs by a factor of 2 in large clusters on EC2.

LATE algorithm works as follows: If a node asks for a new task and there are fewer than Speculative Cap speculative tasks running:

1. Ignore the request if the node's total progress
is below Slow Node Threshold.
2. Rank currently running tasks that are not currently being speculated by estimated time left.
3. Launch a copy of the highest-ranked task with progress rate below Slow Task Threshold.

Like Hadoop's scheduler, we also wait until a task has run for 1 minute before evaluating it for speculation. LATE does not take into account data locality for launching speculative map tasks, although this is a potential extension. The LATE algorithm has several advantages. First, it is robust to node heterogeneity, because it will relaunch only the slowest tasks, and only a small number of tasks. LATE prioritizes among the slow tasks based on how much they hurt job response time. LATE also caps the number of speculative tasks to limit contention for shared resources. LATE takes into account node heterogeneity when deciding where to run speculative tasks. LATE speculatively executes only tasks that will improve job response time, rather than any slow tasks.

III. CONCLUSION

In this paper we briefly described about the MapReduce and Hadoop performance and methods to improve performance.
There are several critical issues faced by the existing Hadoop implementation, including its merge algorithm, its pipeline of shuffle, merge, and reduce phases, as well as its lack of portability for multiple interconnects. We have studied Hadoop-A as an extensible acceleration framework that can allow plug-in components to address all these issues. We also gone through network-levitated merge algorithm, it can significantly reduce disk accesses during Hadoop's shuffling and merging phases. A simple, robust scheduling algorithm, LATE, which uses estimated finish times to speculatively execute the tasks that hurt the response time the most. We have studied factors that affect the performance of MapReduce and investigated alternative implementation strategies for each factor.

References

I. Colin White, "Gaining Value From Big Data: Integrating Relational Systems with Hadoop," BI Research, May 2013.

II. http://aimotion.blogspot.in/2012/08/introduction-to-recommendations-with.html.

III. Weikuan Yu, Yandong Wang, Xinyu Que, "Design and Evaluation of Network-Levitated Merge for Hadoop Acceleration," IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, VOL. 25, NO. 3, MARCH 2014.

IV. Dawei Jiang, Beng Chin Ooi, Lei Shi, Sai Wu, "The Performance of MapReduce: An Indepth Study," The 36th International Conference on Very Large Data Bases, September 1317, 2010, Singapore.

V. G. E. Blelloch, L. Dagum, S. J. Smith, K. Thearling, M. Zagha. An evaluation of sorting as a supercomputer benchmark. NASA Technical Reports, Jan 1993.

VI. Amazon Elastic Compute Cloud, http://aws. amazon.com/ec2.

VII. Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, Ion Stoica,"Improving MapReduce Performance in Heterogeneous Environments," 8th USENIX Symposium on Operating Systems Design and Implementation.