Design and Evaluation of Network-Levitated Merge for Hadoop Acceleration

Weikuan Yu, *Member, IEEE*, Yandong Wang, and Xinyu Que

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 systems. These include a serialization barrier that delays the reduce phase, repetitive merges, and disk accesses, and the lack of portability to different interconnects. To keep up with the increasing volume of data sets, Hadoop also requires efficient I/O capability from the underlying computer systems to process and analyze data. We describe Hadoop-A, an acceleration framework that optimizes Hadoop with plug-in components for fast data movement, overcoming the 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 significantly speeds up data movement in MapReduce and doubles the throughput of Hadoop. In addition, Hadoop-A significantly reduces disk accesses caused by intermediate data.

Index Terms—Hadoop, MapReduce, network-levitated merge, Hadoop acceleration, cloud computing

1 Introduction

MapReduce has emerged as a popular and easy-to-use programming model for cloud computing [1]. It has been used by numerous organizations to process explosive amounts of data, perform massive computation, and extract critical knowledge for business intelligence. Hadoop [2] is an open source implementation of MapReduce, currently maintained by the Apache Foundation, and supported by leading IT companies such as Facebook 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 runtime execution statistics, and handles possible faults and errors through task reexecution. Between the two phases, a ReduceTask needs to fetch a part of the intermediate output from all finished MapTasks. Globally, this leads to the *shuffling* of intermediate data (in segments) from all MapTasks to all ReduceTasks. For many data-intensive MapReduce programs, data shuffling can lead to a significant number of disk operations, contending for the limited I/O bandwidth. This presents a severe problem of disk I/O contention in MapReduce programs, which entails further research on efficient data shuffling and merging algorithms.

A number of studies [3], [4], [5] have been carried out to improve the performance of Hadoop MapReduce framework. Condie et al. [4] 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. It remains as a critical issue 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 MapReduce, no ReduceTasks can start reducing data until all intermediate data have been merged together. This results in a serialization barrier that significantly delays the reduce operation of ReduceTasks. More importantly, the current merge algorithm in Hadoop merges intermediate data segments from MapTasks when the number of available segments (including those that are already merged) goes over a threshold. These segments are spilled to local disk storage when their total size is bigger than the available memory. This algorithm causes data segments to be merged repetitively and, therefore, multiple rounds of disk accesses of the same data (cf. Section 2.2).

To address these critical issues for Hadoop MapReduce framework, we have designed Hadoop-A, a portable acceleration framework that can take advantage of plug-in components for performance enhancement and protocol optimizations. Several enhancements are introduced: 1) a novel algorithm that enables ReduceTasks to perform data merging without repetitive merges and extra disk accesses; 2) a full pipeline is designed to overlap the shuffle, merge, and reduce phases for ReduceTasks; and 3) a portable implementation of Hadoop-A that can support both TCP/IP and remote direct memory access (RDMA). Since ReduceTasks are able to merge data by staying above local

- W. Yu is with the Department of Computer Science & Software Engineering, Auburn University, 3101 Shelby Center, Auburn, AL 36849. E-mail: wkyu@auburn.edu.
- Y. Wang and X. Que are with the Department of Computer Science & Software Engineering, Auburn University, 2105 Shelby Center, Auburn, AL 36849. E-mail: {wangyd, xque}@auburn.edu.

Manuscript received 13 Aug. 2012; revised 2 Feb. 2013; accepted 10 Feb. 2013; published online 27 Feb. 2013.
Recommended for acceptance by J. Wang.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2012-08-0727.
Digital Object Identifier no. 10.1109/TPDS.2013.59.

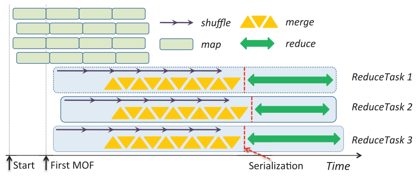


Fig. 1. Serialization between shuffle/merge and reduce phases.

disks, we refer to this new algorithm as network-levitated merge (NLM). We have carried out an extensive set of experiments to evaluate the performance of Hadoop-A. 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 double the throughput of Hadoop data processing.

The rest of the paper is organized as follows. Section 2 provides the motivation. We describe the network-levitated merge algorithm in Section 3, followed by the portable implementation of Hadoop-A in Section 4. Section 5 provides experimental results. We then discuss our perspectives on Hadoop-A scalability in Section 6. Section 7 provides a review of related work. Finally, we conclude the paper in Section 8.

2 Motivation

Hadoop's MapReduce implementation enables a convenient and easy-to-use data processing framework. An overview is provided in the electronic appendix, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2013.59. Our characterization and analysis reveal a number of issues, including 1) the serialization between Hadoop shuffle/merge and reduce phases, 2) repetitive merges and disk access, and 3) the lack of portability to different interconnects.

2.1 A Serialization in Hadoop Data Processing

Hadoop strives to pipeline the data processing. It is indeed able to do so, particularly for map and shuffle/merge phases. As shown in Fig. 1, after a brief initialization period, a pool of concurrent MapTasks starts the map function on the first set of data splits. As soon as *Map Output Files (MOFs)* are generated from these splits, a pool of ReduceTasks starts to fetch partitions from these MOFs. At each ReduceTask, when the number of segments is larger than a threshold, or when their total data size is more than a memory threshold, the smallest segments are merged.

For the 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 Fig. 1, contains an implicit serialization. At each ReduceTask, only until all its segments are available and merged, will the reduce phase start to process data segments via the reduce function. This 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 has revealed that this can increase the total execution time by nearly 40 percent (cf. Table 2 for time dissection in the electronic appendix, available in the online supplemental material).

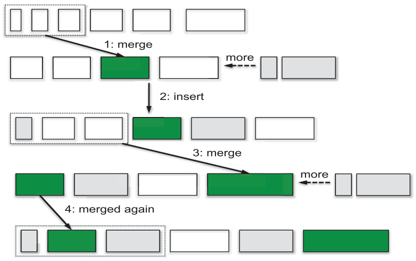


Fig. 2. An illustration of repetitive merges.

2.2 Repetitive Merges and Disk Access

Hadoop 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. Fig. 2 shows a common sequence of merge operations in Hadoop. For the purpose of illustration, we use 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 Fig. 2. Under memory pressure, this will incur disk access. The resulting segment is inserted back into the heap based on its relative size.

When more segments arrive (as shown in Step 2), the threshold is reached 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 segments back if they have been stored on local disks. As even more segments arrive, a previously merged segment will be grouped into another set and merged again, as shown in Step 4. 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, causing degraded performance for Hadoop. Therefore, an alternative merge algorithm is critical for Hadoop to mitigate the impact of repetitive merges and extra disk accesses.

2.3 The Lack of Network Portability

Besides the TCP/IP protocol, Hadoop does not support other transport protocols such as RDMA on InfiniBand [6] and 10-Gigabit Ethernet (10GigE) that have matured in the high-performance computing (HPC) community. Simply replacing the network hardware with the latest interconnect technologies such as InfiniBand and 10GigE and continuing to run Hadoop on TCP/IP will not enable Hadoop to leverage the strengths of RDMA. It is worth noting that despite the high-price differential between RDMA-capable interconnects and traditional commodity Gigabit Ethernets, such price differences have shrunk significantly over the past few years. Many popular commodity interconnects,

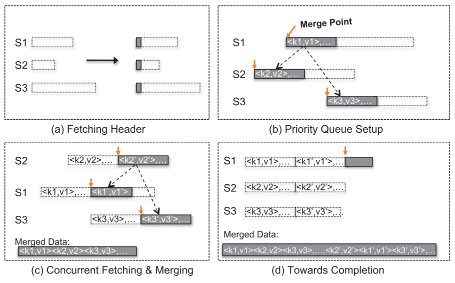


Fig. 3. A network-levitated merge algorithm.

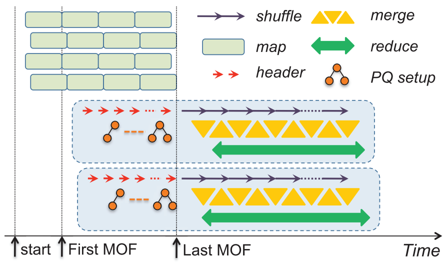


Fig. 4. Pipelined shuffle, merge, and reduce.

such as 10GigE, are becoming RDMA-capable as well. Thus, the lack of portability on multiple interconnects will prevent Hadoop from keeping up with the advances of other computer technologies, particularly when more powerful processors, storage, and interconnect devices are deployed to various computing and data centers.

3 A Network-Levitated Pipeline of Shuffle, Merge, and Reduce Phases

To address the first two issues in Hadoop as mentioned in Section 2, we describe a network-levitated merge algorithm that avoids repeated merges and then detail the construction of a new pipeline to eliminate the serialization barrier.

3.1 Network-Levitated Merge

Hadoop resorts to repetitive merges because of limited memory compared to the size of data. For each remotely completed MOF, each ReduceTask invokes an *HTTP GET* request to query the partition length, pull the entire data, and store locally in memory or on disk. This incurs many memory loads/stores and/or disk I/O operations. We design an algorithm that can merge all data partitions exactly once and, at the same time, stay *levitated* above local disks. Fig. 3 shows our network-levitated merge algorithm. The key idea is to leave data on remote disks until it is time to merge the intended data records.

As shown in Fig. 3a, 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. 3b.

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, our algorithm can fetch the next set of records to resume the merge operation. In fact, our algorithm always ensures that the fetching of upcoming records happens concurrently with the merging of available records. As shown in Fig. 3c, 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. 3d 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.

3.2 Pipelined Shuffle, Merge, and Reduce

Besides avoiding repetitive merges, our algorithm removes the serialization barrier between merge and reduce. As described in Section 3.1, the merged data have <key,val> pairs ordered in their final order and can be delivered to the Java-side ReduceTask as soon as they are available. Thus, the reduce phase no longer has to wait until the end of the merge phase.

In view of the possibility to closely couple the shuffle, merge, and reduce phases, they can form a full pipeline as shown in Fig. 4. In this pipeline, MapTasks map data split as soon as they can. When the first MOF is available, ReduceTasks fetch the headers and build up the PQ. These activities are pipelined. Header fetching and PQ setup are pipelined and overlapped with the map function, but they are very lightweight, compared to shuffle and merge operations. As soon as the last MOF is available, completed PQs are constructed. The full pipeline of shuffle, merge, and reduce then starts. One may notice that there is still a serialization between the availability of the last MOF and the beginning of this pipeline. This is inevitable in order for Hadoop to conform to the correctness of the MapReduce programming model. Simply stated, before all <key,val> pairs are available, it is erroneous to send any <key,val> pair to the reduce function (for final results) because its relative order with future <key,val> pairs is yet to be decided.

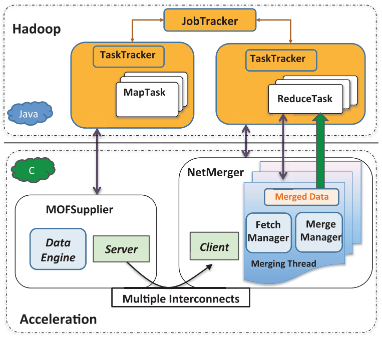


Fig. 5. Software architecture of Hadoop-A.

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.

4 Portable Hadoop Acceleration through Network-Levitated Merge

With the network-levitated merge algorithm, it is also important to design an implementation that can enable Hadoop Acceleration as a portable plug-in on different interconnects without affecting existing Hadoop applications.

4.1 Software Architecture of Hadoop-A

Fig. 5 shows the architecture of Hadoop-A. Two new user-configurable plug-in 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. 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 MOFSupplier and NetMerger plug-ins 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. Additional implementation details are provided in the electronic appendix, available in the online supplemental material.

4.2 Portable Implementation

Hadoop-A is designed to be portable, in which we have developed an implementation that supports both the RDMA protocol for interconnects such as InfiniBand, and the TCP/IP protocol for ubiquitous Ethernet networks. Apart from traditional TCP/IP protocol, InfiniBand Architecture defines RDMA [7] that supports zero-copy data transfer. Through RDMA, applications can directly access memory buffers of remote processes so long as those buffers have to be pinned during the communication.

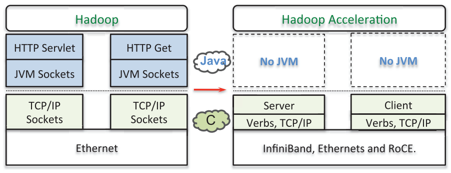


Fig. 6. Portable data movement on different protocols.

The left half of Fig. 6 shows the communication stack currently used for Hadoop data shuffling. When notified of the completion of a MOF, Hadoop ReduceTasks invoke copy threads to fetch their data partitions through Java-based HTTP GET requests. 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.

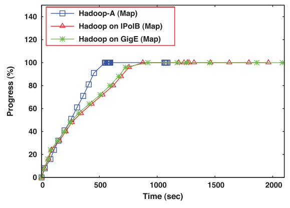
Hadoop-A allows us to introduce alternative communication protocols for data shuffling in Hadoop. To support different interconnects, we design our data shuffling protocol completely in the native C language, as shown on the right of Fig. 6. The new implementation can transfer data on top of both RDMA verbs and TCP/IP protocols. It completely avoids the overhead of JVM for Hadoop data shuffling. With this portable implementation, Hadoop-A can run on both InfiniBand and Ethernet networks.

The transport protocol of Hadoop-A consists of a server in the MOFSupplier and a client in the NetMerger. On the client side, one thread is designed particularly to send fetch/connection requests to remote servers; meanwhile, multiple data threads are running concurrently to retrieve data from remote servers. Correspondingly, on the server side, one thread is dedicated to listening to the incoming connection requests and delegates actual data transfer to data threads.

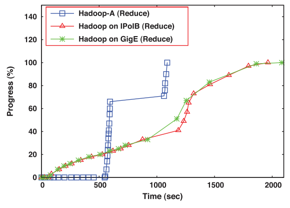
We use the portable RDMA CM protocol for connection establishment on RDMA networks and the socket connection protocol on TCP/IP networks. Once connected, clients and servers communicate data through preallocated memory buffers. To constrain the total number of connections and their memory consumption, we put a configurable threshold (512 by default) on the number of active connections on any interconnect. When the total number of connections reaches this threshold, an old connection will be torn down before establishing a new connection. Hadoop-A create connections on a per node basis. Compared to the original Hadoop, which creates connections per Java Copier thread, our connection model improves the scalability by several folds. Nonetheless, running Hadoop-A across 1,000 s or 10,000 s of nodes will lead to a host of challenging efficiency and scalability issues, which we would like to pursue as our future work.

5 Experimental Results

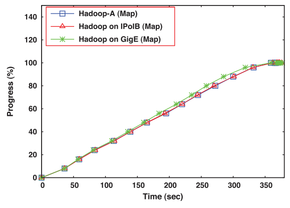
We conduct our experiments on a cluster of 26 nodes. Each node is equipped with dual-socket quad-core 2.13-GHz Intel Xeon processors and 8 GB of DDR2 800-MHz memory, along with 8x PCI-Express Gen 2.0 bus. Four cores on a socket share 4-MB L2 cache. These nodes run



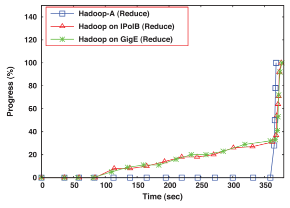
(a) Map Progress of TeraSort



(b) Reduce Progress of TeraSort



(c) Map Progress of WordCount



(d) Reduce Progress of WordCount

Fig. 7. Progress diagrams of TeraSort and WordCount.

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 providing up to 40-Gb/s full bisection bandwidth per port. We use the InfiniBand software stack, OFED [8] version 1.5.2, as released by Mellanox. Each node has two 250-GB, 7,200-RPM, Western Digital SATA hard drives. The basic network performance results are provided in the electronic appendix, available in the online supplemental material. IPoIB (an emulated implementation of TCP/IP on InfiniBand) provides standardized IP encapsulation over InfiniBand links. Therefore, all applications that require TCP/IP can continue to run without any modification. Detailed description of IPoIB can be found in [9].

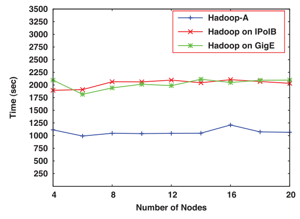
5.1 Benchmarks

In our evaluation experiments, we employ TeraSort and WordCount benchmarks from the default Hadoop package. TeraSort is a *de facto* standard Hadoop I/O benchmark, in which the sizes of intermediate data and the final output are as large as the input size. TeraSort generates a lot of intermediate data and can expose the I/O bottleneck across the Hadoop data processing pipeline. WordCount counts the occurrence of words in the input data and generates relatively smaller intermediate data. Besides experimental results described here, additional results are also provided in the electronic appendix, available in the online supplemental material.

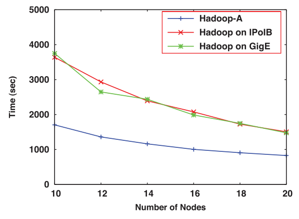
5.2 Overall Performance

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 256 MB. Each slave has eight MapTasks and four ReduceTasks. Fig. 7 shows the performance comparison between Hadoop-A and Hadoop for TeraSort and WordCount programs (Hadoop-A is evaluated with RDMA protocol). 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 in (a) and (b), Hadoop-A speeds up the total execution time significantly for the TeraSort program, by more than 47 percent 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 in (c) and (d). We focus on TeraSort for the rest of the performance evaluation.

Fig. 7a shows that MapTasks of TeraSort complete much faster with Hadoop-A, especially when the percentage of completion goes over 50 percent. This is because Hadoop-A only performs lightweight 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 are 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 in (b) and (d) for TeraSort and WordCount, respectively.



(a) Increasing Data Size



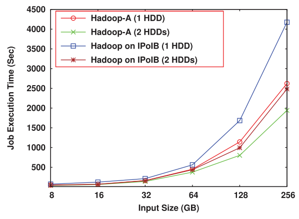
(b) Increasing Number of Nodes

Fig. 8. Hadoop-A scalability evaluation.

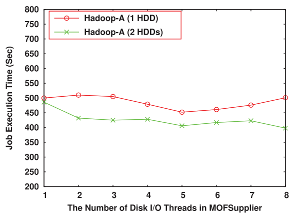
5.3 Scalability of Hadoop-A

Being able to leverage more nodes to process large amounts of data is an essential feature of Hadoop. We want to ensure that 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 (128 GB) and increasing number of nodes, and the other with fixed data (4 GB) per ReduceTask and increasing number of nodes. The aggregated throughput is calculated by dividing the total size with the program execution time.

Fig. 8a 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 reduce the execution time by approximately 50 percent and, therefore, double the throughput. Fig. 8b 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. (The running time of jobs are reduced by 60.6 and 51.5 percent when the number of nodes increases from 10 to 20, in the case of Hadoop and Hadoop-A, respectively.) Hadoop-A can reduce the execution time by up to 40 and 43 percent, compared to Hadoop on IPoIB and GigE, respectively. Conversely, this results in a throughput improvement of 66.7 and 75.4 percent, respectively. These results adequately demonstrate that Hadoop-A is able to efficiently accelerate job execution, meanwhile achieve comparable scalability as Hadoop for large-scale data processing.



(a) Increasing the Number of Disks



(b) Increasing the Number of I/O Threads

Fig. 9. Tuning of I/O performance.

5.4 Performance on Multiple Disks

Slave nodes in a Hadoop cluster may be equipped with multiple disks; therefore, Hadoop MapTasks and ReduceTasks are designed to utilize the bandwidth of all local disks. When several tasks (either MapTask or ReduceTask) are running on the same node, their intermediate data are spread among all disks in a round robin manner. Thus, the I/O traffic to any single disk can be greatly reduced. This can help shorten the wait time of I/O requests. For this reason, Hadoop-A is also implemented with multiple I/O threads to support data accesses to multiple disks. Accordingly, we have measured the performance of Hadoop and Hadoop-A when multiple disks are used to store the intermediate data.

Fig. 9 shows the results of running TeraSort with different input sizes on 16 slave nodes. Increasing the number of disks can improve the performance of both Hadoop and Hadoop-A. When two disks are used for storing intermediate data, although the disk I/O bottleneck problem is significantly alleviated in Hadoop, Hadoop-A is still able to provide up to 21.9 percent better performance for 256-GB data. In addition, we observe that the improvement of Hadoop-A increases with bigger data size. This is because bigger data size leads to more I/O requests, which causes a more severe I/O bottleneck at the disk. Therefore, Hadoop-A can exploit benefits from the network-levitated merge algorithm.

In addition, we also notice that when multiple disks are used to store intermediate data, it is inefficient to use only one thread to read the data from all disks. This can cause underutilization of some disks while others are busy. Therefore, we implement multiple I/O threads to serve the fetch requests in parallel. We measure the performance of Hadoop-A when several threads are used within the MOFSupplier. Fig. 9b shows that multiple I/O threads can accelerate the processing of fetch requests and the improvement can be up to 18 percent when eight threads are used. However, for tests with only one disk, increasing the number of threads has slightly degraded the performance. This is because the only disk is already overloaded. More threads actually introduce higher interferences among requests. Overall, our experiments demonstrate that Hadoop-A is capable of effectively utilizing multiple disks to improve the performance of Hadoop clusters.

It is worth noting that adding more disks improves the performance of Hadoop through reducing the disk I/O contention. This applies to additional investment in other resources as well besides an investment on disks. However, our work is complementary to such hardware investments. Hadoop-A not only makes good use of the network, but also reduces the contention on disk bandwidth, thereby increasing the efficiency of I/O.

TABLE 1
I/O Blocks

	READ (MB)	WRITE (MB)
Hadoop	5,426	36,427
Hadoop-A	2,441	22,713

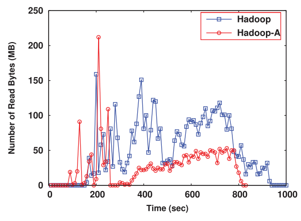
5.5 Improvement on Disk Accesses

Hadoop-A aims to lift the data shuffling and merging above disks for ReduceTasks through network-levitated merge algorithm. It avoids fetching the intermediate data to local disks. Instead, it leaves data on remote disks and only fetches small-size headers that can be stored in memory. In addition, the new merge algorithm sorts and merges all <key, val> pairs in memory. To assess the effectiveness of network-levitated merge, we have also measured the disk accesses during data shuffling under Hadoop-A and compared the results with that of Hadoop. We run TeraSort on 20 slave nodes with 160 GB as input size; each slave node has four MapTasks and two ReduceTasks. On each node, we run *vmstat* and *iostat* to collect I/O statistics and trace the output every 2 seconds.

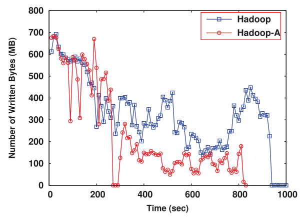
Table 1 shows the comparison of the number of bytes read and written by Hadoop and Hadoop-A into local disks per slave node. Overall, Hadoop-A significantly reduces the number of read blocks by 55.1 percent and write blocks by 37.6 percent. This effectively demonstrates that Hadoop-A reduces the number of I/O operations and relieves the load of underlying disks.

Fig. 10 shows the progressive profile of read and write bytes during the job execution. During the first 200 seconds in which MapTasks are active, there is no substantial difference between Hadoop and Hadoop-A in terms of disk I/O traffic. After the first 200 seconds, ReduceTasks start fetching and merging the intermediate data actively. Because Hadoop-A uses the network-levitated merge algorithm that completely eliminates the disk access for the shuffling and merging of data segments, we observe that Hadoop-A effectively reduces the number of bytes read from or written to the disks. Therefore, disk I/O traffic is significantly reduced during this period.

When disk bandwidth is a scarce resource, high disk I/O traffic can lead to long queuing time of I/O requests. This degrades the performance of the original Hadoop. To further analyze the benefit from the reduced disk accesses,



(a) Read Bytes



(b) Write Bytes

Fig. 10. I/O improvement.

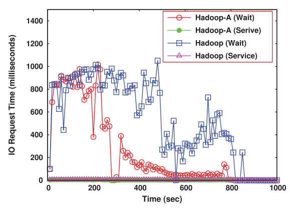


Fig. 11. I/O request service and wait time.

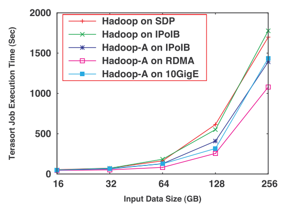


Fig. 12. Network-levitated merge and RDMA.

we measure the service time and wait time of I/O requests for Hadoop and Hadoop-A. The service time is the time taken to complete one I/O request and the wait time includes an I/O request's queuing time and its service time. The result is shown in Fig. 11. A couple of I/O behaviors can be observed from this figure. First, there is a big gap between Hadoop's service time and wait time, which indicates that most I/O requests have spent a huge amount of time waiting in the disk's queue. Second, the I/O service time is comparable between Hadoop and Hadoop-A. Fig. 11 shows that Hadoop-A leads to similar or lower I/O wait time during the first 200 seconds, which corresponds to the mapping phase of the execution. As the execution progresses, the I/O wait time of Hadoop-A is significantly reduced when job enters into the shuffle/merge and reduce phases. This demonstrates that the reduction of disk accesses contributes to the reduction of I/O wait time. Taken together, these experiments indicate that Hadoop-A with network-levitated merge can effectively improve the I/O performance in Hadoop, thereby effectively shortening job execution time.

5.6 Performance Benefits of Network-Levitated Merge and RDMA

To investigate the respective improvement brought by network-levitated merge and RDMA, respectively, we compare the performance of Hadoop-A with different network protocols on InfiniBand. When running on top of TCP/IP, performance improvement is mainly attributed to the NLM. We compare the performance of Hadoop-A when it is running on RDMA with that of running on IPoIB to quantify the improvement introduced by RDMA. The results are shown in Fig. 12.

As shown in the figure, on average, Hadoop-A on IPoIB efficiently reduces the job execution time by 18.9 percent when compared to Hadoop on IPoIB. This demonstrates that NLM is effective at improving the performance of Hadoop by reducing disk accesses on the ReduceTask side and forming a pipelined shuffle, merge, and reduce phases. Fig. 12 also shows that, by leveraging RDMA, Hadoop-A can further lower the job execution time. Compared to Hadoop-A on IPoIB, Hadoop-A on RDMA cuts down on the execution time by 19.9 percent on average.

In addition, Fig. 12 also shows that running Hadoop-A on 10-Gigabit Ethernet with TCP/IP achieves similar performance as Hadoop-A on IPoIB.

6 Discussion

In this section, we discuss our perspectives on the scalability of Hadoop-A for very large data and on its implications to network bandwidth and topology.

6.1 Memory Scalability

Although network-levitated merge is capable of efficiently supporting most Hadoop production jobs (jobs with $\leq$10 GB input make up $\geq$92 percent of total jobs [10]), there is a potential issue of coping with extremely large data sets. In the network-levitated merge algorithm, data are fetched from the remote MOF and stored a block before being merged to its staging buffer. So there must be at least a memory buffer (M_b) per segment to keep up the merging process. For a Hadoop application, the total number of segments (N) is determined by the size of application's data set (S) and the size of a data split (B), i.e., $N = \frac{S}{B}$. Thus, the amount of memory (M_t) per PQ is then given by $M_t = M_b \times \frac{S}{B}$. Conversely, the application data size is given by $S = \frac{M_t}{M_b} \times B$. Because there are multiple ReduceTasks and multiple PQs for one application, the available memory for a PQ is then limited. Assuming M_t equals 4 GB, M_b 8 KB, and B 256 MB, a simple back-of-the-envelope calculation can demonstrate that the maximum supported data size is 128 TB for the network-levitated merge algorithm. This is sufficient for many applications with tens of terabytes of data. However, a key issue here is that the memory requirement grows linearly (i.e., on the order of $O(N)$) with respect to the number of segments. For future 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. 13 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 additional

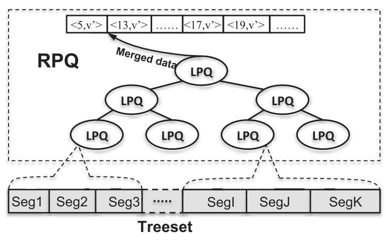


Fig. 13. Hierarchical merge for memory scalability.

staging buffer. The segments are spread into many small LPQs. For simplicity, assuming that all LPQs and the RPQ are of the same size, the number of segments in one LPQ grows in the order of $O(\sqrt{N})$. To keep the pipeline running with overlapped fetching and merging, only one LPQ needs to fetch data actively. Thus, the memory requirement becomes $M_t = \sqrt{\frac{S}{B}} \times M_b$. With such a hierarchical organization, the maximum supported data size for a single Hadoop job can be calculated as 64 exabytes, using the same numbers as before.

For even better scalability, a deeper hierarchy can be exploited for merging the memory segments. However, a hierarchical organization requires more memory copies. While it can improve memory scalability, a two-level tree hierarchy means that each <key, val> pair must be merged twice: one in LPQ and another in RPQ. A deeper hierarchy implies more memory copies. We deem it as a further research topic to investigate the strengths and weaknesses of the hierarchical merge algorithm and find a good tradeoff between the memory scalability and the cost of memory copies. A prototype of hierarchical merge algorithm has been designed and evaluated in [11].

6.2 Network Bandwidth and Topology

One may reasonably wonder that, since network-levitated merge delays data shuffling, the time window in Hadoop-A for shuffling could be possibly shorter than the original Hadoop, causing a higher requirement on the network bandwidth. This actually is not an issue because Hadoop-A overlaps data shuffling with the reducing phase instead of the mapping phase in the original Hadoop. While we change the actual phase that is overlapped with data shuffling, it does not alter the existence of concurrent disk activities. For shuffling to complete, each <key, val> pair is read from disk and copied in memory at least once. So, depending on the relative performance between network I/O and disk I/O, this may or may not cause a performance issue. When a Hadoop cluster can provide sufficient network bandwidth that is much higher than local disk bandwidth, Hadoop-A can efficiently exploit the performance of memory and network to eliminate the slow on-disk merge process, thus accelerating the execution of ReduceTask. On a cluster that is equipped with high-speed storage devices (such as multiple SCSI or Flash disks that are capable of more than 250 MB/sec) but low-bandwidth networks (such as Gigabit Ethernet), the network performance becomes the system bottleneck. In this case, both Hadoop-A and the original Hadoop will be affected. The only sound solution is to correct such imbalance of system resources and relieve the bottleneck on network bandwidth.

A more common situation on networked systems is the presence of network links that connect multiple switches together. Hadoop-A is not aimed to address the problem in a hierarchical network environment. The same challenge applies to the original Hadoop, particularly at the cross-switch links. A solution that can address the bottlenecks at the cross-switch links needs to be aware of the network topology, and design a shuffle/merge algorithm that can avoid overloading such cross-switch links. For example, CamDoop [12] is a recent attempt to perform topology-aware network aggregation of intermediate data for the CamCube network (a direct-connected 3D torus network constructed). It has shed light on intriguing issues that are yet to be investigated for hierarchical networks (such as Ethernet or InfiniBand) that consist of many switches connected by cross-switch links. We consider a topology-aware shuffle/merge algorithm as a very interesting topic to address in future studies and not in the scope of this work.

7 Related Work

MapReduce is a programming model for large-scale arbitrary data processing. To fully take advantage of the multicore and multiprocessor systems, Ranger et al. [13] designed Phoenix, a MapReduce implementation 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. Kaashoek et al. [14] then designed a new MapReduce library with a compromise data structure, which outperforms its simpler peers, including Phoenix. Tiled-MapReduce designed by Chen et al. [15] further improves the Phoenix by leveraging the tiling strategy that is commonly used in complier community. It divides a large MapReduce job into multiple discrete subjobs and extends the Reduce phase to process partial map output. By meticulously reusing memory and threads, Tiled-MapReduce achieves considerable speedup over Phoenix. But our work is completely different from these works in three aspects. First, we aims to improve the Hadoop MapReduce that is designed for large-scale clusters instead of for single machine with multicores. Second, our optimization strategy is to reduce the contention over disk I/O instead of cache and shared data structures. Third, we consider the impact of high-performance interconnects over Hadoop MapReduce. It is also worth mentioning that although Tiled-MapReduce introduces a strategy to pipeline multiple subjobs on each core, such interjob pipelining optimization is in stark contrast with our shuffle/merge/reduce interphase pipelining.

Several studies were published on tuning the performance of MapReduce. These include [3], [16], [17] that tuned different parameters of Hadoop MapReduce for performance improvements. Jiang et al. [3] 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. Herodotou and Babu [18] designed a cost-based optimizer with performance knobs to help choose better Hadoop configurations. Zaharia et al. [5] proposed a new scheduling algorithm, Longest Approximate Time to End (LATE), for environments with heterogeneous server configurations. Ananthanarayanan et al. [19] proposed Mantri to monitor tasks and cull outliers for better job completion time and later proposed Scarlett [20] to replicate data blocks to alleviate hotspots. Jahani et al. [21] applied compiler techniques for Hadoop optimizations. But none of these works investigated the I/O problem caused by MapReduce data shuffling. Our work takes on a different perspective to investigate new strategies for efficient data shuffling and merging in MapReduce, relieving its I/O contention. Li et al. [22] introduce MR-hash and Inc-hash to eliminate the repetitive merge problem, thus improving the I/O efficiency. However, their solution still requires heavy disk I/O within ReduceTask for large key space.

Leveraging RDMA for high-speed data movement has been very popular in various programming models and storage paradigms. Liu et al. [23] designed RDMA-based MPI over InfiniBand. Sur et al. [24] evaluated Hadoop Distributed File system over InfiniBand [6] through using socket direct protocol (SDP) and IPoIB protocols. The same group also leveraged RDMA to accelerate memcached [25]. Our acceleration framework uses both RDMA and TCP/IP protocols, and complements previous efforts to enable RDMA for Hadoop to process large data sets.

8 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 phases, as well as its lack of portability for multiple interconnects. We have designed and implemented Hadoop-A as an extensible acceleration framework that can allow plug-in 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. In addition, Hadoop-A has been designed as a portable framework that can run on both high-performance RDMA protocol and ubiquitous TCP/IP protocol. Our experimental results demonstrate that Hadoop-A doubles the data processing throughput of Hadoop, and that Hadoop-A is capable of effectively utilizing multiple threads to read data from multiple disks. Because of the use of network-levitated merge algorithm, it can significantly reduce disk accesses during Hadoop's shuffling and merging phases, thereby speeding up data movement. Furthermore, we have quantified the performance benefits of network-levitated merge and the RDMA protocol, respectively, on the Hadoop MapReduce.

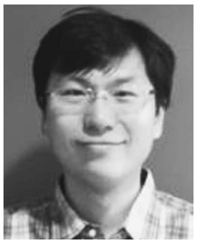
Acknowledgments

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

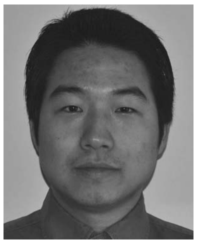
References

[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] T. Condie, N. Conway, P. Alvaro, J.M. Hellerstein, K. Elmeleegy, and R. Sears, "MapReduce Online," *Proc. Seventh USENIX Symp. Networked Systems Design and Implementation (NSDI)*, pp. 312-328, Apr. 2010.
[5] M. Zaharia, A. Konwinski, A.D. Joseph, R.H. Katz, and I. Stoica, "Improving MapReduce Performance in Heterogeneous Environments," *Proc. Eighth USENIX Symp. Operating Systems Design and Implementation (OSDI '08)*, Dec. 2008.
[6] Infiniband Trade Association, http://www.infinibandta.org. 2013.
[7] R. Recio, P. Culley, D. Garcia, and J. Hilland, "An RDMA Protocol Specification (Version 1.0)," Oct. 2002.
[8] Open Fabrics Alliance, http://www.openfabrics.org. 2013.
[9] IP over InfiniBand (IPoIB), http://www.ietf.org/wg/concluded/ipoib.html, 2013.
[10] Y. Chen, S. Alspaugh, and R.H. Katz, "Interactive Query Processing in Big Data Systems: A Cross Industry Study of MapReduce Workloads," Technical Report UCB/EECS-2012-37, EECS Dept., Univ. of California, Berkeley, Apr. 2012.
[11] X. Que, Y. Wang, C. Xu, and W. Yu, "Hierarchical Merge for Scalable MapReduce," *Proc. Workshop Management of Big Data Systems (MBDS '12)*, pp. 1-6, 2012.
[12] P. Costa, A. Donnelly, A. Rowstron, and G. O'Shea, "Camdoop: Exploiting in-Network Aggregation for Big Data Applications," *Proc. Ninth USENIX Conf. Networked Systems Design and Implementation (NSDI '12)*, p. 3, 2012.
[13] 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.
[14] 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.
[15] 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.
[16] S. Babu, "Towards Automatic Optimization of MapReduce Programs," *Proc. First ACM Symp. Cloud Computing (SoCC '10)*, pp. 137-142, 2010.
[17] B. Palanisamy, A. Singh, L. Liu, and B. Jain, "Purlieus: Locality-Aware Resource Allocation for MapReduce in a Cloud," *Proc. Conf. High Performance Computing Networking, Storage and Analysis*, pp. 58:1-58:11, Nov. 2011.
[18] H. Herodotou and S. Babu, "Profiling, What-If Analysis, and Cost-Based Optimization of MapReduce Programs," *Proc. 37th Int'l Conf. Very Large Data Bases*, 2011.
[19] G. Ananthanarayanan, S. Kandula, A.G. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris, "Reining in the Outliers in Map-Reduce Clusters Using Mantri," *Proc. Ninth USENIX Symp. Operating Systems Design and Implementation (OSDI '10)*, pp. 265-278, Oct. 2010.
[20] G. Ananthanarayanan, S. Agarwal, S. Kandula, A.G. Greenberg, I. Stoica, D. Harlan, and E. Harris, "Scarlett: Coping with Skewed Content Popularity in MapReduce Clusters," *Proc. Sixth European Conf. Computer Systems (EuroSys '11)*, Apr. 2011.
[21] E. Jahani, M.J. Cafarella, and C. Re, "Automatic Optimization for MapReduce Programs," *Proc. VLDB Endowment*, vol. 4, pp. 385-396, 2011.
[22] B. Li, E. Mazur, Y. Diao, A. McGregor, and P. Shenoy, "A Platform for Scalable One-Pass Analytics Using MapReduce," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '11)*, pp. 985-996, 2011.
[23] J. Liu, J. Wu, and D.K. Panda, "High Performance RDMA-Based MPI Implementation over InfiniBand," *Int'l J. Parallel Programming*, vol. 32, pp. 167-198, 2004.
[24] S. Sur, H. Wang, J. Huang, X. Ouyang, and D.K. Panda, "Can High-Performance Interconnects Benefit Hadoop Distributed File System?" *Proc. Workshop Micro Architectural Support for Virtualization, Data Center Computing, and Clouds (MASVDC). Held in Conjunction with MICRO*, Dec. 2010.
[25] J. Jose, H. Subramoni, M. Luo, M. Zhang, J. Huang, M. Wasi-ur Rahman, N.S. Islam, X. Ouyang, H. Wang, S. Sur, and D.K. Panda, "Memcached Design on High Performance RDMA Capable Interconnects," *Proc. Int'l Conf. Parallel Processing (ICPP '11)*, pp. 743-752, 2011.

Weikuan Yu received the bachelor degree in genetics from Wuhan University, China, and the master's degree in developmental biology from the Ohio State University. He received the PhD degree in computer science from the Ohio State University in 2006. He is currently an assistant professor in the Department of Computer Science and Software Engineering, Auburn University. Prior to joining Auburn, he served as a research scientist for two and a half years at Oak Ridge National Laboratory (ORNL) until January 2009. He is also a joint faculty at ORNL. At Auburn University, he leads the Parallel Architecture and System Laboratory for research and development on high-end computing, parallel and distributing networking, storage and file systems, as well as interdisciplinary topics on computational biology. He is a member of the AAAS, ACM, and the IEEE.



Yandong Wang received the master's degree in computer science from the Rochester Institute of Technology in 2010. He is currently working toward the PhD degree at the Parallel Architecture and System Laboratory, Department of Computer Science, Auburn University. His research interests include cloud computing, high-speed networking, and file and storage systems.



Xinyu Que received the master's degree in computer science from the University of Connecticut in 2009. He is currently working toward the PhD degree at the Parallel Architecture and System Laboratory, Department of Computer Science, Auburn University. His research interests include high-performance computing, high-speed networking, and network and grid computing.