2015 IEEE 29th International Parallel and Distributed Processing Symposium

High-Performance Design of YARN MapReduce on Modern HPC Clusters with Lustre and RDMA*

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

***Abstract*—The viability and benefits of running MapReduce over modern High Performance Computing (HPC) clusters, with high performance interconnects and parallel file systems, have attracted much attention in recent times due to its uniqueness of solving data analytics problems with a combination of Big Data and HPC technologies. Most HPC clusters follow the traditional *Beowulf* architecture with a separate parallel storage system (e.g. Lustre) and either no, or very limited, local storage. Since the MapReduce architecture relies heavily on the availability of local storage media, the Lustre-based global storage system in HPC clusters poses many new opportunities and challenges. In this paper, we propose a novel high-performance design for running YARN MapReduce on such HPC clusters by utilizing Lustre as the storage provider for intermediate data. We identify two different shuffle strategies, RDMA and Lustre Read, for this architecture and provide modules to dynamically detect the best strategy for a given scenario. Our results indicate that due to the performance characteristics of the underlying Lustre setup, one shuffle strategy may outperform another in different HPC environments, and our dynamic detection mechanism can deliver best performance based on the performance characteristics obtained during runtime of job execution. Through this design, we can achieve 44% performance benefit for shuffle-intensive workloads in leadership-class HPC systems. To the best of our knowledge, this is the first attempt to exploit performance characteristics of alternate shuffle strategies for YARN MapReduce with Lustre and RDMA.**

I. Introduction

Big Data and High Performance Computing are two disruptive technologies that are converging to meet the challenges exposed by large-scale data processing and storage. According to recent IDC reports [1, 2], 67% of HPC centers say that they perform jobs that can be categorized as High Performance Data Analysis (HPDA) workloads. The revenues of these workloads are expected to grow extensively, increasing from $743.8 million in 2012 to almost $1.4 billion in 2017. Additionally, the storage revenue for high performance data analysis on HPC systems will near almost a billion by 2017, IDC says. Hadoop MapReduce [3] is increasingly being used on modern HPC clusters [4, 5] to process HPDA workloads because of its scalability and fault tolerance.

The benefits of running Hadoop MapReduce over modern HPC clusters with high performance interconnects (e.g. InfiniBand) and parallel file systems (e.g. Lustre) have attracted much attention in recent studies [6–13] due to its uniqueness of solving Big Data processing problems with a combination of Big Data and HPC technologies.

A. Motivation

The Hadoop MapReduce framework typically runs over the Hadoop Distributed File System (HDFS) [14], and takes advantage of multiple local disks on compute nodes to achieve better data-locality. However, most modern HPC clusters [4, 5] tend to follow the traditional *Beowulf* architecture [15, 16] model, where the compute nodes are provisioned with a lightweight operating system, and either a disk-less or a limited-capacity local storage [17]. At the same time, they are connected to a sub-cluster of dedicated I/O nodes with enhanced parallel file systems, such as Lustre, which can provide fast and scalable data storage solutions. Figure 1 shows an example of how Lustre is deployed on most modern HPC clusters and how YARN MapReduce jobs are running over it.

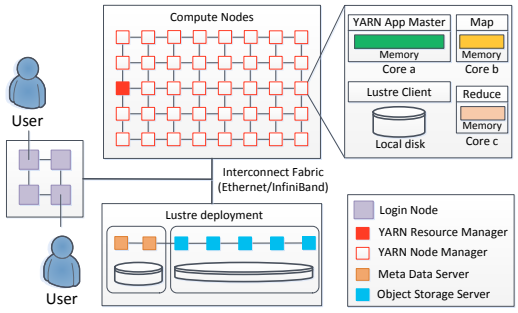


Figure 1: YARN MapReduce running over a typical Lustre setup on modern HPC clusters

This architecture is not naively conducive for default MapReduce because these clusters usually have small-capacity local disks and this prevents default MapReduce jobs with large-scale data sizes to run. Table I shows the usable storage comparison between local disks and Lustre on modern HPC clusters (TACC Stampede [4] and SDSC Gordon [5]) and it clearly proves the above-mentioned scenario. Such a contradiction leads to sub-optimal performance for default MapReduce running on HPC clusters. Recent studies [9, 11] have also demonstrated that default Hadoop MapReduce can not efficiently leverage large capacity parallel file systems (e.g. Lustre) on modern HPC clusters. This leads us

*This research is supported in part by National Science Foundation grants #CCF-1213084 and #IIS-1447804. It used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number OCI-1053575.

1530-2075/15 $31.00 © 2015 IEEE
DOI 10.1109/IPDPS.2015.83

Table I: Storage Capacity Comparison on Typical HPC Clusters

HPC Cluster	Usable Local Disk Capacity	Usable Lustre Capacity	Total Lustre Capacity
TACC Stampede [4]	≈ 80 GB	≈ 7.5 PB	≈ 14 PB
SDSC Gordon [5]	≈ 300 GB	≈ 1.6 PB	≈ 4 PB

Table II: Existing Performance Studies on MapReduce (MR)

	Apache MR	RDMA MR
Apache HDFS	[3, 14]	[7, 13, 18]
RDMA HDFS	[6, 19]	[20]
Lustre with local storage	[9, 21, 22]	[11] (this paper)
Lustre w/o local storage	[23]	N/A (this paper)

to the question: ***Can Lustre-based parallel storage system be used as local storage to run MapReduce jobs efficiently on modern HPC clusters?***

For most of the high-performance Lustre installations on HPC clusters, using Lustre as local storage for MapReduce jobs should be straight-forward since this can be completed only in two steps of file system write and read, which is intuitively seen as a high-speed data shuffle path because of high-throughput read/write operations of Lustre. However, the transmission time inside Lustre (among the Object Storage Servers) depends on many factors such as cluster interconnect, workload variation, etc. These factors may introduce read and write overhead in Lustre for a MapReduce job running with many concurrent maps and reduces, each of them reading from or writing to the file system. On the other hand, high-performance interconnects, such as InfiniBand and 10/40 Gigabit Ethernet, are commonly used for data movement amongst the compute nodes on modern HPC clusters. They can provide extremely low latency and high bandwidth. For example, the latest InfiniBand network provides around 1 μs for point-to-point data transfer latency and up to 100 Gbps bandwidth. Such a high-speed data movement path can also be utilized for data shuffling from mappers to reducers as an alternative strategy where a significantly less number of processes read from Lustre, thereby reduces the contention. Recent studies [7, 13, 18] have proposed enhanced MapReduce designs (e.g. HOMR [13]) to leverage RDMA-enabled high-performance interconnects to significantly speedup MapReduce job performance. This further motivates us to answer the question: ***Can YARN MapReduce be redesigned to fully utilize the benefits of both Lustre (large capacity) and RDMA (ultra performance) in a combined manner?***

To enhance performance of YARN MapReduce over Lustre, we need to choose a higher-throughput shuffle strategy, which leads us to the following broad design challenges:

1) Can alternate shuffle strategies be designed for RDMA-enhanced YARN MapReduce over Lustre while keeping the existing architecture and APIs intact?
2) How to further optimize the design for different shuffle strategies by exploiting the performance characteristics of numerous Lustre installations on HPC clusters?
3) Can we dynamically adapt our design to select the best strategy based on run-time performance behavior?
4) How much performance benefits can be achieved by using different shuffle strategies for YARN MapReduce jobs in leadership-class HPC clusters?

B. Contributions

Table II summarizes the existing research in the literature on different combinations of MapReduce designs with different file systems. As shown in the table, default and RDMA-enhanced MapReduce designs have been well-evaluated with default and RDMA-enhanced HDFS designs. Default Apache MapReduce over Lustre with local storage has also been well studied. A more recent study [23] has examined default MapReduce performance over Lustre without local storage, while our earlier work [11] is the first attempt to exploit performance improvement by leveraging RDMA for MapReduce over Lustre with local disks used as intermediate storage. However, the benefits of RDMA-enhanced MapReduce with Lustre as intermediate storage provider, have not been discovered yet. Also, for this environment, enhanced shuffle algorithms need to be designed to achieve further benefits.

In this context, we propose a new YARN MapReduce design with different shuffle strategies, over Lustre, using the concepts of RDMA-enhanced HOMR [13] architecture. This paper mainly focuses on the architecture where the intermediate data for MapReduce jobs are stored in a Lustre file system or a combination of Lustre and local disks, rather than the default approach of using only the local file system for this purpose. We also propose dynamic adaptation to choose one shuffle strategy over another to use all available resources in the most efficient manner. Our results show that both shuffle strategies have performance benefits while the RDMA-based shuffle approach always scales better compared to Lustre read-based shuffle approach. We achieve up to 44% performance benefit for shuffle-intensive MapReduce applications in leadership-class HPC systems.

The rest of the paper is organized as follows. Section II presents background for this paper. We propose our design with different shuffle strategies for enhanced MapReduce architecture in Section III. Section IV describes our detailed evaluation. Section V provides some related studies currently existing in the literature. We conclude in Section VI with possible future work.

II. Background

A. Hadoop/YARN MapReduce

Hadoop [3] is a popular open source implementation of the MapReduce [24] programming model. The Hadoop Distributed File System (HDFS) [3, 14] is the primary

storage for Hadoop cluster. For most Hadoop applications, it provides the storage for both input and output data. However, the intermediate data is kept on each node's local file system.

In a typical Hadoop-1.x cluster, the master, a.k.a JobTracker is responsible for accepting jobs from clients as well as job scheduling and resource management. Whereas, Hadoop-2.x improves on the scalability limitation by introducing separate node and resource managers. In Hadoop-2.x, YARN (Yet Another Resource Negotiator) [25] which is also known as MapReduce 2.0 (MRv2) decouples the resource management and scheduling functionality of the JobTracker in Hadoop-1.x. There is a global Resource Manager (RM) responsible for assigning resources to all the jobs running on the Hadoop cluster. The Node Manager (NM) is similar to the TaskTracker in the MapReduce v1 framework with one NM running per node. For each application, there is an Application Master (AM). The AM coordinates with the RM and NMs to execute the corresponding tasks.

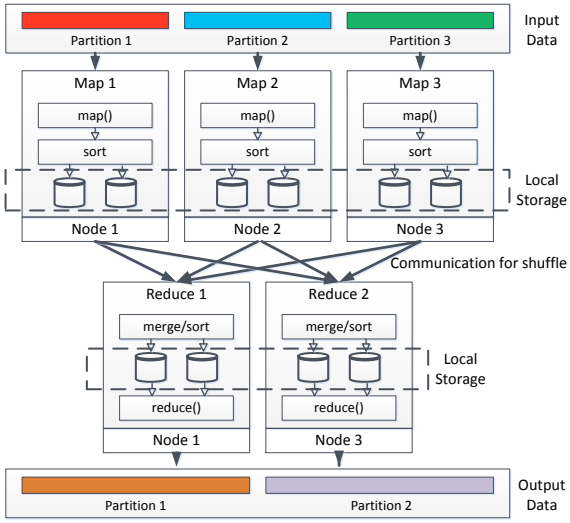


Figure 2: Default MapReduce architecture with local storage

A MapReduce job is divided into several map and reduce tasks. As shown in Figure 2, each map task is assigned a portion of the input file, called split or partition. Each map task reads its split from HDFS and applies the user-defined `map()` function on each key-value pair parsed from this data. Each map task then sorts the data locally and writes the data to the intermediate data directory. After intermediate data is generated and stored in the local file system of each node, reduce tasks start fetching this data from different map output files. This is known as shuffle phase. In YARN, separate service, such as ShuffleHandler exists that runs inside NM and sends this data to each reduce task through HTTP response messages via sockets. During the shuffle phase, a separate thread merges this data periodically and keeps the merged data to local disks. This is also illustrated in Figure 2. After all the data is merged, reduce tasks apply the user-defined `reduce()` function on the merged data and write the final output to the underlying file system.

B. HOMR over HDFS

Since the job execution flow over default Hadoop MapReduce involves many disk operations, it demonstrates performance bottlenecks [7]. Also, default MapReduce is not capable of taking full advantage of high performance interconnect features, such as RDMA. In this context, as our prior work, we propose HOMR [13], a Hybrid approach of obtaining maximum possible Overlapping in MapReduce. With RDMA/Socket-based shuffle engine, it offers the following features: (1) In-memory merge operation, (2) Overlapping among map, shuffle, and merge phases as well as shuffle, merge, and reduce phases, (3) Pre-fetching and caching of map outputs, and (4) Dynamic adjustment of data volume during shuffle. HOMR also provides these features to the default MapReduce architecture that helps to achieve significant performance benefits regardless of the underlying network protocols. However, HOMR with RDMA-based shuffle engine guarantees maximum performance benefits compared to the default architecture over sockets [13].

C. Lustre File System and Its Deployment on HPC Clusters

Lustre is a POSIX-compliant, stateful, object-based parallel file system that has been deployed on several large-scale supercomputing clusters and data centers. This file system has a scalable architecture with three primal components - Meta Data Server (MDS), Object Storage Server (OSS) and the clients that issue the I/O requests. To access a file, a client first obtains its meta data from the primary MDS, including file attributes, file permissions, and the layout of file objects in the form of Extended Attributes (EA). Subsequent file IO (storage) operations are performed directly between the client and the OSS.

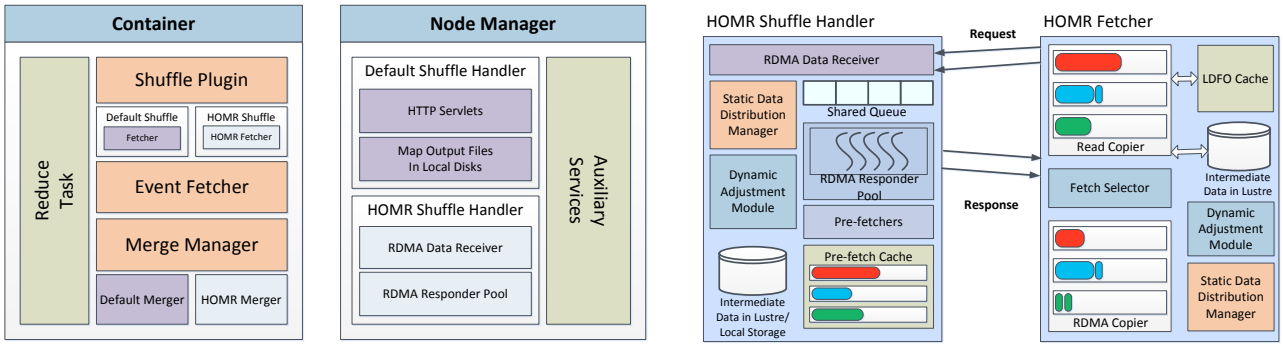
In this paper, we have used Lustre as both the intermediate data directory and the underlying file system to store input and output data for MapReduce applications. Using Lustre as the intermediate storage brings more data space compared to the local file systems used in HPC clusters. However, to ensure performance in this setup, better shuffle algorithms need to be designed to ensure efficient job execution.

III. RDMA-based YARN MapReduce over Lustre

In this section, we introduce the design of RDMA-based YARN MapReduce over Lustre. First, we discuss the architectural overview of our design. We introduce alternate shuffle strategies for this architecture and propose optimizations later.

A. Architectural Overview

Unlike the default MapReduce architecture, YARN MapReduce decouples many functionality in the framework. For example, the shuffle service in YARN is configured as a plug-in so that other shuffle implementations may work without much code changes. Although, HOMR not only enhances the shuffle service, but also introduces overlapping



(a) Architectural overview for HOMR in YARN

(b) HOMR with different shuffle strategies for Lustre

Figure 3: Design of YARN MapReduce over Lustre

among shuffle, merge, and reduce phases, we still keep the pluggable approach throughout our design for this architecture.

Figure 3(a) illustrates the architectural overview of HOMR in YARN. We introduce HOMRShuffle, a pluggable shuffle client that incorporates HOMRFetchers. HOMRFetcher provides the RDMA-based shuffle engine with enhanced shuffle algorithms. It consists of a Static Data Distribution Manager (SDDM) (shown in right side of Figure 3(b)) that assigns an initial set of static weights based on a greedy approach to bring as much data as possible in the beginning. However, as the merge and reduce progresses with shuffle, the Dynamic Adjustment Module may alter the weights for different map outputs to bring specific map output data more compared to others to make the merge and reduce phases progress faster.

Because of the in-memory approach, HOMR merge phase is maintained by HOMRMerger that is different from the default merge process. After performing an in-memory merge operation, it keeps evicting the already sorted key-value pairs and passes these to reduce task for performing user-defined `reduce()` operation on it. It ensures correctness by making sure that it does not evict any key-value pair that is not globally sorted. It also improves performance by identifying which map location output is more preferred for faster execution of the pipeline. To do so, it keeps track of the shuffled key-value pairs after each merge operation.

As opposed to TaskTrackers, YARN introduces Node Managers which act as servers in the shuffle phase. Like the default plug-in approach, we introduce HOMRShuffleHandler here. However, unlike the default ShuffleHandler, it can perform pre-fetching and caching also to provide fast shuffle service to the reduce tasks. The SDDM unit assigns weights here to decide how much data to be pre-fetched for any particular map output.

Running this architecture over parallel global file systems, such as Lustre, opens new possibilities for modified shuffle algorithms. We considered two different shuffle approaches, that can be selected by the Fetch Selector, as shown in Figure 3(b). Fetch Selector unit provides dynamic detection of which shuffle approach is faster and chooses appropriate copiers. The Dynamic Adjustment Module is responsible to provide functionality for dynamic adjustments of statically assigned weights as well as switching from one shuffle strategy to another.

B. Different Shuffle Strategies

In this section, we consider two shuffle strategies. As shown in Figure 4, each map stores the intermediate data to the underlying global file system after the completion of `map()` and local sort. To ensure this, Hadoop's temporary directory is configured with distinct paths in the global file system for each slave node. This ensures that each map stores its output in a separate temporary directory without conflicting with other map tasks' outputs. Optionally, the intermediate directory can also be configured by a list of global file system locations combined with local storage.

1) HOMR over Lustre with Read (HOMR-Lustre-Read): This shuffle strategy is based on Lustre file system read/write operations. Since the data generated from completed maps reside in the underlying global file system, direct read operations from the file system necessarily complete the shuffle process for the reduce task. To facilitate this, we introduce new read-based copier threads, similar to copiers in the default HOMR architecture. These readers can use the Lustre client in the local node to read a particular file residing in the underlying global file system. However, since HOMR employs an SDDM to assign weights for each completed map output, it maintains a limit of how much data a reader can read from a particular map output file. This ensures the availability of data from each completed map location during the merge process as well as minimizes the possibility of swapping out of memory.

In this shuffle strategy, a reader is designed in a way similar to the RDMA copier. Before starting the read operation, these copiers need to know the map output file location information in the Lustre since each slave in the cluster uses a separate and distinct temporary directory space for its intermediate data. Each reader thread thus initially uses

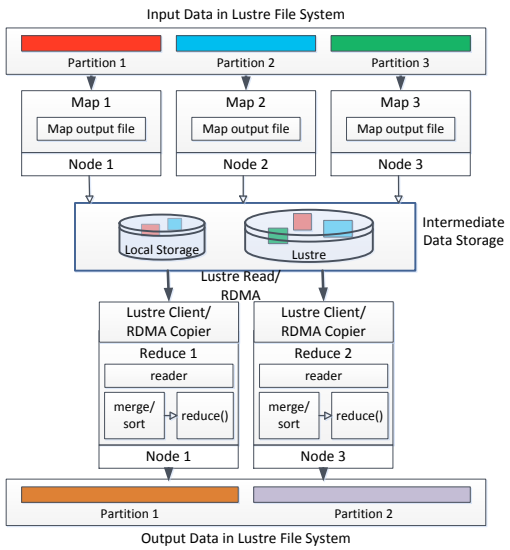


Figure 4: Shuffle strategies for MapReduce over Lustre

RDMA to communicate with the corresponding MapHost to get the file location information. HOMRShuffleHandler detects this type of request and sends a message containing the file location information. Since each reduce task reads the data from the global file system by itself, the pre-fetch and caching mechanism for these map outputs is kept disabled. However, to facilitate in-memory merge operation with overlapping of reduce phase, the reduce tasks read these files in a shuffle packet size granularity. To avoid multiple file location request-response messages, reduce tasks store the map output file location information in a Local Directory File Object (LDFO) cache. This cache, as shown in Figure 3(b), is responsible for keeping track of all map output file location information with an accounting of current read offsets.

2) HOMR over Lustre with RDMA (HOMR-Lustre-RDMA): This shuffle strategy is based on RDMA-based communication among the NodeManagers and the reduce containers. In this strategy, the SDDM assigns weights to different map outputs. The Greedy Shuffle Algorithm presented in [13] calculates these weights for different completed map output locations. As soon as the initial maps start to complete, SDDM assigns the weight of 1.0 so that it brings the entire data from these maps. It continues to do so until the total map data shuffled is close to the memory limit of the reduce task. At this point, it starts to reduce the weight for each map location following exponential back-off approach.[1] Although the map output files are stored in a global file system, reducers fetch these data as if they are residing in a local disk of mappers. Thus, it follows a similar approach to shuffling data from local disks. In this way, the number of reader processes (ShuffleHandlers in Node Managers) reading these files can be kept to a minimum. For fast response during shuffle, pre-fetching and caching of data is kept enabled.

[1]Exponential back-off refers to algorithm commonly used in network congestion, where a rate of repeated transmission of some data is decreased multiplicatively based on some feedback.

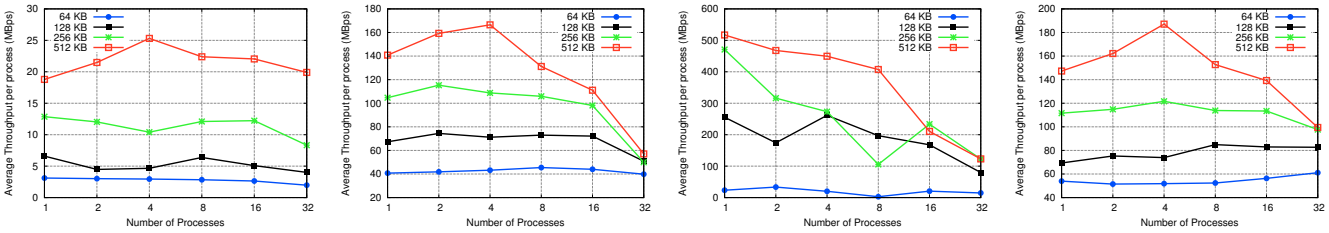
C. Optimization in Shuffle Strategies

Both the shuffle strategies discussed in Section III-B use SDDM to assign fractional weights for completed maps. Since this weight resembles the amount of data that will be shuffled on each request, it maintains the requirement of all data to reside in memory so that the merge operations do not have to spill intermediate merged data to disk. However, this in-memory requirement is also dependent on the shuffle packet size of data. For HOMR-Lustre-RDMA approach, we use the default packet size of 128 KB as shuffle packet size for each completed maps. However, these optimized packet sizes may not be ideal for HOMR-Lustre-Read approach. Since file system I/O operations are mostly optimized with large chunk of read or write operations, a higher packet size value may be appropriate here. However, for keeping the merge operations in-memory, a trade-off between different packet sizes is necessary.

For these reasons, we perform detailed experiments for optimizing the read and write packet sizes. We also choose the optimal number of maps/reduces for our experiments based on the Lustre read/write bandwidth per map/reduce. For this we have used the IOZone [26] benchmark. Figure 5 presents these experiments and the associated results.

To conduct these experiments, we varied the number of writer and reader threads on the compute nodes of two different clusters to write data to or read data from Lustre. The experimental setups for these clusters are described in Section IV-A. We collect average throughput per process for each of these experiments and optimize the number of threads based on these values. We choose this metric to ensure that all maps and reduces get the highest possible throughput while reading from or writing to Lustre. We configure each writer (reader) to write (read) 256 MB (stripe size used for Lustre) of data; we varied the record size from 64 KB to 512 KB. The reason for choosing this range is directly co-related with performing large chunk of I/O operations to minimize the total number of file system operations. Another reason is to keep all the shuffled data in-memory at any point to minimize the risks of getting out of memory while performing merge and reduce in an overlapping manner. Since all these clusters have large Lustre installations with numerous Object Storage Servers, we obtain the largest I/O bandwidth per process with the largest record size, 512 KB. This is evident from the read and write bandwidths presented in Figure 5.

From these experiments, we also select the number of concurrent maps/reduces for a particular MapReduce application. Each map performs sequential writes to its map output file, while several maps can run in parallel per node.



(a) Lustre write (Cluster A) (b) Lustre write (Cluster B) (c) Lustre read (Cluster A) (d) Lustre read (Cluster B)

Figure 5: Optimization in Lustre read and write threads

Thus, to introduce the exact scenario, we vary the number of threads in each of the IOZone experiments from 1 to 32 on each cluster. Each thread is writing a 256 MB file on Lustre with a varying record size from 64 KB to 512 KB. Figures 5(a) and 5(b) present the results from these experiments. For each record size, we observe that the number of threads that provide the highest average throughput per process varies significantly in Clusters A and B.

Similar experiments with read operation over Lustre are performed on the same clusters. These experiments illustrate the scenario of reduce tasks reading the map output files from Lustre for HOMR-Lustre-Read approach. For HOMR-Lustre-RDMA, these experiments resemble the number of threads reading from Lustre inside HOMRShuffleHandlers. For these experiments, with 512 KB record size, we observe a clear trend of decrease in average throughput per process for an increase in number of threads across all the clusters. Figures 5(c) and 5(d) illustrate these observations. This clearly indicates that with more readers per node, the average throughput per reducer decreases for HOMR-Lustre-Read scheme.

From these observations, we tune the read record size for Lustre to be equal to 512 KB in all the clusters. This ensures the Lustre I/O with higher granularity as well as maintaining the data from all map tasks in memory. We configure the concurrent map and reduce containers for each cluster to four so that it achieves the highest write throughput on Lustre. For reader threads in HOMR-Lustre-Read, we choose one. Four concurrent reducers per node essentially means that four readers would be running per node that will access Lustre at the same time. For Cluster B, this choice is obvious since with 4 reader threads per node the write throughput per process achieves the best result. However for Cluster A, even though single thread provides the best read throughput (Figure 5(c)) with 512 KB record size, we choose four concurrent reducers to have the best write throughput (Figure 5(a)) from both maps and reducers as well as obtain more task-level parallelism.

D. Dynamic Adaptation

Both the optimized shuffle strategies discussed in Section III-B employ a static way to choose between RDMA and Lustre Read. However, it may not be an intelligent solution to specifically choose one over the other for all submitted jobs in a typical HPC cluster. For example, if several jobs are running concurrently, the Lustre read and write throughput may vary significantly. Also, depending on the Lustre read size granularity, each read operation may provide variable performance. Similarly, if all jobs are trying to utilize RDMA for data shuffle, it may saturate the network bandwidth very fast. We validate one of these claims in Figure 6. Here, we ran a small TeraSort experiment in Cluster C with a data size of 10 GB. We conduct two sets of experiments. In one experiment, we ran a single TeraSort benchmark in the cluster to have exclusive access on Lustre read. However, in the other experiment, we use eight other jobs running and accessing the Lustre concurrently while TeraSort is running. We simulated this scenario with IOZone running multiple processes where each of them reading from and writing to Lustre.

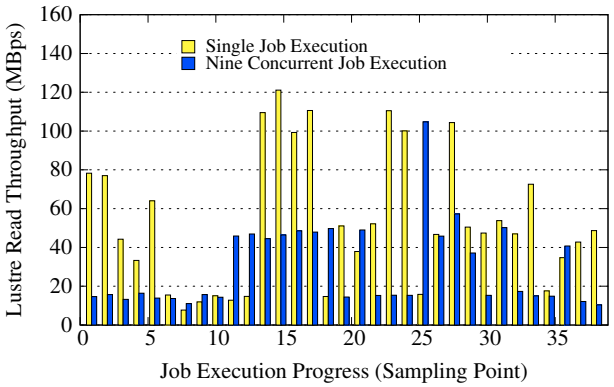


Figure 6: Lustre read performance profiling in Read Copier

We profiled Lustre read throughput in both the experiments. In Figure 6, we show the first few read throughputs. Here, we can observe that, with nine concurrent job execution, average read throughput decreases. From this experiment, we also verify that Lustre read performance can vary significantly. So, depending on the cluster and job configurations, this can affect the job execution performance.

To overcome this by utilizing both RDMA and Lustre bandwidth in the most efficient manner, we introduce dynamic adaptation in choosing shuffle policy. Since, Lustre read is more intuitive, we initially assign all the map output files to Read copiers. Thus, all reduce tasks start fetching data through Lustre Read operation as mentioned in Section III-B1. However, the Fetch Selector unit performs profiling on these read operations, thus measuring the read

latency and accumulating it from Read copier. If read latency keeps increasing for consecutive fetch requests over a pre-specified number of Read operations (in our experiments, we set this threshold to three), it informs the Dynamic Adjustment Module to switch the shuffle policy to HOMR-Lustre-RDMA for all the rest of the shuffle execution. Another alternative is to choose a random number of copier threads to start RDMA-based shuffle, while the other copiers can still do Lustre Read. However, to keep things simple and avoid book-keeping of meta information for all map locations in both HOMRFetcher and HOMRShuffleHandler, we choose this switching mechanism to operate only once. By doing this, we can also stop Fetch Selector to perform profiling after the shuffle switches to RDMA.

IV. Performance Evaluation

In this section, we present detailed performance evaluation for our design of YARN MapReduce over Lustre. First we show the performance characteristics of the two shuffle strategies over Lustre and compare the performance with that of the default MapReduce architecture.

A. Experimental Setup

For our experiments, we have used three different clusters.

(1) TACC Stampede [4] (Cluster A): The Stampede supercomputing system at TACC [4], which is the 7th fastest supercomputer in the Top500 [27] list of November 2014, has 6,400 compute nodes. The compute nodes are provisioned with Intel Sandy Bridge (E5-2680) dual octa-core processors, 32GB of memory, 80 GB local HDD, a SE10P (B0-KNC) co-processor and a Mellanox IB FDR MT4099 HCA. The host processors are running CentOS release 6.3 (Final). This system provides 14 PBs of global storage managed as three Lustre file systems.

(2) SDSC Gordon [5] (Cluster B): The Gordon Compute Cluster at SDSC [5], which is the 192nd fastest supercomputer on the same list, is composed of 1,024 dual-socket compute nodes connected by a dual rail, QDR InfiniBand, 3D torus network. Each compute node in this cluster has two eight-core Intel EM64T Xeon E5 2.6 GHz (Sandy Bridge) processors, 64 GB of DRAM, and 300 GB local SSD with CentOS 6.4 operating system release. The 4 PB Lustre file system is accessible from each compute node via two 10 GigE network interfaces.

(3) Intel Westmere (Cluster C): This in-house cluster has Xeon Dual quad-core operating at 2.67 GHz. Each node is equipped with 12 GB RAM and one 160 GB HDD. Each node is also equipped with MT26428 QDR ConnectX HCAs (32 Gbps data rate) with PCI-Ex Gen2 interfaces. The nodes are interconnected with a Mellanox QDR switch. Each node runs Red Hat Enterprise Linux Server release 6.1 (Santiago). Lustre installation in this cluster has a capacity of 12 TB and accessible through IB QDR.

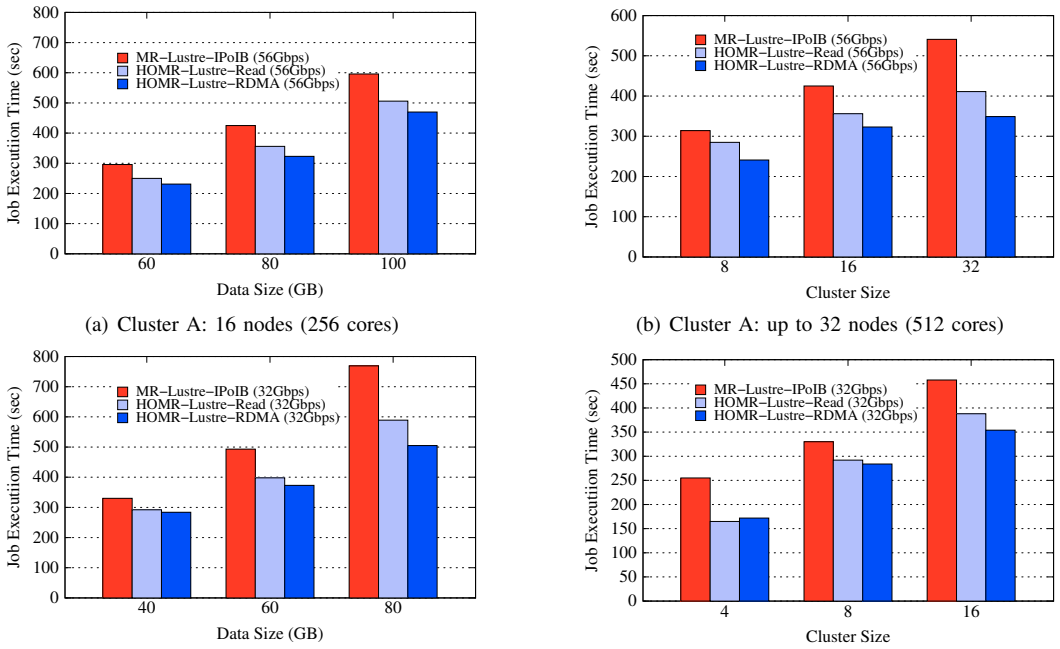
In all our experiments, we have used Hadoop-2.5.0 and JDK 1.7.0. Our HOMR implementation over YARN is based on the same Hadoop version. For the figures presented in this section, we have used MR-Lustre-IPoIB as the legend for default MapReduce using IPoIB over Lustre. For the two shuffle approaches with HOMR, we use HOMR-Lustre-RDMA and HOMR-Lustre-Read legends to represent the RDMA-based shuffle approach for HOMR and Lustre read-based approach for HOMR, respectively. We have used HOMR-Adaptive legend to show the overall design performance. Since in our experiments, all input, output, and intermediate data are stored in Lustre or local file system, we do not deploy HDFS at any time. We use the local file system block size of 256 MB and set the Lustre stripe size equal to it.

B. Comparison between Two Shuffle Strategies

In this section, we show the performance characteristics of two shuffle approaches presented in Section III-B, using the Sort benchmark. We have performed these evaluations on Clusters A and B. We choose this benchmark since it represents a shuffle-intensive work-flow; thus, clearly depicts the performance characteristics of the corresponding shuffle and merge algorithms used in the framework.

We perform two sets of experiments on each cluster. First, we vary the data size with a fixed cluster configuration and observe the performance trends. For the second set of experiments, we perform weak scaling tests and vary both the cluster and the data size to provide performance insights for scale-out environment. On Cluster A, we have performed Sort experiments on a cluster size of 16 (256 cores) with a variation in data size from 60 GB to 100 GB. As shown in Figure 7(a), we observe that HOMR-Lustre-RDMA performs better compared to HOMR-Lustre-Read for each of the data sizes. For example, HOMR-Lustre-RDMA outperforms HOMR-Lustre-Read by 8% for 100 GB Sort in a cluster size of 16. However, both shuffle approaches have higher performance benefits compared to MR-Lustre-IPoIB. For example, HOMR-Lustre-RDMA has a performance benefit of 21% compared to MR-Lustre-IPoIB.

For the second set of experiments on Cluster A, we use three different cluster sizes, 8, 16, and 32. We vary the data size from 40 GB to 160 GB. Figure 7(b) shows the results of these evaluations. Because of the fast network bandwidth from IB FDR, here also, we observe better performance benefits for HOMR-Lustre-RDMA compared to HOMR-Lustre-Read as we scale up the cluster size. As observed from Figure 7(b), on 32-node cluster, HOMR-Lustre-RDMA has a 15% benefit compared to HOMR-Lustre-Read for a Sort with 160 GB data size. These results indicate that the read-based shuffle approach is performing worse as we scale compared to RDMA-based shuffle. Scaling the cluster size means large number of I/O readers and writers accessing Lustre simultaneously (256 writers and 128 readers for 32 node experiment). However, both approaches still observe performance benefits compared to MR-Lustre-IPoIB.



(a) Cluster A: 16 nodes (256 cores)

(b) Cluster A: up to 32 nodes (512 cores)

(c) Cluster B: 8 nodes (128 cores)

(d) Cluster B: up to 16 nodes (256 cores)

Figure 7: Evaluating Sort in Clusters A and B

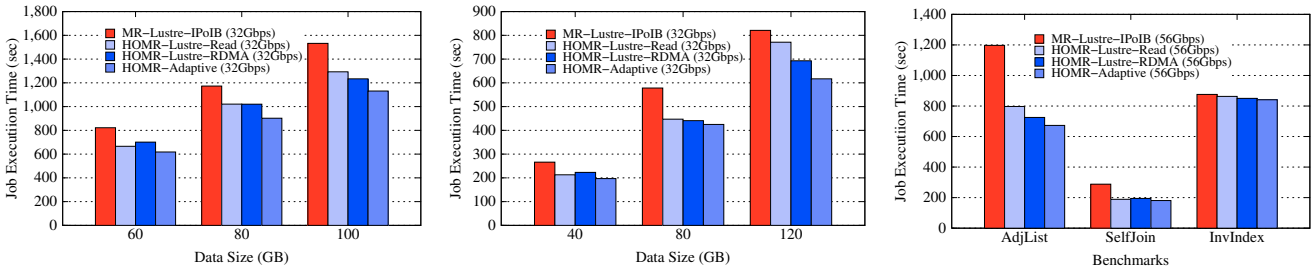
Figures 7(c) and 7(d) show the performances of the similar Sort experiments on Cluster B. In this cluster, we have performed the first set of experiments on a cluster size of 8 (128 cores). We vary the data size from 40 GB to 80 GB. Here, we observe that the RDMA-based shuffle approach, HOMR-Lustre-RDMA, performs better for each experiment compared to HOMR-Lustre-Read. For an 80 GB Sort experiment in a cluster size of 8, we observe 15% improved performance benefit for HOMR-Lustre-RDMA compared to HOMR-Lustre-Read. For scale out experiments in this cluster, we vary the cluster size from 4 to 16, while increasing the data up to 80 GB. Here, we can see that for a cluster size of 4, Read-based shuffle performs better compared to RDMA. However, as we scale, HOMR-Lustre-RDMA performs much better than HOMR-Lustre-Read, similar to our observations in Cluster A. The major reason behind this observation is similar to that in Cluster A. Moreover, unlike Cluster A, Cluster B Lustre is accessible through 10 GigE connections. Thus, in HOMR-Lustre-Read, the intermediate data is transferred over the comparatively slower network. Also, during IOZone experiments in Cluster B (shown in Figure 5), we observe variations in average read and write throughputs from different compute nodes. For these reasons, shuffle over RDMA proves to be faster than Lustre read over 10GigE.

C. Performance Improvement for Dynamic Adaptation

As discussed in Section III-D, our dynamic adaptation in choosing shuffle policy ensures the optimum usage of network and file system bandwidth. We verify this for three set of experiments in three different clusters. Figure 8 presents these results. In the first set of experiments, we use the same Sort benchmark in Cluster C. We use 16 nodes from this cluster. Here, we vary the data size from 60 GB to 100 GB. As shown in Figure 8(a), we can see that our adaptive design, HOMR-Adaptive, ensures equal or better performance compared to the two separate shuffle approaches. For 100 GB Sort experiment, we observe that by incorporating dynamic adaptation, it gets 8% better performance benefit compared to HOMR-Lustre-RDMA. Overall, it observes a 26% performance benefit compared to default MR over Lustre with IPoIB.

The second set of experiments is conducted in Cluster B with 16 nodes. The benchmark used in this experiment is TeraSort, which is a special case of the more generic benchmark, Sort. Unlike Sort, TeraSort uses fixed size key-value pair of 100 bytes. For this experiment, we vary the data size up to 120 GB. Here also, we observe that HOMR-Adaptive can choose the shuffle policy optimally to observe 25% benefit compared to default YARN MR over Lustre. These results are shown in Figure 8(b).

We also evaluate other benchmarks apart from Sort and TeraSort. We choose three different benchmarks from the PUMA [28] benchmark repository. Two of these benchmarks, AdjacencyList (AL) and SelfJoin (SJ), represent shuffle-intensive work-flow, while InvertedIndex (II) represents compute-intensive behavior. Since, we enhance mostly in shuffle phase, we observe larger benefits for AL and SJ. We conduct these experiments in 8 nodes of Cluster A and observe a maximum of 44% performance benefit for AL. Since the data sets that we have used for these experiments are relatively small (30 GB) compared to the other experiments, we observe better performance benefits



(a) Sort, Cluster C, 16 nodes (128 cores) (b) TeraSort, Cluster B, 16 nodes (256 cores) (c) PUMA [28], Cluster A, 8 nodes (128 cores)

Figure 8: Performance improvement for dynamic adaptation

through effective caching for RDMA shuffle.

D. System Resource Utilization

In this section, we discuss CPU and memory usage for our design and compare with those of the default MR-Lustre-IPoIB. Also, we show the data shuffled vs. data read from Lustre in our adaptive approach. Figure 9 presents these results from Cluster A. To measure different system resource parameters, we use Linux performance monitoring tool, `sar`, provided as a part of the `sysstat` package. We measure these parameters for a Sort benchmark running on 4 nodes in Cluster A with a data size of 40 GB.

Figure 9(a) shows the CPU utilization for the entire job duration of the Sort benchmark. Here, we observe that the CPU usage for MR-Lustre-IPoIB is more during the early stage of job execution and it reduces throughout the rest of the execution. However, our shuffle design has a higher CPU usage at the end of the job execution. During this period, the shuffle, merge, and reduce phases overlap with each other to complete the job execution fast.

As shown in Figure 9(b), although our design uses slightly more memory compared to default MR-Lustre-IPoIB, it can progress much faster. The major reason behind memory consumption is caching for both shuffle approaches.

In Figure 9(c), we compare the amount of data shuffle through Lustre and RDMA in our adaptive approach. As we can see, the initial stage uses Lustre read and to get maximum throughput, our design tries to read as much data as possible during the early stage of job execution. However, as soon as more concurrent tasks start reading from Lustre, our design switches to RDMA-based data transfer and thus achieves better network efficiency during the end stage of job execution. For default Hadoop, we skip this usage report since the data read from Lustre is exactly similar to the data shuffled over the network.

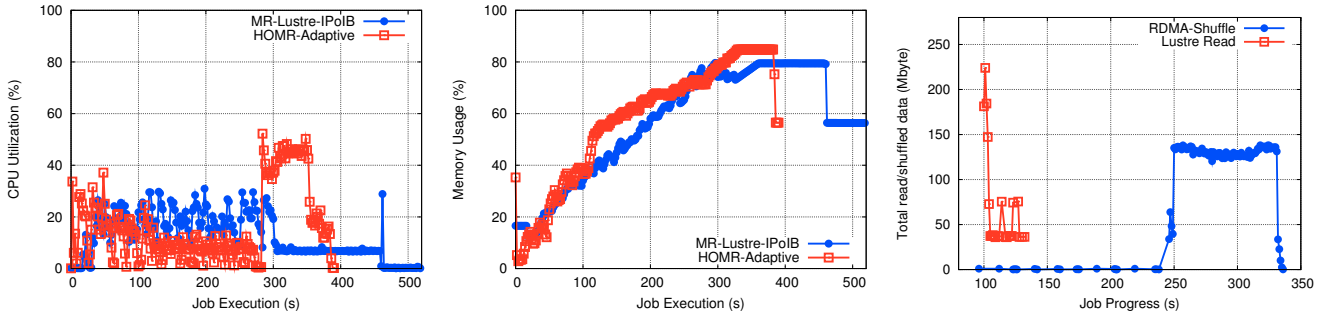
V. Related Work

In modern HPC clusters, the compute nodes are provisioned with a lightweight operating system, and zero or limited-capacity local storage. This prevents the default MapReduce framework from processing jobs with large scale data sizes when local disks are required. Prior research has dealt with improving the performance of HDFS by exploring the interoperability between HDFS and existing POSIX-compliant parallel file systems such as PVFS [29], Ceph [30], GPFS [31], GFarm [32], etc. These studies either provide interfaces that the Hadoop environment can use with the corresponding file system, or propose HDFS-specific optimizations in their file system which makes it compatible for use in the Hadoop ecosystem. A SEDA (Staged Event Driven Architecture) based approach for HDFS is presented in [19] that presents a new architecture for HDFS which can bring down significant bottlenecks in the existing architecture. Other storage architectures, like MixApart [33], enable scale-out of data analytics by using a cache layer at the compute nodes and intelligent schedulers to utilize the shared storage efficiently for MapReduce framework.

However, the use of Lustre, with the MapReduce architecture, also has attracted significant attention within the HPC community. Research work like [21, 22] have studied the implications of using Lustre as the back-end file system for MapReduce jobs and showed that with the availability of high-bandwidth network, significantly better clustered file system performance with Hadoop is possible. In [23], the authors have evaluated the performance of the YARN architecture over Lustre file systems. In one of our prior work [11], we have shown that RDMA-enhanced MapReduce can provide much higher performance gains compared to the default MapReduce over Lustre. In this prior work, the intermediate data of MapReduce jobs are stored in local disks and the final outputs reside in Lustre. On the other hand, in this paper, we propose alternative shuffle strategies for using Lustre as local storage. Also, our designs are transparent to the underlying global file system and thus any POSIX-compliant parallel file system can be used.

VI. Conclusion and Future Work

In this paper, we propose an enhanced YARN MapReduce architecture over parallel file system, Lustre, with two different shuffle strategies. We focus on the architecture where the intermediate data for MapReduce jobs are placed in Lustre file system as well as local disks, contrasting the default approach of using only the local file system for these data. The shuffle approaches proposed in this paper differ on the way of accessing data from the reduce tasks. We also perform detailed optimization for this architecture to exploit the differences in performance characteristics of Lustre file



(a) CPU (b) Memory (c) RDMA Shuffle vs. Lustre Read

Figure 9: Resource utilization in Cluster A

system used in different leadership-class HPC clusters. Our results indicate that both the shuffle strategies have influence on the performance of MapReduce jobs while the RDMA-based shuffle approach always scales better compared to Lustre read-based shuffle approach. We also designed a dynamic adaptation technique to choose the better shuffle strategy during run-time of job execution. Our results ensure a maximum of 44% performance benefit for shuffle-intensive benchmarks. In the future, we plan to make this design publicly available to the community to help HPC clusters with Lustre to handle big data processing efficiently.

REFERENCES

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

[2] N. Hemsoth, "HPC Roots Feed Big Data Branches," http://www.hpcwire.com/2014/02/09/hpc-roots-feed-big-data-branches/.

[3] The Apache Software Foundation, "The Apache Hadoop Project," http://hadoop.apache.org/.

[4] Stampede at TACC, http://www.tacc.utexas.edu/resources/hpc/stampede.

[5] Gordon at San Diego Supercomputer Center, http://www.sdsc.edu/us/resources/gordon/.

[6] 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 *SC*, Salt lake City, Utah, 2012.

[7] M. W. Rahman, N. S. Islam, X. Lu, J. Jose, H. Subramoni, H. Wang, and D. K. Panda, "High-Performance RDMA-based Design of Hadoop MapReduce over InfiniBand," in *HPDIC, in conjunction with IPDPS*, Boston, MA, 2013.

[8] X. Lu, N. S. Islam, M. W. Rahman, J. Jose, H. Subramoni, H. Wang, and D. K. Panda, "High-Performance Design of Hadoop RPC with RDMA over InfiniBand," in *ICPP*, France, 2013.

[9] R. H. Castain and O. Kulkarni, "MapReduce and Lustre: Running Hadoop in a High Performance Computing Environment," https://intel.activeevents.com/sf13/connect/sessionDetail.ww?SESSION_ID=1141.

[10] 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 *IPDPS*, Shanghai, China, 2012.

[11] M. W. Rahman, X. Lu, N. S. Islam, R. Rajachandrasekar, and D. K. Panda, "MapReduce over Lustre: Can RDMA-based Approach Benefit?" in *Euro-Par*, Porto, Portugal, 2014.

[12] X. Lu, M. W. Rahman, N. S. Islam, D. Shankar, and D. K. Panda, "Accelerating Spark with RDMA for Big Data Processing: Early Experiences," in *HotI*, California, USA, 2014.

[13] M. W. Rahman, X. Lu, N. S. Islam, and D. K. Panda, "HOMR: A Hybrid Approach to Exploit Maximum Overlapping in MapReduce over High Performance Interconnects," in *ICS*, Munich, Germany, 2014.

[14] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *MSST*, Incline Village, Nevada, 2010.

[15] T. Sterling, E. Lusk, and W. Gropp, "Beowulf Cluster Computing with Linux," in *MIT Press*, Cambridge, MA, 2003.

[16] T. L. Sterling, J. Salmon, D. J. Becker, and D. F. Savarese, "How to Build a Beowulf: A Guide to the Implementation and Application of PC Clusters," in *MIT Press*, Cambridge, MA, 1999.

[17] C. Engelmann, H. Ong, and S. L. Scott, "Middleware in Modern High Performance Computing System Architectures," in *ICCS*, Beijing, China, 2007.

[18] Y. Wang, X. Que, W. Yu, D. Goldenberg, and D. Sehgal, "Hadoop Acceleration through Network Levitated Merge," in *SC*, Seattle, WA, 2011.

[19] N. S. Islam, X. Lu, M. W. Rahman, and D. K. Panda, "SOR-HDFS: A SEDA-based Approach to Maximize Overlapping in RDMA-Enhanced HDFS," in *HPDC*, 2014.

[20] OSU NBC Lab, "High-Performance Big Data (HiBD)," http://hibd.cse.ohio-state.edu.

[21] N. Rutman, "Map/Reduce on Lustre," http://www.xyratex.com/sites/default/files/Xyratex_white_paper_MapReduce_1-4.pdf.

[22] O. Kulkarni, "Hadoop MapReduce over Lustre," http://www.opensfs.org/wp-content/uploads/2013/04/LUG2013_Hadoop-Lustre_OmkarKulkarni.pdf.

[23] W. Yu and O. Kulkarni, "Progress Report on Efficient Integration of Lustre and Hadoop/YARN," Lustre User Group Meeting, April 2014.

[24] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *OSDI*, Boston, MA, 2004.

[25] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache Hadoop YARN: Yet Another Resource Negotiator," in *SOCC*, Santa Clara, CA, 2013.

[26] IOzone, "IOzone Filesystem Benchmark," http://www.iozone.org/.

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

[28] Purdue MapReduce Benchmarks Suite (PUMA), https://sites.google.com/site/farazahmad/pumabenchmarks.

[29] W. Tantisiriroj, S. Patil, G. Gibson, S. W. Son, S. Lang, and R. Ross, "On the Duality of Data-Intensive File System Design: Reconciling HDFS and PVFS," in *SC*, Seattle, WA, 2011.

[30] C. Maltzahn, E. Molina-Estolano, A. Khurana, A. J. Nelson, S. A. Brandt, and S. Weil, "Ceph as a Scalable Alternative to the Hadoop Distributed File System," in *USENIX*, Boston, MA, 2010.

[31] K. Gupta, R. Jain, H. Pucha, P. Sarkar, and D. Subhraveti, "Scaling Highly-Parallel Data-Intensive Supercomputing Applications on a Parallel Clustered Filesystem," in *SC*, New Orleans, LA, 2010.

[32] S. Mikami, K. Ohta, and O. Tatebe, "Using the Gfarm File System as a POSIX Compatible Storage Platform for Hadoop MapReduce Applications," in *GRID*, France, 2011.

[33] M. Mihailescu, G. Soundararajan, and C. Amza, "MixApart: Decoupled Analytics for Shared Storage Systems," in *HotStorage*, Boston, MA, 2012.