On Exploring Efficient Shuffle Design for In-Memory MapReduce

Harunobu Daikoku
University of Tsukuba
daikoku
@hpcs.cs.tsukuba.ac.jp

Hideyuki Kawashima
University of Tsukuba
kawasima
@cs.tsukuba.ac.jp

Osamu Tatebe
University of Tsukuba
tatebe
@cs.tsukuba.ac.jp

ABSTRACT

MapReduce is commonly used as a way of big data analysis in many fields. Shuffling, the inter-node data exchange phase of MapReduce, has been reported as the major bottleneck of the framework. Acceleration of shuffling has been studied in literature, and we raise two questions in this paper. The first question pertains to the effect of Remote Direct Memory Access (RDMA) on the performance of shuffling. RDMA enables one machine to read and write data on the local memory of another and has been known to be an efficient data transfer mechanism. Does the pure use of RDMA affect the performance of shuffling? The second question is the data transfer algorithm to use. There are two types of shuffling algorithms for the conventional MapReduce implementations: *Fully-Connected* and more sophisticated algorithms such as *Pairwise*. Does the data transfer algorithm affect the performance of shuffling? To answer these questions, we designed and implemented yet another MapReduce system from scratch in C/C++ to gain the maximum performance and to reserve design flexibility. For the first question, we compared RDMA shuffling based on *rsocket* with the one based on IPoIB. The results of experiments with GroupBy showed that RDMA accelerates *map+shuffle* phase by around 50%. For the second question, we first compared our in-memory system with Apache Spark to investigate whether our system performed more efficiently than the existing system. Our system demonstrated performance improvement by a factor of 3.04 on Word Count, and by a factor of 2.64 on BiGram Count as compared to Spark. Then, we compared the two data exchange algorithms, Fully-Connected and Pairwise. The results of experiments with BiGram Count showed that Fully-Connected without RDMA was 13% more efficient than Pairwise with RDMA. We conclude that it is necessary to overlap map and shuffle phases to gain performance improvement. The reason of the relatively small percentage of improvement can be attributed to the time-consuming insertions of key-value pairs into the hash-map in the map phase.

BeyondMR'16, June 26-July 01 2016, San Francisco, CA, USA

DOI: http://dx.doi.org/10.1145/2926534.2926538

CCS Concepts

•**Information systems** → **MapReduce-based systems;**

Keywords

MapReduce; Shuffle; RDMA; Pairwise, Fully-Connected

1. INTRODUCTION

1.1 Background

MapReduce is commonly used as a way of big data analysis in many fields. All that the users have to do is to write two routines, called *map* and *reduce*. The system then distributes tasks across computation nodes taking into consideration the data locality. These tasks can easily be scaled out as they are independent of each other. It is widely known that MapReduce can be applied to SQL [3] and sophisticated algorithms of machine learning, such as Logistic Regression, Naive Bayes, and K-means [8, 29].

Shuffling, the inter-node data exchange stage of MapReduce, has been reported as the major bottleneck of the framework in most cases. A number of studies have addressed this issue and have tried to improve the performance of shuffling by either revising the existing implementations [11, 24, 17, 13], or by introducing new shuffling schemes [15, 1, 23]. Since the proposal of MapReduce in 2004 [12], a variety of implementations have been developed. On the basis of the underlying hardware systems, these implementations can be classified into two classes.

The first class is for the cloud computing infrastructure constructed by conventional machines and network systems. This class of implementations is currently major in MapReduce because popular open-source implementations, such as Apache Hadoop [2] and Apache Spark [4] are available on a variety of cloud services. According to a survey conducted by Databricks Inc., one of the major contributors to Spark, nearly 50% of Spark users are running it on public clouds, which are composed of such hardware [10]. These machines and networks are affordable although they do not provide high reliability and performance. Hence, some kind of recovery mechanism for partial failure is crucial. To maintain availability, the results of mappers are often spilled to disk before the shuffle phase begins.

The second class is designed for the supercomputing infrastructure constructed from the finest machines and network systems. Use of shared-memory multiprocessor [21, 7], GPU [14, 6], Xeon Phi [16], and InfiniBand [17, 20, 18] are attempts to accelerate MapReduce with such excellent

hardware. This class is currently not as popular in MapReduce because the usage of supercomputers is limited to researchers in certain fields, such as genome, meteorology, and cosmology. Such hardware is expensive while it provides high performance and reliability. The shuffle phase in these systems does not need to spill out the results of mappers to disk because machines and network rarely fail in supercomputing systems.

1.2 Problem

There is no doubt that, as the market prices go down, the expensive hardware of today will be affordable and common in the future. It seems likely that InfiniBand, the high performance interconnect, will be employed in the majority of computing clusters in the near future.

InfiniBand has been adopted in some conventional systems to accelerate performance. MapReduce-MPI Library (MR-MPI) [20] and K MapReduce (KMR) [18] build new systems that provide all of the map, shuffle, and reduce phases using MPI (Message Passing Interface) for shuffling. Another approach is to attach a pluggable shuffling module to the conventional MapReduce systems. Both the Hadoop MapReduce [25, 26] and the Spark MapReduce [17] are accelerated by this approach.

Although these conventional studies introduce notable performance improvement, none of these have been built from the ground up. Hence, we argue that the following two questions still remain for the optimal design of shuffling.

Q1: Pure Effect of RDMA. RDMA has been widely adopted by supercomputing systems for its high efficiency. However, it is not clear how major improvement can be achieved by RDMA itself in the shuffle phase.

HOMR [25] is a pluggable shuffling module for Hadoop MapReduce. It accelerates shuffling with multiple approaches. In addition to the use of RDMA, it also presents a scheme that overlaps the different phases. Similarly, a pluggable shuffling module tailored for Spark is introduced in [17]. It replaces the original module in Spark with the proposed one. The work reports that the proposed module is more efficient than the conventional Spark with IPoIB (IP over InfiniBand) that emulates IP communication over InfiniBand network.

These pluggable module approaches are effective for practitioners because they accelerate original Spark and Hadoop systems. For exploring optimal design, however, these do not largely contribute to clarify the achieved performance improvement provided by the pure use of RDMA. Besides, invocation of RDMA requires Java Native Interface (JNI) that should generate some overhead but has not been evaluated in those studies. Does the pure use of RDMA affect the performance of shuffling?

Q2: Appropriate Data Exchange Algorithm. Spark assumes Ethernet as the network, and high speed interconnect such as InfiniBand is out of scope. This is because such hardware is expensive and not adopted in major cloud infrastructures. The communication algorithm in the shuffle phase is the **Fully-Connected** algorithm. In the Fully-Connected algorithm, each process establishes connections to the others and fetches data from them, and it requires $(n-1) \times n$ links in total.

InfiniBand is the basic interconnect and MPI is the standard communication framework in the field of supercomputing. In MPI-based implementations, such as MR-MPI [20] and KMR [18], the data exchange algorithm in the shuffle phase is mainly based on the all-to-all communication scheme. One of the all-to-all communication algorithms is **Pairwise**, which has been exploited in this work. The Pairwise algorithm achieves all-to-all communication in a series of steps. In each step, processes form pairs and exchange data with each other. Thus, the Pairwise algorithm is valid when there are power-of-two number of processes. In general, the Pairwise algorithm requires $n-1$ steps when there are n processes.

The Fully-Connected algorithm requires a larger number of connections than the Pairwise. Overlapping of different execution phases for the Fully-Connected can be implemented more efficiently than the Pairwise. However, a detailed comparison of Pairwise and Fully-Connected algorithms in the shuffle phase has not been analyzed in literature. Does the difference of data transfer algorithm affect the performance of shuffling?

1.3 Contribution

To address these questions, we design and implement yet another in-memory MapReduce engine from scratch. We have published our source code on Bitbucket [9].

To address Q1, we first measured the performances of communication methods on InfiniBand, including RDMA with *rsocket* API, and evaluated the pure effect of RDMA on shuffling. We used the **GroupBy** workload that groups input words by their lengths and is known to be shuffle-heavy [17]. Through the breakdown, we show that RDMA demonstrates higher performance than that of IPoIB in this workload.

To address Q2, we integrated both the data exchange algorithms into our system, and compared them with each other. We evaluated the algorithms by using two types of workloads: **Word Count** and **BiGram Count**. The Word Count workload counts the number of occurrences of each word from a text file. The BiGram Count workload counts the number of occurrences of each bigram, which is a pair of two words, and thereby tends to be more shuffle-heavy as compared to Word Count. We first compared the performance of our system with that of the Spark 1.5 that adopts Project Tungsten [27], and showed that Spark degrades in larger size of datasets because of disk I/Os invoked by shuffle read and write. Further, we compared the performances of Pairwise and Fully-Connected, and showed that Fully-Connected is the winner in terms of performance.

It should be noted that this paper focuses on shuffling performance, and recovery system of MapReduce is beyond the scope of this work.

1.4 Organization

This paper is structured as follows: Section 2 introduces related work. We explain conventional MapReduce systems and attempts that accelerate shuffling. In Section 3, we present an overview of our approaches to realize an efficient in-memory MapReduce system, followed by its design and implementation in Section 4. Section 5 describes the results of the evaluations we conducted to address the two questions. Finally, concluding remarks and future work are described in Section 6.

2. RELATED WORK

To accelerate shuffling, there are two approaches. The first approach is to design the entire MapReduce system, and the second approach is to attach a pluggable module to a conventional MapReduce system. We describe them in the following two subsections.

2.1 Designing The Entire System

2.1.1 Cloud Computing Field

The very first implementation by Google was written in C++ [12]. Open-source implementations, such as Java-based Apache Hadoop [2] and Scala-based Apache Spark [4], are popular and widely used in the industrial world. Hadoop is an on-disk engine, that is, it writes out the results of the map and the reduce stages onto disk, while Spark holds them in memory. Spark, therefore, is superior to Hadoop, especially on iterative workloads such as K-means. Spark further accelerates its performance by designing its data access CPU cache conscious in Project Tungsten [27]. Both systems have some solid recovery mechanisms since they are designed for commodity hardware configurations where machines often fail during executions.

2.1.2 Supercomputing Field

In the supercomputing field, MPI is the standard communication library. There are two well-known MapReduce engines that exploit MPI to implement the shuffle phase.

MapReduce-MPI Library (MR-MPI) [20] is an open-source C/C++ MapReduce library which exploits similarity between the MapReduce framework and MPI. It does not only implement MPI-based MapReduce engine, but also has some unique features which distinguish itself from the original MapReduce framework. One example is that users can call MPI collective communication routines inside map and reduce procedures. This allows users to write more flexible data processing routines.

K MapReduce (KMR) [18] is an MPI-based MapReduce implementation optimized for K supercomputer that has 82,944 computing nodes and some unique components, such as file systems, interconnect, and network topology. For file accesses, it is well optimized for the particular file system of K. K has two file systems: *Global-FS* and *Local-FS*. The former is accessible from all the nodes and the latter is managed by the I/O nodes dedicated to every 2 racks. To overcome heavy loads on the I/O nodes caused by concentration of accesses, KMR proposes a clever method, in which only one mapper reads from Local-FS and distributes it among others using the collective communications of MPI.

MR-MPI and KMR are considered to have two weak points. First, they are not robust enough since there is no provision for a mechanism to ensure fault tolerance. This is due to the most current implementations of MPI, in which all the processes die if one of them fails. However, processes on supercomputers rarely fail, and therefore this is not a severe problem. Second, they do not overlap map and shuffle phases since synchronization is required before shuffling begins.

2.2 Designing Pluggable Shuffle Modules

The designing the entire system approach enables us to fully optimize the performance, with absolute flexibility in terms of design. However, it cannot directly contribute to the existing projects which already have thousands of users, such as Hadoop and Spark. By adopting pluggable module design approach, such widely used systems can be accelerated, without losing their usability.

Work in [17] implements RDMA Shuffle Server and Client as plug-ins of Apache Spark, and compares them with Spark running on 1/10 Gig Ethernet and IPoIB. They use JNI to invoke RDMA calls from Spark, which runs on JVM. They also design some additional mechanisms, such as off-heap buffer management and connection pooling, to minimize overhead. They evaluate their implementation by running *groupByKey*, which is one of Spark RDD's operators, and the results show that it is 80% faster than the original Spark. However, since the network topology is still Fully-Connected, it is a concern that, as the number of nodes grows, network can be a bottleneck. What is more, they do not refer to the random disk I/Os before and after shuffling, which is known to be the biggest problem of Spark's shuffle stage [11].

3. APPROACH

Shuffling is known to be the major bottleneck of MapReduce. To overcome it, we exploit high speed interconnect, InfiniBand. For the use of InfiniBand, there are two parameters that should be considered. The first parameter is data transfer mechanism, and the second is data exchange algorithm. Our choices for those parameters are described below.

3.1 Data Transfer Mechanism

InfiniBand is a network transport standard that enables much higher throughput and lower latency than traditional Ethernet. It is now adopted as interconnect on nearly 50% of supercomputers listed in TOP500 [22]. InfiniBand supports RDMA, which permits a machine directly read from and write on the local memory of another machine without involving CPUs. RDMA-enabled network adapters directly send and receive data on application buffers, and thereby can omit the costs of context switches and copying data from application buffers to OS buffers. Although users need to use dedicated APIs to enjoy the maximum performance of RDMA, InfiniBand also provides a service called IPoIB, which is fairly faster than ordinary Ethernet. Users can benefit from faster communication of IPoIB by just specifying the address assigned to InfiniBand card, without any modifications on their codes.

Figure 1 shows the measurement results we collected on the cluster used for evaluations on Section 5. We measured Ping-Pong bandwidths on 1 Gig Ethernet, IPoIB, and RDMA implemented with rsocket API. The graph indicates that, as the message size grows bigger, the difference between RDMA and the others becomes greater. The bandwidth of RDMA where the message size is 1 MB is about 46x higher than that of Ethernet. IPoIB is also faster than Ethernet according to our measurements, but still it is necessary to utilize the native RDMA APIs for the maximum performance of InfiniBand. From this result, we adopt RDMA with rsocket to accelerate the shuffle phase.

3.2 Data Exchange Algorithm

For data exchange algorithms, we tested two types of algorithms in this paper: *Fully-Connected* and *Pairwise*. This is because Spark and Hadoop adopt plain Fully-connected, while MR-MPI and KMR use more sophisticated ones such

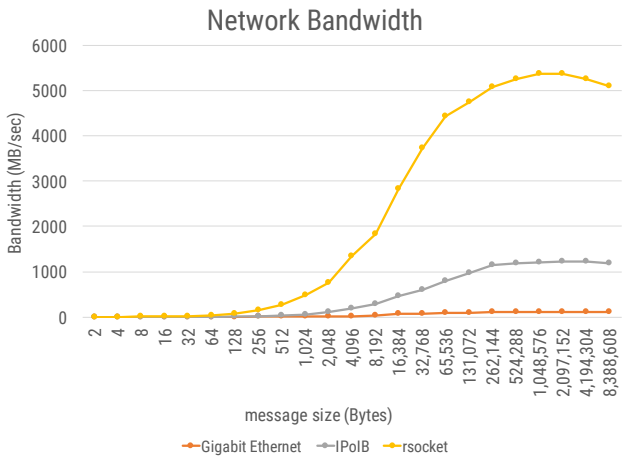

Figure 1: Network Bandwidth of 1Gig Ethernet, IPoIB, and RDMA

as Pairwise, and it is not yet clear which algorithm is more appropriate for the shuffling. The next two subsections explain these two algorithms in detail.

3.2.1 *Fully-Connected*

In the Fully-Connected algorithm, each process establishes connections to the others and fetches data from them. While it requires $(\#process - 1) \times \#process$ links in total, it can be easily adapted to the shuffling technique that overlaps map stage with shuffling, as each communication happens independently. The existing MapReduce engines, such as Hadoop and Spark, employ this method for shuffling.

3.2.2 *Pairwise*

The Pairwise algorithm is one of MPI's all-to-all communication algorithms. In each step of this algorithm, processes form pairs and exchange data with each other. Thus, Pairwise is only valid when there are power-of-two number of processes, though, with a slight modification, it can be utilized for non power-of-two number of processes. Pairwise is also known to be effective for longer messages and suited for RDMA since it directly transfers data without any proxy nodes. *MPICH*, one of the most popular implementations of MPI, employs Pairwise algorithm for power-of-two number of processes and long messages [19].

Figure 2 shows a working example of Pairwise algorithm. Initially, each node holds 4 blocks, that is, 0, 1, 2, and 3. Through steps, a set of blocks sharing the same number is put into the same process. In general, Pairwise algorithm requires $n-1$ steps when there are n processes. For example, in the case of Figure 2, the algorithm completes in three steps since there are 4 processes. In step 1, the adjacent processes form pairs and exchange data, in step 2, pairs are made alternately, and in step 3, the remaining patterns of pair are formed. These steps can be generalized as follows: *In step k, a process whose rank is i forms a pair with process (i XOR k).* The algorithm can be slightly modified for non power-of-two number of processes as follows: *In step k, a process whose rank is i sends data to process $(i+k)\%n$, and receives from process $(i-k)\%n$.*

3.2.3 *Comparison*

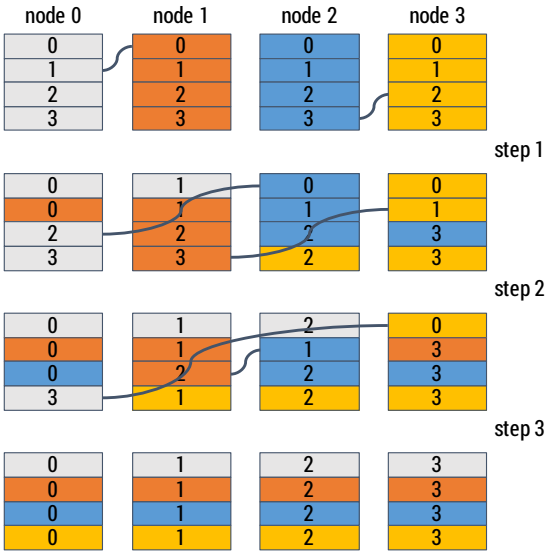

Figure 2: Pairwise All-to-All Algorithm

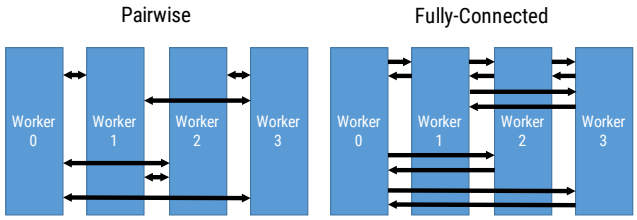

Figure 3: Overview of links generated by Pairwise and Fully-Connected algorithms

One of the advantages of the Pairwise algorithm is that it requires the minimum number of links to achieve all-to-all communication. The number of links established by Pairwise is $(\#process - 1) \times (\#process/2)$ as it makes $n/2$ links in each step. Fully-Connected, on the other hand, requires twice as many links as Pairwise. Figure 3 presents an overview of links generated by Pairwise and Fully-Connected. Since there are 4 processes, Pairwise only needs $(4-1) \times (4/2) = 6$ links, while Fully-Connected requires $(4-1) \times 4 = 12$.

Meanwhile, as the number of all-to-all communication executions increases, the cost of the Pairwise algorithm grows rapidly. Most current MapReduce engines, as mentioned earlier, overlap map and shuffle stages. If we try to implement the same mechanism by using the Pairwise algorithm, we must repeat the series of steps again and again. This solution is not efficient, since it requires all the $n-1$ steps even when there is no data to exchange, and results in exhaustion of CPU and network resources required by connection establishments. A more sophisticated solution is needed to achieve resource-friendly shuffling overlapped by map computation.

4. IN-MEMORY MAPREDUCE

We introduce a prototype of our new in-memory MapRe-

duce engine in this section. We designed and implemented this system for an experimental tool to study the effect of our approaches. Attaching a shuffling module to Spark itself is another important approach since it can accelerate the product, which already has thousands of users. On the other hand, our approach of developing another MapReduce system from scratch, enables us to measure the detailed behavior of MapReduce, as well as to test our ideas without being limited by the existing framework. To examine the effect of data exchange algorithms, we insist that modifying Spark itself is not adequate since, as presented in [11], the major factor responsible for the performance of the shuffling in Spark is disk I/Os, and not network. The source code of our implementation is available on Bitbucket [9].

4.1 Design

The design of our system was inspired by Resilient Distributed Datasets (RDD) [28] - the very core technology of Apache Spark that abstracts datasets spread across computing nodes. Once an RDD is distributed among nodes, it is further split into partitions on each node. When an operator is called on RDD, the system schedules one task per partition, and does parallel execution on each node according to the parallelism set by user. MapReduce in Spark is defined as operators on RDD, namely, *map* and *reduce(ByKey)*. Below is a list of some other characteristics of RDD.

immutable: Once defined, RDDs cannot be updated. What users can do is to apply operators on an existing RDD and generate a new one. By repeating these procedures, they can achieve their intended processing.

lazy evaluation: Some operators on RDD are not executed at the time they are called. This feature gives the job scheduler a chance to organize an optimum execution plan. Operators that are executed at the time of calls are referred to as *actions*, while the rest are called *transformations*.

DAG scheduler: The process of transformations of RDDs are tracked by Directed Acyclic Graph (DAG), which is utilized by the job scheduler as well. The benefit of DAG scheduling is that when some partitions are lost on failure during execution, the system can efficiently recover by going up the DAG, identifying parent partitions, and recomputing only missing ones.

At the moment, our implementation inherits only immutableness of RDD.

Figure 4 shows an overview of our implementation. More detailed information of the implementation is shown in Appendix A. The next subsection describes Master and Worker programs in detail.

4.2 Master and Worker

4.2.1 Master

The RPC Client module handles communications between master and worker nodes. It tries to connect to worker nodes listed in the configuration file specified in a runtime argument. If users provide a text file as an input data to MapReduce, it is indexed based on the partition size (default: 128 MiB). RPC Client then sends indices to workers through RPC. On worker nodes, the worker instances open the file and perform actual reads according to received indices.

The RDDStub module defines the available operators, such as *Map* and *Reduce*, as stubs. It also manages IDs

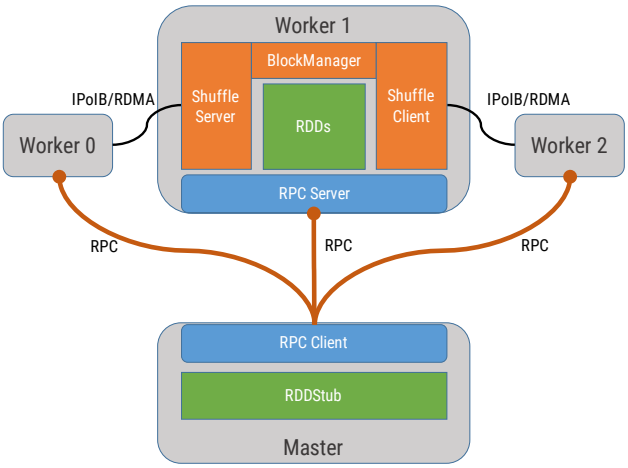

Figure 4: Overview of Our MapReduce System

of the worker nodes holding the corresponding RDD. When user applications call stubs defined in this module, actual operators are called on worker nodes through RPC.

4.2.2 Worker

The RPC Server module receives RPCs from the master node and calls operators on specified RDDs.

The RDD module implements some part of Spark's RDD. We offer two types of RDDs: *KeyValueRDD* and *KeyValuesRDD*. The former holds one value per key and defines *Map* operator, while the latter holds several values per key and defines *Reduce* and *Combine* operators. The actual *Map* and *Reduce* routines implemented by users are compiled and modularized as shared libraries. Users must place those binaries on some kind of global file system such as NFS, and make them accessible with the same path on every worker node. Those routines are dynamically linked on runtime and multi-threaded by the Intel®TBB library. As many threads as the number of CPU cores are executed concurrently.

The Block Manager module holds intermediate data before and after shuffling. It internally uses thread-safe queues provided by the Intel®TBB library to handle simultaneous pushes and pops from several threads. Before shuffling begins, key-values pairs managed by RDDs are serialized using MessagePack library and pushed into this module. The target queue is specified by the value which is the remainder from the division of the hash value of key by the number of worker nodes. Shuffle Server and Client that are described below pop serialized data from Block Manager, send it to the destination, and push received data into queues. Afterwards, reducers pop data from this module, deserialize them, and pass them to the reduce routine.

4.3 Shuffle Server/Client

4.3.1 Fully-Connected

The Fully-Connected Server and Client have almost identical structures as those of Spark. Before the map stage begins, each worker program starts one server thread and one client thread. The client sends fetch requests to servers on the other worker nodes. On reception of a request from the client, the server pops blocks from Block Manager and sends them to the client.

Table 1: Cluster Configuration

# of nodes	32
OS	CentOS release 6.5 (Final)
Kernel	Linux 2.6.32-431.el6.x86_64
Apache Hadoop	2.6.2
Apache Spark	1.5.2 (Standalone Deploy Mode)
CPU	Intel(R) Xeon(R) CPU E5-2650 v3 @ 2.30GHz x 2
# of CPU cores	10 x 2
Memory	64 GB
InfiniBand	Mellanox Technologies MT27500 Family [ConnectX-3]

After all map tasks have been finished, the Block Manager is set to *FINALIZED* state. The server thread then sends a completion signal to a client if Block Manager is *FINALIZED* and the queue for the client is empty. After having sent signals to all the clients, the server thread exits. On the client side, the thread finishes when it has received completion signals from all the servers.

4.3.2 Pairwise

In Pairwise, every worker node starts Shuffle Server or Client based on RPC message from master node in the first place. It then connects to the partner node, exchanges data with each other, and notifies master node about its completion. After having received completion messages from all the pairs, master node proceeds to the next step.

We implement two types of Pairwise shuffling: the one based on BSD socket API and the one based on *rsocket* API provided by the *rdmacm* library. The former can invoke IPoIB communication by specifying address assigned to InfiniBand card.

The current Pairwise implementation starts shuffling after all mappers have finished their tasks, since, as mentioned in the previous section, overlapping Pairwise shuffling ends up in consuming computing resources. Fully-Connected, on the other hand, overlaps map and shuffle by default.

5. EVALUATION

To evaluate the proposed MapReduce engine, we executed three types of workloads on the following four systems: Apache Spark (with IPoIB), Prototype (Fully-Connected with IPoIB), Prototype (Pairwise with IPoIB), and Prototype (Pairwise with RDMA). The three types of workloads that were executed included GroupBy for Q1, Word Count for Q2, and BiGram Count for Q2. The configuration of the cluster we used for these evaluations is shown in Table 1. Although recovery issues are beyond the scope of this paper, we did not observe any failures during this evaluation.

5.1 GroupBy for Q1

5.1.1 Overview

To provide the evidence to support the superiority of RDMA over IPoIB, we performed GroupBy workload on our 2 prototypes, namely, Pairwise (RDMA) and Pairwise (IPoIB). The workload groups words written in a text file by their lengths, and thus the execution times are ruled by the performance of shuffling. This is because the workload shuffles the entire input data and the map computation of the workload is relatively light. We measured execution times of 2, 4, 8, 16, and 32 GiB of text files. Those files were generated using "RandomTextWriter" [5] that is distributed in Hadoop package as an example program. For 2 GiB, we used only 16 out of 32 nodes since all these systems we evaluated generate 1 Mapper task for 128 MB of input data, and therefore there are only 16 tasks. The Mapper code we implemented is shown in Figure 5.

```
void Map(unordered_map<int, vector<string>> &kvs,
         const long long int &key,
         const string &val) {
  size_t cur = 0, pos;
  while ((pos = val.find_first_of(' ', cur))
         != val.npos) {
    auto word = val.substr(cur, pos - cur);
    kvs[word.length()].push_back(word);
    cur = pos + 1;
  }
  auto word = val.substr(cur, val.length() - cur);
  kvs[word.length()].push_back(word);
}
```

Figure 5: Mapper code for GroupBy

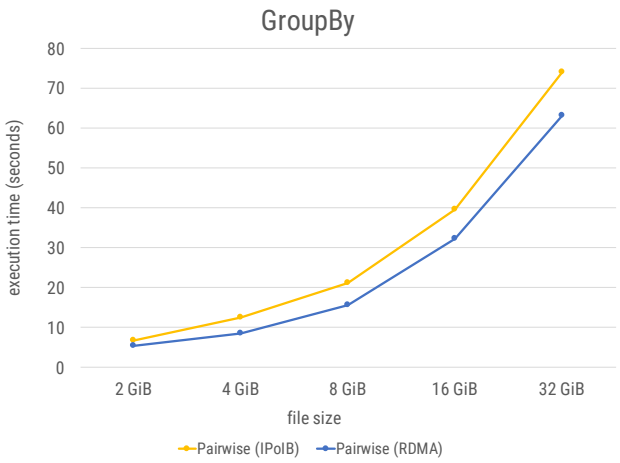

Figure 6: Execution times of GroupBy

5.1.2 Results

Figure 6 shows that, throughout the graph, RDMA outperforms IPoIB. On 32 GiB, RDMA-based shuffling is about 20% (10 sec) faster than that of IPoIB-based shuffling. This result is seemingly inconsistent with the result of bandwidth measurement shown in Figure 1 where RDMA demonstrates more than 3x higher throughput than IPoIB.

To understand the reason, we measured the breakdown of 32 GiB execution, and it is shown in Figure 7. In the figure, we can see that the 10 second performance improvement is achieved in *map+shuffle* phase. By optimizing the implementation and reducing the ratio of the reduce phase, the improvement in shuffling can be more effective.

5.2 Word Count for Q2

5.2.1 Overview

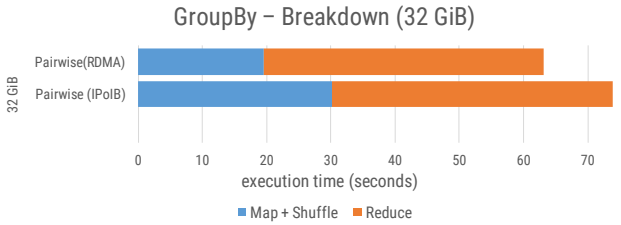

Figure 7: Breakdown of GroupBy execution time when file size is 32 GiB

```
void Map(unordered_map<string, vector<int>> &kvs,
         const long long int &key, const string &val) {
  size_t cur = 0, pos;
  while ((pos = val.find_first_of(' ', cur))
         != val.npos) {
    kvs[val.substr(cur, pos - cur)].emplace_back(1);
    cur = pos + 1;
  }
  kvs[val.substr(cur, val.size() - cur)].emplace_back(1);
}
```

Figure 8: Mapper Code for Word Count

This workload counts the number of occurrences of each word appearing in a text file. For each system, we adopted *Combiner*, which partially reduces the output from Mappers before shuffling.

We measured execution times of 2, 4, 8, 16, 32, 64, 128, 256, and 512 GiB of text files. The Mapper and the Reducer (Combiner) codes we implemented for our own system are shown in Figure 8 and 9, respectively.

5.2.2 Results

Figure 10 shows the execution times of Word Count on each system. As can be seen, the bigger the file size, the faster our three prototypes are, as compared to Spark. For 32 GiB, Pairwise (RDMA) is 2.35x faster than Spark, and for 512 GiB, it is 3.04x faster. This is because our system does not perform disk access which is, as described in [11], the major bottleneck in Spark's shuffling performance.

As the lines of our three prototypes are overlapping with each other, we can see that there is not much difference between those systems on this workload. This is because of the characteristic of Word Count which Combiner reduces the output from Mapper significantly, and therefore the shuffling method does not make a great difference. In general, if there are m Mappers and k kinds of words in a text, Combiners reduce the output from Mappers to $m \times k \times$ ($pair\_of\_one\_string\_and\_one\_int$).

The breakdowns of the execution times for 2, 32, and 512 GiB are shown in Figure 11. We can see that Pairwise (RDMA) is slower than Pairwise (IPoIB) on 2 and 32

```
pair<string, int>
Reduce(const string &key, const vector<int> &vals) {
  return make_pair(
    key, accumulate(vals.begin(), vals.end(), 0)
  );
}
```

Figure 9: Reducer Code for Word Count

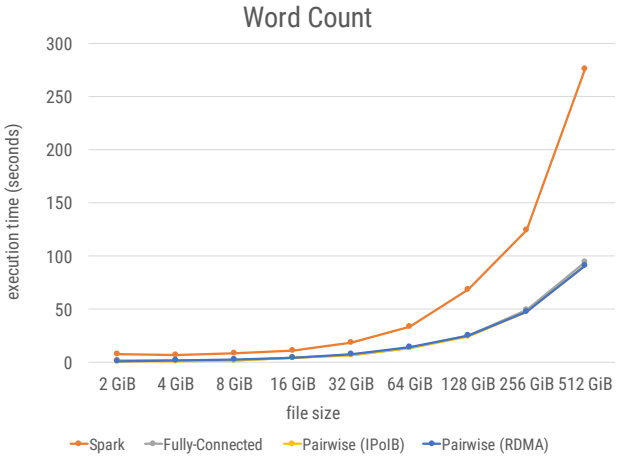

Figure 10: Execution times of Word Count

GiB, but RDMA excels on 512 GiB. This is assumed to be due to the size of data transferred per communication. The bandwidth measurements shown in Figure 1 indicate that when the message size is 2 KB, the difference is only 700 MB/sec, and when 70 KB, it is as great as 3,600 MB/sec. Thus, we can assume that when the file size is small, the improvement of bandwidth is not apparent due to the connection overhead required by RDMA.

5.3 BiGram Count for Q2

5.3.1 Overview

This workload counts the number of occurrences of each bigram, which is a pair of two words appearing in a text file. The only difference between Word Count and this workload is the type of key. In Word Count, we use words themselves for keys, and in BiGram Count, we use pairs of words. This difference makes the workload more shuffle-heavy not only because the size of each key increases, but also because the effect of the Combiner drops as the number of keys increases and occurrences of each key decrease. In general, if there are m Mappers and k kinds of words in a text, the variety of keys is at most $k \times k$. Therefore, Combiners reduce the output from Mappers to $m \times k \times k \times$ ($pair\_of\_two\_words\_and\_one\_int$). We also used only 16 nodes for 2 GiB on this workload as well.

The Mapper and the Reducer (Combiner) codes implemented for our own engine are shown in Figure 12 and 13, respectively.

5.3.2 Results

Figure 14 shows the execution times of BiGram Count on each system. On this workload, due to the shuffle-heavy property, we can see the difference between our three prototypes more clearly. In most cases, Pairwise (RDMA) is faster than Pairwise (IPoIB) and Fully-Connected is the fastest. Yet, the difference between Pairwise and Fully-Connected is modest. One of the advantages of Pairwise, as previously described, is that it can achieve all-to-all communication with only half as many connections as Fully-Connected. For this evaluation, since we only used 32 nodes, the number of connections should not have a definitive influence on the

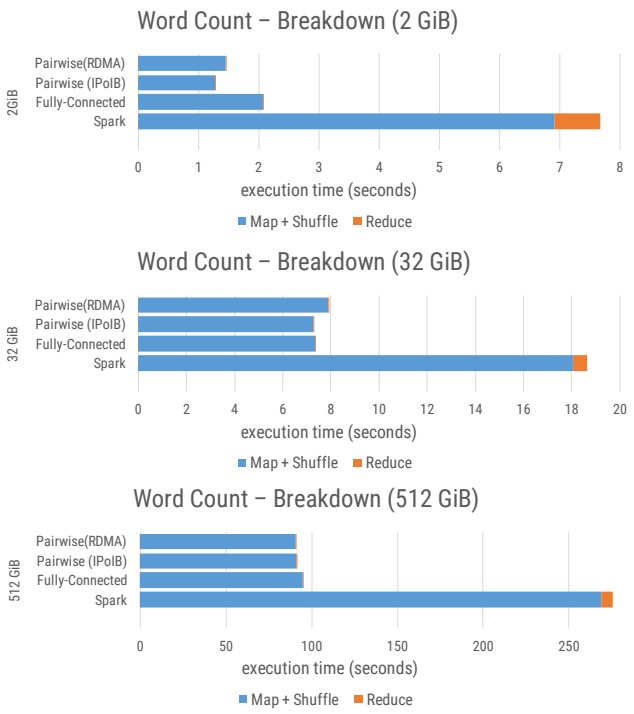

Figure 11: Breakdown of Word Count execution times when file sizes are 2, 32, and 512 GiB

performance. However, when there are thousands of nodes communicating at the same time, Pairwise may gain potential advantage over Fully-Connected. We need to further investigate how each method performs under such conditions.

Our three prototypes outperform Spark, except for 32 and 64 GiB. For 512 GiB, Pairwise (RDMA) is 2.64x faster than Spark. Also, note that the performance of Spark deteriorates when the file size is 512 GiB, while that does not apply to our systems. We discuss this phenomenon in Section 5.3.3.

The breakdowns of the execution times for 2, 32, and 512 GiB are presented in Figure 15. The results show that, for 512 GiB, map and shuffle phase of Pairwise (RDMA) is about 6% faster than that of Pairwise (IPoIB), while the difference was less than 1% on Word Count. This indicates that we can effectively utilize RDMA when the workload

```
void Map(unordered_map<string, vector<int>> &kvs,
         const long long int &key, const string &val) {
  size_t cur = 0, p1, p2 = 0;
  p1 = val.find_first_of(' ', cur);
  while (true) {
    p2 = val.find_first_of(' ', p1 + 1);
    if (p2 == val.npos) {
      kvs[val.substr(cur, val.size() - cur)]
      .emplace_back(1);
      break;
    }
    kvs[val.substr(cur, p2 - cur)].emplace_back(1);
    cur = p1 + 1;
    p1 = p2;
  }
}
```

Figure 12: Mapper code for BiGram Count

```
pair<string, int> Reduce(const string &key,
                         const vector<int> &vals) {
  return make_pair(
    key, accumulate(vals.begin(), vals.end(), 0)
  );
}
```

Figure 13: Reducer code for BiGram Count

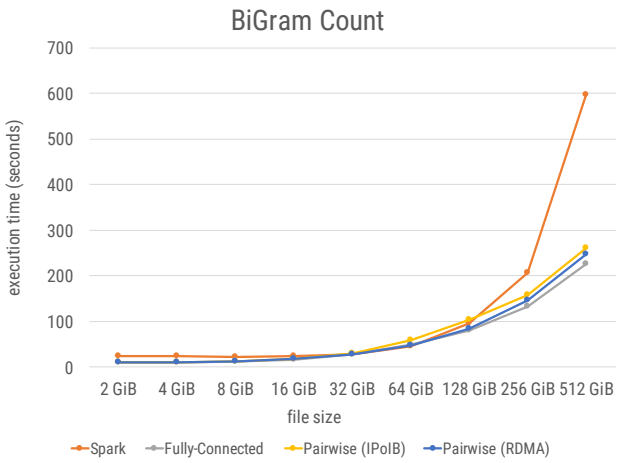

Figure 14: Execution times of BiGram Count

is shuffle-heavy and there is enough amount of data to exchange. In addition, Fully-Connected further improves the performance by 13% as compared to Pairwise (RDMA). This result verifies that overlapping map and shuffling phase is effective when shuffling phase is relatively heavy.

5.3.3 *Discussion for Small Improvement*

It appears that the 6% performance improvement of RDMA over IPoIB is relatively small compared to the 3,600 MB/sec bandwidth difference between the two. This is because the map phase occupies nearly 80% of the *map+shuffle* execution time. Table 2 shows a part of the profiling data we collected from our prototype system using Intel®VTune™ Amplifier. The results show that the top five CPU consuming spots of the process, and it is insertions of key-value pairs into the hash map that tops the list.

As can be seen from the 512 GiB data in Figure 15, the

Table 2: Profiling data of the prototype collected using Intel®VTune™(extracted)

Function	Module	CPU Time
std::unordered_map <...>::operator[]	libBiGramMapper.so	41.356s
operator new	libstdc++.so.6	21.335s
_int_free	libc.so.6	14.313s
std::__detail:: _Hashtable_base <...>:: _M_equals	libBiGramMapper.so	13.736s
std::char_traits <char>::compare	RDDSlave	8.667s

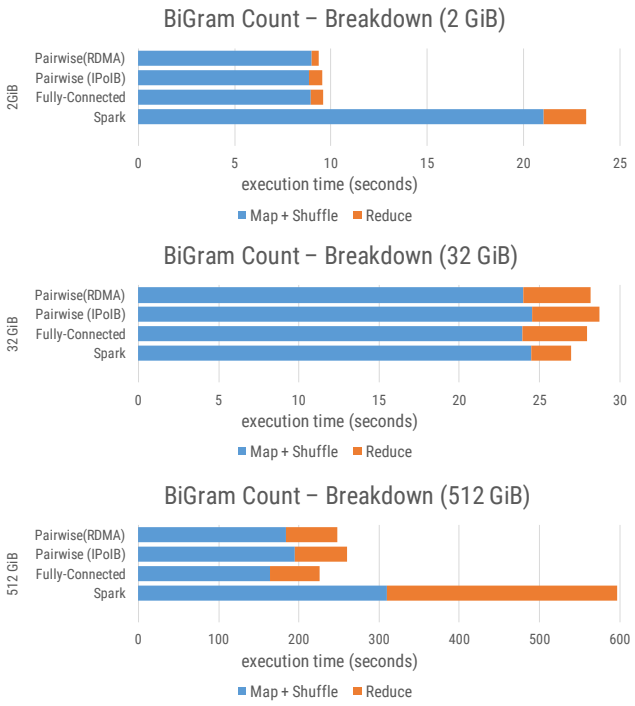

Figure 15: Breakdown of BiGram execution times when file sizes are 2, 32, and 512 GiB

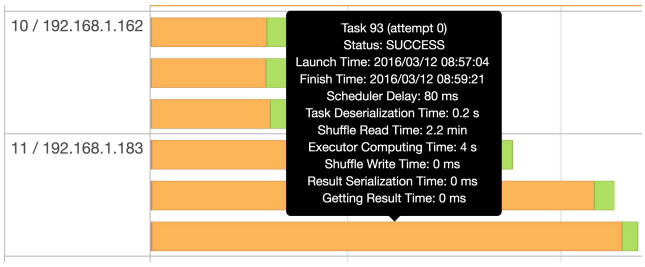

Figure 16: Breakdown of Spark's reduceByKey execution time when the file size is 512 GiB

reduce phase of Spark accounts for nearly 50% of the execution time. Figure 16, which visualizes the execution logs of Spark's reduce phase, implies that it spends most of the reduce phase (2.2 minutes) on *Shuffle Read*. This is the phase the system reads shuffled data from disk. This explains the significant performance degradation shown in Figure 14. Our prototypes, on the other hand, do not show this tendency because they do not access to disk when shuffling. Thus, our proposed system can enjoy a greater advantage over Spark as the data size gets larger.

6. CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

Acceleration of shuffling in MapReduce has been addressed in the concerned literature because it is the major performance bottleneck. In this paper, we raise two questions with the issue. The first question is the pure effect of RDMA, and the second question pertains to the use of the appropriate data exchange algorithm.

To answer these questions, we designed and implemented yet another in-memory MapReduce system that includes shuffling, in C/C++. For the first question, we compared RDMA shuffling based on *rsocket* with the shuffling based on IPoIB. The results of experiments with GroupBy workload showed that RDMA accelerates *map+shuffle* phase by around 50%.

For the second question, we first investigated if the avoidance of spill can benefit performance of our new in-memory MapReduce system. Our system demonstrated performance improvement by a factor of 3.04 on Word Count workload, and by a factor of 2.64 on BiGram Count. Then we compared the two data exchange algorithms, Fully-Connected and Pairwise. The results of experiments with BiGram Count showed that Fully-Connected without RDMA is 13% more efficient than Pairwise with RDMA. Therefore, it is necessary to overlap map and shuffle phases for gaining performance improvement, and Fully-Connected is more appropriate than Pairwise as the data exchange algorithm.

6.2 Future Work

Simply executing Pairwise shuffling in the background is inefficient since it requires $n-1$ steps every time data is exchanged between n nodes. These steps are executed even when there is no data to transfer and can end up with consuming precious CPU resources. Thus, we need to investigate a better shuffle algorithm which is RDMA-aware and can be efficiently overlapped by map computation. Since RDMA is more CPU-efficient than the ordinary Ethernet communication, we can expect performance improvement not just in the shuffle phase but also in the map phase.

Acknowledgements

This work is partially supported by JST CREST "System Software for Post Petascale Data Intenseive Science" and JST CREST "Extreme Big Data (EBD) Next Generation Big Data Infrastructure Technologies Towards Yottabyte / Year". It is also supported by JSPS KAKENHI 25280043HA and JST CREST "Statistical Computational Comsmology with Big Astronomical Imaging Data".

7. REFERENCES

[1] F. Ahmad, S. Lee, M. Thottethodi, and T. Vijaykumar. Mapreduce with communication overlap (marco). *Journal of Parallel and Distributed Computing*, 73(5):608–620, 2013.

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

[3] Apache. Apache hive. http://hive.apache.org/.

[4] Apache. Apache spark. http://spark.apache.org/.

[5] Apache. Randomtextwriter.java. https://github.com/apache/hadoop/blob/trunk/hadoop-mapreduce-project/hadoop-mapreduce-examples/src/main/java/org/apache/hadoop/examples/RandomTextWriter.java.

[6] L. Chen and G. Agrawal. Optimizing mapreduce for gpus with effective shared memory usage. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '12, pages 199–210, New York, NY, USA, 2012. ACM.

[7] R. Chen and H. Chen. Tiled-mapreduce: Efficient and flexible mapreduce processing on multicore with tiling. *ACM Trans. Archit. Code Optim.*, 10(1):3:1–3:30, Apr. 2013.

[8] C. Chu, S. K. Kim, Y. Lin, Y. Yu, G. Bradski, A. Y. Ng, and K. Olukotun. Map-reduce for machine learning on multicore. In *Annual Conference on Neural Information Processing Systems*, pages 281–288. MIT Press, 2006.

[9] H. Daikoku. Source codes of the prototype implementation. https://bitbucket.org/hdaikoku/rdd.

[10] Databricks. Spark survey 2015 infographic. http://cdn2.hubspot.net/hubfs/438089/DataBricks_Surveys_-_Content/Spark-Survey-2015-Infographic.pdf?t=1443057549926.

[11] A. Davidson and A. Or. Optimizing shuffle performance in spark. *University of California, Berkeley-Department of Electrical Engineering and Computer Sciences, Tech. Rep*, 2013.

[12] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[13] Y. Guo, J. Rao, and X. Zhou. ishuffle: Improving hadoop performance with shuffle-on-write. In *International Conference on Autonomic Computing*, pages 107–117. USENIX, 2013.

[14] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: a mapreduce framework on graphics processors. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 260–269. ACM, 2008.

[15] M. Lin, L. Zhang, A. Wierman, and J. Tan. Joint optimization of overlapping phases in mapreduce. *Performance Evaluation*, 70(10):720–735, 2013.

[16] M. Lu, Y. Liang, H. P. Huynh, Z. Ong, B. He, and R. S. M. Goh. Mrphi: An optimized mapreduce framework on intel xeon phi coprocessors. *IEEE Transactions on Parallel and Distributed. Systems*, 26(11):3066–3078, 2015.

[17] X. Lu, M. W. Rahman, N. Islam, D. Shankar, and D. K. Panda. Accelerating spark with rdma for big data processing: Early experiences. In *Annual Symposium on High-Performance Interconnects (HOTI)*, pages 9–16. IEEE, 2014.

[18] M. Matsuda, N. Maruyama, and S. Takizawa. K mapreduce: A scalable tool for data-processing and search/ensemble applications on large-scale supercomputers. In *International Conference on Cluster Computing*, pages 23–27. IEEE, 2013.

[19] MPICH. alltoall.c. https://trac.mpich.org/projects/mpich/browser/src/mpi/coll/alltoall.c.

[20] S. J. Plimpton and K. D. Devine. Mapreduce in mpi for large-scale graph algorithms. *Parallel Computing*, 37(9):610–632, 2011.

[21] J. Talbot, R. M. Yoo, and C. Kozyrakis. Phoenix++: modular mapreduce for shared-memory systems. In *Proceedings of the second international workshop on MapReduce and its applications*, pages 9–16. ACM, 2011.

[22] TOP500.org. List statistics. http://www.top500.org/statistics/list/.

Table 3: Prototype Implementation

Language	C/C++
Lines of Code	2574
Tested Compilers	clang 7.0.2, gcc/g++ 5.0.2, Intel®Compiler 16.0.1
Dependencies	MessagePack 0.5.9, jubatus-mpio 0.4.5, jubatus-msgpack-rpc 0.4.4, Intel®TBB (Threading Building Blocks) 4.4-20150728, google-sparsehash 2.0.3, rdmacm 1.0.21mlnx-OFED.3.0.1.5.2

[23] J. Ullman. Mapreduce algorithms. In *Proceedings of the 2Nd IKDD Conference on Data Sciences*, CODS-IKDD '15, pages 1:1–1:1, New York, NY, USA, 2015. ACM.

[24] M. Wasi-ur-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 *International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, pages 1908–1917. IEEE, 2013.

[25] M. Wasi-ur-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 *International Conference on Supercomputing*, pages 33–42. ACM, 2014.

[26] M. Wasi-ur-Rahman, X. Lu, N. S. Islam, R. Rajachandrasekar, and D. K. Panda. High-performance design of yarn mapreduce on modern hpc clusters with lustre and rdma. In *International Conference on Supercomputing*, pages 291–300. IEEE, 2015.

[27] R. Xin and J. Rosen. Project tungsten: Bringing spark closer to bare metal. https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html. Accessed: 2016-01-21.

[28] 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 *Symposium on Networked Systems Design and Implementation*, pages 15–28. USENIX, 2012.

[29] W. Zhao, H. Ma, and Q. He. Parallel k-means clustering based on mapreduce. In *International Conference on Cloud computing*, pages 674–679. Springer, 2009.

APPENDIX

A. IMPLEMENTATION DETAILS

Table 3 shows detailed information about our implementation described in Section 4.

The source code of our implementation is available on Bitbucket [9].