2014 IEEE 22nd Annual Symposium on High-Performance Interconnects

Accelerating Spark with RDMA for Big Data Processing: Early Experiences*

Xiaoyi Lu, Md. Wasi-ur-Rahman, Nusrat Islam, Dipti Shankar and Dhabaleswar K. (DK) Panda
Department of Computer Science and Engineering, The Ohio State University
{luxi, rahmanmd, islamn, shankard, panda}@cse.ohio-state.edu

***Abstract*—Apache Hadoop MapReduce has been highly successful in processing large-scale, data-intensive batch applications on commodity clusters. However, for low-latency interactive applications and iterative computations, Apache Spark, an emerging in-memory processing framework, has been stealing the limelight. Recent studies have shown that current generation Big Data frameworks (like Hadoop) cannot efficiently leverage advanced features (e.g. RDMA) on modern clusters with high-performance networks. One of the major bottlenecks is that these middleware are traditionally written with sockets and do not deliver the best performance on modern HPC systems with RDMA-enabled high-performance interconnects. In this paper, we first assess the opportunities of bringing the benefits of RDMA into the Spark framework. We further propose a high-performance RDMA-based design for accelerating data shuffle in the Spark framework on high-performance networks. Performance evaluations show that our proposed design can achieve 79-83% performance improvement for GroupBy, compared with the default Spark running with IP over InfiniBand (IPoIB) FDR on a 128-256 core cluster. We adopt a plug-in-based approach that can make our design to be easily integrated with newer Spark releases. To the best our knowledge, this is the first design for accelerating Spark with RDMA for Big Data processing.**

***Keywords*—Apache Spark; RDMA; InfiniBand**

I. Introduction

As the explosive growth of Big Data continues, there is an increasing demand for Big Data analytics stacks to deliver high-performance on modern HPC clusters. According to the most recent IDC figures [5], 67% of HPC centers claim to perform what can be categorized as Big Data analysis. Hadoop MapReduce has revolutionized Big Data processing by enabling users to store and process huge amounts of data at very low costs. It is easily parallelizable, scales out and is highly optimized for analytical workloads. While MapReduce [3] has been highly successful in implementing large-scale and data-intensive batch applications on commodity clusters, it is a poor fit for low-latency interactive applications and iterative computations, such as machine learning and graph algorithms. As a result, newer data-processing frameworks such as Apache Spark [19, 20] have been stealing the limelight.

Spark is an in-memory, data-processing framework that is compatible with Hadoop data sources. It is particularly well suited for iterative machine learning jobs, as well as interactive data analytics. Along with providing high-level APIs in Scala, Java, and Python, which make parallel jobs easy to write, Spark retains the scalability and fault tolerance of MapReduce. Therefore, Spark, with its fast and generalized cluster computing platform, shows great promise for the next-generation Big Data applications.

On the other hand, a majority of the existing clusters today are equipped with modern high-speed interconnects, such as InfiniBand and 10 Gigabit Ethernet/RoCE/iWARP, that offer high-bandwidth and low-latency data transmission. In addition, they provide advanced features, such as Remote Direct Memory Access (RDMA), that enable the design of novel communication protocols and libraries. Recent studies [7, 9, 10, 12, 13, 15] have shed light on the possible performance improvements for different Big Data computing middleware using InfiniBand networks, including MapReduce [12, 13, 17].

Spark is a communication intensive framework because of the shuffle operations that repartition Resilient Distributed Datasets (RDD) [19]. Wide dependencies between RDDs, require data from all parent partitions to be available and to be shuffled across the nodes using a MapReduce-like operation. These communication protocols are layered on top of Sockets, which usually need multiple data copies, resulting in poor performance in terms of both latency and throughput [9, 10]. Consequently, even though the underlying system is equipped with high-performance interconnects such as InfiniBand, Spark cannot fully utilize the hardware's capabilities to obtain optimal performance. These issues lead us to believe that Spark needs to adopt RDMA to serve as a part of its communication infrastructure. In order to do so, we must consider the following broad questions and design challenges:

- **Is it worth it?** Is the performance improvement potential high enough, if we can successfully adapt RDMA to Spark? Will the potential improvement be a few percentage points or orders of magnitude?
- **How difficult is it to adapt RDMA to Spark?** Can RDMA be adapted to suit the communication needs of Spark? Is it viable to have to rewrite portions of the Spark code with RDMA?
- **Can Spark applications benefit from an RDMA-enhanced design?** What are the performance benefits that can be achieved by using RDMA feature for Spark applications on modern HPC clusters?

To address the above challenges, this paper first investigates the performance improvement potential for adapting RDMA to Spark through primitive-/application-level benchmark evaluations on different communication protocols (10GigE, IPoIB, and RDMA). In addition, we further propose an RDMA-based design for accelerating data shuffle in Spark on high-performance networks. In this design, we propose a high-throughput data shuffle framework based on Staged Event-Driven Architecture (SEDA) [18]. Within the framework, we have designed an RDMA-based shuffle engine with advanced

*This research is supported in part by National Science Foundation grants #OCI-1148371, #CCF-1213084 and #CNS-1347189. It used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number OCI-1053575.

978-1-4799-5860-3/14 $31.00 © 2014 IEEE
DOI 10.1109/HOTI.2014.15

features, such as dynamic RDMA connection management and sharing, non-blocking and out-of-order data transfer, and off-JVM-heap buffer management. Performance evaluations show that our proposed design can achieve 79-83% performance improvement for GroupBy, compared with the default Spark running with IP over InfiniBand (IPoIB) FDR on a 128-256 core cluster. These tests show that Spark augmented with RDMA can significantly speed up application performance. This paper adopts a plug-in based approach that can make our design to be easily integrated with newer Spark releases. To the best our knowledge, this is the first design to accelerate Spark with RDMA for Big Data processing.

The rest of the paper is organized as follows. Section II presents the background about Spark. We further discuss the opportunities in accelerating Spark on top of high-performance interconnects in Section III. Section IV presents our proposed design. Section V describes our detailed evaluation. Section VI discusses related work. We conclude in Section VII with future work.

II. Background

A. Spark Overview

Spark [19, 20] is an open source data analytics cluster computing framework originally developed in the AMPLab at UC Berkeley. It was designed for specific types of workloads in cluster computing namely — iterative workloads such as machine learning algorithms that reuse a working set of data across parallel operations and interactive data mining. To optimize for these types of workloads, Spark employs the concept of in-memory cluster computing, where datasets can be cached in memory to reduce their access latency. Spark's architecture revolves around the concept of a Resilient Distributed Dataset (RDD) [19], which is a fault-tolerant collection of objects distributed across a set of nodes that can be operated on in parallel. These collections are resilient, because they can be rebuilt if a portion of the dataset is lost. To achieve fault tolerance efficiently, RDDs provide a restricted form of shared memory based on coarse-grained transformations.

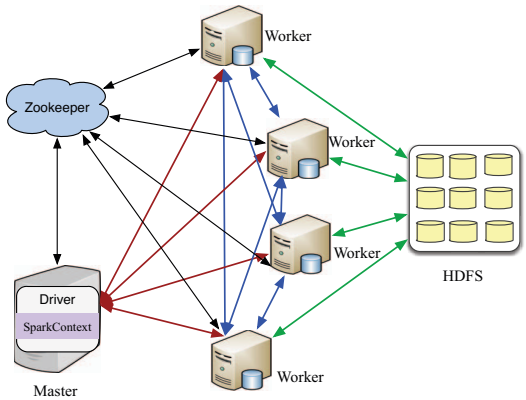


Fig. 1: Overview of Spark Architecture

Spark applications run as independent sets of processes on a cluster, coordinated by the *SparkContext* object in your main program called the *driver program*. On a cluster, *SparkContext* can connect to several types of cluster managers, either Stand-alone Cluster Manager, Apache Mesos or Hadoop YARN. In stand-alone mode, the environment consists of one Spark Master and several Spark Worker processes, as shown in Figure 1. Spark also utilizes Zookeeper to enable high availability. Once the cluster manager allocates resources across applications, Spark will acquire executors on the worker nodes, which are responsible for running computations and storing data for the application. It then sends the application code to the executors. Finally, *SparkContext* sends tasks for the executors to run. At the end of the execution, actions are executed at the workers and results are returned to the *driver program*. As this paper mainly focuses on network communication aspects of Spark, we only consider the Stand-alone Cluster Manager.

B. Dependencies in Spark

The tasks run by the executor are made up of two types of operations, supported by the RDDs: an *action* which performs a computation on a dataset and returns a value to the *driver* and a *transformation*, which creates a new dataset from an existing dataset. The *transformation* operation specifies the processing dependency Directed Acyclic Graph (DAG) among RDDs.

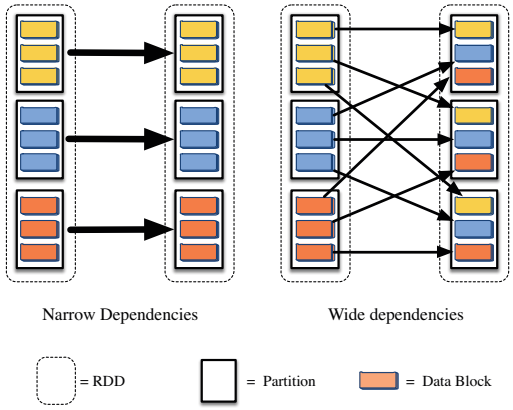


Fig. 2: Spark dependencies

As illustrated in Figure 2, these dependencies come in two forms [19]: *narrow* dependencies (e.g. Map, Filter), where each partition of the parent RDD is used by at most one partition of the child RDD and *wide* dependencies (e.g. GroupByKey, Join), where multiple child partitions may depend on the same partition of the parent RDD. Evidently, *wide* dependencies involve data shuffle across the network. Thus, *wide* dependencies are communication intensive and can turn out to be a performance bottleneck for Spark applications.

C. GroupBy Test

GroupBy Test is a commonly used Spark benchmark that uses the *groupByKey* operation. The *groupByKey* operation is a complex transformation that groups the values for each key in the RDD into a single sequence and results in *wide* dependencies. As mentioned above, this operation utilizes the network extensively for shuffling data across the worker nodes.

III. Assessment of Performance Improvement Potential

In this section, we discuss the opportunities to bring the benefits of RDMA in the Spark framework.

To assess the performance improvement potential, we first need to determine how much benefit RDMA can bring in the communication times of Spark applications compared to other interconnects. For this, we have performed some primitive-level micro-benchmark evaluations to identify how much performance improvement is achievable from the use of RDMA in Java-based environments. This micro-benchmark is written in Java and message transfers occur in a ping-pong manner. The library for RDMA communication is invoked via Java Native Interface (JNI). Figures 3(a) and 3(b) show the point-to-point latencies of small and large message sizes over different interconnects and protocols. As observed from these figures, RDMA can reduce the latencies significantly for all the message sizes. Figure 3(c) shows the point-to-point peak bandwidth achievable over different interconnects and protocols. These numbers are also taken with a Java-based micro-benchmark wherein the client sends messages to the server in a ping-ping manner. The figure clearly shows that RDMA can significantly improve the peak bandwidth compared to the other interconnects and protocols.

We also analyzed the communication performance of the Spark framework using the *GroupBy Test* workload over different interconnects. Figure 4(a) shows that the execution time of the GroupBy operation is significantly improved over high-performance networks and protocols (e.g. 10GigE, IPoIB). Figures 4(b) and 4(c) present network usage profiling data for the test with 10GB data size on 4 nodes of Cluster A (described in Section V-A). As observed from these figures, the network throughput for high-performance interconnects like InfiniBand is much higher than that of traditional Ethernet (e.g. 1GigE) for both `send` and `receive`. All of these prove that the Spark framework can benefit from high-performance interconnects and protocols to a greater extent. However, current Spark design can only use IPoIB or 10GigE by Java Sockets. As shown in Figure 3, RDMA-based communication provides much higher performance than IPoIB and 10GigE. This motivates us to answer the question: *Can RDMA further benefit Spark's performance compared with IPoIB and 10GigE?*

IV. Proposed Design

In this section, we will describe the architecture and design details of RDMA-based data shuffle for Spark over InfiniBand.

A. Architecture Overview

In this section, we first give an overview of the RDMA-based data shuffle architecture for Spark. From the perspective of the entire ecosystem, the new proposed design should provide better performance, while keeping the existing Spark architecture and interface intact. In this way, the existing Spark applications on top of this ecosystem will get the benefits of our designs transparently. In order to achieve this, we adopt the plug-in based approach to extend current shuffle framework in Spark. By utilizing class inheritance and method overriding features supported in both Scala and Java, we define new classes and implement some methods to override the important functions in existing shuffle framework. At the same time, through around 100 lines of code changes inside Spark original files, we can integrate our new designs into the Spark shuffle framework as plug-ins.

Figure 5 depicts a high-level overview of the RDMA-based data shuffle architecture for Spark. Spark can currently support Scala, Java, and Python applications based on Spark APIs. These applications can be executed by a set of tasks, which will be scheduled and launched according to their dependencies in application DAGs. When wide dependencies exist between two groups of tasks, it will cause a global many-to-many data shuffle process, which is communication intensive. The default Spark design provides two approaches to perform data shuffle. The first one is Java NIO (New IO) based data shuffle, which is the default and corresponds to the components of NIO Shuffle Server and Fetcher as shown in Figure 5. The second one is based on the Netty [1] OIO (Old IO) model, which is an optional approach, including two components namely Netty Shuffle Server and Fetcher in Spark. Even though the Spark source code mentions that the Netty OIO communication channel could provide better performance in some cases, we have seen that the default NIO-based design performs better on our testbeds. Spark also indicates that they would like to converge the two approaches and use the NIO-based communication channel eventually. In this paper, we focus on comparing the performance of our design with the default NIO-based data shuffle scheme.

In our approach, we redesign the communication layer of Spark data shuffle by RDMA over InfiniBand instead of the Java Socket interface. We propose a hybrid approach for Spark that uses RDMA for data shuffle via JNI, while all other Spark operations go over the Java Socket interface. One of the major objectives of this research is to find out how much performance improvement can be achieved for data shuffle in Spark by enhancing the communication layer. Therefore, we have kept the existing Spark architecture and interface intact.

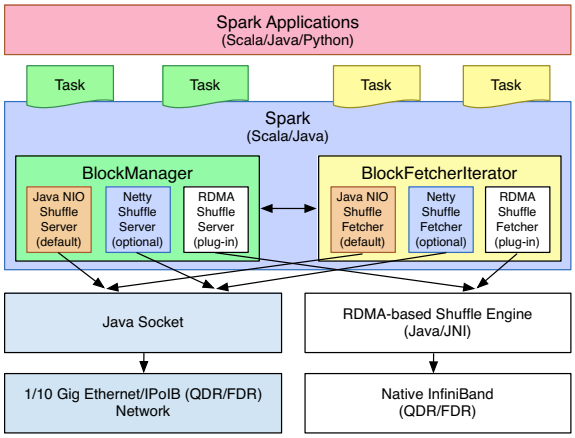
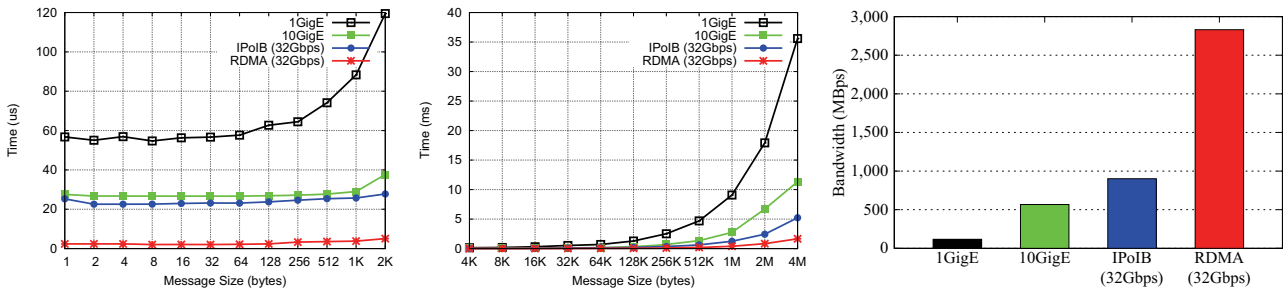


Fig. 5: Architecture of RDMA-based data shuffle for Spark

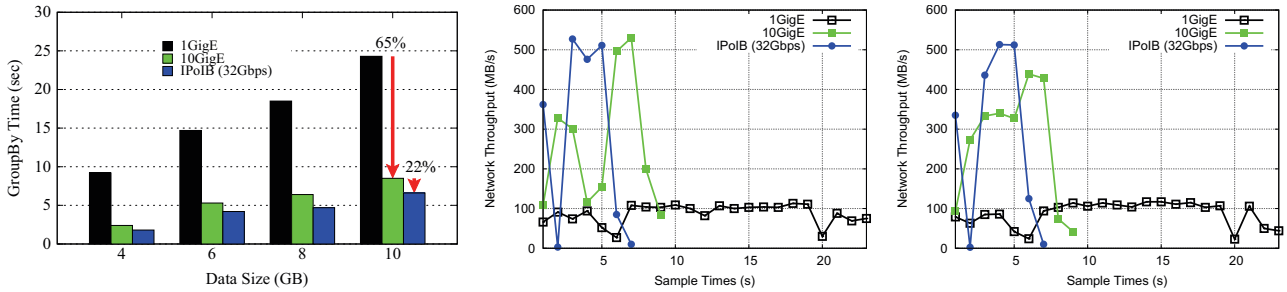
B. SEDA-based Data Shuffle Plug-ins

High throughput is one of the major goals in designing data shuffle systems. In the distributed computing field, Staged Event-Driven Architecture (SEDA) [18] is widely used for designing these kinds of systems. SEDA has been considered a high-throughput software architecture. Its basic principle is to decompose a complex processing logic into a set of stages connected by queues. A dedicated thread pool will be in charge of processing events on the corresponding queue for each stage. By performing admission controls on these event queues, the whole system achieves high throughput through maximally overlapping different processing stages.



(a) Latency (small messages) (b) Latency (large messages) (c) Bandwidth

Fig. 3: Micro-benchmark evaluation (Cluster A)



(a) Performance comparison for GroupBy (b) Network throughput in receive (c) Network throughput in send

Fig. 4: Opportunities for performance improvement with Spark over high-performance interconnects (Cluster A)

Our previous efforts on improving Hadoop components [6, 10] have also shown the SEDA approach is applicable for RDMA-based data processing systems. In this paper, we choose SEDA to design high-throughput data shuffle plug-ins.

Figure 6 shows the event-driven architecture of the data shuffle server plug-in with RDMA. This server side plug-in consists of a listener for accepting new RDMA connections, an RDMA-based receiver that receives data requests from all RDMA connections, a set of handler threads for the request processing stage, and a set of responder threads for serving responses. All of the actual connection establishment and data exchanging are implemented by the RDMA-based shuffle engine, which will be introduced in Section IV-C. Through interaction with low-layer shuffle engine, the upper layer threads can focus on Spark-level processing while offloading the actual communication process to the engine. In this way, we can achieve a good trade-off between performance and implementation complexity. Each accepted connection object has one end-point which connects to the client-side fetcher plug-in, and multiple connections with clients support multi-endpoint based RDMA communication. All of these connections are put in the connection pool. Whenever a request is received, the receiver will just put it into the request queue and notify one of the handlers to handle it. The receiver will then continue to receive more requests to follow. When a handler thread is triggered, it will start to handle a received request by getting the requested block information, reading the block data into RDMA buffers for shuffle, and putting a response object with the buffer pointer (without data copy) into the response queue. After this, one of the responder threads will be awakened. This responder thread will get the connection from the connection pool for current response and then send the response out through the selected connection. Thus, we divide the whole process into multiple stages and each stage can overlap with the others by using events and thread pools. In this manner, our proposed design can achieve the default task-level parallelism in Spark, as well as overlapping within block processing with our shuffle engine.

In the client side, we have chosen to utilize a similar idea to design high-throughput RDMA-based data shuffle fetchers. To save the space, we skip the client-side design discussion.

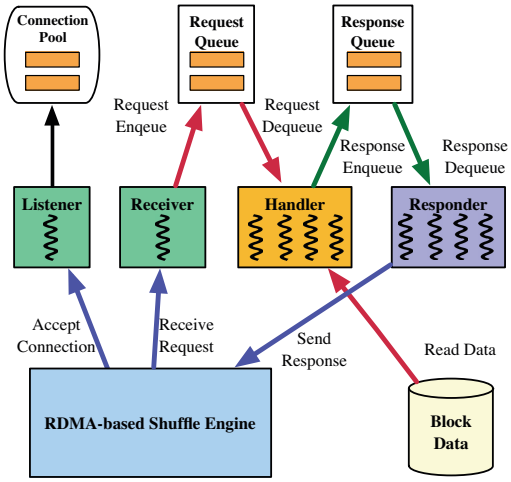


Fig. 6: SEDA-based Data Shuffle Server Plug-in with RDMA

C. RDMA-based Shuffle Engine

In this section, we discuss the major features of our RDMA-based shuffle engine design, which aims for high-

performance data shuffle over InfiniBand.

Dynamic RDMA Connection Management and Sharing: As presented in our earlier work [7, 12], the overhead of the RDMA connection establishment is a little higher than that of Sockets connection establishment. One alternative approach to hide this kind of overhead is to choose a pre-connection mechanism, i.e., before the actual communication, we pre-connect the processes to each other. Though we can hide the connection establishment overhead by this approach for the actual communication, we end up sacrificing more resources (e.g. memory) to keep all of these connections alive, even when there is no communication in progress. Since these resources are expensive for Spark, the pre-connection mechanism does not fit here well. In this paper, we choose dynamic RDMA connection management, wherein a connection will be established if and only if an actual data exchange is going to take place. However, a naive dynamic connection design is not optimal because we need to allocate resources for every data block transfer. We therefore need to design an advanced dynamic connection scheme that reduces the number of connections. Unlike MapReduce, Spark uses multi-threading approach to support multi-task execution in a single JVM, by default, which provides a good opportunity to share resources. This inspires us to share the RDMA connection among different tasks as long as they want to transfer data to the same destination. This leads us to consider another issue: how long should the connection be kept alive for possible re-use? If we do not close the connection till the job finishes, it will fall back to the pre-connection scheme mentioned above. In order to resolve this, we design a time out mechanism for connection destroy. The time out threshold can be assigned by the users through a configuration parameter. When the time out event occurs, the connection will be automatically garbage collected by our engine. In this way, we can achieve a design trade-off between resource utilization and performance.

Non-blocking and Out-of-order Data Transfer: Since we choose dynamic connection management and sharing, each connection may be used by multiple tasks (threads) to transfer data concurrently. In this case, packets over the same communication lane will go to different entities in both server and fetcher sides. In order to achieve high throughput, it is not feasible to perform sequential transfers of complete blocks over a communication lane, as this may cause long wait times for some tasks that are ready to transfer data over the same connection. In contrast, our design will divide a large data block to a sequence of chunks (default chunk size is 512 KB) and send them out in a non-blocking fashion. This can improve the network bandwidth utilization by offloading more packets to the NIC, while it also brings more challenges to our design, since the messages over the connection are out-of-order. In order to guarantee both performance and ordering, we need more designs from the buffer management perspective, which will be discussed later. In short, through non-blocking and out-of-order data transfer, our design can efficiently work with the dynamic connection management and sharing mechanism.

Off-JVM-Heap Buffer Management: In our data shuffle engine, we have a pool of buffers that can be used for both sending and receiving data, at any end-point. The buffer pool is constructed from a ring of off-JVM-heap buffers that are also mapped to the Java/Scala layer as shadow buffers based on the Java NIO Direct Buffer mechanism. This helps JNI and Java/Scala layers to obtain data directly from each other. These buffers are also registered for RDMA communication and can be used in a round-robin fashion. We decouple the buffer management and connection management to provide more flexibility for upper layer design choices. For example, when we decide to choose connection sharing mechanism, we only need to change buffer layout by incorporating a request ID into the buffer header. In this way, every message over the same connection can be differentiated based on its request ID. In addition, by including a sequence number, we can guarantee that the packets that arrive out-of-order on the receiver side can be reordered easily, without much overhead. Furthermore, for non-blocking send, we need a flag to indicate the status of buffer. We implement a callback mechanism to update the buffer status. Whenever one send operation is completely finished, the callback function will change the flag to indicate that the buffer is free for reuse. Through these, we can see that having a pool of off-JVM-heap buffers is reasonable for high-performance and robust handling of data packets in our event-driven communication architecture, while providing scope for more advanced communication features.

V. Performance Evaluation

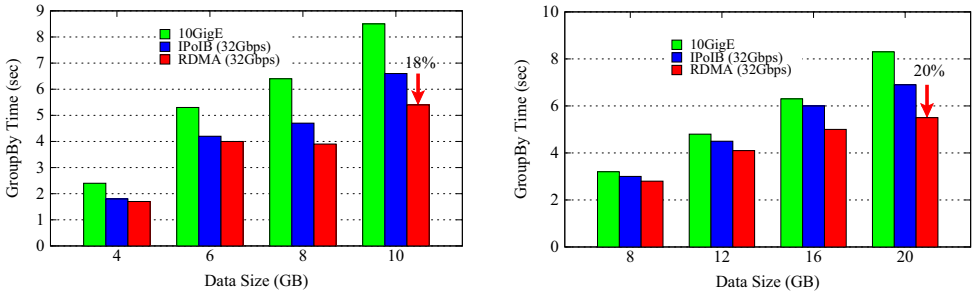
In this section, we present the detailed performance evaluations of our RDMA-based design for Spark and compare the performance of the RDMA-based design with that of the default architecture over various interconnects and protocols. We have used the GroupBy Test benchmark for our evaluations. We have also performed profiling of system resource utilization to analyze the performance of our proposed design.

A. Experimental Setup

(1) Intel Westmere Cluster (Cluster A): This cluster has nine nodes. Each node has Intel Westmere Xeon dual quad-core processors operating at 2.67 GHz. Each node is equipped with 24 GB and two 1TB HDDs. Nodes in this cluster also have NetEffect NE020 10Gb Accelerated Ethernet Adapters that are connected using a 24 port Fulcrum Focalpoint switch. The nodes are also interconnected with Mellanox InfiniBand QDR (32Gbps) HCAs. Each node runs Red Hat Enterprise Linux Server release 6.1.

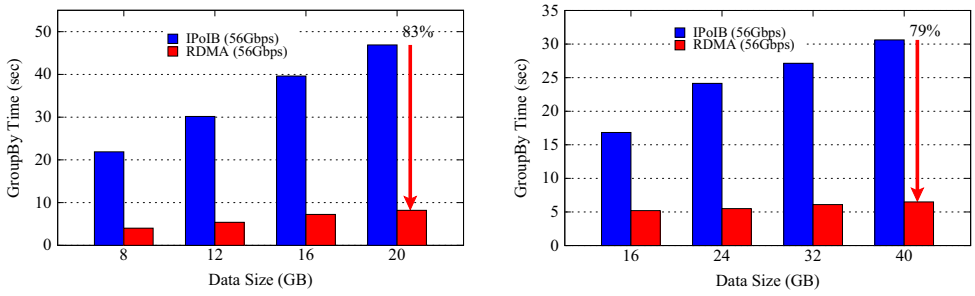
(2) TACC Stampede [14] (Cluster B): We use the Stampede supercomputing system at TACC [14] for our experiments. According to TOP500 [16] list in June 2014, this cluster is listed as the 7th fastest supercomputer worldwide. Each node in this cluster has Intel Sandy Bridge (E5-2680) dual octa-core processors, running at 2.70GHz. It has 32GB of memory, a SE10P (B0-KNC) co-processor and a Mellanox InfiniBand FDR (56Gbps) MT4099 HCA. The host processors are running CentOS release 6.3 (Final). Each node is equipped with a single 80 GB HDD.

For Cluster A, we can show performance comparisons over 10GigE, IPoIB (QDR), and RDMA (QDR) on Intel Westmere architecture. For Cluster B, we can show the numbers on a newer and larger hardware platform by comparing the performance between IPoIB (FDR) and RDMA (FDR) on Intel Sandy Bridge architecture. In all our experiments, we have used Spark 0.9.1, Scala 2.10.4, and JDK 1.7.0. Since our main focus is to optimize data shuffle, we report the



(a) Cluster A with 4 nodes, **32** cores (b) Cluster A with 8 nodes, **64** cores

Fig. 7: Performance evaluation for GroupBy on Cluster A



(a) Cluster B with 8 nodes, **128** cores (b) Cluster B with 16 nodes, **256** cores

Fig. 8: Performance evaluation for GroupBy on Cluster B

GroupBy operation time, instead of the entire GroupBy Test batch job execution time, which includes data generation and computation times.

B. Evaluation with GroupBy

In this section, we discuss the performance results of the GroupBy Test benchmark using our proposed design and compare them with those from default Spark.

Figure 7(a) shows the performance of GroupBy on Cluster A. These experiments are run on 4 nodes (32 cores) and we vary the data size from 4GB to 10GB with 32 maps and 32 reduces. As observed from this figure, our design can reduce the GroupBy operation time by up to 18% over IPoIB (32Gbps) and up to 36% over 10GigE. We perform similar experiments on 8 nodes (64 cores) of Cluster A. Figure 7(b) shows the results of our evaluations on 8 nodes of Cluster A. Here, we vary the data size from 8GB to 20GB with 64 maps and 64 reduces and obtain a performance improvement of up to 20% over IPoIB (32Gbps) and 34% over 10GigE.

We perform similar experiments on Cluster B with varying cluster sizes. On 8 nodes (128 cores), we vary the data size of GroupBy from 8GB to 20GB with 128 maps 128 reduces. As shown in Figure 8(a), our design can reduce the GroupBy operation time by up to 83% over IPoIB (56Gbps) on this cluster size. We also run GroupBy on a 16-node (256 cores) environment with 256 maps and 256 reduces and vary the data size from 16GB to 40GB. As observed from Figure 8(b), the performance of GroupBy is improved by up to 79% over IPoIB (56Gbps) by our design.

We observe that our RDMA-based design shows higher benefits on Cluster B and then we further analyze the performance benefits of our design. We first run some Java-level bandwidth tests using the micro-benchmark described in Section III. Table I shows the point-to-point peak bandwidth of both IPoIB (56Gbps) and RDMA (56Gbps) on Cluster B. As observed from this table, RDMA (56Gbps) can achieve a much higher peak bandwidth than IPoIB (56Gbps), which leads to large performance benefits for our design.

	IPoIB (56Gbps)	RDMA (56Gbps)
Peak Bandwidth	1741.46MBps	5612.55MBps

TABLE I: Peak Bandwidth on Cluster B

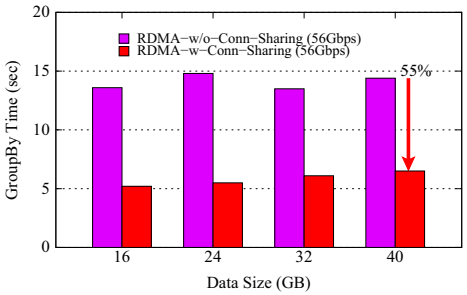
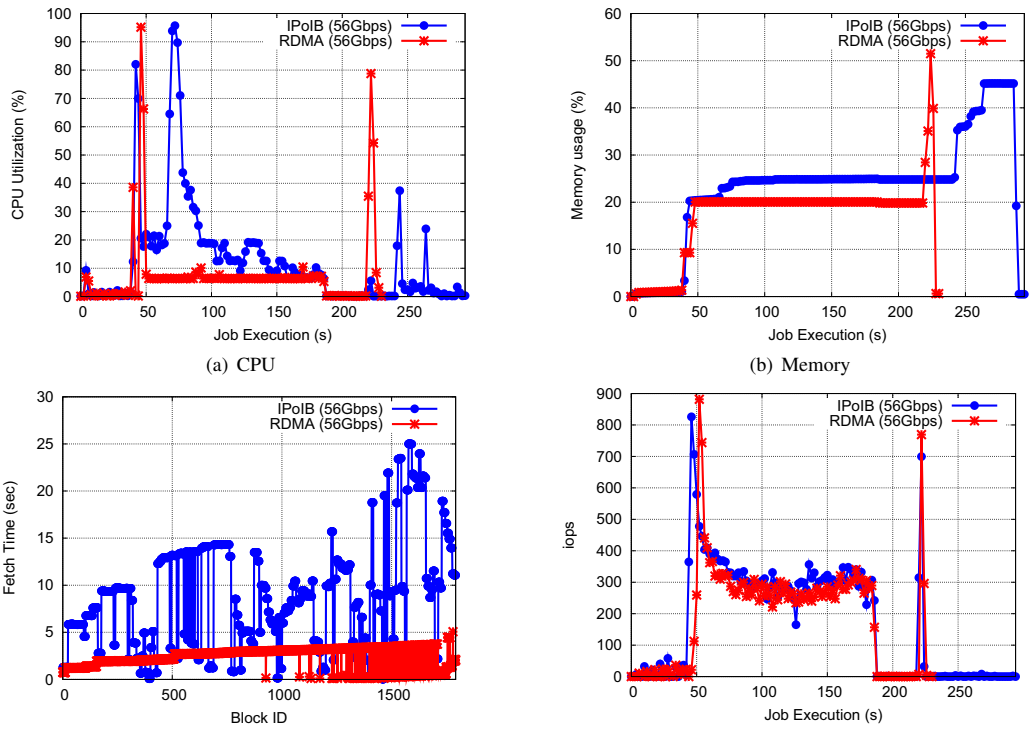


Fig. 9: Performance benefit of connection sharing

Another reason for the performance benefits on Cluster B is the RDMA connection sharing design as mentioned in Section IV-C. Figure 9 illustrates this scenario. Here, we observe that by enabling connection sharing, our design can achieve 55% performance benefit for 40 GB data size on 16 nodes of Cluster B compared to the same design with connection sharing disabled. This validates that leveraging maximum possible benefit from IB FDR interconnect is possible through our design.



(a) CPU (b) Memory (c) Network (d) I/O

Fig. 10: Resource utilization on a worker node of Cluster B

C. System Resource Utilization

In this section, we discuss different system resource utilization for our proposed design and compare with those of the default. Figure 10 presents all these sampling data for the test with 20GB data size on 8 nodes of Cluster B. We skip similar experiments for other cases due to space limitations. To measure different system resource parameters, we use Linux performance monitoring tool, `sar`, provided as a part of the `sysstat` package.

Figure 10(a) shows the CPU utilization for the entire execution of the GroupBy Test benchmark. To evaluate the CPU utilization, we measure the system idle% using `sar` and subtract this from 100 to obtain CPU usage at each point. We use an interval of 2 seconds to report CPU usage. Here, we observe that the CPU usage for our design is almost similar to that of the default one during the early stages. However, at the end, our design has a higher CPU usage. Because of the latency-sensitivity of this benchmark, we use polling-based mechanism in our RDMA design and thus, during this part of execution, the CPU usage is high.

To measure memory usage, we use `sar` and report the average memory usage. As shown in Figure 10(b), both the default approach and our design consume similar memory space. However, our design has a little higher peak memory usage at the end and provides faster execution of the benchmark.

We also provide the network usage during shuffle stage. We provide this measurement by profiling both our design and default approach during run-time. Since, for RDMA, we cannot get the true picture of network activity using `sar`, we use profiling approach. For this, we profile the shuffled data on each block during the shuffle progress. Figure 10(c) illustrates these results. Here, on X-axis, we present each block id and on Y-axis, we show the corresponding fetching time. Here, the fetch time includes all the processing times in addition to the communication, queuing and connection setup time. From this figure, we can clearly observe that RDMA-based design achieves much faster fetch times compared to the default approach running over IPoIB.

For I/O usage, we report iops (I/O transactions per second) by `sar`. As shown in Figure 10(d), we can see that the RDMA-based design has similar I/O activity compared to the default approach. Since our approach is not minimizing any disk activity inside Spark, this behavior is as expected.

VI. Related Work

We summarize and discuss the related work along the following two different categories.

Spark and its related work: With the increasing demand to discover and explore data for real-time insights, the need to extend MapReduce became apparent and this led to the emergence of Spark [20], an in-memory cluster computing framework that allows user programs to load data into a cluster's memory and query it repeatedly, making it well-suited to machine learning algorithms. Spark caches the intermediate data into memory and provides the abstraction of Resilient Distributed Datasets (RDDs) [19] to support lineage-based data recovery (re-computation). Furthermore, studies [2] have been carried out to identify that the shuffle phase incurs considerable overhead and explore several alternatives to mitigate the associated bottlenecks. This paper attempts to explore

the performance benefits by using RDMA to improve the performance of Spark over high-performance networks.

Optimizing Hadoop over high-performance networks: Data movement in Hadoop has proven to be expensive and lots of effort has been dedicated to accelerate this over high-performance networks that are prevalent in most modern clusters today. For instance, one of our earlier work [15] examined the impact of high-speed interconnects such as 10 GigE and InfiniBand (IB) using protocols such as TCP, IP-over-IB (IPoIB), and Sockets Direct Protocol (SDP), on the performance of HDFS and found that the results are promising. We further proposed RDMA-enhanced HDFS designs [7, 8] to improve HDFS performance with pipelined and parallel replication schemes, respectively. Recent studies [12, 13, 17] have shed light on the possible performance improvements for Hadoop MapReduce using InfiniBand networks. Similarly, some recent studies [4, 10] have shown that in addition to MapReduce and HDFS, other Hadoop components such as HBase, RPC, etc., can be improved significantly by leveraging the advanced features offered by high-performance networks. The RDMA for Apache Hadoop package developed by our research group is publicly available from [11].

VII. Conclusions

In this paper, we have analyzed the performance improvement potential for Spark over high-performance interconnects on modern HPC clusters. We have proposed an RDMA-based design for data shuffle of Spark over InfiniBand. Based on our study, we make the following three conclusions:

- **RDMA and high-performance interconnects can benefit Spark.** Through assessment of performance improvement potential and proposing a new RDMA-enhanced design for Spark, we have actually improved the performance of GroupBy by up to 83% over IPoIB on an FDR cluster.
- **Plug-in based approach with SEDA-/RDMA-based designs provides both performance and productivity.** With around 100 lines of code changes in Spark, we have successfully integrated our plug-in into the whole Spark framework. At the same time, the SEDA based plug-in design with advanced features of RDMA-based data shuffle engine has been demonstrated to deliver performance benefits.
- **Spark applications can benefit from an RDMA-enhanced design.** Our proposed new design keeps the default Spark architecture and its APIs intact, which can provide performance benefits to the end-applications transparently.

In the future, we plan to investigate more bottlenecks of Spark over high-performance networks and propose new designs along these directions. We will also evaluate our designs with other benchmarks and applications (both interactive and iterative) on large-scale clusters.

References

[1] "The Netty Project," http://netty.io.

[2] A. Davidson and A. Or, "Optimizing Shuffle Performance in Spark," University of California, Berkeley - Department of Electrical Engineering and Computer Sciences, Tech. Rep., 2013.

[3] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design and Implementation (OSDI)*, CA, December 2004.

[4] J. Huang, X. Ouyang, J. Jose, M. W. Rahman, H. Wang, M. Luo, H. Subramoni, C. Murthy, and D. K. Panda, "High-Performance Design of HBase with RDMA over InfiniBand," in *Proceedings of IEEE 26th International Parallel and Distributed Processing Symposium (IPDPS)*, Shanghai, China, May 2012.

[5] International Data Corporation (IDC), "New IDC Worldwide HPC End-User Study Identifies Latest Trends in High Performance Computing Usage and Spending," http://www.idc.com/getdoc.jsp?containerId=prUS24409313.

[6] N. S. Islam, X. Lu, M. W. Rahman, and D. K. Panda, "SOR-HDFS: A SEDA-based Approach to Maximize Overlapping in RDMA-Enhanced HDFS," in *Proceedings of The 23rd International ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, Vancouver, Canada, June 2014.

[7] 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 The International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Salt Lake City, November 2012.

[8] N. S. Islam, X. Lu, M. W. Rahman, and D. K. Panda, "Can Parallel Replication Benefit Hadoop Distributed File System for High Performance Interconnects?" in *Proceedings of IEEE 21st Annual Symposium on High-Performance Interconnects (HOTI)*, San Jose, CA, August 2013.

[9] J. Jose, H. Subramoni, M. Luo, M. Zhang, J. Huang, M. W. Rahman, N. S. Islam, X. Ouyang, H. Wang, S. Sur, and D. K. Panda, "Memcached Design on High Performance RDMA Capable Interconnects," in *Proceedings of IEEE 40th International Conference on Parallel Processing (ICPP)*, Sept 2011.

[10] X. Lu, N. S. Islam, M. W. Rahman, J. Jose, H. Subramoni, H. Wang, and D. K. Panda, "High-Performance Design of Hadoop RPC with RDMA over InfiniBand," in *Proceedings of IEEE 42nd International Conference on Parallel Processing (ICPP)*, France, October 2013.

[11] OSU NBC Lab, "RDMA for Apache Hadoop: High-Performance Design of Apache Hadoop over RDMA-enabled Interconnects," http://hadoop-rdma.cse.ohio-state.edu.

[12] M. W. Rahman, N. S. Islam, X. Lu, J. Jose, H. Subramoni, H. Wang, and D. K. Panda, "High-Performance RDMA-based Design of Hadoop MapReduce over InfiniBand," in *Proceedings of International Workshop on High Performance Data Intensive Computing (HPDIC), in conjunction with IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Boston, May 2013.

[13] M. W. Rahman, X. Lu, N. S. Islam, and D. K. Panda, "HOMR: A Hybrid Approach to Exploit Maximum Overlapping in MapReduce over High Performance Interconnects," in *Proceedings of the 28th ACM International Conference on Supercomputing (ICS)*, Munich, Germany, 2014.

[14] Stampede at Texas Advanced Computing Center, http://www.tacc.utexas.edu/resources/hpc/stampede.

[15] S. Sur, H. Wang, J. Huang, X. Ouyang, and D. K. Panda, "Can High Performance Interconnects Benefit Hadoop Distributed File System?" in *Proceedings of the Workshop on Micro Architectural Support for Virtualization, Data Center Computing, and Clouds, in Conjunction with MICRO*, Atlanta, GA, December 2010.

[16] Top500 Supercomputing System, http://www.top500.org.

[17] Y. Wang, X. Que, W. Yu, D. Goldenberg, and D. Sehgal, "Hadoop Acceleration through Network Levitated Merge," in *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Seattle, WA, November 2011.

[18] M. Welsh, D. Culler, and E. Brewer, "SEDA: An Architecture for Well-Conditioned, Scalable Internet Services," in *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, Banff, Alberta, Canada, 2001.

[19] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, San Jose, CA, 2012.

[20] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster Computing with Working Sets," in *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud)*, Boston, MA, 2010.