2013 IEEE 27th International Symposium on Parallel & Distributed Processing Workshops and PhD Forum

High-Performance RDMA-based Design of Hadoop MapReduce over InfiniBand*

Md. Wasi-ur-Rahman, Nusrat Sharmin Islam, Xiaoyi Lu, Jithin Jose,
Hari Subramoni, Hao Wang and Dhabaleswar K. (DK) Panda

Department of Computer Science and Engineering,
The Ohio State University
{rahmanmd, islamn, luxi, jose, subramon, wangh, panda} @cse.ohio-state.edu

***Abstract*—MapReduce is a very popular programming model used to handle large datasets in enterprise data centers and clouds. Although various implementations of MapReduce exist, Hadoop MapReduce is the most widely used in large data centers like Facebook, Yahoo! and Amazon due to its portability and fault tolerance. Network performance plays a key role in determining the performance of data intensive applications using Hadoop MapReduce as data required by the map and reduce processes can be distributed across the cluster. In this context, data center designers have been looking at high performance interconnects such as InfiniBand to enhance the performance of their Hadoop MapReduce based applications. However, achieving better performance through usage of high performance interconnects like InfiniBand is a significant task. It requires a careful redesign of communication framework inside MapReduce. Several assumptions made for current socket based communication in the current framework do not hold true for high performance interconnects. In this paper, we propose the design of an RDMA-based Hadoop MapReduce over InfiniBand and several design elements: data shuffle over InfiniBand, in-memory merge mechanism for the Reducer, and pre-fetch data for the Mapper. We perform our experiments on native InfiniBand using Remote Direct Memory Access (RDMA) and compare our results with that of Hadoop-A [1] and default Hadoop over different interconnects and protocols. For all these experiments, we perform network level parameter tuning and use optimum values for each Hadoop design. Our performance results show that, for a 100 GB TeraSort running on an eight node cluster, we achieve a performance improvement of 32% over IP-over InfiniBand (IPoIB) and 21% over Hadoop-A. With multiple disks per node, this benefit rises up to 39% over IPoIB and 31% over Hadoop-A.**

I. Introduction

Hadoop [2] is one of the most popular open-source frameworks to handle big data analytic applications. It provides support for MapReduce [3] programming model and Hadoop Distributed FileSystem (HDFS) [4] for big data storage. Hadoop is being used in many research projects and many companies such as Facebook, Yahoo!, etc., due to its scalability, fault-tolerance, and productivity.

The Hadoop MapReduce framework provides application developers several critical functions such as key-based sorting, multi-way merging, data shuffling, etc. When the MapTasks finish the user defined map operation and set key-value pairs into context for the ReduceTasks, the framework executes the shuffle operation which sends key-value pairs from the MapTasks to the appropriate ReduceTask. It also performs the merge of the key-value pairs from multiple mappers on the reducer side. As the size of the dataset increases, the entire operation becomes very communication-intensive.

As big data applications begin scaling to tera bytes and peta bytes of data, the socket based communication model, which is the default implementation in Hadoop MapReduce, demonstrates performance bottleneck. To avoid this potential bottleneck, designers of high end enterprise data centers and clouds have been looking towards high performance interconnects such as InfiniBand to allow the unimpeded scaling of their big data applications [5]. The "Greenplum Analytics Workbench" [6], a 1000+ node InfiniBand based cluster is one of the latest in a series of clusters being deployed to design, develop, and test InfiniBand based solutions for the Hadoop ecosystem. Although such systems are being used, the current Hadoop middleware components do not leverage the high performance communication features offered by InfiniBand. Without high performance InfiniBand support in the vanilla Hadoop system, the big data analytic applications using Hadoop cannot get the best performance on such systems.

Recent research works [1, 7–10] analyze on the huge performance improvements possible for different cloud computing middlewares using InfiniBand networks. The research project Hadoop-A [1] provides native IB verbs support for data shuffle in Hadoop MapReduce on InfiniBand network. However, it does not exploit all the potential performance benefits of high performance networks. As a result, it is necessary to rethink the Hadoop system design when a high performance network is available. This leads to the following research challenges:

1) How do we re-design Hadoop MapReduce to take advantage of high performance networks such as InfiniBand and exploit advanced features such as RDMA?
2) What will be the performance improvement of

*This research is supported in part by National Science Foundation grants #CCF-0937842, #OCI-0926691, #OCI-1148371 and #CCF-1213084.

978-0-7695-4979-8/13 $26.00 © 2013 IEEE
DOI 10.1109/IPDPSW.2013.238

MapReduce applications with the new RDMA based design in Hadoop MapReduce over InfiniBand?

3) Can the new design lead to consistent performance improvement on different hardware configurations, such as with multiple HDD or SSD on the compute node of Hadoop cluster?

In this paper, we address these research challenges. First, we present a detailed RDMA-based design of MapReduce shuffle engine to provide faster data communication. We use Unified Communication Runtime (UCR) [11], a light-weight communication library, to provide faster availability of the data. Second, we design and implement an efficient key-value pair pre-fetching and caching mechanism inside the TaskTracker of Hadoop, which is responsible for fast data shuffle. This mechanism helps reduce the overhead in the reducer side when the shuffle and the merge procedures run in an overlapping manner. Finally, we investigate the advantage of increasing disk bandwidth for our design using multiple HDD and SSD configurations in Hadoop DataNodes.

We perform extensive performance evaluation with popular benchmarks used in Hadoop literature. We compare our performance with that of 1 GigE, IP-over-InfiniBand (IPoIB), 10 GigE, and Hadoop-A [1]. These evaluations show that for TeraSort benchmark, our implementation of Hadoop MapReduce over InfiniBand achieves 21% and 32% benefit in execution time over Hadoop-A and IPoIB (32Gbps), respectively, for 100GB sort. With multiple disks deployed in our design, improvement increases up to 31% over Hadoop-A and 39% over IPoIB. Also, it outperforms Hadoop-A by 32% in regular Sort benchmark with 40GB sort size.

The rest of the paper is organized into sections: in section II, we present background about the key components involved in our design; in section III, we provide the detailed design for both default MapReduce and RDMA-based approach; and in section IV, we describe our experiments and evaluations. Related works are discussed in section V and in section VI we present conclusions and future work.

II. Background

A. Hadoop MapReduce and Benchmarks

Hadoop [2] is a popular open source implementation of the MapReduce [12] programming model. The Hadoop Distributed File System (HDFS) [2, 4] is the primary storage for Hadoop cluster. An HDFS cluster consists of two types of nodes: NameNode and DataNode. The NameNode manages the file system namespace and the DataNodes store the actual data.

Figure 1(a) shows a typical MapReduce cluster. The NameNode has a JobTracker process and all the DataNodes can run one or more TaskTracker processes. These processes, together, act as a master-slave architecture for a MapReduce job. A MapReduce job usually consists of three basic stages: map, shuffle/merge/sort and reduce. Figure 1(b) shows these stages in details.

JobTracker is the service within Hadoop that farms out tasks to specific nodes in the cluster. A single JobTracker and a number of TaskTrackers are responsible for successful completion of a MapReduce job. Each TaskTracker can launch several MapTasks, one per split of data. The map function converts the original records into intermediate results and stores them onto the local file system. Each of these files are sorted into many data partitions, one per ReduceTask. The JobTracker then launches the ReduceTasks as soon as the map outputs are available from the MapTasks. TaskTracker can spawn several concurrent ReduceTasks. Each ReduceTask starts fetching the map outputs from the map output locations that are already completed. This stage is the shuffle/merge period, where the data from various map output locations are sent and received via HTTP requests and responses. While receiving these data from various locations, a merge algorithm is run to merge these data to be used as an input for the reduce operation. Then each ReduceTask loads and processes the merged outputs using the user defined reduce function. The final result is then stored into HDFS.

Some popular MapReduce benchmarks from Apache Hadoop community are mentioned here.

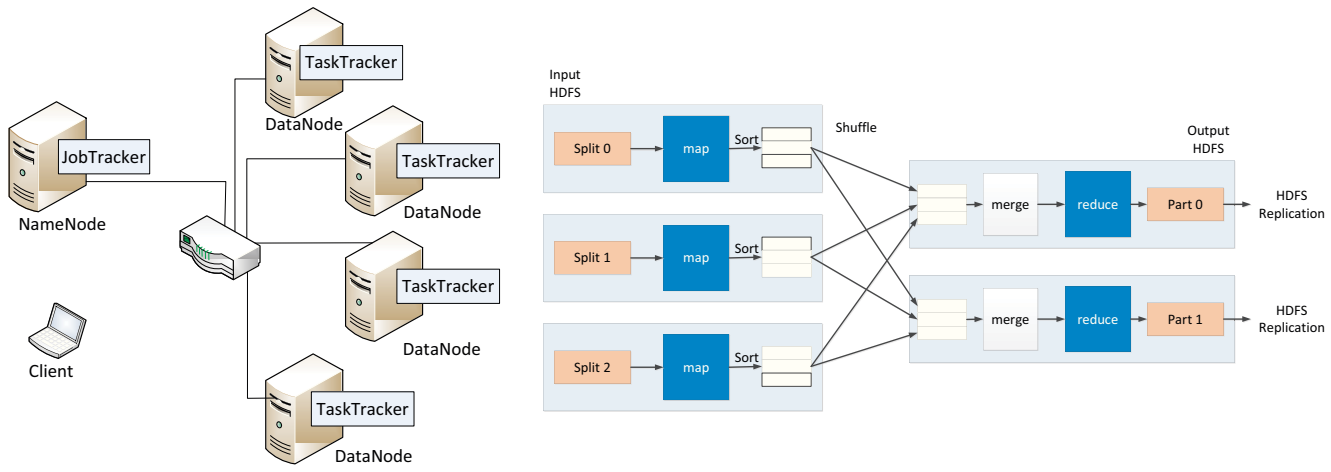
1) TeraSort: TeraSort [2] is probably the most well-known Hadoop benchmark. It is a benchmark that combines testing the HDFS and MapReduce layers of a Hadoop cluster. The input data for TeraSort can be generated by TeraGen [2] tool, which writes the desired number of rows of data in the input directory. By default, the key and value size is fixed for this benchmark at 100 bytes. The output of the TeraSort can be validated by a TeraValidate tool which is also implemented in Hadoop trunk.

2) Sort: The Sort [13] benchmark uses the MapReduce framework to sort the input directory into the output directory. The input of this benchmark can be generated by RandomWriter [14] which writes random-sized, key value pairs in HDFS. This benchmark is a very useful tool to measure the performance efficiency of MapReduce cluster.

B. High Performance Networks

In this section, we present an overview of different networking technologies that can be utilized in a data center for high-performance communication. During the past decade, the field of High-Performance Computing (HPC) has been witnessing a transition to commodity clusters with modern interconnects such as InfiniBand and 10 Gigabit Ethernet.

1) InfiniBand: InfiniBand [15] is an industry standard switched fabric that is designed for interconnecting nodes in HPC clusters. It is a high-speed, general purpose I/O interconnect that is widely used by scientific computing centers world-wide. The recently released TOP500 rankings



(a) High-level overview of Hadoop MapReduce architecture

(b) MapReduce data flow with multiple map and reduce tasks

Figure 1. Overview of MapReduce and its components

in November 2012 indicate that more than 44% of the computing systems use InfiniBand as their primary interconnect. One of the main features of InfiniBand is Remote Direct Memory Access (RDMA). This feature allows software to remotely read or update memory contents of another remote process without any software involvement at the remote side. This feature is very powerful and can be used to implement high-performance communication protocols. InfiniBand offers various software layers through which it can be accessed, described below:

(a) **InfiniBand Verbs Layer**: The *verbs* layer is the lowest access layer to InfiniBand. Verbs are used to transfer data and are completely OS-bypassed, i.e. there are no intermediate copies in the OS. The verbs interface follows the Queue Pair (or communication end-points) model. Upper-level software using verbs can place a work request on a queue pair. The work request is then processed by the Host Channel Adapter (HCA). When work is completed, a completion notification is placed on the completion queue. Upper level software can detect completion by polling the completion queue or by asynchronous events (interrupts). The OpenFabrics interface [16] is the most popular verbs access layer due to its applicability to various InfiniBand vendors.

(b) **InfiniBand IP Layer:** InfiniBand software stacks, such as OpenFabrics [16], provide driver for implementing the IP layer. This makes it possible to use the InfiniBand device as just another network interface available from the system with an IP address. Such IB devices are presented as `ib0, ib1` and so on just like other Ethernet IP interfaces. Although the verbs layer in InfiniBand provides OS-bypass, the IP layer does not provide so. This layer is often called "IP-over-IB" or IPoIB for short. We use this terminology in this paper.

2) 10 Gigabit Ethernet: In data center environments, to achieve better performance with respect to higher bandwidth, 10 Gigabit Ethernet is typically used. It is also realized that traditional sockets interface may not be able to support high communication rates [17]. Towards that effort, iWARP (Internet Wide Area RDMA Protocol) standard was introduced for performing RDMA over TCP/IP [18]. In fact, the OpenFabrics [16] network stack provides a unified interface for both iWARP and InfiniBand. In addition to iWARP, there are also hardware accelerated versions of TCP/IP available. These are called TCP Offload Engines (TOE), which use hardware offload.

C. Solid State Drive (SSD) Overview

Solid State Drives have amassed a lot of attention over the recent past owing to significant data-throughput and efficiency gains over traditional spinning-disks. Although it is a mere physical array of fast flash-memory packages, the core-intelligence of an SSD can be attributed to its Flash Translation Layer (FTL) which plays a vital role in the adoption of this technology. Some of major functionality of an SSD, such as Wear leveling, Garbage collection and Logical Block Mapping, is packed into the FTL. High bandwidth as well as low latency makes SSD an ideal candidate to be used in the DataNodes of the Hadoop cluster in order to lessen the I/O bottlenecks for MapReduce applications.

D. Unified Communication Runtime (UCR)

The Unified Communication Runtime (UCR) [11] is a light-weight, high performance communication runtime, designed and developed at The Ohio State University. It aims to unify the communication runtime requirements of scientific parallel programming models, such as MPI and Partitioned

Global Address Space (PGAS) along with those of data-center middle-ware, such as Memcached [19], HBase [20], MapReduce [12], etc.

UCR is designed as a native library to extract high performance from advanced network technologies. The design of UCR has evolved from MVAPICH and MVAPICH2 software stack. MVAPICH2 and MVAPICH2-X [21] are popular implementations of MPI-2 specification (with initial support for MPI-3). These libraries are being used by more than 2,000 organizations in 70 countries and also distributed by popular InfiniBand software stacks and Linux distributions like RedHat and SUSE.

III. Existing and Proposed Design

In this section, we first discuss the default design of Hadoop MapReduce and its various components. Then we propose our design for MapReduce that takes the benefits of RDMA over InfiniBand. We discuss the key components of this design and describe the related challenges and our approach to find the solution. We also contrast our design with that of Hadoop-A in Section III-C.

Figure 2 shows a hybrid version of MapReduce consisting of both the default vanilla design and our proposed RDMA-based design. Here, we keep the existing design and add our RDMA-based communication features as a configurable option with a selection parameter, `mapred.rdma.enabled`. With this parameter set to true, user can enable RDMA-based features and vice versa.

A. Existing Design

Figure 2 shows the hybrid version of MapReduce which has both the existing design and our proposed RDMA-based approach. As discussed in Section II-A, in the default MapReduce workflow, shown in Figure 1(b), map outputs are kept in files residing in local disks which can be accessed by TaskTrackers. During shuffle, reduce processes, that are running in ReduceTasks, ask TaskTrackers to provide these map outputs over HTTP. All the related components that are responsible for these operations are shown in the left side of both ReduceTask and TaskTracker in Figure 2. Some of these components are described here.

HTTP Servlet: HTTP Servlets run in TaskTracker to handle incoming requests from the ReduceTasks. Upon HTTP request, the servlets get the appropriate map output file from local disk and send the output in an HTTP response message.

Copier: Inside the ReduceTask, the copiers are responsible for requesting appropriate map outputs from TaskTracker in an HTTP request message. Upon arrival of the data, it keeps the data in memory, if a sufficient amount of memory is available, or in a local disk, otherwise.

In-Memory Merger: Inside ReduceTask, there are two level of Merge operations. If the data received during shuffle can be held in memory, the in-memory merger merges these data from memory and then keeps the merged output to local disk. Otherwise, only the Local FS Merger is used.

Local FS Merger: Local FS Merger gets the data from local disk which are kept either by Copier or In-Memory Merger. It merges and stores these data in local disk in an iterative manner, minimizing the total number of merged output files in local disk each time.

Map Completion Fetcher: Inside ReduceTask, a Map Completion Fetcher keeps track of currently completed map tasks. After a map completion event occurs, it signals the Copiers to start copying the data for this map output by sending out HTTP request.

In the existing implementation of MapReduce, TaskTracker and ReduceTask exchange their information over Java Sockets. Each HTTP Servlet sends an acknowledgment to the appropriate reducer with the meta information for the data requested and then begins sending data packets one after another. ReduceTask, after receiving the acknowledgment, extracts the meta information and sets appropriate parameters and buffers for receiving the data packets next.

B. Proposed Design

We introduce the major components in our RDMA based MapReduce design and then explain our design details. We first discuss our updated Shuffle design followed by the Merge design.

*1) **RDMA based Shuffle**:* In the shuffle stage, both TaskTracker and ReduceTask are modified to achieve the benefits of RDMA. We have added the following new components in the TaskTracker side:

RDMAListener: Each TaskTracker initiates an RDMAListener during its startup. RDMAListener in TaskTracker waits for incoming connection requests from the ReduceTask side, adds the connection to a pre-established queue, and starts an RDMAReceiver if necessary. We use UCR as the library for communication which is an end-point based library. An end-point is analogous to socket connection. RDMAListener provides end-point for UCR communication and allocates an appropriate buffer for each end-point.

RDMAReceiver: Each RDMAReceiver is responsible for receiving requests from ReduceTasks. RDMAReceiver gets the end-point list that are currently being used and receives request from those end-points. After receiving the request, it places the request in DataRequestQueue.

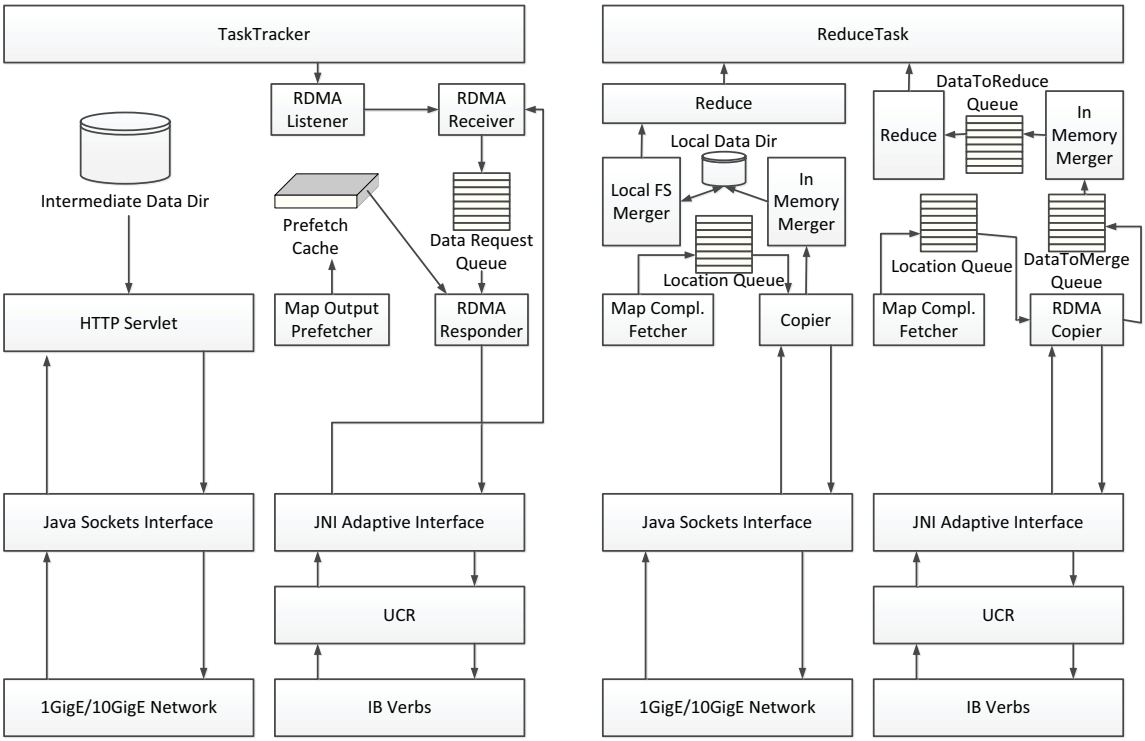


Figure 2. RDMA-based MapReduce architecture and communication characteristics

DataRequestQueue: DataRequestQueue is used to hold all the requests from ReduceTasks. It gets these requests from RDMAReceiver and stores it until one of the RDMAResponders take it for further processing.

RDMAResponder: RDMAResponder belongs to a pool of threads that wait on DataRequestQueue for incoming requests. Whenever a new request gets inserted in Data Request Queue, one of the RDMAResponders responds to that request. It is a very light-weight thread and after sending the response, it immediately goes to wait state if no other requests are available.

The ReduceTask side is also enhanced by adding RDMACopier.

RDMACopier: In the default design of MapReduce, the copier threads are responsible for requesting data to TaskTracker and storing data for Merge. Likewise, RDMACopier sends requests to TaskTrackers and upon arrival of data, it stores data to DataToMergeQueue for merge operation.
Both TaskTracker and ReduceTask have **JNI Adaptive Interface** which enables the Java code to make use of UCR library functions, implemented in native C, for communication. Initially, RDMACopier sends end point information to RDMAListener in TaskTracker to establish the connection. From each ReduceTask, one RDMACopier sends such information to all available TaskTrackers. After a successful map completion event detection by Map Completion Fetcher in the ReduceTask side, RDMACopier sends data request to the TaskTracker for this map's output. RDMAReceiver gets this request and puts the request into DataRequestQueue. RDMAResponder then responds with this request and extracts the required data from PrefetchCache. It then sends the data to the ReduceTask using the same end point. For successful and reliable transmission of data, each request and response messages consist of various identification and control parameters such as map id, reduce id, job id, number of key value pairs sent etc. All these data in shuffle is transferred using RDMA over InfiniBand to leverage the benefits of high performance interconnect in terms of latency and throughput.

*2) **Faster Merge**:* In the default design of MapReduce, each HTTP response consists of the entire map output file, dividing it into packets of the default packet size, 64 KB. With the use of high performance networks, the communication time reduces significantly, which in turn creates the opportunity to transfer one map output file in multiple communication steps instead of one. By doing this, we can start the merge process as soon as some key-value pairs from all map output files reach at the reducer side. For this reason, in our design RDMAResponder in TaskTracker

side sends a certain number of key-value pairs starting from the beginning of map output file instead of sending the entire map output file at once. While receiving these key-value pairs from all map locations, a ReduceTask now merges all these data to build up a Priority Queue. It then keeps extracting the key-value pairs from the Priority Queue in sorted order and puts these data in a first in first out structure, named as DataToReduceQueue. As each map output file is already sorted in the mapper side, the merger in the ReduceTask can only extract the data from Priority Queue until the point when the number of key-value pairs from a particular map decreases to zero. At that point, it needs to get next set of key-value pairs from that particular map task to resume extracting from Priority Queue.

3) ***Intermediate Data Pre-fetching and Caching****:* For faster response in TaskTracker side, we propose an efficient caching mechanism for the intermediate data residing in map output files stored in local disk. User can enable caching in TaskTracker side through a configuration parameter `mapred.local.caching.enabled`. The following is a new component in the TaskTracker side for caching:

MapOutputPrefetcher: MapOutputPrefetcher is a daemon threadpool which caches intermediate map output as soon as it gets available. After finishing a map task, one of the daemons starts to fetch the data from this map output and caches it in PrefetchCache. The novel feature for this cache is that it can adjust caching based on data availability and necessity. It can also prioritize which data to cache more frequently based on the demand from the ReduceTasks. Depending on heap size availability it can limit the amount of data to be cached in PrefetchCache.
Even with caching, cache misses may occur as ReduceTasks' requests may arrive faster than caching. In that case, TaskTracker fetches data directly from disk itself without waiting for caching. But after disk fetch, it requests MapOutputPrefetcher to cache this particular map output data with more priority so that successive requests for this output file can be served from the cache.

Intermediate data pre-fetching and caching plays an important role for achieving better performance with respect to job execution time. For large number of ReduceTasks, more requests for map output files arrive to a single TaskTracker, which can swamp out the I/O bandwidth in TaskTracker side. So, an efficient cache implemented here can alleviate such bottleneck and provide better performance. In Section IV-D, we show how our caching mechanism provides significant performance improvement.

4) ***Overlap of Shuffle, Merge and Reduce****:* Figure 3 shows our design which enhances performance by efficiently overlapping shuffle, merge, and reduce operations in ReduceTask. In the default design, the merge process starts immediately with shuffle. However, there exists an implicit barrier for the start of reduce operation: the reduce operations can only be started after all the merge operations have completed. In our design, we start reduce operation as soon as the first merge completes. In this way, we can achieve maximum benefit by introducing pipelining between merge and reduce stages.

C. Differences with respect to Hadoop-A [1]

Here, we distinguish our proposed RDMA-based design for MapReduce with that of Hadoop-A [1]. Some major differences are as follows:

1) *Data Pre-fetching and Caching*: with RDMA-based communication in the shuffle stage, we redesign the merge mechanism to get more overlapping of the reduce stage with the shuffle and merge stages. In order to respond to the key-value pairs requested from the reducer as soon as possible, we design the intermediate data pre-fetching and caching mechanism in Hadoop TaskTrackers. On the other hand, although Hadoop-A has proposed a DataEngine tool for intermediate data access on the disk of the mapper side [1], DataEngine doesn't provide data caching to decrease the disk access based on the paper. In section IV, we compare our design with Hadoop-A. The better performance of our design is a result of the data pre-fetching and caching mechanism.
2) *JAVA Implementation vs C Implementation*: Hadoop-A has implemented a separate merge algorithm and the DataEngine tool in native C; and, has provided a plug-in based approach to provide RDMA operations on both TaskTracker and ReduceTask side. However, in order to provide minimal code changes in the existing Hadoop codebase, our implementation for merge algorithm still makes use of the existing Merge operation with minimal code changes. And the data pre-fetching and caching mechanism is implemented as a daemon in TaskTracker side which does not require any separate tools.
3) *Configuration and Tuning Interfaces*: Between these two RDMA based implementations, our design has more user level flexibility in terms of configuration and tuning. We provide a number of configuration parameters such as RDMA packet size, enabling of caching, number of key, value pairs transmitted in each packet, etc. to give user the flexibility and options of choosing the best configuration for a particular workload. Tuning of these parameters can also play a major role on achieving better performance in terms of job execution time. Section IV-C shows one of these examples where better parameter tuning achieves better benefits. On the other hand, we don't find the similar interfaces in Hadoop-A implementation.

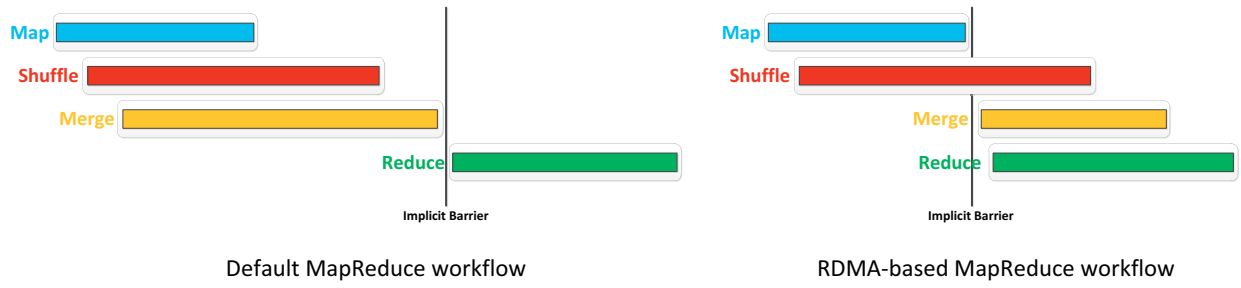


Figure 3. Overlapping of different processes in MapReduce workflow

IV. Performance Evaluation

In this section, we present the detailed performance evaluations of our RDMA-based design of Hadoop MapReduce and its impact on different Hadoop benchmarks. We compare the performance of our design with socket based interconnects (10 GigE and IPoIB) and Hadoop-A [1]. We have performed the experiments on different storage platforms (single/multiple HDDs or SSD per node) in order to illustrate the effect of I/O on our design.

In this study, we perform the following set of experiments: (1) Evaluation with the TeraSort benchmark and (2) Evaluation with the Sort benchmark. These benchmarks are described in section II. For each of the benchmarks, we have identified the optimal values of HDFS block-size for different interconnects as well as for Hadoop-A and our design. Additionally, in our experimental setup we have also determined that four is the maximum number of map and reduce tasks that can be run simultaneously to achieve the optimal performance by a TaskTracker.

In all our experiments, we have used Hadoop 0.20.2, Hadoop-A and JDK 1.7.0.

A. Experimental Setup

We have used an Intel Westmere cluster for our evaluations. This cluster consists of compute nodes with Intel Westmere series of processors using Xeon Dual quad-core processor nodes operating at 2.67 GHz with 12GB RAM and 160GB HDD. Each node is equipped with MT26428 QDR ConnectX HCAs (32 Gbps data rate) with PCI-Ex Gen2 interfaces. The nodes are interconnected using a Mellanox QDR switch. Each node runs Red Hat Enterprise Linux Server release 6.1 (Santiago) at kernel version 2.6.32-131 with OpenFabrics version 1.5.3. This cluster also has dedicated storage nodes with the same configuration, but with 24GB of RAM each. Additionally, eight of the storage nodes are equipped with two 1TB HDD each. Four of the storage nodes also have Chelsio T320 10GbE Dual Port Adapters with TCP Offload capabilities.

In the figures presented in this section, we have mentioned OSU-IB to indicate our RDMA-based design of MapReduce and Hadoop-A to indicate the design in [1]. 32 Gbps indicates InfiniBand QDR card speed.

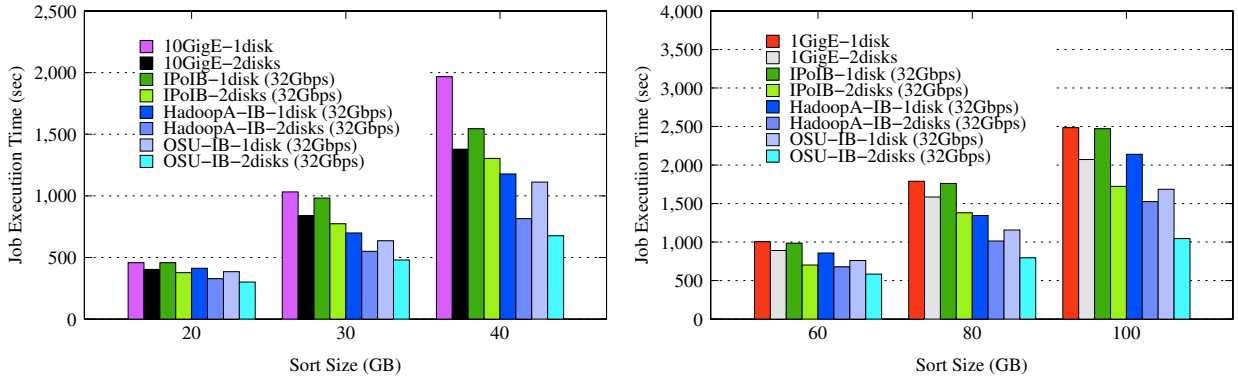
B. Evaluation with the TeraSort Benchmark

We have performed this experiment in 10 GigE, IPoIB (32 Gbps) with vanilla Hadoop and Hadoop-A and compared the results with our design. For this experiment, we have found that the optimal HDFS block-size for 10 GigE, IPoIB (32 Gbps), and our design is 256 MB, whereas it is 128 MB for Hadoop-A. We have used TeraGen to generate the input data for TeraSort.

Figure 4 shows the job execution times of the TeraSort benchmark in a four-DataNode cluster. In the experiments for Figure 4(a), we show performance results with single and dual HDDs for each interconnect. For single HDD, 30 GB sort size, our design reduces the job execution time by 9% over Hadoop-A (32 Gbps), 35% over IPoIB (32 Gbps) and 38% over 10 GigE. Compared with IPoIB and 10 GigE, our design uses native IB verbs communication for the data shuffle, which is much better than the socket based communication on IPoIB and 10 GigE. Although Hadoop-A also uses native IB verbs communication, the data prefetching and caching mechanism of our design improves the performance.

On the other hand, if two HDDs are used per node, our design improves the execution time by 13% over Hadoop-A (32 Gbps), 38% over IPoIB (32 Gbps), and 43% over 10 GigE for the same sort size. For 40 GB sort size, our design achieves an improvement of 17%, 48%, and 51% over Hadoop-A (32 Gbps), IPoIB (32 Gbps), and 10 GigE, respectively. When multiple HDDs are used per node, the performance bottleneck of the local disk read and write bandwidth is alleviated. Compared to Hadoop-A, our design can utilize the improved bandwidth more efficiently to overlap the data shuffle, merge, and reduce further as illustrated in section III-B.

We have performed similar experiments with the TeraSort benchmark using eight DataNodes. In this case, we varied the sort size from 60 GB to 100 GB. As shown in Figure 4(b), our design reduces the job execution time 21% over Hadoop-A (32 Gbps) for 100 GB sort size with single



(a) Total job execution times in 4 nodes cluster

(b) Total job execution times in 8 nodes cluster

Figure 4. TeraSort benchmark evaluation

HDD. From Figure 4(b), we observe that, with two HDDs per node, our design achieves an improvement of 31% over Hadoop-A (32 Gbps).

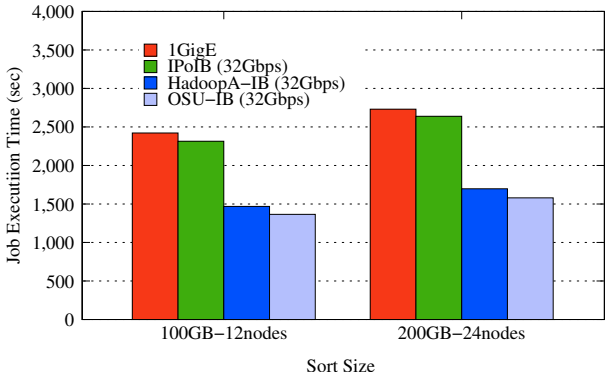


Figure 5. TeraSort benchmark evaluation with larger sort size

We have also evaluated TeraSort with larger sized cluster. In this case, we have varied the sort size from 100 GB to 200 GB with 12 and 24 compute nodes in the cluster, respectively. Figure 5 shows these results. For 100 GB sort size, we achieve 41% benefit over IPoIB (32Gbps) and 7% benefit over Hadoop-A (32Gbps). For 200 GB sort size also, we achieve similar benefits.

From Figure 4 and Figure 5, we observe that Hadoop-A performs better with respect to IPoIB in a bigger cluster with the same sort size, whereas, our implementation achieves better performance in both cases compared to Hadoop-A. As in our setup, storage nodes have twice as much memory as compute nodes, our implementation has more benefits in storage nodes compared to those in compute nodes. This clearly depicts the efficiency of the caching mechanism implemented in our design.

C. Evaluation with the Sort Benchmark

We perform the regular Sort benchmark in 1 GigE, IPoIB (32 Gbps) with vanilla Hadoop and Hadoop-A and compare the results with our design. For this experiment, the optimal value of HDFS block-size is 64 MB. We have used RandomWriter to generate the input data for the Sort benchmark.

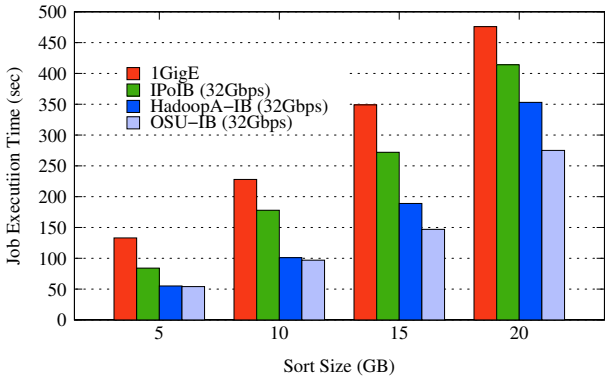
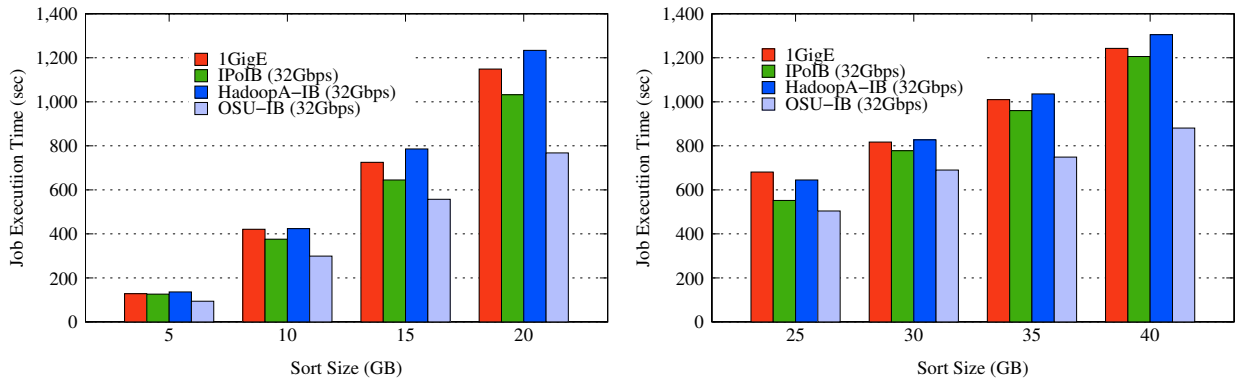


Figure 7. Sort benchmark evaluation with SSD

Figure 6(a) shows the performance results of the Sort benchmark using four DataNodes. For this we have used four compute nodes in our cluster. Each DataNode has single HDD per node. In this case, our design reduces the job execution time by 26% over IPoIB (32 Gbps) and 38% over Hadoop-A for 20 GB sort. From Figure 6(b) we observe that, with eight DataNodes, our design can achieve an improvement of 27% over IPoIB (32 Gbps) and 32% over Hadoop-A. Compared with the TeraSort benchmark, the difference in the Sort benchmark is the variable size of the key-value pairs. In Sort, the combined length of key-value pairs can be as large as 20,000 bytes. From the results, we can observe that Hadoop-A performs worse than IPoIB even after configuring all the tunable parameters with optimum values as mentioned in Hadoop-A release [22]. It reveals that only substituting the socket based communication with the native IB verbs, some applications such as the Sort benchmark cannot get better performance due to the inefficiency in number of key-value pairs transferred each time that also affects proper overlapping between all the stages. Our design with the efficient caching mechanism can get better performance in



(a) Total job execution times in 4 nodes cluster

(b) Total job execution times in 8 nodes cluster

Figure 6. Sort benchmark evaluation

both these cases as it considers the size of the key-value pair before the transfer.

We also evaluate Sort benchmark using SSD as HDFS data stores. Figure 7 shows the comparison for this evaluation. In this case, our design achieves a benefit of 22% over Hadoop-A (32Gbps) and 46% over IPoIB (32Gbps) in job execution time for 15 GB sort.

D. Benefits of Caching

In our design, we have implemented efficient map output pre-fetching and caching mechanism. We have also provided a configuration parameter to enable/disable the caching. In this experiment, we have evaluated the performance improvement that we can get through caching enabled.

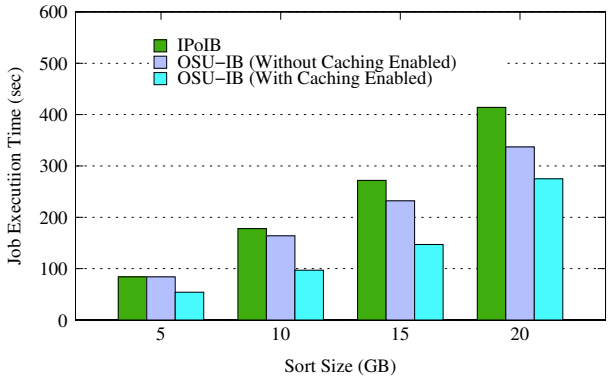


Figure 8. Effect of Caching Mechanism

Figure 8 shows the performance comparison between caching enabled and caching disabled in our RDMA-based design for Sort benchmark. We perform this experiment using SSD as HDFS data store. In this case, enabling caching with our design can enhance performance by 18.39% over caching disabled in the same design for 20 GB sort size. It reveals that for big workload as in sort, an efficient caching mechanism can significantly improve the performance for an RDMA-based design using a high performance interconnect.

V. Related Work

Many studies have paid attention to improve the performance of MapReduce in recent years. As mentioned before, the most analogous one is Hadoop-A [1], which provides a new merge method to Hadoop MapReduce framework by utilizing RDMA over InfiniBand. Our work has some major enhancements and differences with respect to their work along pre-fetching, caching, codebase modification, etc. We have discussed these detail in section III-C.

In [23], the authors have proposed techniques of pre-fetching and pre-shuffling into MapReduce. From their results, the techniques can improve the overall performance of Hadoop. We have focused on implementing an efficient key-value pairs pre-fetching and caching mechanism inside TaskTracker, which is responsible for fast data shuffle when the mappers are yet to be completed. Such a design can help reduce the overhead in the reducer side when the shuffle and merge procedures run in an overlapped manner. The research [24] has demonstrated that there is an impressive space for performance improvement in Hadoop MapReduce compared with traditional HPC technologies, such as MPI. Our earlier work [7, 10] revealed that SSD can reduce the I/O cost and make the overheads involved in data-transmission over the network prominent. In this paper, we have conducted experiments with multiple HDDs and SSDs per node to lessen the I/O bottleneck when studying the effect of communication over different interconnects.

VI. Conclusion

In this paper, we have presented an RDMA-based design of MapReduce over InfiniBand. We have also proposed efficient pre-fetching and caching mechanisms for retrieving of the intermediate data. Our performance evaluation shows that we achieve 21% benefit in terms of execution time over Hadoop-A for 100 GB TeraSort. For regular Sort benchmark, our design outperforms Hadoop-A by 32% for 40 GB sort. We also observe an improvement of 22% over Hadoop-A for the same sort size using SSD. In future, we plan to extend

our design to handle faster recovery in case of task failures. We will also evaluate our design on larger clusters with a range of applications.

REFERENCES

[1] 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*, ser. SC '11, 2011.

[2] Apache Hadoop, http://hadoop.apache.org/.

[3] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *Operating Systems Design and Implementation (OSDI)*, 2004.

[4] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, 2010.

[5] J. Appavoo, A. Waterland, D. Da Silva, V. Uhlig, B. Rosenburg, E. Van Hensbergen, J. Stoess, R. Wisniewski, and U. Steinberg, "Providing A Cloud Network Infrastructure on A Supercomputer," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC '10. New York, NY, USA: ACM, 2010, pp. 385–394.

[6] Greenplum Analytics Workbench, http://www.greenplum.com/news/greenplum-analytics-workbench.

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

[8] 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 *International Conference on Parallel Processing (ICPP)*, Sept 2011.

[9] 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 *IEEE International Parallel and Distributed Processing Symposium (IPDPS'12)*.

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

[11] J. Jose, M. Luo, S. Sur, and D. K. Panda, "Unifying UPC and MPI Runtimes: Experience with MVAPICH," in *Fourth Conference on Partitioned Global Address Space Programming Model (PGAS)*, Oct 2010.

[12] Hadoop Map Reduce, "The Apache Hadoop Project," http://hadoop.apache.org/mapreduce/.

[13] Sort, http://wiki.apache.org/hadoop/Sort.

[14] RandomWriter, http://wiki.apache.org/hadoop/RandomWriter.

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

[16] OpenFabrics Alliance, http://www.openfabrics.org/.

[17] P. Balaji, H. V. Shah, and D. K. Panda, "Sockets vs RDMA Interface over 10-Gigabit Networks: An In-depth analysis of the Memory Traffic Bottleneck," in *Workshop on Remote Direct Memory Access (RDMA): Applications, Implementations, and Technologies (RAIT), in conjunction with IEEE Cluster*, 2004.

[18] RDMA Consortium, "Architectural Specifications for RDMA over TCP/IP," http://www.rdmaconsortium.org/.

[19] B. Fitzpatrick, "Distributed Caching with Memcached," *Linux Journal*, vol. 2004, pp. 5–, August 2004. [Online]. Available: http://portal.acm.org/citation.cfm?id=1012889.1012894

[20] Apache HBase, "The Apache Hadoop Project," http://hbase.apache.org/.

[21] MVAPICH2: MPI over InfiniBand, 10GigE/iWARP and RoCE, http://mvapich.cse.ohio-state.edu/.

[22] Mellanox Technologies, "Unstructured Data Accelerator," http://www.mellanox.com/page/products_dyn?product_family=144.

[23] 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 Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, Sep. 2009, pp. 1–8.

[24] X. Lu, B. Wang, L. Zha, and Z. Xu, "Can MPI Benefit Hadoop and MapReduce Applications?" in *IEEE 40th International Conference on Parallel Processing Workshops (ICPPW)*, 2011.