Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 18 (2013) 2458 – 2467

Procedia Computer Science

Adaptive Preshuffling in Hadoop clusters

Jiong Xie, Yun Tian, Shu Yin, Ji Zhang, Xiaojun Ruan,
and Xiao Qin

Department of Computer Science and Software Engineering
Auburn University, Auburn, AL 36849-5347
Email: {jzx0009, tianyun, szy0004, Jzz0014, xzr0001}@eng.auburn.edu,
xqin@auburn.edu http://www.eng.auburn.edu/∼xqin

***Abstract*—MapReduce has become an important distributed processing model for large-scale data-intensive applications like data mining and web indexing. Hadoop–an open-source implementation of MapReduce is widely used for short jobs requiring low response time. In this paper, We proposed a new preshuffling strategy in Hadoop to reduce high network loads imposed by shuffle-intensive applications. Designing new shuffling strategies is very appealing for Hadoop clusters where network interconnects are performance bottleneck when the clusters are shared among a large number of applications. The network interconnects are likely to become scarce resource when many shuffle-intensive applications are sharing a Hadoop cluster. We implemented the push model along with the preshuffling scheme in the Hadoop system, where the 2-stage pipeline was incorporated with the preshuffling scheme. We implemented the push model and a pipeline along with the preshuffling scheme in the Hadoop system. Using two Hadoop benchmarks running on the 10-node cluster, we conducted experiments to show that preshuffling-enabled Hadoop clusters are faster than native Hadoop clusters. For example, the push model and the preshuffling scheme powered by the 2-stage pipeline can shorten the execution times of the WordCount and Sort Hadoop applications by an average of 10% and 14%, respectively.**

I. INTRODUCTION

In the past decade, the MapReduce framework has been employed to develop a wide variety of data-intensive applications in large-scale systems. In this paper, we focus on a new reshuffling scheme to further improve Hadoop's system performance.

II. MOTIVATIONS

A. *Shuffle-Intensive Hadoop Applications*

Recall that a Hadoop application has two important phases - map and reduce. The execution model of Hadoop can be divided into two separate steps. In the first step, a map task loads input data and generates some ¡key,value¿ pairs. In this step, multiple map tasks can be executed in parallel on multiple nodes in a cluster. In step two, all the pairs for a particular key are pulled to a single reduce task after the reduce task communicates and checks all the map tasks in the cluster.

Reduce tasks depend on map tasks; map tasks are followed by reduce tasks. This particular sequence prevents reduce tasks from sharing the computing resources of a cluster with map tasks, because there is no parallelism between a pair of map and reduce tasks. During an individually communication between a set of map tasks and a reduce task, an amount of intermediate data (i.e., result generated by the map tasks) is transferred from the map tasks to the reduce task through the network interconnect of a cluster. This communication between the map and reduce tasks is also known as the shuffle phase of a Hadoop application.

In an early stage of this study, we observe that a Hadoop application's execution time is greatly affected by the amount of data transferred during the shuffle phase. Hadoop applications generally fall into two camps, namely, non-shuffle-intensive

1877-0509 © 2013 The Authors. Published by Elsevier B.V.
Selection and peer review under responsibility of the organizers of the 2013 International Conference on Computational Science
doi:10.1016/j.procs.2013.05.422

and shuffle-intensive applications. Non-shuffle-intensive applications transfer a small amount of data during the shuffle phase. For instance, compared with I/O-intensive applications, computation-intensive applications may generate a less amount of data in shuffle phases. On the other hand, shuffle-intensive applications move a large amount of data in shuffle phases, imposing high network and disk I/O loads. Typical shuffle-intensive applications include the inverted-index tool used in search engines and the k-means tool applied in the machine learning field. These two applications transfer more than 30% data through network during shuffle phases.

B. Alleviate Network Load in the Shuffle Phase

In this paper, we propose a new shuffling strategy in Hadoop to reduce heavy network loads caused by shuffle-intensive applications. The new shuffling strategy is important, because network interconnects in a Hadoop cluster is likely to become a performance bottleneck when the cluster is shared among a large number of applications running on virtual machines. In particular, the network interconnects become scarce resource when many shuffle-intensive applications are running on a Hadoop cluster in parallel.

We propose the following three potential ways of reducing network loads incurred by shuffle-intensive applications on Hadoop clusters.

1) First, decreasing the amount of data transferred during the shuffle phase can effectively reduce the network burden caused by the shuffle-intensive applications. To reduce the amount of transferred data in the shuffle phase, combiner functions can be applied to local outputs by map tasks prior to storing and transferring intermediate data. This strategy can minimize the amount of data that needs to be transferred to the reducers and speeds up the execution time of the job.
2) Second, there is no need for reduce tasks to wait for map tasks to generate an entire intermediate data set before the data can be transferred to the reduce tasks. Rather, a small portion of the intermediate data set can be immediately delivered to the reduce tasks as soon as the portion becomes available.
3) Third, heavy network loads can be hidden by overlapping data communications with the computations of map tasks. To improve the throughput of the communication channel among nodes, intermediate results are transferred from map tasks to reduce tasks in a pipelining manner. Our preliminary findings show that shuffle time is always much longer than map tasks' computation time; this phenomenon is especially true when network interconnects in a Hadoop cluster are saturated. A pipeline in the shuffle phase can help in improving throughput of Hadoop clusters.
4) Finally, map and reduce tasks allocated within a single computing node can be coordinated in a way to have their executions overlapped. Overlapping these operations inside a node can efficiently shorten the execution times of shuffle-intensive applications. A reduce task checks all available data from map nodes in a Hadoop cluster. If reduce and map tasks can be grouped with particular key-value pairs, network loads incurred in the shuffle phase can be alleviated.

C. Benefits and Challenges of the Preshuffling Scheme

There are three benefits of our preshuffling scheme:

- Data movement activities during shuffle phases is minimized.
- Long data transfer times are hidden by a pipelining mechanism.
- Grouping map and reduce pairs to reduce network load.

Before obtaining the above benefits from the preshuffling scheme, we face a few design challenges. First, we have to design a mechanism allowing a small portion of intermediate data to be periodically transferred from map to reduce tasks without waiting an entire intermediate data set to be ready. Second, we must design a grouping policy that arranges map and reduce tasks within a node to shorten the shuffle time period by overlapping the computations of the map and reduce tasks.

D. *Organization*

The rest of the paper is organized as follows. Section III describes the design of our preshuffling algorithm after presenting the system architecture. Section IV presents the implementation details of the preshuffling mechanism in the Hadoop system. In Section V, we evaluate the performance of our preshuffling scheme. Section VI reviews related work and Section VII concludes the paper with future research directions.

III. Design Issues

In this section, we first present the design goals of our preshuffling algorithm. Then, we describe how to incorporate the preshuffling scheme into the Hadoop system. We also show a way of reducing the shuffling times of a Hadoop application by overlapping map and reduce operations inside a node.

A. *Push Model of the Shuffle Phase*

A typical reduce task consists of three phases, namely, the shuffle phase, the sort phase, and the reduce phase. After map tasks generate intermediate (key, value) pairs, reduce tasks fetch in the shuffle phase the (key, value) pairs. In the shuffle phase, each reduce task handles a portion of the key range divided among all the reduce tasks. In the sort phase, records sharing the same key are groups together; in the reduce phase, a user-defined reduce function is executed to process each assigned key and its list of values.

To fetch intermediate data from map tasks in the shuffle phase, HTTP requests are issued by a reduce task to five (this default value can be configured) number of TaskTrackers. The locations of these TaskTrackers are managed by the JobTracker located in the Master node of a Hadoop cluster. When a map or reduce TaskTracker finishes, the TaskTracker sends a heartbeat to the JobTracker in the master node, which assigns a new task to the TaskTracker. The master node is in charge of determining time when reduce tasks start running and data to be processed. Map task and reduce tasks are stored in two different queues.

Reduce tasks pull intermediate data (i.e., (key, value) pairs) from each TaskTracker that is storing the intermediate data. In this design, application developers can simply implement separate map tasks and reduce tasks without dealing with the coordination between the map and reduce tasks. In the shuffle phase the above pull model is not efficient, because reduce tasks are unable to start their execution until the intermediate data are retrieved. To improve the performance of the shuffle phase, we change the pull model into a push model. In the push model, map tasks automatically push intermediate data in the shuffle phase to reduce tasks. Map tasks start pushing (key, value) pairs to reduce tasks as soon as the pairs are produced.

We refer to the above new push model in the shuffle phase as the preshuffling technique. In what follows, we describe the design issues of our preshuffling scheme that applies the push model in the shuffle phase.

B. *A Pipeline in Preshuffling*

When a new job submitted to a Hadoop cluster, the JobTracker assigns map and reduce tasks to available TaskTrackers in the cluster. Unlike the pulling model, the pushing model of preshuffling push intermediate data produced by map tasks to reduce tasks. The preshuffling scheme allows the map tasks to determine a partition records to be transferred a reduce task. Upon the arrival of the partition records, the reduce task sorts and stores these records into the node hosting the reduce task. Once the reduce task is informed that all the map tasks have been completed, the reduce task performs a user-defined function to process each assigned key and its list of values. The map tasks continue generating intermediate records to be delivered the reduce tasks.

Let us consider a simple case where a cluster has enough free slots allowing all the tasks of a job to run after the job is submitted to the cluster. In this case, we establish communication channels between a reduce task and all the map tasks pushing intermediate data to the reduce task. Since each map task decides reduce tasks to which the intermediate data should be pushed, the map task transfers the intermediate data to the corresponding reduce tasks immediately after the data are produced by the map task.

In some cases, there might not be enough free slots available to schedule every task in a new Hadoop job. If a reduce task can not be executed due to limited number of free slots, map tasks can store intermediate results in memory buffers or local disks. After a free slot is assigned to the reduce task, the intermediate results buffered in the map tasks can be sent to the reduce task.

Shuffle phase time in many cases is much longer than map phase time (i.e., tasks' computation time); this problem is more pronounced true when network interconnects are scarce resource in a Hadoop cluster. To improve the performance of the preshuffling scheme, we build a pipeline in the shuffle phase to proactively transfer intermediate data from map tasks to reduce tasks. The pipeline aims at increasing the throughput of preshuffling by overlapping data communications with the computations of map tasks.

We design a mechanism to create two separate threads in a map task. The first thread processes input data, generates intermediate records, and completes the sort phase. The second thread manages the aforementioned pipeline that sends intermediate data from map tasks to reduce tasks immediately when the intermediate outputs are produced. The two threads can work in parallel in a pipelining manner. In other words, the first thread implements the first stage of the pipeline; the second thread performs the second stage of the pipeline. In this pipeline, the first stage is focusing on producing intermediate results to be stored in the memory buffers, whereas the second stage periodically retrieves the intermediate results from the buffers and transfers the results to the connected reduce tasks.

C. In-memory Buffer

The push model does not require reduce tasks to wait a long time period before map tasks complete the entire map phase. Nevertheless, pushing intermediate data from map to reduce tasks in the preshuffling phase is still a time-consuming process. The combiner process in a map task is an aggregate function (a reduce-like function) that groups multiple distinct values together as input to form a single value. If we plan to implement the preshuffling mechanism to directly send intermediate outputs from map to reduce tasks, we will have to ignore the combiner process in map tasks. In the native Hadoop system, the combiner can help map tasks to illuminate relevant data, thereby reducing data transfer costs. Sending all the data generated from map tasks to reduce tasks increases response time and downgrades the performance of Hadoop applications. Without the pre-sorting and filtering process in the combiner stage, reduce tasks should spend much time in sorting for merging values.

Instead of sending an entire buffered content to reduce tasks directly, we design a buffer mechanism to temporary collect intermediate data. The buffer mechanism immediately sends a small portion of the intermediate data to reduce tasks as soon as the portion is produced. A configurable threshold is used to control the size of the portion. Thus, once the size of buffered intermediate results reaches the threshold, the map task sorts the intermediate data based on reduce keys. Next, the map task writes the buffer to its local disk. Then, the second stage of the pipeline is invoked to check whether reduce tasks have enough free slots. If nodes hosting reduce tasks are ready, a communication channel between the map and reduce tasks are established. The combined data produced in the first stage of the pipeline can be passed to reduce tasks in the second stage of the pipeline.

In cases where nodes hosting reduce tasks are not ready, the second stage of the pipeline will have to wait until the reduce tasks are available to receive the pushed data. This pipeline mechanism aims to improve the throughput of the shuffling stage, because the pipeline makes it possible for map tasks to send intermediate data as soon as a portion of the data is produced by map functions.

In the design of our preshuffling scheme, it is flexible to dynamically control the amount of data pushed from map to reduce tasks by adjusting the buffer's threshold. A high threshold value means that each portion to be pushed from map tasks in the second stage of the pipeline is large; a small threshold value indicates that each portion shipped to reduce tasks is large. If network interconnects are not overly loaded, map tasks may become a performance bottleneck. This bottleneck problem can be addressed by increasing the buffer's threshold so that each data portion pushed to reduce tasks is large. A large threshold is recommended for Hadoop clusters with fast network interconnects; a small threshold is practical for Hadoop clusters where networks are a performance bottleneck.

IV. IMPLEMENTATION

In Hadoop, reduce tasks will not start their executions until entire intermediate output of all map tasks have been produced, although some map tasks may generate some intermediate results earlier than the other map tasks. In our preshuffling scheme, map tasks do not need to be synchronized in the way to produce a group of intermediate data to be sent to reduce tasks at the same time. Thus, a reduce task can immediately receive corresponding intermediate data generated by map tasks. However, the reduce task is unable to apply the reduce function on the intermediate data until all the date produced by every map task become available. Like reduce tasks, a Hadoop job must wait for all map tasks to finish before producing a final result.

As described in Section III, a map task consists of two phases: map and map-transfer. The map phase processes an entire input file, sorts intermediate results, and then sends them to an output buffer. The sort phase in the map task groups records sharing the same key together; this group procedure otherwise should be performed in the reduce phase. In the map-transfer phase, intermediate data is transferred from buffer in map tasks to reduce tasks.

A reduce task consists two main phases - shuffle and reduce. In the shuffle phase, the reduce task not only receives its portion of intermediate output from each map task, but also performs a merge sort on the intermediate output from map tasks. In reduce tasks, the shuffle phase time accounts for a majority of the total reduce tasks' execution time. For example, 70% of a reduce task's time is spent in the shuffle phase. The shuffle phase is time consuming, because a large amount of intermediate output from map tasks must be merged and sorted in this phase. To improve the performance of the shuffle phase, we implement a preshuffling scheme where intermediate data are immediately merged and sorted when the data are produced by map tasks. After receiving required intermediate data from all map tasks, the reduce task performs a final merge sort function based on intermediate output produced by the preshuffling scheme. When the reduce task completes its final merge sort, the task reaches the reduce phase.

In a Hadoop cluster, a master node monitors the progress of each task's execution. When a map task starts its execution, the master node assigns a progress score anywhere in the range between 0 and 1. The value of a progress score is assigned based on how much of the input data the map task has processed [3]. Similarly, we introduce a progress score, allowing the preshuffling scheme to monitor the progress of reduce tasks. Progress scores of reduce tasks are assigned based on how much intermediate data of each portion has been consumed by the reduce tasks. The progress score is incorporated with the data structure of intermediate data. Thus, when a partition of intermediate file is transferred to a reduce task, the progress score of this partition is also received by the reduce task. The average progress score of all relevant partitions in each intermediate data file can be considered as the progress of a reduce task.

Each node hosting reduce tasks individually runs the tasks. In heterogeneous Hadoop clusters, nodes may run tasks at different speed. Once a reduce task has made sufficient progress, the task reports its progress score written to a temporary file on HDFS. For example, we can set several granularity; the user can set the default value as 20%, 40%, 60%, 80%, and 100%. When reduce progress reaches this value, the progress score will be automatically written down to HDFS.

By aggressively pushing data from map tasks to reduce tasks, the push model can increase the throughput of the Hadoop system by partially overlapping communication and transfer times among the map and reduce tasks. The preshuffling scheme, when used in combination with the push model, can boost the performance of Hadoop clusters. The performance improvement offered by preshuffling and the push model becomes more pronounced when network interconnection is a performance bottleneck of the clusters.

V. Evaluation Performance

A. Experimental Environment

To evaluate the performance of the proposed preshuffling scheme incorporated in the push model with a pipelining technique, we run Hadoop benchmarks on a 10-node cluster. Table I summarizes the configuration of the cluster used as a testbed for the performance evaluation. Each computing node in the cluster is equipped with two dual-core 2.4 GHz Intel processors, 2GB main memory, 146 SATA hard disk, and a Gigabit Ethernet network interface card.

TABLE I: Test Bed

CPU	Intel Xeon 2.4GHz
Memory	2GB Memory
Disk	SEGATE 146GB
Operation System	Ubuntu 10.4
Hadoop version	0.20.2

In our experiments, we vary the block size in HDFS to evaluate the impacts of block size system performance. In this study, we focus on impact of preshuffling and the push model on Hadoop and; therefore, we disable the data replica feature of HDFS. Nevertheless, using the preshuffling mechanism in combination with the data replica mechanism can significantly improve performance of Hadoop clusters.

We test the following two Hadoop benchmarks running on the cluster, in which the preshuffling scheme is integrated with the push model to improve the performance of the shuffle phase in Hadoop applications.

1) WordCount (WC): This Hadoop application counts the frequency of occurrence for each word in a text file. Map tasks process different sections of input files and return intermediate data that consists of several pairs word and frequency. Then, reduce tasks add up the values for each identity word. The Word-Count is a memory-intensive application.
2) Sort: This Hadoop application puts elements of a list in a certain order. The most-used orders are numerical order and lexicographical order. The output list of this application is in a non-decreasing order.

B. In Cluster

We compare the overall performance between the native Hadoop and the preshuffling-enabled Hadoop on a 10-node cluster. We measure the execution times of the two tested Hadoop benchmarks running on the Hadoop cluster, where the default block size is 64 MB.

Figure **??** illustrates the progress trend of WordCount processing 1GB data on the native Hadoop. The progress trend shown in Figure **??** indicates how the map and reduce tasks are coordinating. For example, Figure 1(a) shows that in the native Hadoop system, the reduce task does not start its execution until the all the map tasks complete their executions at time 50. Figure 1(b) proves that in the preshuffling-enabled Hadoop, our push model makes it possible for the reduce task in WordCount to begins its execution almost immediately after the map task gets started.

Our solution shortens the execution time of WordCount by approximately 15.6%, because the reduce task under the push model receives intermediate output produced by the map tasks as soon as the output become available.

Figures 1(a) and 1(b) show that it takes 50 seconds to finish the map task in the native Hadoop and its takes about 60 seconds to complete the map in the preshuffling-enabled Hadoop. The preshuffling-enabled Hadoop system has a longer map task than the native Hadoop, because in our push model part of the shuffle phase is handled by the map task rather than the reduce task in the native Hadoop. Forcing the map task to process the preshuffling phase is an efficient way of reducing heavy load imposed on reduce tasks. As a result, the preshuffling-enabled Hadoop cluster can complete the execution of WordCount faster than the native Hadoop cluster.

Fig. 2: Impact of block size on the preshuffling-enabled cluster running WordCount.

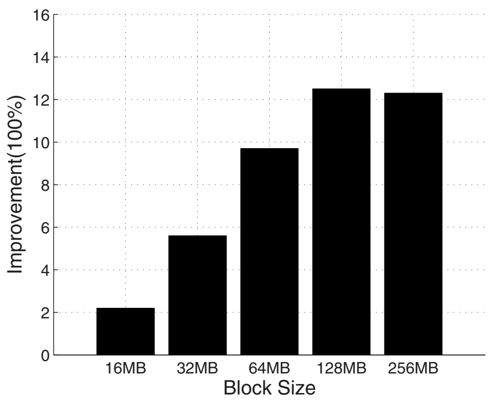
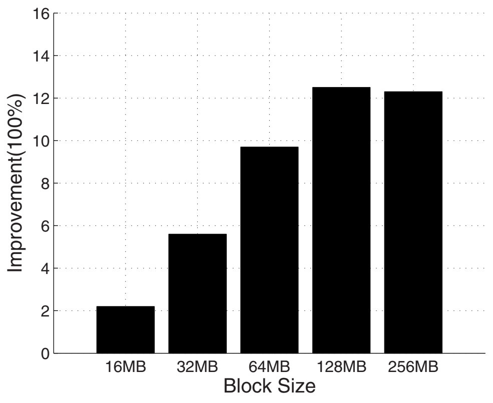
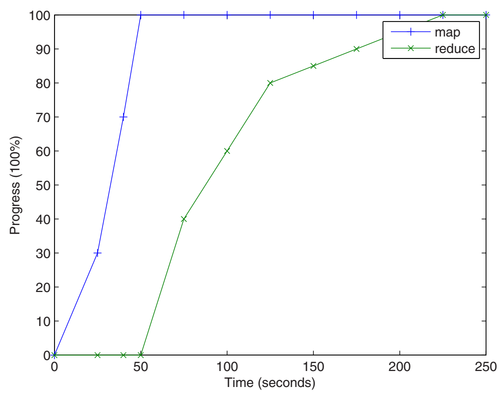
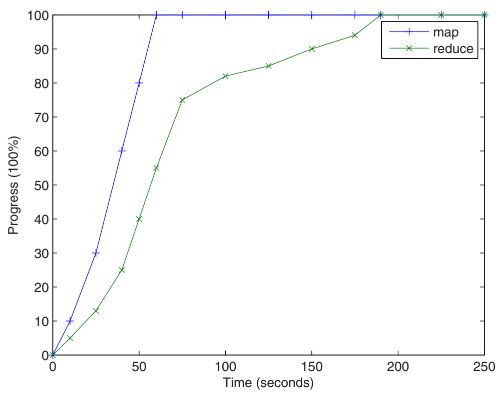


Fig. 3: Impact of block size on the preshuffling-enabled Hadoop cluster running Sort.





(a) The execution time of WordCount processing 1GB data on the native Hadoop system is 450 seconds.



(b) The execution time of WordCount processing 1GB on the preshuffling-enabled Hadoop system is 380 seconds.

Fig. 1: The progress trend of WordCount processing 1GB data on the 10-node Hadoop cluster.

C. Large Blocks vs. Small Blocks

Now we evaluate the impact of block size on the performance of preshuffling-enabled Hadoop clusters. The goal of this set of experiments is to quantify the sensitivity of our preshuffling scheme on the block size using the two Hadoop benchmarks. We run the WordCount and Sort benchmarks on both the native Hadoop and the preshuffling-enabled Hadoop clusters when the block size is set to 16MB, 32MB, 64MB, 128MB, and 256MB, respectively.

Figures 2 and 3 shows the performance improvement of the preshuffling-enabled Hadoop cluster over the native Hadoop cluster as a function of the block size. Figure 2 demonstrates that the improvement offered by preshuffling in case the of WordCount increases when the block size goes up from 16 MB to 128 MB. However, increasing the block size from 128 MB to 256 MB does not provide a higher improvement percentage. Rather, the improvement slightly drops from 12.5% to 12.2% when the block size is changed from 128 MB to 256 MB. The experimental results plotted in Figure 2 suggest that a large block size allows the preshuffling scheme to offer good performance improvement. The improvement in terms of percentage is saturated when the block size is larger than 128 MB.

Figure 3 shows the performance improvement of preshuffling on the 10-node cluster running the Sort application. The results plotted in Figure 3 are consistent with those shown in Figure 2. For the two Hadoop benchmarks, the performance improvement offered by preshuffling is sensitive to block size when the block size is smaller than 128 MB.

VI. Related work

Implementations of MapReduce. MapReduce framework is inspired by the map and reduce functions commonly used in functional programming [4]. MapReduce is useful in a wide range of applications including: distributed grep, distributed sort, web link-graph reversal, term-vector per host, web access log stats, inverted index construction, document clustering, machine learning [2], and statistical machine translation. Moreover, the MapReduce model has been adapted to several computing environments like multi-core and many-core systems [1][11], desktop grids, volunteer computing environments [9], and dynamic cloud environments [10].

Shuffling. Duxbury *et al.* built a theoretical model to analyze the impacts of MapReduce on network interconnects [12]. There are two new findings in their study. First, during the shuffle phase, each reduce task communicates with all map tasks in a cluster to retrieve required intermediate data. Network load is increased during the shuffle phase due to intermediate data transfers. Second, at the end reduce phase, final results of the Hadoop job is written to HDFS. Their study shows evidence that the shuffle phase can cause high network loads. Our experimental results confirm that 70% of a reduce task's time is spent in the shuffle phase. In this paper, we propose a preshuffling scheme combined with a push model to release the network burden imposed by the shuffle phase.

Pipeline. Dryad [8] and DryadLINQ [13] offer a data-parallel computing framework that is more general than MapReduce. This new framework enables efficient database joins and automatic optimizations within and across MapReductions using techniques similar to query execution planning. In the Dryad-based MapReduce implementation, outputs produced by multiple map tasks are combined at the node level to reduce the amount of data transferred during the shuffle phase. Compared with this combining technique, partial hiding latencies of reduce tasks is more important and effective for shuffle-intensive applications. Such a latency-hiding technique may be extended to other MapReduce implementations.

Recently, researchers extended the MapReduce programming model to support database management systems in order to process structured files [7]. For example, Olston *et. al* developed the Pig system [6], which is a high-level parallel data processing platform integrated with Hadoop. The Pig infrastructure contains a compiler that produces sequences of Hadoop programs. Pig Latin - a textual language - is the programming language used in Pig. The Pig Latin language not only makes it easy for programmers to implement embarrassingly parallel data analysis applications, but also offer performance optimization opportunities.

VII. CONCLUSION

A Hadoop application's execution time is greatly affected by the shuffling phase, where an amount of data is transferred from map tasks to reduce tasks. Moreover, improving performance of the shuffling phase is very critical for shuffle-intensive applications, where a large amount of intermediate data is delivered in shuffle phases. Making a high-efficient shuffling scheme is an important issue, because shuffle-intensive applications impose heavy network and disk I/O loads during the shuffle phase. In this paper, we proposed a new push model, a new preshuffling module, and a pipelining mechanism to efficiently boost the performance of Hadoop clusters running shuffle-intensive applications.

In the push model, map tasks automatically send intermediate data in the shuffle phase to reduce tasks. Unlike map tasks in the traditional pull model, map tasks in the push model proactively start sending intermediate data to reduce tasks as soon as the data are produced. The push model allows reduce tasks to start their executions earlier rather than waiting until an entire intermediate data set becomes available. The push model improves the efficiency of the shuffle phase, because reduce tasks do not need to be strictly synchronized with their map tasks waiting for the entire intermediate data set.

Our preshuffling scheme aims to release the load of reduce tasks by moving the pre-sorting and filtering process from reduce tasks to map tasks. As a result, reduce tasks with the support of the preshuffling scheme spend less time in sorting to merge values.

In the light of the push model and the preshuffling scheme, we built a 2-stage pipeline to efficiently move intermediate data from map tasks to reduce tasks. In stage one, local buffers in a node hosting map tasks temporarily store combined intermediate data. In stage two, a small portion of the intermediate data stored in the buffers is sent to reduce tasks as soon as the portion is produced. In the second stage of the pipeline, the availability of free slots in nodes hosting reduce tasks are checked. If there are free slots, a communication channel between the map and reduce tasks are established. In the 2-stage pipeline, the combined data produced in the first stage of the pipeline can be passed to reduce tasks in the second stage of the pipeline.

We implemented the push model along with the preshuffling scheme in the Hadoop system, where the 2-stage pipeline was incorporated with the preshuffling scheme. Our experimental results based on two Hadoop benchmarks shows that preshuffling-enabled Hadoop clusters are significantly faster than native Hadoop clusters with the same hardware configurations. For example, the push model and the preshuffling scheme powered by the 2-stage pipeline can shorten the execution times of the two Hadoop applications (i.e., WordCount and Sort) by an average of 10% and 14%, respectively.

ACKNOWLEDGMENTS

This research was supported by the U.S. National Science Foundation under Grants CCF-0845257 (CAREER), CNS-0917137 (CSR), CNS-0757778 (CSR), CCF-0742187 (CPA), CNS-0831502 (CyberTrust), CNS-0855251 (CRI), OCI-0753305 (CI-TEAM), DUE-0837341 (CCLI), and DUE-0830831 (SFS), as well as Auburn University under a startup grant, and a gift (No. 2005-04-070) from the Intel Corporation.

REFERENCES

[1] B.He, W.Fang, Q.Luo, N.Govindaraju, and T.Wang. *Mars: a MapReduce framework on graphics processors*. ACM, 2008.

[2] Cheng-Tao Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary R. Bradski, Andrew Y. Ng, and Kunle Olukotun. Map-reduce for machine learning on multicore. In *NIPS*, pages 281–288, 2006.

[3] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. Mapreduce online. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, NSDI'10, pages 21–21, Berkeley, CA, USA, 2010. USENIX Association.

[4] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *OSDI '04*, pages 137–150, 2008.

[5] Adam Dou, Vana Kalogeraki, Dimitrios Gunopulos, Taneli Mielikainen, and Ville H. Tuulos. Misco: a mapreduce framework for mobile systems. In *Proceedings of the 3rd International Conference on PErvasive Technologies Related to Assistive Environments*, PETRA '10, pages 32:1–32:8, 2010.

[6] Apache Software Foundation. The pig project. `http://hadoop.apache.org/pig`.

[7] Eric Friedman, Peter Pawlowski, and John Cieslewicz. Sql/mapreduce: a practical approach to self-describing, polymorphic, and parallelizable user-defined functions. *Proc. VLDB Endow.*, 2:1402–1413, August 2009.

[8] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *SIGOPS Oper. Syst. Rev.*, 41:59–72, March 2007.

[9] Heshan Lin, Xiaosong Ma, Jeremy Archuleta, Wu-chun Feng, Mark Gardner, and Zhe Zhang. Moon: Mapreduce on opportunistic environments. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 95–106, New York, NY, USA, 2010. ACM.

[10] Fabrizio Marozzo, Domenico Talia, and Paolo Trunfio. A peer-to-peer framework for supporting mapreduce applications in dynamic cloud environments. In Nick Antonopoulos and Lee Gillam, editors, *Cloud Computing*, volume 0 of *Computer Communications and Networks*, pages 113–125. Springer London, 2010.

[11] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. *High-Performance Computer Architecture, International Symposium on*, 0:13–24, 2007.

[12] Raplesf. Analyzing network load in map/reduce. `http://blog.rapleaf.com/dev/2010/08/24/analyzing-network-lo`

[13] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. Dryadlinq: a system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association.