JVM-Bypass for Efficient Hadoop Shuffling

Yandong Wang Cong Xu Xiaobing Li Weikuan Yu

Department of Computer Science, Auburn University, AL 36849, USA
{*wangyd,congxu,xbli,wkyu*}@*auburn.edu*

***Abstract*—Hadoop employs Java-based network transport stack on top of the Java Virtual Machine (JVM) for its data shuffling and merging purposes. Our examination reveals that JVM introduces a significant amount of overhead to data processing capability of the native interface. Furthermore, JVM constrains the use of high-performance networking mechanisms such as RDMA (Remote Direct Memory Access) which has established itself as an effective data movement technology in many networking environments because of its low-latency, high bandwidth, low CPU utilization, and energy efficiency. In this paper, we introduce a plug-in library called JVM-Bypass Shuffling (JBS) for Hadoop data shuffling. JBS helps Hadoop data shuffling by avoiding Java-based transport protocols, removing the overhead and limitations of the JVM. In addition, we design JBS as a portable library that can leverage both TCP/IP and RDMA on different network systems such as InfiniBand and 1/10 Gigabit Ethernet. We have designed and implemented JBS as part of Hadoop acceleration. It has been transferred to Mellanox as the software product UDA (Unstructured Data Accelerator) and used to enable our studies on a variety of merging algorithms. Our performance evaluation demonstrates that JBS can effectively reduce the execution time of Hadoop jobs by up to 66.3% and lower the CPU utilization by 48.1%.**

I. INTRODUCTION

Nowadays, a growing number of organizations center their business around the collection and analysis of enormous data sets. MapReduce is a popular programming model [7] that provides a simple and scalable parallel data processing framework for large-scale off-the-shelf clusters. Hadoop [1], as an open-source implementation of MapReduce, has been widely adopted by leading companies, such as Yahoo! and Facebook, for big data analytics. Two kinds of tasks, MapTasks and ReduceTasks, are employed in the MapReduce programming model for data processing. A data shuffling phase is required to move the intermediate data generated by the MapTasks to the ReduceTasks as their input. However, such data shuffling can cause a great volume of network traffic, imposing a serious constraint on the efficiency of data analytics applications. The performance issue of data shuffling has been identified by many previous works [7], [21], [5], [25]. However, few studies have been carried out to optimize the intermediate data transfer.

Hadoop currently relies on a stack of transport protocols in the Java Virtual Machine (JVM), including Java HTTP and network libraries. However, JVM introduces significant overhead in managing Java objects. For example, for every 8-byte *double* object, it requires another 16 bytes for data representation, an overhead of 67% [20]. Such inflated memory consumption quickly shrinks the available memory to Hadoop, and prolongs Java garbage collection for reclaiming memory. This JVM issue has also been documented by [4], [15].

Contemporary high speed networks, such as InfiniBand [13], provide Remote Direct Memory Access (RDMA) [24] that is capable of up to 56Gbps bandwidth, sub-microsecond latency and low CPU utilization. RDMA is also available through the RoCE (RDMA over Converged Ethernet) protocol [10] on 10 Gigabit Ethernet (10GigE). The performance advantage of RDMA advocates it as a compelling solution to speed up Hadoop intermediate data shuffling. Unfortunately, the existing Hadoop is designed to rely on the legacy TCP/IP protocol to transfer intermediate data and is incapable of utilizing RDMA. Such a limit prevents Hadoop from relishing high bandwidth and low latency from InfiniBand and 10GigE networks.

In order to address the JVM overhead issue and the need of portable network support, we have examined the software architecture of Hadoop for data shuffling. Accordingly we propose a solution, called **J**VM-**B**ypass **S**huffling (JBS), for Hadoop to avoid the overhead of JVM in data shuffling and enable fast data movement on both RDMA and TCP/IP protocols. JBS has actually been in use as part of our Hadoop Acceleration project. An initial version of it was released by Mellanox as the UDA (Unstructured Data Accelerator) software product [19]. It was also used to enable the network-levitated merge algorithm [29] and its followup hierarchical merge algorithm [22]. Compared to our earlier works on merge algorithms, this paper describes the design and implementation details of JVM-bypass and its portability on both TCP/IP and RDMA networks.

Overall, we have made four contributions in this research:

- We have examined the performance of Hadoop data shuffling and revealed that JVM can impose significant overhead on high-speed networks.
- We have designed and implemented JVM-Bypass Shuffling to avoid JVM in the critical path of Hadoop intermediate data shuffling.
- We have designed JBS as a portable plug-in library that can enable Hadoop to leverage both the traditional TCP/IP protocol and the advanced RDMA protocol.
- We have carried out a systematic performance evaluation of JBS on different networks. Our results demonstrate that JBS improves the execution time of Hadoop jobs by up to 66.3% and reduces the CPU utilization by 48.1%.

The remainder of the paper is organized as follows. We

introduce the motivation and describe the existing issues in Hadoop in Section II. We then present the design and implementation of JVM-Bypass Shuffling in Section III and Section IV. The experimental results are provided in Section V. Finally, we provide a review of related work in Section VI and then conclude the paper in Section VII.

II. Motivation

In this section, we present an overview of Hadoop intermediate data shuffling and then describe in detail the problems associated with its current design.

A. Hadoop Intermediate Data Shuffling

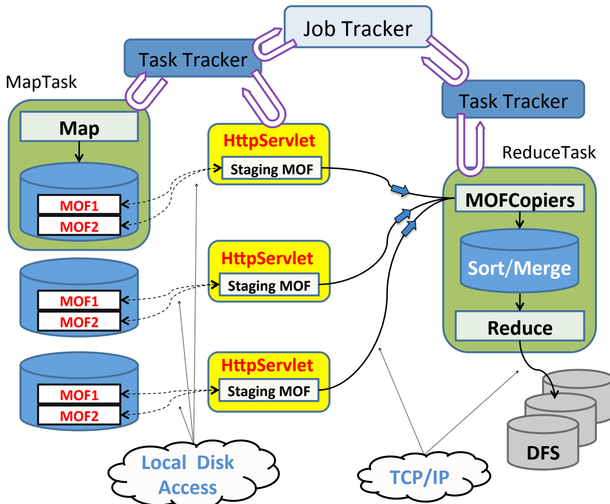


Fig. 1: Details of Intermediate Data Shuffling

Hadoop exposes two simple interfaces: *map* and *reduce*, to application users but hides processing complexities, such as data distribution, task parallelization, fault tolerance, etc. Its runtime system consists of four major components: JobTracker, TaskTracker, MapTask, and ReduceTask. These components are shown in Figure 1. JobTracker assigns one TaskTracker per slave node and orchestrates TaskTrackers to launch MapTasks and ReduceTasks for job execution. A MapTask reads an input split from Hadoop Distributed File System (HDFS) [27], runs the map function, and stores intermediate data as a **M**ap **O**utput **F**ile (MOF) to local disks. A MOF is divided into multiple *segments*, each of which is for a specific ReduceTask. Each ReduceTask fetches the segments from all the MOFs, sorts/merges them, and then reduces the merged results. The output generated by a ReduceTask is stored back to the HDFS as a part of the final result.

Hadoop has been highly optimized to reduce the amount of network traffic when reading input data for MapTasks and writing output from ReduceTasks. For instance, delay scheduling [31] helps improve the data locality and reduce data movement in the network. According to [31], up to 98% of MapTasks can be launched with inputs on local disks. In addition, ReduceTasks usually generate and store the final outputs to the disks local to themselves in the HDFS.

However, Hadoop intermediate data shuffling still causes a large volume of network traffic. Every ReduceTask fetches data segments from all map outputs, resulting in a network traffic pattern from all MapTasks to all ReduceTasks, which grows in the order of $O(N^2)$ assuming that MapTasks and ReduceTasks are both a factor of N total tasks. As reported by [23] from Yahoo!, the intermediate data shuffling from 5% of large jobs can consume more than 98% network bandwidth in a production cluster, and worse yet, Hadoop performance degrades non-linearly with the increase of intermediate data size. As pointed out by [6], network bandwidth oversubscription can quickly saturate the network links of those machines that participate in the reduce phase. This intermediate data shuffling essentially becomes the dominant source of network traffic and performance bottleneck in Hadoop.

B. Issues of Java Virtual Machine

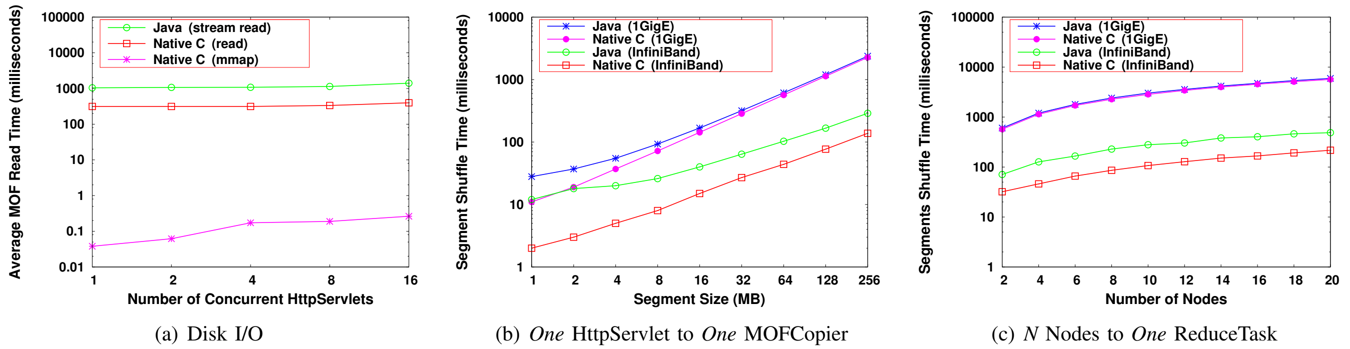
Figure 1 also shows the flow of data shuffling in Hadoop. An HttpServer is embedded inside each TaskTracker, and this server spawns multiple HttpServlets to answer incoming fetch requests for segment data. On the other side of the data shuffling, within each ReduceTask, multiple MOFCopiers are running concurrently to gather all segments.

Inside HttpServlets and MOFCopiers, Hadoop employs Java streams to simultaneously access and move data. However, such kind of Java I/O can perform 40%$\sim$60% worse than that written in native C, as pointed by [26], [8]. In order to shed a light on the overhead imposed by Java Virtual Machine(JVM) on Hadoop data shuffling, we have examined data movement between HttpServlets and MOFCopiers using both Java and native C languages. Figure 2(a) shows the performance of disk I/O using Java and native C. Currently Hadoop HttpServlets use traditional Java *FileInputStreams* to retrieve content from Map output. The results in Figure 2(a) show that, on average, using Java-based HttpServlets to read MOFs can be 3.1$\times$ worse than using native C language.

Figure 2(b) shows the time to shuffle data between one HttpServlet and one MOFCopier. On 1 Gigabit Ethernet (1GigE), the constraining effect of JVM is hidden because of the limited network bandwidth. On InfiniBand, Java-based data shuffling leads to a performance degradation of as much as 3.4$\times$, compared to native C. Furthermore, Figure 2(c) shows that on the InfiniBand cluster, when one ReduceTask (with multiple MOFCopiers) is fetching segments simultaneously from multiple nodes, JVM imposes above 2.5$\times$ overhead. Again, this overhead is hidden when the network bandwidth is bottlenecked on 1GigE.

III. JVM-Bypass Shuffling of Intermediate Data

In this section, we firstly describe the software architecture of JVM-Bypass Shuffling and then several salient features of its internal design.



(a) Disk I/O (b) *One* HttpServlet to *One* MOFCopier (c) *N* Nodes to *One* ReduceTask

Fig. 2: Simulation Results of Intermediate Data Shuffling

A. Architecture of JVM-Bypass Shuffling

To address the issues we have discussed in Section II, we design a shuffling scheme called JVM-Bypass Shuffling for moving Hadoop intermediate data. Figure 3 shows the changes in the software architecture from the original Hadoop to JBS. The main objective of JBS is to avoid JVM's heavy overhead caused by its deep stack of transport protocols without changing the user programming interfaces such as the user-defined *map* and *reduce* functions. Instead of going through a stack of Java HTTP and socket libraries (shown at the bottom left of Figure 3), JBS is designed to bypass the JVM from the critical path of the intermediate data shuffling.

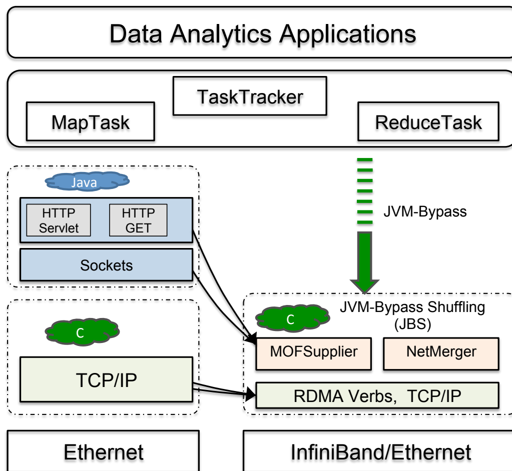


Fig. 3: Software Architecture of JBS

Additionally, to expand the network portability of Hadoop and compensate its lack of RDMA support, we have designed JBS as a portable layer on top of any network transport protocol. As shown in the figure, both RDMA and TCP/IP protocols are integrated as the underlying network mechanisms for data transfer in JBS. As a result, the JBS library is designed to avoid the overhead of JVM on data shuffling and accelerate Hadoop on two most popular commodity cluster networks including Ethernet (1Gigabit Ethernet and 10Gigabit Ethernet) and InfiniBand.

JBS as a Transparent Plugin Library – While the main objective of JBS is to bypass JVM, it is also important to design it as a transparent plugin library to Hadoop. In doing so, not only Hadoop programs can leverage JBS without any change, but also JBS can work with other Hadoop internal components as a plugin module. To this end, two components, namely MOFSupplier and NetMerger, are introduced to undertake the movement of intermediate data for Hadoop. These two components are standalone native C processes. They are launched by the local TaskTracker, with which they communicate via loopback sockets. These two components replace the HttpServlets in the TaskTracker and the MOFCopiers in the ReduceTasks, respectively, thereby bypassing the need of JVM when moving data between MOFSupplier and NetMerger. As a plugin module [2] for Hadoop MapReduce, JBS is invoked based on a runtime user parameter. When it is not loaded, it does not change the execution of the original Hadoop.

B. Pipelined Segment Prefetching

In the original Hadoop, for each fetch request, the HttpServlet finds the MOF and its corresponding Index file from disk devices. It then retrieves the location of the targeted segment in the MOF based on the Index file. Note that an *IndexCache* is usually maintained to cache the entries from the Index file and speed up the identification of MOF segments. From the MOF, a segment is read via disk I/O and then transmitted through network I/O. As shown in Figure 4, disk read and network transmit (*Xmit*) are completely serialized in the HttpServlet. No effort is made to correlate and batch multiple requests to improve the locality of disk accesses. As a result, each request can experience a long delay in the queue, degrading the performance of data shuffling.

To improve the efficiency of serving intermediate data, we design the MOFSupplier with a pipelined prefetching scheme. Besides providing an IndexCache for quick identification of MOF segments, we also design a *DataCache* to prefetch MOF segments for request processing. As shown at the bottom of the Figure 5, with dedicated memory space as the DataCache, requests are grouped based on their targeted MOF, and those in the same group are ordered based on their intended segments. Segments for several requests are prefetched to the DataCache.

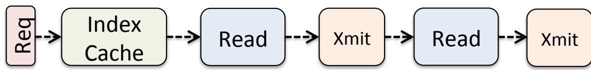


Fig. 4: Serialized Request Processing in HttpServlet

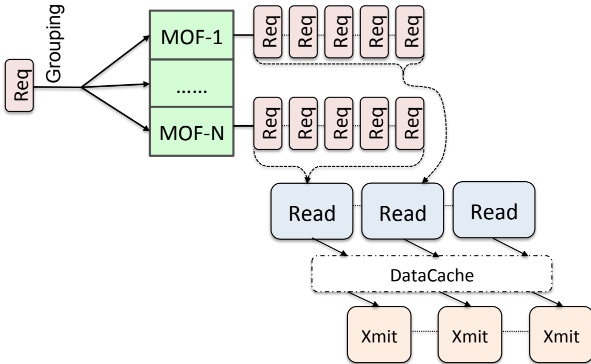


Fig. 5: Pipelined Segment Prefetching in MOFSupplier

All groups are served by the disk prefetch server in a round-robin manner. When a batch of segments are ready in the DataCache, they are transmitted over by asynchronous network operations. With the DataCache as the buffer, fetch requests are served in a batched and pipelined manner, thereby increasing the locality of disk accesses and reducing the average delay of requests.

C. *Consolidated and Balanced Data Fetching*

As mentioned earlier, a Hadoop ReduceTask employs multiple MOFCopier threads to concurrently fetch independent segment data to local file system. Several merging threads are running in the background to merge available segments. When faced with large data sets, both MOFCopier and merging threads spill data to local disks. As part of our Hadoop Acceleration project, the NetMerger for JBS is designed to undertake the shuffling (a.k.a fetching) and merging of intermediate data. The details of the network-levitated merging algorithm used by NetMerger to conduct the segments merging has been described in our previous paper [29]. Here we provide more details on the arrangement of network requests to fetch data from remote MOFs.

We design the NetMerger as a component that can consolidate network fetching requests from all ReduceTasks on a single node. As mentioned in Section III-A, only one NetMerger is created by TaskTracker on the single node. Thus all segments needed by multiple ReduceTasks on the same node will be served by the NetMerger. Within this shared NetMerger, we consolidate and group all requests based on their targeted remote nodes. Requests to the same node are ordered based on their time of arrival. Using this organization, we are able to consolidate a number of network connections, which is no longer the total amount of MOFCopiers from all ReduceTasks. This consolidation also reduces the resource requirements from creating and sustaining many network channels and their associated memory to buffer data. In addition, across different groups, we adopt a simple round-robin mechanism to balance the injection of fetching requests to different nodes, mitigating the impact of burst requests from an aggressive ReduceTask.

IV. Implementation

We have implemented JBS as a portable library for different network environments including the traditional TCP/IP protocol, the RDMA protocol and the latest RoCE (RDMA over Converged Ethernet) protocol. The implementation of JBS is same for both RDMA and RoCE, except that their activation is different. In this paper, we refer to RDMA as the protocol activated on InfiniBand and RoCE as the protocol activated on 10Gigabit Ethernet.

The TCP/IP protocol and the RDMA-like (RDMA and RoCE) protocols are very different in their ways of establishing network connections. Accordingly, we provide some implementation details for their respective connection establishment.

A. *Connection Establishment for RDMA and RoCE*

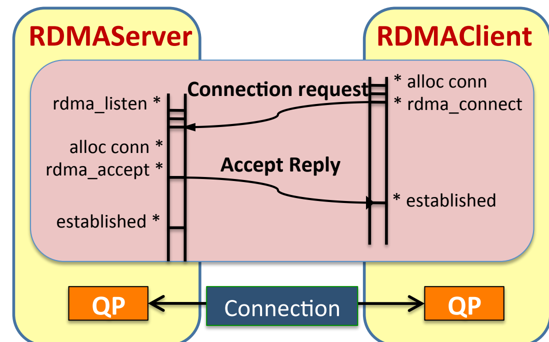


Fig. 6: Connection Establishment for RDMA and RoCE

Figure 6 illustrates our implementation of connection establishment on RDMA and RoCE. A pair of RDMAServer and RDMAClient are designed to handle RDMA connection establishment. An additional thread managing network events is created for both RDMAServer and RDMAClient. The first fetching request triggers a RDMAClient to initiate the process of connection establishment with the remote RDMAServer. In the case of a RDMAClient, it allocates a new connection (a.k.a Queue Pair) and sends a connection request via *rdma_connect()* to the RDMAServer. The network thread listening for incoming requests on the RDMAServer receives this connection request and handles a series of events that are detected on the associated RDMA event channel. This RDMAServer then allocates a new RDMA connection. Via *rdma_accept()*, it accepts and confirms the connection request to the RDMAClient. The successful completion of the *accept()* call will be detected via an *established* event by network threads at both RDMAServer and RDMAClient. This completes the establishment of a queue pair (QP), i.e., a new RDMA connection.

Currently we use only the Reliable Connection (RC) service provided by RDMA-capable interconnects. Since the cost of

setting up RDMA connection is relatively high, we keep newly created connections for reuse by default. We allow a maximum of 512 active connections. When this threshold is reached, connections are torn down based on the LRU (Least Recently Used) order.

B. TCP/IP-Based Communication

To support the TCP/IP protocol, JBS employs conventional TCP/IP sockets to establish connections. It makes use of an event-driven model and multiple threads to achieve good parallelism and communication throughput. On the client side, one thread is dedicated to prepare connection requests to different nodes and monitor their status. The actual connection requests are made by the client's data threads to the remote servers. On the server side, one thread is listening for client connection requests. It accepts a client's connection request after validating its legitimacy. Both client and server use the *epoll* interface to monitor and detect events from concurrent connections, and rely on their data threads to perform the network communication for data transfer.

Similar to the case of RDMA, a client's first TCP/IP fetch request triggers the creation of a TCP connection to a remote node. A connection is torn down when the total of connections exceed the threshold of 512.

V. Performance Evaluation

All experiments are conducted on two clusters. Each cluster features 23 compute nodes. All compute nodes in both clusters are identical, each with four 2.67GHz hex-core Intel Xeon X5650 CPUs, two Western Digital SATA 500GB hard drives and 24GB memory. In the Ethernet environment, all compute nodes are interconnected by 1/10 Gigabit Ethernet. In the InfiniBand environment, all nodes are equipped with Mellanox ConnectX-2 QDR HCAs and connected to a 108-port InfiniBand QDR switch.

JBS can be adapted to various Hadoop versions. In our evaluation, we use the stable version Hadoop 0.20.3. During the experiments, one node is dedicated as both the NameNode of HDFS and the JobTracker of Hadoop MapReduce. On each of the 22 slave nodes, we run 4 MapTasks and 2 ReduceTasks. The HDFS block size is chosen as 256MB to balance the parallelism and performance of MapTasks.

We run a group of benchmarks which include Terasort, WordCount, Grep from standard Hadoop package and Self-Join, AdjacencyList, InvertedIndex, SequenceCount benchmarks from Tarazu benchmark suite [3]. Tarazu benchmarks represent typical jobs in production clusters. Since JBS specifically aims to improve I/O during the intermediate data shuffling, among different benchmarks, we focus on the data-intensive Terasort, whose size of intermediate data is equal to its input size.

Many names are used in this section to describe the test cases, for example, *Hadoop on SDP* means we run original Hadoop on InfiniBand through Socket Direct Protocol. To avoid confusion, we list the protocol and network environment used for each test case in Table I.

TABLE I: Test Case Description

Test Cases	Transport Protocol	Network
Hadoop on 1GigE	TCP/IP	1GigE
Hadoop on 10GigE	TCP/IP	10GigE
Hadoop on IPoIB	IPoIB	InfiniBand
Hadoop on SDP	SDP	InfiniBand
JBS on 10GigE	TCP/IP	10GigE
JBS on IPoIB	IPoIB	InfiniBand
JBS on RoCE	RoCE	10GigE
JBS on RDMA	RDMA	InfiniBand

A. Benefits of JVM-Bypass

We start our experiments by examining the efficiency of JBS for intermediate data of different sizes. We run Tersort jobs of different input sizes on both InfiniBand and Ethernet clusters. For each data size, we conduct 3 experiments and report the average job execution time.

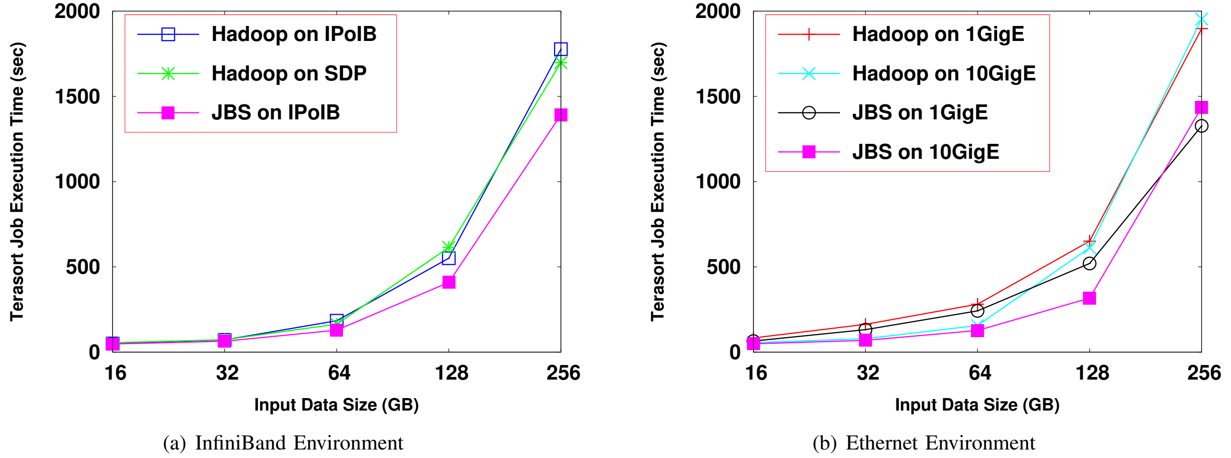
Figure 7 shows the experimental results. In the InfiniBand environment, compared to Hadoop on IPoIB and Hadoop on SDP, JBS on IPoIB reduces job execution time by 14.1% and 14.8%, respectively, on average. During these experiments, we notice that the performance of Hadoop on IPoIB is very close to that of Hadoop on SDP. Therefore, in the following sections, we use mostly Hadoop on IPoIB as the reference case to show the benefits of JBS in the InfiniBand environment.

In the Ethernet environment, JBS on 1GigE and JBS on 10GigE reduce the job execution times by 20.9% and 19.3% on average, compared to Hadoop on 1GigE and Hadoop on 10GigE, respectively.

In both environments, we observe that Hadoop on high-speed networks can speed up jobs with small sizes of intermediate data ($\leq$ 64GB). For instance, when the data size is 32GB, compared to Hadoop on 1GigE, Hadoop on IPoIB and Hadoop on 10GigE achieve improvements of 55.2% and 51.5%, respectively, on average. This is because the movement of small size data is less dependent on disks and most of them reside in disk cache or system buffers. Thus high-performance networks can exhibit better benefits for data shuffling.

Nevertheless, even with high-speed networks, Hadoop is still constrained from exploiting the full performance potentials of network hardware by the presence of JVM. In contrast, JBS eliminates the overhead of JVM, and improves the jobs with small size intermediate data. For 32GB input, JBS reduces the execution time by 11.1% and 12.7% on average, compared to Hadoop on IPoIB and Hadoop on 10GigE, respectively. For jobs with even smaller data sets, the costs of task initialization and destruction become dominant, JBS does not exhibit benefits.

Furthermore, as shown in Figure 7, the cases using high-performance networks (Hadoop on IPoIB and Hadoop on 10GigE) without JBS, compared to the Hadoop on 1GigE. do not provide noticeable improvements for large data sets ($\geq$ 128GB) due to disk I/O bottleneck caused by large data sets. JBS alleviates these issues through its batched and pipelined prefetching. Therefore, with a data size of 256GB, JBS on IPoIB and JBS on 10GigE improve the performance by 21.7%



(a) InfiniBand Environment

(b) Ethernet Environment

Fig. 7: Benefits of JVM-Bypass

and 26.5%, respectively on average, compared to Hadoop on IPoIB and Hadoop and 10GigE.

Another interesting observation as shown in Figure 7(b) is that when data size grows close to 256GB, JBS performs similarly on 1GigE and 10GigE. This is because that the overhead incurred by large amount of memory copies for TCP/IP transportation becomes a severe bottleneck. Therefore, in the next section, we explore the benefit provided by RDMA protocol.

B. Benefits of RDMA

To evaluate the benefit of RDMA, we compare the performance of JBS on RDMA and JBS on RoCE to that of JBS on IPoIB and JBS on 10GigE. Figure 8 shows the experiment results. In the InfiniBand environment, JBS on RDMA outperforms JBS on IPoIB by 25.8% on average. In the Ethernet environment, compared to JBS on 10GigE, JBS on RoCE speeds up the job executions by 15.3% on average. Note that in both environments, running JBS on RDMA and RoCE achieve better performance for all data sizes.

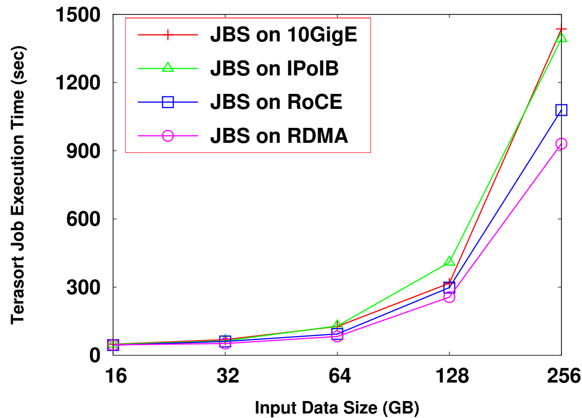


Fig. 8: Benefits of RDMA

The reason that JBS delivers better performance with RDMA protocol is two-fold. Firstly, RDMA has significant performance advantages over TCP/IP because of its higher bandwidth and lower latency. Secondly, RDMA also reduces the number of memory copies for the movement of intermediate data.

C. Scalability

High scalability is a critical feature that leads Hadoop to its success. JBS aims to preserve this feature. In this section, we evaluate the scalability of JBS by two scaling patterns: *Strong Scaling* and *Weak Scaling*. In the case of strong scaling, we use a fixed-size data (256GB) as Terasort input while increasing the number of compute nodes. In the case of weak scaling, we use a fixed-size data (6GB) for each ReduceTask of a Terasort job, so the total input size increases linearly when we increase the number of nodes, reaching 264GB when 22 slave nodes are used.

Figure 9(a) shows the results of the strong scaling experiments on the InfiniBand cluster. On average, JBS on RDMA and JBS on IPoIB outperform Hadoop on IPoIB by 49.5% and 20.9%, respectively. It achieves a linear reduction of the execution time with an increasing number of slave nodes. Figure 9(b) shows the results of the weak scaling experiments. JBS on RDMA and JBS on IPoIB reduce the execution time by 43.6% and 21.1%, respectively on average, compared to Hadoop on IPoIB. As also shown in the Figure, JBS maintains stable improvement ratios with a varying number of slave nodes.

Figures 9(c) and (d) show the results of scaling experiments in the Ethernet environment. JBS accomplishes similar performance improvement as in the InfiniBand environment. Compared to Hadoop on 10GigE, JBS on RoCE reduces the execution time by up to 41.9% for strong scaling tests and up to 40.4% for weak scaling tests on average. Compared to Hadoop on 10GigE, JBS on 10GigE reduces the execution time by 17.6% and 23.8%, respectively on average, for strong

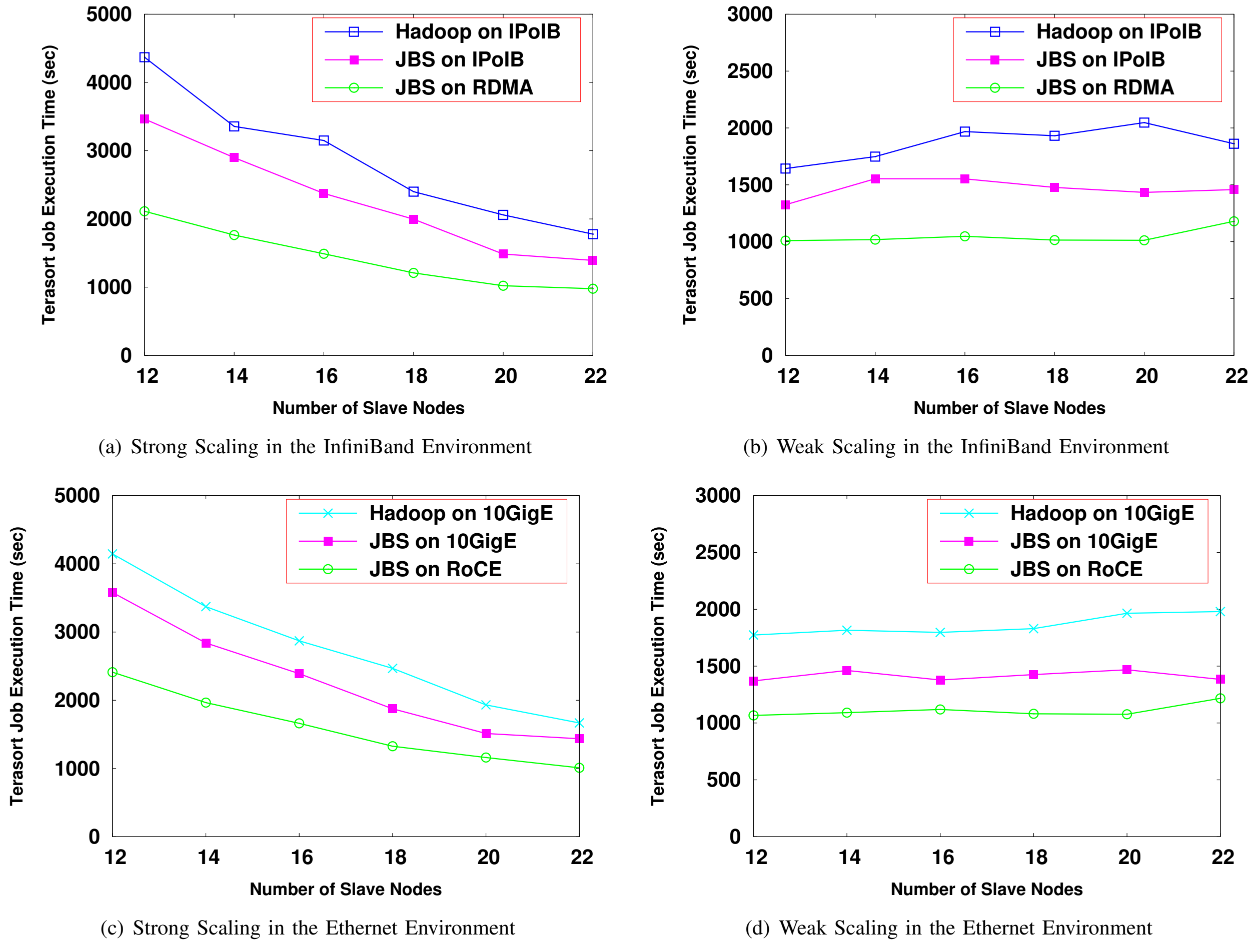


Fig. 9: Scalability Evaluation

and weak scaling tests. Taken together, our results demonstrates that JBS is capable of providing better scalability than Hadoop in terms of the job execution time regardless of the underlying networks and protocols.

D. CPU Utilization

CPU utilization is another important performance metric. In this section, we measure the CPU utilization of JBS by using Terasort benchmark with 128GB input data. Low CPU utilization during data shuffling can spare more CPU cycles for Hadoop applications. On each node, we run *sar* in the background to collect CPU statistics and trace the output every 5 seconds. In the results, we report the average CPU utilization across all 22 slave nodes.

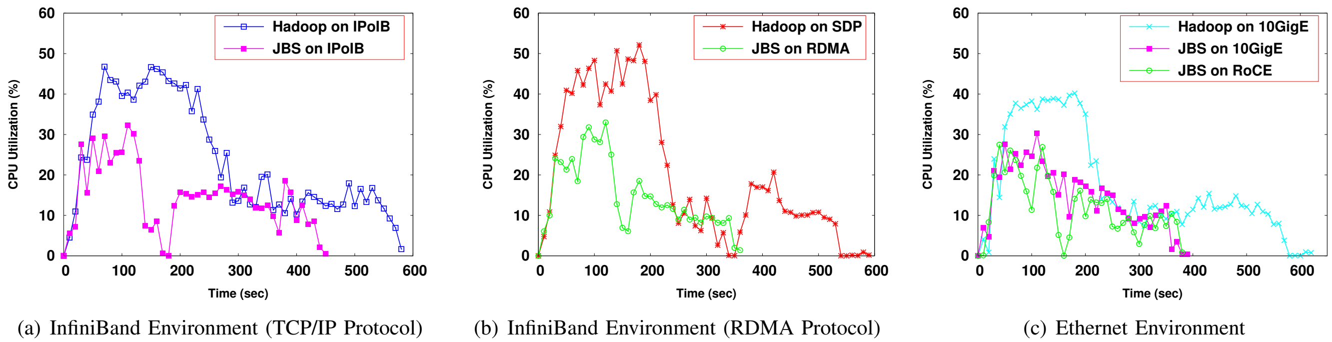
Figures 10(a) and (b) show the results of comparing CPU utilization between Hadoop and JBS in the InfiniBand environment. For fair comparison, we only consider CPU utilization in the same execution period. By eliminating the overhead of JVM and reducing the disk I/O, JBS on IPoIB greatly lowers the CPU utilization by 48.1% compared to Hadoop on IPoIB. In addition, even though the SDP protocol provides Java stream sockets to take advantage of RDMA through the socket interface, Hadoop on SDP can only reduce CPU utilization by 15.8%, compared to Hadoop on IPoIB. In contrast, JBS on RDMA significantly reduces the CPU utilization by 44.8%, compared to Hadoop on SDP. Note that, besides the RDMA benefit of low CPU utilization, another major factor that contributes to low utilization is the use of less number of threads in JBS. Unlike the original Hadoop in which each ReduceTask spawns more than 8 JVM threads for the purpose of data shuffling, JBS only requires 3 native C threads for the same.

Figure 10(c) shows the CPU utilization of JBS in the Ethernet environment. Compared to Hadoop on 10GigE, JBS on RoCE and JBS on 10GigE reduce the CPU utilization by 46.4% and 33.9%, respectively on average. In addition, JBS on RoCE reduces CPU utilization by about 18.7% due to RoCE's low CPU utilization and less memory copies.

Taken together, these results adequately demonstrate that JBS not only reduces the job execution time, but also achieves much lower CPU utilization.

E. Impact of JBS Transport Buffer Size

The size of the buffer for network transportation has critical impact on the performance. Large buffer size can better utilize



(a) InfiniBand Environment (TCP/IP Protocol)

(b) InfiniBand Environment (RDMA Protocol)

(c) Ethernet Environment

Fig. 10: CPU Utilization

the bandwidth and reduce overheads due to less number of fetch requests for each segment, but it also results in less number of available buffers to be shared by data threads, causing more resource contention. To understand the impact of the buffer size, we measure the execution time of Terasort with 128GB input while changing the buffer size.

Figure 11 shows the results. For JBS on RDMA and JBS on RoCE, the execution time goes down with an increasing transport buffer size, and gradually levels off from 128KB and beyond. Compared to the 8KB buffer size, the 256KB buffer size improves the performance of JBS on RDMA by 53%. This performance difference is more significant for JBS on IPoIB. When the buffer size increases from 8KB to 128KB, execution time is reduced by up to 70.3%. However, when the size reaches 512KB, the performance is slightly degraded. This is because the use of very large buffers increases the contention between communication threads, and reduces the pipelining effects of many buffers.

Overall this evaluation demonstrates that a large buffer size up to 128KB can effectively improve job execution time. For this reason, we choose the default transport buffer size as 128KB for the JBS library.

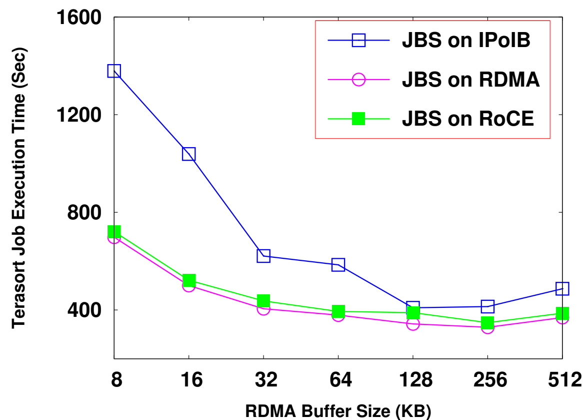


Fig. 11: Impact of Different RDMA Buffer Sizes

F. Effectiveness on Different Benchmarks

To assess the optimization effectiveness of JBS to other Hadoop applications, we have evaluated JBS with Tarazu benchmark suite in addition to Terasort. The input for those benchmarks are 30GB, using either wikipedia data or database data. Figures 12 (a) and (b) present the job execution times in both InfiniBand and Ethernet environments. Overall, these benchmarks can be categorized into two types.

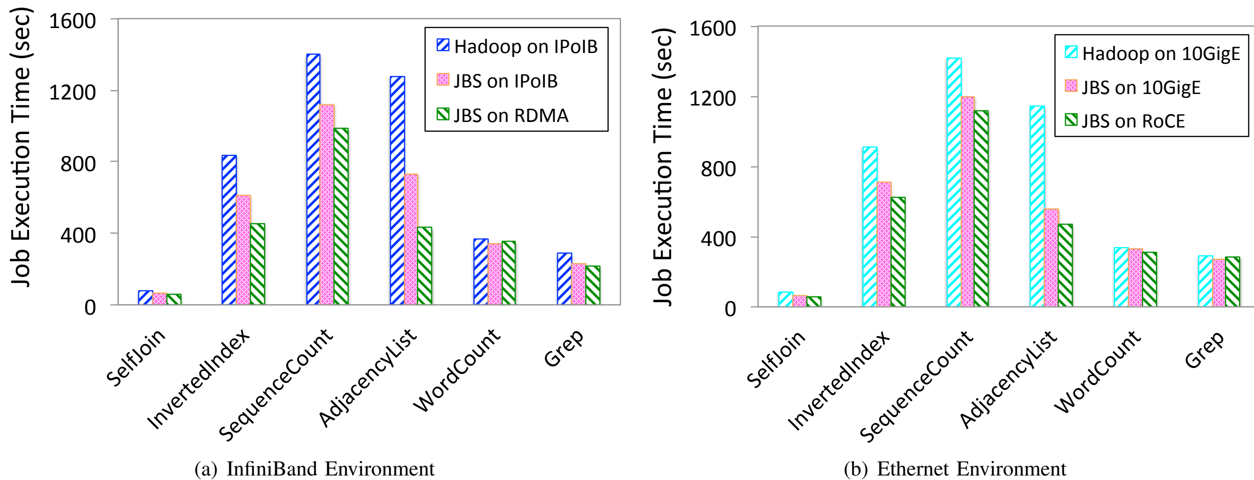
For the first type of benchmarks including SelfJoin, InvertedIndex, SequenceCount, and AdjacencyList, each MapTask generates a lot of intermediate data to be shuffled to ReduceTasks. Because of its strength in accelerating the shuffling phase of Hadoop, JBS is geared to provide good performance benefits for these applications.

In the InfiniBand environment, we observe that JBS on RDMA achieves an average of 41% reduction in the execution time for these four benchmarks, and reaches up to 66.3% improvement for AdjacencyList. JBS on IPoIB reduces the execution times of these benchmarks by 26.9% on average. In the Ethernet environment, compared to Hadoop on 10GigE, JBS on RoCE reduces the execution times by 36.1% on average for these benchmarks. JBS on 10GigE only reduces the execution times by 29.8% on averaged compared to the same.

In contrast, for the second type of benchmarks, WordCount and Grep, only a small amount of intermediate data is generated. As a result, JBS does not gain performance improvement for these two benchmarks.

VI. Related Work

Leveraging high performance interconnects to move data in the Hadoop ecosystem has attracted numerous research interests from many organizations. Huang *et al.* [11] designed an RDMA-based HBase over InfiniBand. In addition, they also mentioned the disadvantages of using Java Socket Interfaces. Jose *et al.*[17], [16] implemented a scalable memcache through taking advantage of performance benefits provided by high-speed interconnects. Sur *et al.* [28] studied the potential benefit of running HDFS over InfiniBand. Furthermore, Islam *et al.* [14] enhances the HDFS using RDMA over InfiniBand via JNI interfaces. However, although Hadoop MapReduce



(a) InfiniBand Environment

(b) Ethernet Environment

Fig. 12: Performance of Different Benchmarks

is a fundamental basis of Hadoop ecosystem, there is lack of research on how to efficiently leverage high performance interconnects in Hadoop MapReduce.

Solving the intermediate data shuffling bottleneck is another interesting research topic about Hadoop. Camdoop [6] is designed to decrease the network traffic caused by intermediate data shuffling through applying a hierarchical aggregation during the data forwarding. However, Camdoop is only effective in special network topology, such as 3D torus network, and its performance degrades sharply in common network topologies adopted by data centers. MapReduce online [5] attempts to directly send the intermediate data from MapTasks to ReduceTasks to avoid touching disks on the MapTasks sides. In order to do so, it requires large number of sustained TCP/IP connections between MapTasks and ReduceTasks. However, it severely restricts the scalability of Hadoop MapReduce. In addition, when the data size is large and network cannot keep up with the MapTask processing speed, intermediate data still needs to be spilled to disks. Furthermore, it fails to identify the I/O bottleneck problem in HttpServlet and MOFCopier. So for the above reasons, MapReduce online has to fall back onto the original Hadoop execution mode. Different from MapReduce online, JBS completely re-designs both server and client sides and eliminates the JVM overhead associated with the data shuffling. Seo *et al.* [25] improved the performance of MapReduce by reducing redundant I/O in the software architecture. But it did not study the I/O issues caused by the data shuffling between MapTasks and ReduceTasks. [12] replaces HDFS with high-performance Lustre file system, and stores intermediate data in Lustre. However there is no efficient performance improvement reported.

Leveraging RDMA from high speed networks for high-performance data movement has been very popular in various programming models and storage paradigms. [9] studied the pros and cons of using RDMA capabilities. Liu *et al.* [18] designed RDMA-based MPI over InfiniBand. Yu *et al.* [30] implemented a scalable connection management strategy for high-performance interconnects. Our work is based on these mature studies of RDMA technology.

VII. Conclusions

In this paper, we have comprehensively analyzed the performance of Hadoop running on InfiniBand and 1/10 Gigabit Ethernet. Our experiment results reveal that simply switching to the high-performance interconnects cannot effectively boost the performance of Hadoop. To investigate the cause, We identify the overhead imposed by JVM on Hadoop intermediate data shuffling. We have designed and implemented JVM-Bypass Shuffling (JBS) to avoid JVM in the critical path of Hadoop data shuffling. Our implementation of JBS also enables it as a portable library that can leverage both conventional TCP/IP protocol and high-performance RDMA protocol in different network environments. Our experimental evaluation demonstrates that JBS can effectively reduce the execution time of Hadoop jobs by up to 66.3% and lower the CPU utilization by 48.1%. Furthermore, with a set of different application benchmarks, we demonstrate that JBS can significantly reduce the CPU utilization and job execution time for Hadoop jobs that generate a large amount of intermediate data.

Acknowledgments

This work is funded in part by a National Science Foundation award CNS-1059376. We are very thankful for an InfiniBand equipment donation from Mellanox Technologies Inc. to Auburn University.

References

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

[2] Plugin for Generic Shuffle Service. https://issues.apache.org/jira/browse/MAPREDUCE-4049.

[3] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. N. Vijaykumar. Tarazu: optimizing mapreduce on heterogeneous clusters. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'12, pages 61–74, New York, NY, USA, 2012. ACM.

[4] Baidu, Inc. Hadoop C++ Enhancement. http://issues.apache.org/jira/browse/MAPREDUCE-1270.

[5] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, NSDI'10, pages 21–21, Berkeley, CA, USA, 2010. USENIX Association.

[6] P. Costa, A. Donnelly, A. Rowstron, and G. O'Shea. Camdoop: exploiting in-network aggregation for big data applications. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, NSDI'12, pages 3–3, Berkeley, CA, USA, 2012. USENIX Association.

[7] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[8] M. Ding, L. Zheng, Y. Lu, L. Li, S. Guo, and M. Guo. More convenient more overhead: the performance evaluation of hadoop streaming. In *Proceedings of the 2011 ACM Symposium on Research in Applied Computation*, RACS '11, pages 307–313, New York, NY, USA, 2011. ACM.

[9] P. W. Frey and G. Alonso. Minimizing the hidden cost of rdma. In *Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems*, ICDCS'09, pages 553–560, Washington, DC, USA, 2009. IEEE Computer Society.

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

[11] J. Huang, X. Ouyang, J. Jose, M. W. ur Rahman, H. Wang, M. Luo, H. Subramoni, C. Murthy, and D. K. Panda. High-performance design of hbase with rdma over infiniband. In *26th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2012, Shanghai, China, May 21-25, 2012*, IPDPS'12, pages 774–785, 2012.

[12] S. M. Inc. Using Lustre with Apache Hadoop. http://wiki.lustre.org.

[13] InfiniBand Trade Association. The InfiniBand Architecture. http://www.infinibandta.org.

[14] N. S. Islam, M. W. Rahman, J. Jose, R. Rajachandrasekar, H. Wang, H. Subramoni, C. Murthy, and D. K. Panda. High performance rdma-based design of hdfs over infiniband. In *Proceedings of 2012 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC'12. ACM, 2012.

[15] R. J.Chanslet. Data availability and durability with the hadoop distributed file system. ;login' 12. USENIX Association, 2012.

[16] J. Jose, H. Subramoni, K. Kandalla, M. Wasi-ur Rahman, H. Wang, S. Narravula, and D. K. Panda. Scalable memcached design for infiniband clusters using hybrid transports. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2012)*, CCGRID'12, pages 236–243, Washington, DC, USA, 2012. IEEE Computer Society.

[17] J. Jose, H. Subramoni, M. Luo, M. Zhang, J. Huang, M. W. ur Rahman, N. S. Islam, X. Ouyang, H. Wang, S. Sur, and D. K. Panda. Memcached design on high performance rdma capable interconnects. In *ICPP*, pages 743–752. IEEE, 2011.

[18] J. Liu, J. Wu, and D. K. Panda. High performance rdma-based mpi implementation over infiniband. *International Journal of Parallel Programming*, 32:167–198, 2004.

[19] Mellanox. Mellanox Announces Availability of UDA 2.0 for Big Data Analytic Acceleration. http://ir.mellanox.com/releasedetail.cfm?ReleaseID=621771, November 2011.

[20] M. Nick and S. Gary. Building memory-efficient java application:practices and challenges. PLDI '09. ACM, 2009.

[21] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the 35th SIGMOD international conference on Management of data*, SIGMOD '09, pages 165–178, New York, NY, USA, 2009. ACM.

[22] X. Que, Y. Wang, C. Xu, and W. Yu. Hierarchical merge for efficient mapreduce. In *Proceedings of 2012 International Workshop on Management of Big Data Systems (MBDS)*, Held In Conjunction with ICAC'12, San Jose, CA, 2012.

[23] S. Rao. I-files: Handling Intermediate Data In Parallel Dataflow Graphs (Sailfish).

[24] R. Recio, P. Culley, D. Garcia, and J. Hilland. An rdma protocol specification (version 1.0), October 2002.

[25] S. Seo, I. Jang, K. Woo, I. Kim, J.-S. Kim, and S. Maeng. HPMR: Prefetching and pre-shuffling in shared MapReduce computation environment. In *CLUSTER*, pages 1–8, August 2009.

[26] J. Shafer, S. Rixner, and A. L. Cox. The hadoop distributed filesystem: Balancing portability and performance. In *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2010, www.ispass.org, 28-30 March 2010, White Plains, NY, USA*, pages 122–133. IEEE Computer Society, 2010.

[27] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[28] S. Sur, H. Wang, J. Huang, X. Ouyang, and D. K. Panda. Can High-Performance Interconnects Benefit Hadoop Distributed File System? In *MASVDC-2010 Workshop in conjunction with MICRO*, Dec 2010.

[29] Y. Wang, X. Que, W. Yu, D. Goldenberg, and D. Sehgal. Hadoop acceleration through network levitated merge. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 57:1–57:10, New York, NY, USA, 2011. ACM.

[30] W. Yu, Q. Gao, and D. K. Panda. Adaptive connection management for scalable mpi over infiniband. In *Proceedings of the 20th international conference on Parallel and distributed processing*, IPDPS'06, pages 102–102, Washington, DC, USA, 2006. IEEE Computer Society.

[31] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems*, EuroSys'10, pages 265–278, New York, NY, USA, 2010. ACM.