International Journal of Science and Research (IJSR)
ISSN (Online): 2319-7064
Index Copernicus Value (2013): 6.14 | Impact Factor (2013): 4.438

A Survey on Hierarchical Merge for Hadoop-A

Vaibhav Dhore[1], Sonali R. Jagtap[2]

[1]Assistant Professor,Department of Computer Engineering, RMD Sinhgad School of Engineering, University of Pune, India

[2]Department of Computer Engineering, RMD Sinhgad School of Engineering, University of Pune, India

Abstract: ***Hadoop is a popular open source implementation of the Map Reduce programming model for cloud computing. However, it faces a number of issues to achieve the best performance from the underlying systems. These include a serialization barrier that delays the reduce phase, repetitive merges, and disk accesses, and the lack of portability to different interconnects. We describe Hadoop-A, an acceleration framework that optimizes Hadoop with plug-in components for fast data movement, overcoming the existing limitations. A Hierarchical merge algorithm is introduced to merge data without repetition and disk access. In addition, we are using virtual shuffling to reduce disk access.***

Keywords: Hadoop, MapReduce, virtual shuffling, hierarchical merge, Hadoop acceleration

1. Introduction

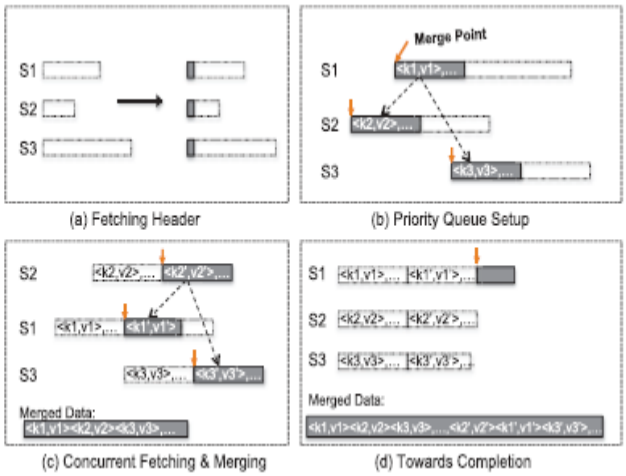
Today's era of a Big Data processing explosive amounts of data in a scalable, reliable and effcient manner to mine critical knowledge for human intelligence is becoming one of the most important challenges. For example, AT&T currently processes close to 20 petabytes of data every 24 hours, and Google processes more than 1 petabytes of information every hour [11]. To be able to process data-intensive analysis in a scalable and fault-tolerant manner in a distributed environment, Google introduced a distributed and parallel programming model called MapReduce [12]. Due to its ease of programming, scalability, especially highly fault-tolerant and applicability on low-cost hardware MapReduce paradigm has become the favor of many commercial enterprises such as web crawling, fnancial services and telecommunications.

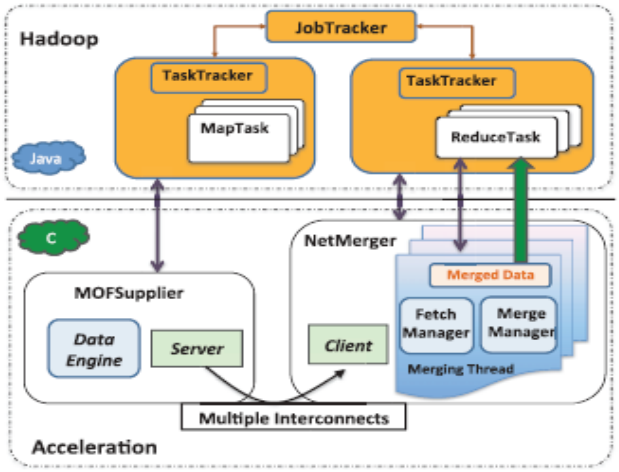
Hadoop is an open-source implementation of MapReduce, supported by leading IT companies such as Google and Yahoo!, and widely adopted and deployed in industry on several thousands of commodity machines. Hadoop implements MapReduce framework with two categories of components: a JobTracker and many TaskTrackers. TaskTrackers are managed by the JobTracker and launched on each computational node to perform the tasks they receives from JobTracker. Data processing is performed in parallel through two main functions: map and reduce. The JobTracker is in charge of scheduling the map tasks (MapTasks) and reduce tasks (ReduceTasks) to TaskTrackers. It also monitors job progress, collects run-time execution statistics, and handles possible faults and errors through task re-execution.

2. Current Methodology

2.1 Network-Levitated Merge

In network-levitated merging algorithm. the idea is to leave data on remote disks until it is time to merge the intended data records. As shown in Fig. 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, we 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. b. 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. We extract the root pair as the first <key,val> in the final merged data. Then we 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, 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. c, 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. d 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. Further details are available in the following section.





2.2 Queued Shuffle, Merge, and Reduce

In Queue, MapTasks map data split as soon as they can. When the first MAP OUTPUT FILE is available, ReduceTasks fetch the headers and build up the PQ. These activities are Queued. Header fetching and PQ setup are Queued and overlapped with the map function, but they are very lightweight, compared to shuffle and merge operations. As soon as the last MAP OUTPUT FILE is available, completed PQs are constructed. The full Queue of shuffle, merge, and reduce then starts. One may notice that there is still a serialization between the availability of the last MAP OUTPUT FILE and the beginning of this Queue. This is inevitable in order for Hadoop to conform to the correctness of the apReduce 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.

3. Propose Methodology

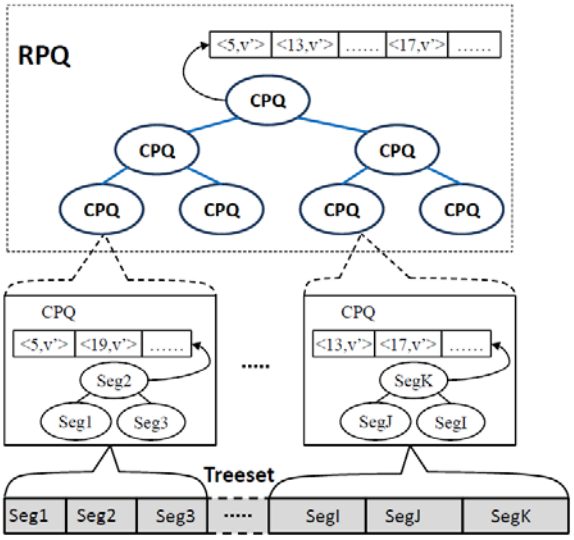
3.1 System Architecture of Hadoop-A

Fig. shows the architecture of Hadoop-A. Two new userconfigurable plug-in components, MAP OUTPUT FILESupplier and Net-Merger, are introduced to leverage RDMA-capable interconnects and enable alternative data merge algorithms. Both MAP OUTPUT FILESupplier and NetMerger 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 MAP OUTPUT FILESupplier and NetMerger plugins as native C programs that can be launched by TaskTrackers. 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.

3.2 Memory Scalability

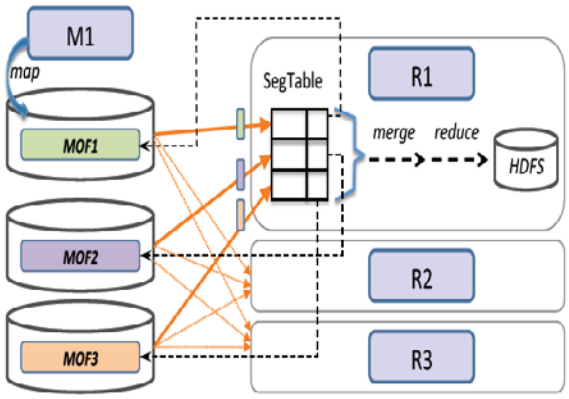
Although network-levitated merge is capable of efficiently supporting most Hadoop production jobs (jobs with _10 GB input make up _92 percent of total jobs),there is a potential issue of coping with extremely large data sets. In the network-levitated merge algorithm, data are fetched from the remote MAP OUTPUT FILE and stored a block before being merged to its staging buffer applications with exascale data sets, the linear growth of memory requirement does not promise good scalability.

For better scalability in the future, a hierarchical merge algorithm is needed to organize memory buffers. Then, we can activate the data shuffling for only one branch of the tree and leave the other branch temporarily inactive, i.e., not holding any data in memory. Fig. shows a general idea with a two-level tree organization. At the very bottom, a linear array (called treeset) is used to sort the incoming segments based on their size. Once the number of segments goes over a threshold, the segments are moved into a leaf priority queue (LPQ). More segments will lead to the creation of more LPQs. After all segments have arrived, the remaining segments in treeset are moved to the last LPQ. All LPQs are then organized into a root priority queue (RPQ), which merges data from LPQs into an dditional staging buffer. The segments are spread into many small LPQs . Virtual shuffling is applied to apReduce through hierarchical merge technique using a two-level hierarchy of priority queues. At the very bottom, alinear array (called treeset) is used to sort the incoming segments based on their size. Once the number of segments goes over a threshold, the segments are moved into a Child Priority Queue (CPQ). More segments will lead to the creation of more CPQs. After all segments have arrived, the remaining segments in treeset are moved to the last CPQ. All CPQs are then organized into a root priority queue (RPQ), which merges data from CPQs into and additional staging buffer. The segments are spread into many small CPQs. The novelty of Hierarchical Merge is to minimize the number of merges down to 2 and yet keep the merging process in memory.



3.3 Virtual Shuffling for Data Movement in Hadoop -A Map Reduce

Virtual shuffling strategy to enable efficient moving of data for Map Reduce programs. Figure shows the general idea. Instead of moving data segments to local disks before starting the reduce function, virtual shuffling allows a Reduce Task to fetch only a minimal set of segment attributes and create a virtual segment table that records the actual locations of remote segments. Virtual shuffling delays the actual movement of data until the Reduce Task requests data to be reduced. At that point, virtual shuffling employs on-demand merging to fetch data in small blocks into memory, merge and send them directly to the reduce function. In doing so, virtual shuffling greatly reduces the number of disk accesses of physical shuffling, and enables efficient data movement.



4. Conclusion

We have examine Hadoop-A as an extensible acceleration framework that can allow plug-in components to address all hadoop issues. By introducing a new hierarchical merge algorithm that merges data without touching disks and designing a full Queue of shuffle, merge, and reduce phases for Reduce Tasks, we have successfully accomplished an accelerated Hadoop framework, Hadoop-A also using virtual shuffling we can reduce disk access .

References

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," Proc. Sixth Symp. Operating System Design and Implementation (OSDI '04), pp. 137-150, Dec. 2004.
[2] Apache Hadoop Project, http://hadoop.apache.org/, 2013.
[3] D. Jiang, B.C. Ooi, L. Shi, and S. Wu, "The Performance of MapReduce: An In-Depth Study," Proc. VLDB Endowment, vol. 3, no. 1, pp. 472-483, 2010.
[4] C. Ranger, R. Raghuraman, A. Penmetsa, G.R. Bradski, and C. Kozyrakis, "Evaluating MapReduce for Multi-Core and Multiprocessor Systems," Proc. IEEE 13th Int'l Symp. High Performance Computer Architecture (HPCA '07), pp. 13-24, 2007.
[5] Y. Mao, R. Morris, and F. Kaashoek, "Optimizing MapReduce for Multicore Architectures," Technical Report MIT-CSAIL-TR-2010- 020, Massachusetts Inst. of Technology, May 2010.
[6] R. Chen, H. Chen, and B. Zang, "Tiled-MapReduce: Optimizing Resource Usages of Data-Parallel Applications on Multicore with Tiling," Proc. 19th Int'l Conf. Parallel Architectures and Compilation Techniques (PACT '10), pp. 523-534, 2010.
[7] S. Babu, "Towards Automatic Optimization of MapReduce Programs," Proc. First ACM Symp. Cloud Computing (SoCC '10),pp. 137-142, 2010.
[8] Global arrays toolkit, http://www.emsl.pnl.gov/docs/global.
[9] Report on experimental language X10, http://dist.codehaus.org/x10/documentation/languagespec/x10-170.pdf (2008).
[10] A. Shet, V. Tipparaju, R. Harrison, Asynchronous programming in upc: A case study and potential for improvement, in: Workshop on Asynchrony in the PGAS Programming Model Collocated with ICS 2009, 2009.