Hadoop Acceleration Through Network Levitated Merge

Yandong Wang
Auburn University
wangyd@auburn.edu

Xinyu Que
Auburn University
xque@auburn.edu

Weikuan Yu
Auburn University
wkyu@auburn.edu

Dror Goldenberg
Mellanox Technologies
gdror@mellanox.co.il

Dhiraj Sehgal
Mellanox Technologies
dhirag@mellanox.com

Abstract

Hadoop is a popular open-source implementation of the MapReduce programming model for cloud computing. However, it faces a number of issues to achieve the best performance from the underlying system. These include a serialization barrier that delays the reduce phase, repetitive merges and disk access, and lack of capability to leverage latest high speed interconnects. We describe Hadoop-A, an acceleration framework that optimizes Hadoop with plugin components implemented in C++ for fast data movement, overcoming its existing limitations. A novel network-levitated merge algorithm is introduced to merge data without repetition and disk access. In addition, a full pipeline is designed to overlap the shuffle, merge and reduce phases. Our experimental results show that Hadoop-A doubles the data processing throughput of Hadoop, and reduces CPU utilization by more than 36%.

1. Introduction

MapReduce [6] has emerged as a popular and easy-to-use programming model for numerous organizations to process explosive amounts of data, perform massive computation, and extract critical knowledge for business intelligence. Hadoop [1] is an open-source implementation of MapReduce, currently maintained by the Apache Foundation, and supported by leading IT companies such as Google and Yahoo!. Hadoop implements MapReduce framework with two categories of components: a JobTracker and many TaskTrackers. The JobTracker commands TaskTrackers (a.k.a. slaves) to process data in parallel through two main functions: map and reduce. In this process, the JobTracker is in charge of scheduling map tasks (MapTasks) and reduce tasks (ReduceTasks) to TaskTrackers. It also monitors their progress, collects run-time execution statistics, and handles possible faults and errors through task re-execution.

Performance and scalability are critical to ensure Hadoop's continuing success to various industry and scientific users. A number of studies have been carried out to improve the performance of Hadoop MapReduce framework. Jiang et al. [11] have tuned the parameters of Hadoop for better performance. Condie et al. [5] have proposed the *MapReduce Online* architecture to open up direct network channels between MapTasks and ReduceTasks and speed up the delivery of data from MapTasks to ReduceTasks. While their work reduces job completion time and improves system utilization, it cannot cope with a gigantic dataset that does not fit in memory, and also complicates the fault tolerance handling of Hadoop tasks. Furthermore, it demands a large number of network channels for data movement. For these reasons, MapReduce Online has to fall back onto the default MapReduce execution mode.

Little work has been carried out to examine the relationship of Hadoop MapReduce's three data processing phases, i.e., shuffle, merge, and reduce, and their implication to the efficiency of Hadoop. With an extensive examination of Hadoop MapReduce framework, particularly its ReduceTasks, we 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 two-phase MapReduce protocol, no ReduceTasks can start reducing data until all intermediate data has been merged together. This results in a serialization barrier that significantly delays the reduce operation of ReduceTasks.

More importantly, the current merge algorithm 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 (c.f. Section 2.2.2). It poses a severe threat to the efficiency of Hadoop data processing.

To address these critical issues for Hadoop MapReduce Framework, we have designed Hadoop-A, an acceleration framework that can take advantage of plugin components for performance enhancement and protocol optimizations. Several enhancements

SC11, November 12-18, 2011, Seattle, Washington, USA

are introduced: (1) a novel algorithm that enables ReduceTasks to perform data merging without repetitive merges and disk access; (2) a full pipeline is designed to overlap the shuffle, merge and reduce phases for ReduceTasks; and (3) besides the TCP/IP protocol in the original Hadoop, an alternative protocol is introduced in Hadoop-A to enable data movement via RDMA (Remote Direct Memory Access). Since ReduceTasks are able to merge data by staying above local disks, we refer to this new algorithm as network-levitated merge. We have carried out an extensive set of experiments to evaluate the performance of Hadoop-A compared to the original Hadoop. Our 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 improve the throughput of Hadoop data processing by more than 100%. Its RDMA-based data movement also reduces CPU utilization by more than 36%.

The rest of the paper is organized as follows. Section 2 provides the motivation. We then describe the Hadoop-A acceleration framework in Section 3, followed by Section 4 that details the network-levitated merge algorithm. Section 5 provides experimental results. Section 6 provides a review of related work. Finally, we conclude the paper in Section 7.

2. Motivation

Hadoop's MapReduce implementation enables a convenient and easy-to-use data processing framework. However, our characterization and analysis reveal a number of issues in the existing architecture. In this section, we provide an overview of the Hadoop MapReduce framework, and then shed light on existing limitations in the current design.

2.1 Overview of Hadoop MapReduce Framework

A key feature of the Hadoop MapReduce framework is its pipelined data processing. As shown in Figure 1, Hadoop consists of three main execution phases: map, shuffle/merge, and reduce. When a user job is submitted to the JobTracker, its input dataset is divided into many data splits. In a split, user data is organized as many records of <key,val> pairs. In the first phase, the JobTracker selects a number of TaskTrackers and schedules them to run the map function. Each TaskTracker launches several MapTasks, one per split of data. The mapping function in a MapTask converts the original records into intermediate results, which are data records in the form of <key',val'> pairs. These new data records are stored as a MOF (Map Output File), one for every split of data. A MOF is organized into many data partitions, one per ReduceTask. Each data partition contains a set of data records. When a MapTask completes one data split, it is rescheduled to process the next split.

In the second phase, when MOFs are available, the JobTracker selects a set of TaskTrackers to run the ReduceTasks. TaskTrackers can spawn several concurrent ReduceTasks. Each ReduceTask starts by fetching a partition that is intended for it from a MOF (also called segment). Typically, there is one segment in each MOF for every ReduceTask. So, a ReduceTask needs to fetch such segments from all MOFs. Globally, these fetch operations lead to an all-to-all **shuffle** of data segments among all the ReduceTasks. While the data segments are being shuffled, they are also merged based on the order of keys in the data records. As more remote segments are fetched and merged locally, a ReduceTask has to spill, i.e., store, some segments to local disks in order to alleviate memory pressure. Shuffle and merge of data segments by ReduceTasks is called the copy phase in Hadoop. It is also commonly referred as the **shuffle/merge** phase.

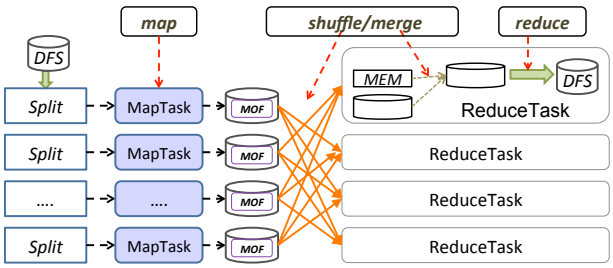


Figure 1. An Overview of Data Processing in Hadoop MapReduce Framework

In the third, or **reduce** phase, each ReduceTask loads and processes the merged segments using the reduce function. The final result is then stored to Hadoop Distributed File System [17].

2.2 Issues in the Existing Hadoop Framework

There are several issues in the existing Hadoop framework, including (a) the serialization between Hadoop shuffle/merge and reduce Phases, (b) repetitive merges and disk access, and (c) the lack of support for RDMA interconnects.

2.2.1 A Serialization in Hadoop Data Processing

Hadoop strives to pipeline the processing of large datasets. It is indeed able to do so, particularly for map and shuffle/merge phases. As shown in Figure 2, after a brief initialization period, a pool of concurrent MapTasks starts the map function on the first set of data splits. As soon as the 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.

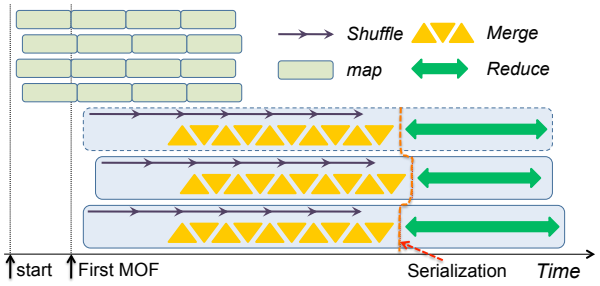


Figure 2. Serialization between Shuffle/Merge and Reduce Phases

To guarantee correctness of the MapReduce programming model, it is necessary to ensure that the reduce phase does not start until the map phase is done for all data splits. However, the pipeline as shown in Figure 2 contains an implicit serialization. At each

ReduceTask, not until all its segments are available and merged, will the reduce phase start to process data segments via the reduce function. These essentially enforce a serialization between the shuffle/merge phase and the reduce phase. When there are many segments to process (which is often the case), it takes a significant amount of time for a ReduceTask to shuffle and merge them. As a result, the reduce phase will be significantly delayed. Our analysis (c.f. Section 5.4) has revealed that this can delay the reduce phase by 668 seconds, i.e., more than 39.4% of the total execution time for a Hadoop program that sorts 192GB of data on 24 nodes (c.f. the last row of Table 2).

2.2.2 Repetitive Merges and Disk Access

As mentioned earlier in the overview, ReduceTasks merge data segments when the number of segments or their total size goes over a threshold. A newly merged segment has to be spilled to local disks due to memory pressure. However, the current merge algorithm in Hadoop often leads to repetitive merges, thus extra disk accesses. Figure 3 shows a common sequence of merge operations in Hadoop. For the purpose of illustration, we hereby choose a very small threshold parameter *io.sort.factor* = 3. A ReduceTask fetches its data segments and arranges them in the order of their size. When the number of data segments reaches six, i.e., twice the threshold, the smallest three segments are merged, shown as Step 1 in Figure 3. Under memory pressure, this will incur disk access. The resulting segment is inserted back into the heap based on its relative size.

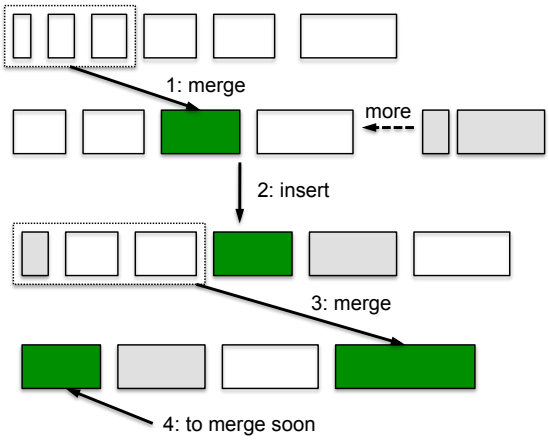


Figure 3. Repetitive Merging and Disk Access

When more segments arrive (as shown in Step 2), the threshold will be broken again. It is then necessary to merge another set of segments, shown as Step 3. This again causes additional disk access, let alone the need to read some segments if they have been stored on local disks. Depending on its relative size, a previously merged segment is likely to be grouped into another set and merged again, shown as Step 4. Since the smallest segments are usually selected for merge, chances are rather high for a segment to be merged repetitively. Furthermore, any segment merged from a subset of segments eventually needs to be merged for final results. Altogether, this means repetitive merges and disk access, therefore degraded performance for Hadoop.

It is tempting to choose a different policy for merge. This can lead to a similar problem of essentially the same nature. The key constraint is that, if any merge happens before a global order of segments is established, it ought to be re-merged into the final result before the reduce function. Therefore, an alternative merge algorithm is critical for Hadoop to mitigate the impact of repetitive merges and their associated disk access.

2.2.3 Unable to Use RDMA Interconnects

Hadoop does not take advantage of high-performance RDMA interconnect technologies such as InfiniBand [9] that have matured in the HPC (High Performance Computing) community. For example, a node in a hierarchical Hadoop cluster is typically equipped with one or more Gigabit Ethernet (GigE) network interface cards and connected to the lowest tier GigE switch. With this configuration, one card can only add an upper bound of 125 MB/sec to the data movement throughput of Hadoop. Given a multi-socket and multi-core server, such capacity has to be shared and very thinly divided amongst all cores. Worse yet, advances in processor technology will soon deliver compute servers with hundreds of cores to the mass market. Furthermore, RDMA supports high bandwidth data movement with very little CPU involvement. Simply replacing the network hardware with the latest interconnect technologies such as InfiniBand and 10 Gigabit Ethernet, and continuing to run Hadoop on TCP/IP will not enable Hadoop to leverage the strengths of RDMA. Thus, the lack of support for RDMA interconnects will become a severe threat for Hadoop to keep up with the advances of other computer technologies, particularly when more highly capable processors, storage, and interconnect devices are deployed to various computing and data centers.

3. Design of Hadoop Acceleration

In view of the issues discussed in Section 2, we deem that it is important to design a solution that can accelerate Hadoop's MapReduce framework and overcome existing limitations. We have designed Hadoop-A, an acceleration framework that allows Hadoop to take advantage of RDMA-capable interconnects, experiment with different plugin merge algorithms, and retain the same user interface. In this section, we will describe the architecture of Hadoop-A, and the exploitation of RDMA for data shuffling.

3.1 Software Architecture of Hadoop-A

Figure 4 shows the architecture of Hadoop-A. Two new user-configurable plugin components, *MOFSupplier* and *NetMerger*, are introduced to leverage RDMA-capable interconnects and enable alternative data merge algorithms. Both MOFSupplier and NetMerger are threaded C++ implementations, with all components following the object-oriented principle. 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. We briefly describe several features of this acceleration framework without going too much into the technical details of their implementations.

User-Transparent Plugins – A primary requirement of Hadoop-A is to maintain the same programming and control interfaces for users. To this end, we design the MOFSupplier and NetMerger plugins as C++ programs that can be launched by TaskTrackers. A user can choose to enable or disable the acceleration, which is

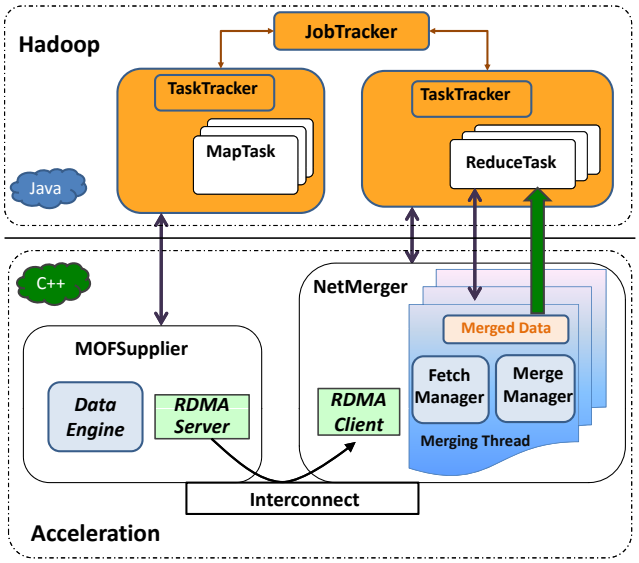


Figure 4. Software Architecture of Hadoop-A

controlled by a parameter in the configuration file. With these runtime plugins, we ensure that Hadoop-A is user-transparent in two ways: (1) no changes are introduced for the scheduling and monitoring interface between TaskTracker and MapTask, and the same for TaskTracker and ReduceTask; and (2) no modification is made to the submission and control interface between a user program and the JobTracker. All MapReduce programs written for Hadoop will continue to function with Hadoop-A.

Multithreaded and Componentized MOFSupplier and Netmerger – MOFSupplier contains an RDMA server that handles fetch requests from ReduceTasks. It also contains a data engine that manages the index and data files for all MOFs that are generated by local MapTasks. Both components are implemented with multiple threads in MOFSupplier. NetMerger is also a multithreaded program. It provides one thread for each Java ReduceTask. It also contains other threads, including an RDMA client that fetches data partitions and a staging thread that uploads data to the Java-side ReduceTask.

Event-Driven Progress and Coordination – To synchronize with Java-side components, we provide event channels between MOFSupplier/NetMerger plugins and Hadoop. These event channels are also used to coordinate activities and monitor progress for internal components of MOFSupplier and NetMerger. All channels are implemented through asynchronous loopback sockets that can wake up threads when there are tasks, and allow them to go back to sleep when tasks are not available. Run-time progress reports and execution statistics are collected and stored as a part of Hadoop logging files. Such logging utilities are capable of monitoring and dissecting the execution of Hadoop jobs. For example, results in Section 5.4 are collected by using this feature.

3.2 RDMA-Accelerated Data Shuffling

As mentioned in Section 2, Hadoop cannot make use of RDMA interconnects such as InfiniBand. The left half of Figure 5 shows the communication stack currently used for Hadoop data shuffling. When notified of the completion of a MOF, Hadoop ReduceTasks invoke a copy thread to fetch its data partition through a Java-based HTTP GET request. On the server side, a Java-based HTTP server is launched by every TaskTracker. A specific HTTP servlet is attached to this server to handle HTTP GET requests and serve data partitions from the MOF files accordingly.

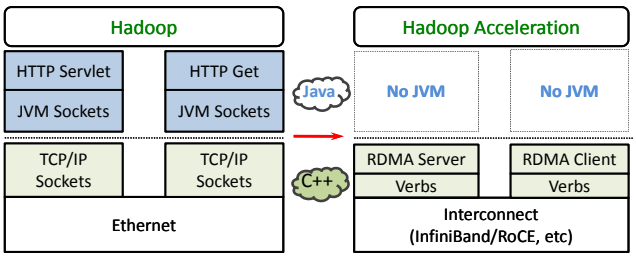


Figure 5. RDMA-Accelerated Data Shuffling

Hadoop-A component architecture allows us to introduce alternative communication protocols for data shuffling in Hadoop. To exploit the benefit of RDMA-capable interconnects, we design our RDMA-based data shuffling protocol completely in the native C++ language, as shown on the right of Figure 5. The new protocol directly builds the RDMA-based communication on top of the verbs protocol, and completely avoids the overhead of JVM for Hadoop data shuffling.

RDMA-based shuffling protocol consists of an RDMA server in the MOFSupplier and an RDMA client in the NetMerger. InfiniBand Reliable Connected (RC) service are established on a per-node basis for RDMA clients and servers. The RDMA CM protocol is used for connection establishment. Connections are retained for the lifetime of an RDMA client, and will be torn down and re-established for a revived client. Once connected, RDMA clients and servers communicate data through pre-registered memory buffers. The data engine in the MOFSupplier always prefetches data segments. It retrieves data from disk when the requested data is not yet available in memory. Such data movement is realized through a direct request and reply protocol. An RDMA client sends a request along with the information of the available memory buffer, and the RDMA server locates the data and writes it to the client buffer via a zero-copy RDMA write operation. More implementation details of the RDMA protocol (and the data engine) are omitted here as they are not the focus of this paper.

4. A Network-Levitated Shuffle, Merge and Reduce Pipeline

As discussed in Section 2, there exist two critical issues in Hadoop: (1) the serialization barrier between shuffle/merge and reduce phases, and (2) repeated merges and disk access. We first describe a network-levitated merge algorithm that avoids repeated merges, and then detail the construction of a new pipeline to eliminate the serialization.

4.1 Network-Levitated Data Merge

Hadoop resorts to repetitive merges because of limited memory compared to the size of data. For each remotely completed MOF, it 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.

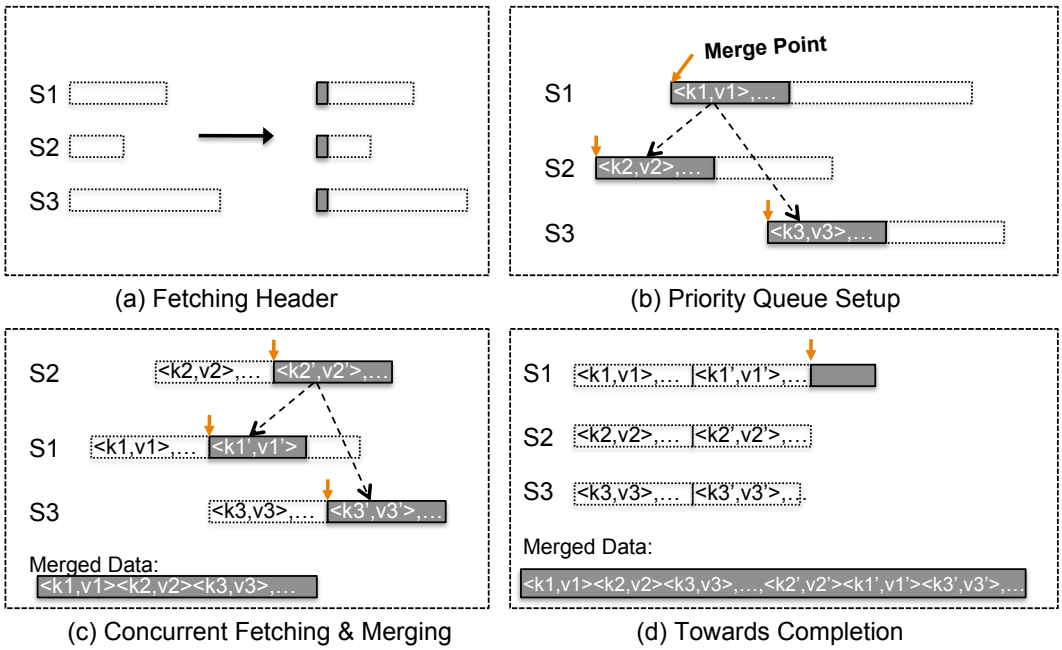


Figure 6. A Network-Levitated Merge Algorithm

With the performance of RDMA interconnects comes so close to memory, it is now unnecessary, even unwise, to pull data partitions locally before merging. Therefore we design an algorithm that can merge all data partitions exactly once, and at the same time stay *levitated* above local disks.

Figure 6 shows our network-levitated merge algorithm. Our algorithm is modified from Hadoop's Priority Queue-based merge sort algorithm. The key idea is to leave data on remote disks until it is time to merge the intended data records.

As shown in Figure 6(a), 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 specially 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 Figure 6(b).

Our algorithm merges the available <key,val> pairs in the same way as is done in Hadoop. Each segment is a part of a MOF produced by the map function of Hadoop, which means that it is composed of data records ordered by their keys. Thus the root of a complete PQ is the first record among all segments. So, it is safe to store this root record as the first record in the final merged data. When the PQ is updated, the next root will be the first <key,val> among all remaining segments. It is then stored to the final merged data as well. 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 Figure 6(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. Note that data records are merged exactly once and stored as part of the merged results. Figure 6(d) shows a possible state of the three segments when their merge completes. Naturally, one may ask where the merged data is stored and what happens if it cannot be contained in memory. Since the merge data has 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 below in Section 4.2.

4.2 Pipelined Shuffle, Merge and Reduce

Besides avoiding repetitive merges, our algorithm removes the serialization barrier between merge and reduce. As described in Section 4.1, the merged data has <key,val> records 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.

In view of the possibility to closely couple the shuffle, merge and reduce phases, we design Hadoop-A with a full pipeline, which is shown in Figure 7. In this pipeline, MapTasks map data splits 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 light-weight, compared to shuffle and merge operations. As soon as the last MOF is available, completed PQs are constructed. The full pipeline of shuffle, merge, and reduce then starts. One may notice that there is

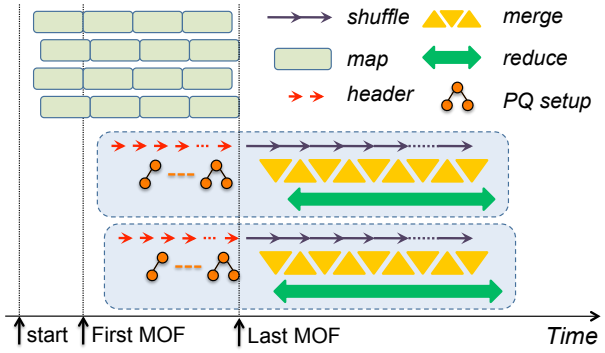


Figure 7. Pipelined Shuffle, Merge and Reduce

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, it is erroneous to send any data to the reduce function (for final results), while the intermediate result is yet to be produced by the map function.

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 as possible for final results.

5. Experimental Results

In this section, we show experimental results from our evaluation of Hadoop-A on InfiniBand, compared to the original Hadoop on Gigabit Ethernet and IPoIB.

5.1 Testbed and Performance of Network Devices

We conduct our experiments on a cluster of 26 nodes. Each node is equipped with dual-socket quad-core 2.13GHz Intel Xeon processors and 8 GB of DDR2 800 MHz memory, along with 8x PCI-Express Gen 2.0 bus. Four cores on a socket share 4 MB L2 cache. These nodes run Linux 2.6.18-164.el5 kernels. All nodes are equipped with Mellanox ConnectX-2 QDR Host Channel Adaptors and are connected to a 36-port InfiniBand QDR switch. We use the InfiniBand software stack, OFED [2] version 1.5.2, as released by Mellanox. Each node has a 250GB, 7200 RPM, Western Digital SATA hard drive.

The performance of RDMA is measured using the perf_test from OFED [2], that of IPoIB and Gigabit Ethernet (GigE) using the netperf [4] benchmark. For the performance of IPoIB and Gigabit Ethernet in Java, we use a Java-based TTCP benchmark [3].

Table 1. Comparison of Network Performance

Devices	Bandwidth (MB/sec)	
	Java	C
IB (RDMA)	–	3239.21
IB (IPoIB)	1078.40	1220.39
Gigabit Ethernet	122.31	124.13

Table 1 shows the comparison of peak throughput for three network protocols: RDMA, IPoIB and GigE. RDMA delivers much higher throughput in the C environment, but it is not available in a Java environment. IPoIB can achieve a peak performance of 1078.40 MB/sec and 1220.39 MB/sec, respectively, when running in Java and C. For all our tests, we use the default *connected* mode of IPoIB. GigE can achieve a peak of 122.31 MB/sec and 124.13 MB/sec in Java and C, respectively. Note that compute nodes in our system have relatively slow processors and memory buses compared to the best available in the market. Thus these numbers may differ slightly from vendors' advertised performance numbers. Nonetheless, these network protocols provide a good set to compare the performance of Hadoop and Hadoop-A.

5.2 Overall Performance

We run Hadoop TeraSort and WordCount programs with different data sizes and different numbers of slave nodes. We choose the data size per split as 256MB. Each slave has 8 MapTasks and 4 ReduceTasks. Figure 8 shows the performance comparison between Hadoop-A and Hadoop for TeraSort and WordCount programs. The Y-axis shows the percentage of completion for Map and Reduce Tasks. The X-axis shows the progress of time during execution. As shown by (a) and (b), Hadoop-A speeds up the total execution time significantly for the TeraSort program, by more than 47% compared to Hadoop over IPoIB or GigE. WordCount, on the other hand, does not benefit much from Hadoop-A because of the small size of its intermediate data and low requirement on data movement, as shown by (c) and (d). We focus on TeraSort for the rest of the performance evaluation.

Figure 8(a) shows that MapTasks of TeraSort complete much faster with Hadoop-A, especially when the percentage of completion goes over 50%. This is because Hadoop-A only performs light-weight operations such as fetching headers and setting up PQ, thereby leaving more resources such as disk bandwidth for MapTasks. Note that Hadoop reports the progress of ReduceTasks as soon as data is being merged. Hadoop-A implements the same. Because Hadoop-A waits until the completion of last MOF before merge, this results in seemingly slow progress of ReduceTasks in Hadoop-A. Hadoop-A still makes progress on ReduceTasks. Once it begins reporting, its progress in terms of percentage jumps up quickly, as shown by (b) and (d) for TeraSort and WordCount, respectively.

5.3 Tuning the Size of Data Splits

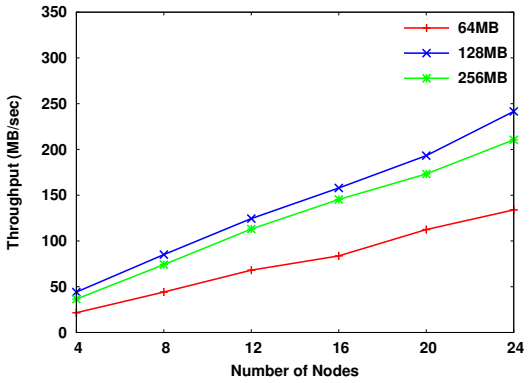
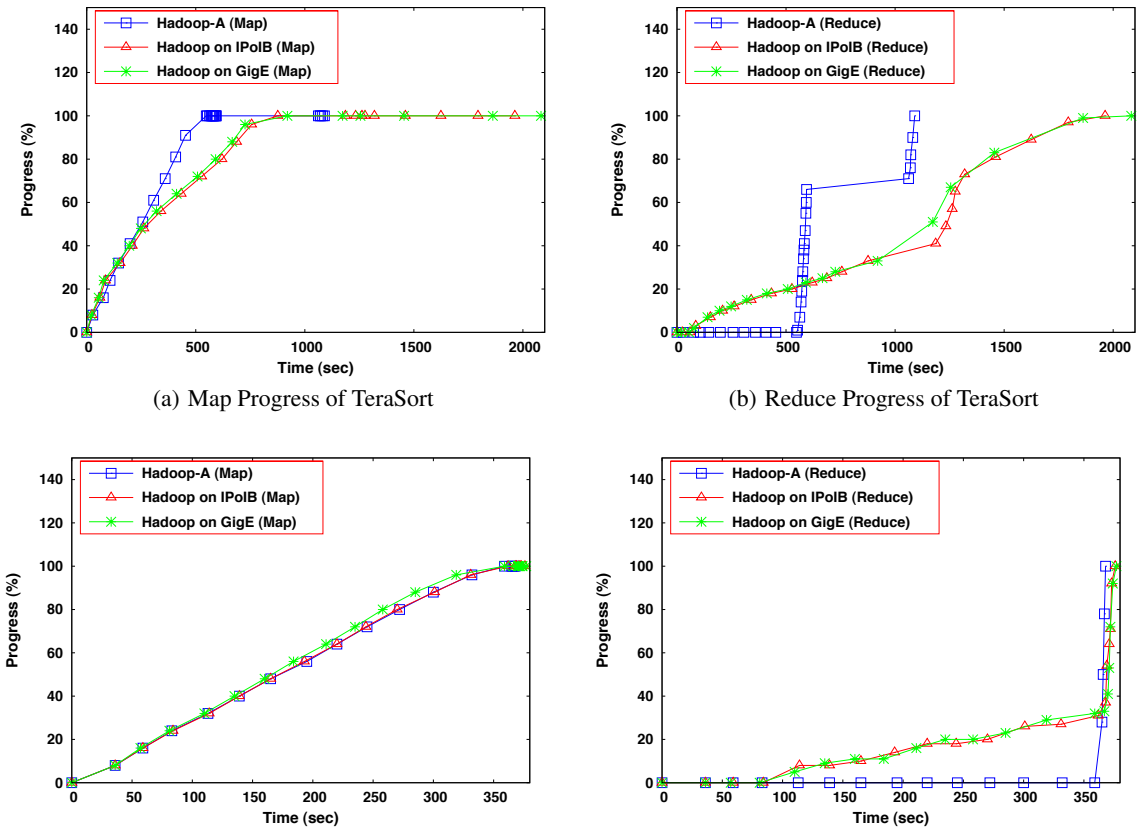


Figure 9. Performance with Different Block Sizes



(a) Map Progress of TeraSort

(b) Reduce Progress of TeraSort

(c) Map Progress of WordCount

(d) Reduce Progress of WordCount

Figure 8. Progress Diagrams of TeraSort and WordCount

The size of data splits determines the granularity of the pipeline for Hadoop and Hadoop-A. To understand the impact of split size to the performance of Hadoop-A, we measure the throughput of TeraSort using different split sizes. Figure 9 shows our evaluation results. Among three split sizes, 64MB, 128MB and 256MB, Hadoop-A achieves the best throughput for TeraSort when the split size is 128MB.

5.4 Dissection of ReduceTask Data Processing

As shown in Figures 2 and 7, Hadoop-A avoids the serialization barrier between shuffle/merge and reduce phases of Hadoop ReduceTasks. Instead, it has a separate, light-weight phase to fetch headers and set up PQ. To shed light on how well the full pipeline of shuffle, merge and reduce in Hadoop-A benefits the performance of Hadoop, we run TeraSort with 4GB per ReduceTask and measure the time of different phases in Hadoop and Hadoop-A. The way that Hadoop and Hadoop-A maintain their execution statistics makes it possible and greatly simplifies this measurement. We collect timestamps at the begin and end of individual phases, and calculate the elapsed time. The timestamps across different ReduceTasks are close to each other. We take the average across different ReduceTasks.

Table 2 shows our measurement results, including both the absolute time in seconds and the percentage of different phases during the execution of ReduceTasks. Note that the execution of ReduceTasks differs from that of the entire program by only a small duration, roughly the time taken to complete the first MOF. This can also be validated from Figures 2 and 7.

Hadoop-A significantly cuts down on the execution time of ReduceTasks. The shuffle/merge phase in Hadoop dominates the execution of ReduceTasks. Hadoop-A avoids shuffle/merge and performs only light-weight tasks such as header fetching and PQ setup. The PQ setup phase (including header fetching) is much faster compared to the shuffle/merge phase in Hadoop. Interestingly, even though Hadoop-A delays the start of merge until the completion of last MOF and overlaps the merge operation together with shuffle and reduce, the execution of the full pipeline for ReduceTasks can still be as much as 18% faster than the stand-alone reduce phase in Hadoop. This is mainly because our merge algorithm is levitated above local disks and avoids repetitive merges.

Hadoop Fetching Throughput – One puzzling observation from Table 2 is the comparison between IPoIB and GigE for Hadoop. As listed in Table 1, IPoIB delivers more than 8 times higher throughput than GigE, but such performance power does not result in performance advantage for Hadoop. We wonder if IPoIB can really speed up data movement for TeraSort with the original Hadoop. We create a Java program to mimic the execution of HTTP GET operations in Hadoop ReduceTasks and measure the data fetching throughput. In this program, a client makes iterative requests to fetch data of a certain size, ranging from 4MB to 256MB. The server then retrieves data of the same size from a disk

Table 2. Time Breakdown of ReduceTask Execution (Seconds)

Slaves	Hadoop on GigE		Hadoop on IPoIB		Hadoop-A	
	Shuffle/Merge	Reduce	Shuffle/Merge	Reduce	PQ Setup	Shuffle/Merge/Reduce
4	1238.70 (66.2%)	633.45 (33.8%)	1179.76 (65.7%)	615.09 (34.3%)	452.92 (42.5%)	613.70 (57.5%)
6	1066.44 (67.7%)	522.79 (32.3%)	1152.52 (65.7%)	602.20 (34.3%)	426.91 (45.5%)	511.07 (54.5%)
8	1016.43 (65.0%)	551.38 (35.0%)	1190.05 (65.0%)	641.79 (35.0%)	441.47 (44.2%)	556.81 (55.8%)
10	1137.70 (64.1%)	629.21 (36.0%)	1208.32 (65.1%)	649.44 (34.9%)	437.97 (44.6%)	543.89 (55.4%)
12	1143.04 (65.0%)	607.22 (35.0%)	1208.22 (65.4%)	639.27 (34.6%)	442.98 (45.1%)	538.11 (54.9%)
14	1194.74 (66.0%)	622.86 (34.1%)	1166.19 (65.6%)	612.02 (34.4%)	446.10 (45.3%)	539.89 (54.7%)
16	1182.15 (65.7%)	616.34 (34.2%)	1169.19 (64.9%)	631.51 (35.0%)	455.07 (47.6%)	501.21 (52.4%)
18	1192.02 (65.6%)	624.65 (34.4%)	1195.53 (65.6%)	628.12 (34.4%)	461.09 (45.7%)	547.89 (54.3%)
20	1158.60 (65.8%)	602.68 (34.2%)	1178.89 (65.7%)	614.57 (34.3%)	461.91 (46.3%)	535.33 (53.7%)
22	1164.56 (66.3%)	593.30 (33.7%)	1170.11 (65.8%)	608.39 (34.2%)	463.11 (45.2%)	563.76 (54.8%)
24	1150.01 (65.0%)	599.97 (35.0%)	1148.26 (65.9%)	597.09 (34.1%)	460.01 (47.4%)	509.81 (52.6%)

file (8GB or 16GB) and replies back to the client.

Figure 10 shows the comparison of fetching throughput between IPoIB and GigE. With an 8GB file, both IPoIB and GigE can achieve a throughput nearly 96MB/sec. This is a bit higher than the peak rate of our disks because of operating system cache effects, given a memory size of 8GB. With a 16GB file, however, the throughput drops below 80MB/sec for both IPoIB and GigE because of the loss of cache effects. In both case, the throughput results reveal that fast network throughput of IPoIB does not benefit data-intensive TeraSort programs in Hadoop.

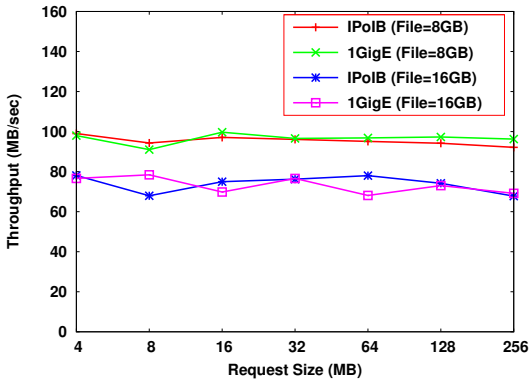
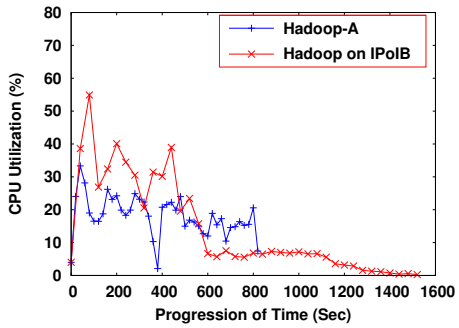
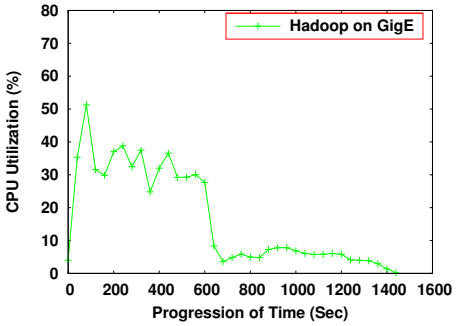


Figure 10. Data Fetching Throughput via HTTP Get



(a) Hadoop-A vs. Hadoop-IPoIB



(b) Hadoop-GigE

Figure 11. Comparison of CPU Utilization

5.5 CPU Utilization

We measure CPU utilization during the execution of TeraSort every 2 seconds. The percentage of CPU usage for 8 cores is recorded. We then take the average across all slaves at the same timestamp. Figure 11(a) shows the comparison of the average CPU utilization between Hadoop-A and Hadoop on IPoIB. Figure 11(b) shows that of Hadoop on GigE. These results are from a TeraSort program on 20 slave nodes. Similar comparisons are observed for TeraSort on different number of nodes. Clearly, Hadoop-A has less CPU utilization compared to Hadoop. Cumulatively, Hadoop-A has a CPU utilization of 18.7% at the time of its job completion, compared to 29.3% and 33.5% for Hadoop-IPoIB and Hadoop-GigE, respectively, at the same time point. Relatively, the reduction is 36.2% and 44.2%, respectively. Note that Hadoop-A has higher CPU usage towards the end of its completion, during which it is running a pipeline of shuffle/merge/reduce operations. The CPU utilization curve reveals that Hadoop-A is able to leverage system resource and sustain this pipeline, thereby shortening the execution time of TeraSort.

5.6 Scalability of Hadoop-A

Being able to leverage more nodes to process large amounts of data is an essential feature of Hadoop. We want to ensure Hadoop-A can deliver scalability in a similar manner. So we measure

the total execution time of TeraSort in two scaling patterns: one with fixed amount of total data (128GB) and increasing number of nodes, and the other with fixed data (4GB) per ReduceTask and increasing number of nodes. The aggregated throughput is calculated by dividing the total size with the program execution time.

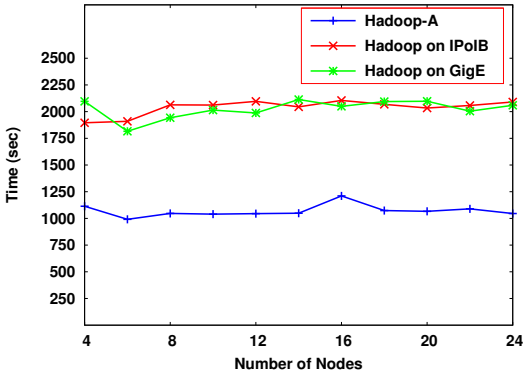


Figure 12. Hadoop-A Scalability with Increasing Data Size

Figure 12 shows the scalability comparison between Hadoop-A and Hadoop with a fixed data size per node. Both Hadoop and Hadoop-A can achieve linear scalability. Hadoop-A can cut the execution time by approximately 50% and therefore double the throughput. Figure 13 shows the scalability comparison between Hadoop-A and Hadoop with a fixed size of total data. Again both Hadoop and Hadoop-A can achieve good scalability. Hadoop-A can cut the execution time by up to 40% and 43%, compared to Hadoop on IPoIB and GigE, respectively. Conversely, this results in an throughput improvement of 66.7%, and 75.4%, respectively. These results adequately demonstrate better scalability of Hadoop-A for large-scale data processing compared to the original Hadoop.

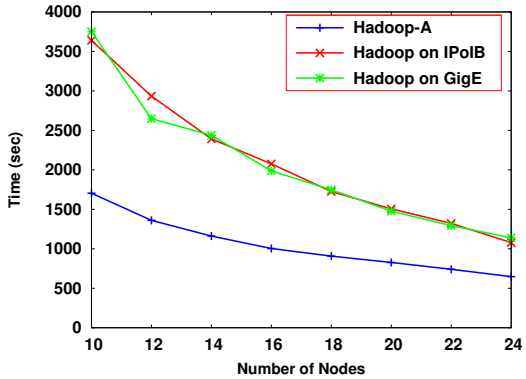


Figure 13. Hadoop-A Scalability with Increasing Number of Nodes

6. Related Work

MapReduce is a programming model for large-scale arbitrary data processing. The model popularized by Google provides very simple but powerful interfaces, while hiding complex details of parallelizing computation, fault-tolerance, distributing data and load balancing [6]. Its open-source implementation, Hadoop, provides a software framework for distributed processing of large datasets [1].

A rich set of research has been published on improving the performance of MapReduce recently. Originally, the Hadoop scheduler assumed that all nodes in a cluster were homogeneous and made progress with the same speed. Jiang et al. [11] conducted a comprehensive performance study of MapReduce (Hadoop), concluding that the total performance could be improved by a factor of 2.5 to 3.5 by carefully tuning the factors, including: I/O mode, indexing, data parsing, grouping schemes and block-level scheduling. Zaharia et al. [21] designed a new scheduling algorithm, Longest Approximate Time to End (LATE), for heterogeneous environments where ideal application environment might not be available. To fully take advantage of the multicore and multiprocessor systems, Ranger et al. [15] designed Phoenix, a programming API and runtime system for shared-memory systems. In Phoenix, users only need to write simple parallel code without considering the complexity of thread creation, dynamic task scheduling, data partitioning, and fault tolerance across processor nodes. Considering the fact that original data structure used to group key/value pairs would be a primary performance bottleneck on the clusters of multicore architecture, Kaashoek et al. [13] designed a new MapReduce library with a compromise data structure, which outperforms its simpler peers, including Phoenix. Seo et al. [16] has leveraged prefetching and pre-shuffling techniques into MapReduce. They have shown that these techniques can improve the overall performance of Hadoop in shared environment. In [8], Sun engineers describe their work on executing Hadoop over Lustre File System.

The closest work to ours is MapReduce online as proposed by Condie et al. [5]. As discussed in Section 1, it focuses on fast data output from MapTasks and changes Hadoop fault handling mechanism. Our work addresses the similar performance issue of data movement, but differs from these studies by enabling network-levitated merge to avoid disk access and overlapping merge and reduce at ReduceTasks.

Leveraging RDMA from high speed networks for high-performance data movement has been very popular in various programming models and storage paradigms. Liu *et al.* [12] designed RDMA-based MPI over InfiniBand. Implementations of PVFS [14] on top of RDMA networks such as InfiniBand and Quadrics were described in [19] and [20], respectively. A recent evaluation [18] of Hadoop Distributed File system (HDFS) [17] used the SDP [10] and IPoIB protocols of InfiniBand [9], and the authors showed that MapReduce is still unable to leverage the RDMA (Remote Direct Memory Access) communication mechanism available from high-performance RDMA interconnects such as InfiniBand and RoCE [7] (RDMA over Converged Ethernet). Our acceleration framework uses RDMA as its first communication protocol besides the TCP/IP protocol in the original Hadoop. Our work complements previous efforts to enable RDMA for Hadoop large-scale data processing programming model. Particularly, we show that RDMA is very beneficial in reducing Hadoop CPU utilization.

7. Conclusions

We have examined the design and architecture of Hadoop's MapReduce framework in great detail. Particularly, our analysis has focused on data processing inside ReduceTasks. We reveal that

there are several critical issues faced by the existing Hadoop implementation, including its merge algorithm, its pipeline of shuffle, merge, and reduce operations, as well as its lack of support for RDMA interconnects. We have designed and implemented Hadoop-A as an extensible acceleration framework that can allow plugin components to address all these issues. By introducing a new network-levitated algorithm that merges data without touching disks and designing a full pipeline of shuffle, merge, and reduce phases for ReduceTasks, we have successfully accomplished an accelerated Hadoop framework, Hadoop-A. Our experimental results show that Hadoop-A doubles the data processing throughput of Hadoop, and also reduces CPU utilization by more than 36% by leveraging RDMA-based data movement. For future work, we plan to investigate the scalability of Hadoop-A on large-scale systems that are equipped with RoCE networks such as RDMA-capable 10Gigabit Ethernet.

Acknowledgments

We are very thankful to Dr. Douglas Thain from University of Notre Dame for his help on finalizing the paper.

This work is funded in part by a Mellanox grant to Auburn University, and by National Science Foundation awards CNS-0917137 and CNS-1059376.

8. References

[1] Apache Hadoop Project. http://hadoop.apache.org/.

[2] Open Fabrics Alliance. http://www.openfabrics.org.

[3] Test-TCP. http://www.pcausa.com/Utilities/pcattcp.htm.

[4] The Public Netperf Homepage. http://www.netperf.org/netperf/NetperfPage.html.

[5] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. MapReduce Online. In *7th USENIX Symp. on Networked Systems Design and Implementation (NSDI)*, pages 312–328, April 2010.

[6] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Sixth Symp. on Operating System Design and Implementation (OSDI)*, pages 137–150, December 2004.

[7] HPC Wire. RoCE: An Ethernet-InfiniBand Love Story. http://www.hpcwire.com/blogs/.

[8] Sun Microsystems Inc. Using Lustre with Apache Hadoop. http://wiki.lustre.org.

[9] Infiniband Trade Association. http://www.infinibandta.org.

[10] InfiniBand Trade Association. Socket Direct Protocol Specification V1.0, 2002.

[11] Dawei Jiang, Beng Chin Ooi, Lei Shi, and Sai Wu. The performance of mapreduce: An in-depth study. In *Proceedings of the 36th International Conference on Very Large Data Bases (VLDB)*, volume 3, pages 472–483, 2010.

[12] Jiuxing Liu, Jiesheng Wu, and Dhabaleswar K. Panda. High Performance RDMA-Based MPI Implementation over InfiniBand. *International Journal of Parallel Programming (IJPP)*, 32:167–198, 2004.

[13] Yandong Mao, Robert Morris, and Frans Kaashoek. Optimizing mapreduce for multicore architectures. Technical Report MIT-CSAIL-TR-2010-020, MIT, May 2010.

[14] P. H. Carns and W. B. Ligon III and R. B. Ross and R. Thakur. PVFS: A Parallel File System For Linux Clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, GA, October 2000.

[15] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary R. Bradski, and Christos Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *Int'l Symp. on High Performance Computer Architecture (HPCA)*, pages 13–24. IEEE Computer Society, 2007.

[16] Sangwon Seo, Ingook Jang, Kyungchang Woo, Inkyo Kim, Jin-Soo Kim, and Seungryoul Maeng. HPMR: Prefetching and pre-shuffling in shared MapReduce computation environment. In *IEEE Cluster Conference*, pages 1–8, August 2009.

[17] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[18] S. Sur, H. Wang, J. Huang, X. Ouyang, and D. K. Panda. Can High-Performance Interconnects Benefit Hadoop Distributed File System? In *Workshop on Micro Architectural Support for Virtualization, Data Center Computing, and Clouds (MASVDC). Held in Conjunction with MICRO*, Dec 2010.

[19] Jiesheng Wu, Pete Wychoff, and Dhabaleswar K. Panda. PVFS over InfiniBand: Design and Performance Evaluation. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, Kaohsiung, Taiwan, October 2003.

[20] Weikuan Yu, Shuang Liang, and Dhabaleswar K. Panda. High Performance Support of Parallel Virtual File System (PVFS2) over Quadrics. In *Proceedings of The 19th ACM International Conference on Supercomputing (ICS)*, Boston, Massachusetts, June 2005.

[21] Matei Zaharia, Andrew Konwinski, Anthony D. Joseph, Randy H. Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. Technical Report UCB/EECS-2008-99, EECS Department, University of California, Berkeley, Aug 2008.