# Design and Evaluation of Hadoop on the Spider Filesystem

Seung-Hwan Lim, James Horey, and Raghul Gunasekaran
Oak Ridge National Laboratory
Oak Ridge, TN 37831
{lims1, horeyjl, gunasekaranr}@ornl.gov

## ABSTRACT

We describe the deployment of an on-demand Hadoop tool on a shared, high-performance parallel filesystem. Our filesystem, Spider, is a Lustre-based parallel filesystem actively used across several computing clusters, including Titan, one of the world's fastest supercomputers. Although Hadoop is designed for a very different architecture than ours (i.e., shared-nothing), we demonstrate that Hadoop running over a shared filesystem is practical and can yield some significant performance benefits.

We evaluated our implementation across both Lustre and Hadoop-specific parameters using several standard benchmarks. We collected experimental data while Spider was in active use, thus allowing us to capture realistic performance data. In addition to experimental data, we also discuss architectural details of our on-demand Hadoop tool, with the aim that other HPC institutions may benefit. To our knowledge, this paper is the first to provide such a rigorous experimental evaluation of an on-demand Hadoop tool running over a shared parallel filesystem.

## 1. INTRODUCTION

Constructing data analytics platforms that can handle massive amounts of data is an emerging requirement for many organizations including national laboratories, businesses (both small and large), and academia. Among data processing platforms, Hadoop MapReduce has emerged as one of the most active open-source "big data" tools, allowing users to process both structured and unstructured data. Although Hadoop was initially designed to analyze business-oriented data (logs, transactional data), Hadoop has been used increasingly to analyze scientific data (graphs, images) as well. As Hadoop evolves to meet these additional demands, it has continued to interface with a variety of system architectures and applications as shown in Figure 1.

The current Hadoop implementation is based upon a shared-nothing architecture. In such an architecture, each node is

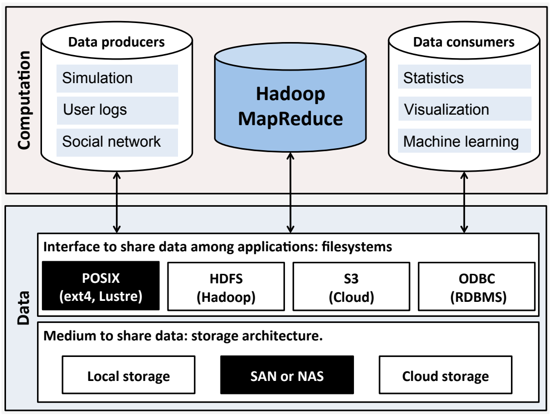


Figure 1: Motivation and architecture: users face a variety of choices in both interfaces and hardware to share data across an analytics pipeline.

equipped with its own local processors, memory, and storage. To take advantage of this architecture, clusters powered by Hadoop typically use the Hadoop Distributed File System (HDFS) [30] to serve data. These nodes are networked together using commodity interconnects (1/10 GigE) in various topologies (multi-level, fat-tree topologies, and more advanced setups) [16, 24]. The primary benefits of such architectures are simplicity, low initial cost, and horizontal scalability. In addition to these benefits, shared-nothing architectures also enable a great deal of optimization opportunities since the user is able to configure the number and type of disks associated with each system, thereby minimizing storage performance variability. However, while initially appealing, this architecture often creates challenges in optimizing application performance, since data locality must be carefully considered [27].

In contrast, shared storage architectures (storage area networks (SAN), network attached storage (NAS), or parallel filesystems) are commonly employed in both data centers for business applications [17,31] and high-performance computing (HPC) centers for scientific applications [13]. Typically, storage is shared among a set of compute clusters via a high-performance interconnect such as Infiniband. In such environments, compute nodes normally do not have any local storage. Instead, the shared filesystem is mounted to each compute node, in a manner similar to a networked filesystem such as NFS for regular files [29] and VMFS for virtual block devices [33].

Such a shared storage architecture provides several advantages over the shared-nothing architecture, including the flexibility in provisioning compute and storage resources [31]. The shared storage architecture also adds the benefit of simplicity from the application's perspective such as more uniform storage access time as data-locality is no longer a strict requirement. Further discussion of such benefits as applied to data infrastructures can be found in recent work by Nightingale et al [23]. While these benefits have made shared storage an attractive option for many organizations, users trying to run Hadoop in such environments are still left with many technical challenges [9]. Challenges in such an architecture center around providing sufficient bandwidth between compute nodes and storage nodes while controlling the quality of service, mainly credited by the fact that all I/O is essentially transformed into network communication [31, 36].

In this paper, we discuss the implementation of an on-demand Hadoop service that executes over a shared, parallel POSIX-compliant filesystem. Our filesystem, Spider, is a Lustre-based [34] filesystem actively used across several computing clusters at Oak Ridge National Laboratory, including Titan [7], one of the world's fastest supercomputers. Spider is currently provisioned with approximately 10 PB of active storage, and can support up to 240GB/s of throughput. While Spider was designed to support I/O for peta-scale scientific simulations, we show that Spider can also be used for data intensive applications. Specifically, we developed an on-demand Hadoop service that users can deploy dynamically on ORNL's compute clusters that leverages Spider to store both temporary and active data. Using our tool, users can dynamically instantiate Hadoop clusters, specify the desired number of nodes, and provide details of the MapReduce job in a manner similar to Amazon's Elastic MapReduce service [1]. This feature makes our tool useful for users in shared computing environments as they only need to reserve the computing resources for a limited duration. Input and output for the data analytics is made simple since all the files already reside on a shared system. This makes it possible to connect multiple applications (including HPC simulations) into a flexible data analysis pipeline. While we have found performance to be quite competative with stand-alone Hadoop clusters, we emphasize that a large shared-storage architecture may not be the most appropriate architecture for small and medium scale Hadoop deployments. Instead, our work demonstrates that for organizations that are already invested in shared, parallel filesystems, Hadoop can be a reasonable choice as a data analysis tool.

The rest of the paper is organized as follows. § 2 provides an overview of Hadoop and reviews related work. § 3 describes the hardware and software architecture of our on-demand Hadoop tool. This discussion is followed by a series of in-depth experimental evaluations (§ 4). Our experimental evaluation focuses on the performance effect of Lustre and Hadoop-specific parameters on Smoky, one of ORNL's development clusters. We demonstrate that I/O intensive applications can benefit greatly with our implementation, while CPU bound applications still perform reasonably well. In § 5, we discuss some promising areas of development. Finally, we offer brief conclusions in § 6.

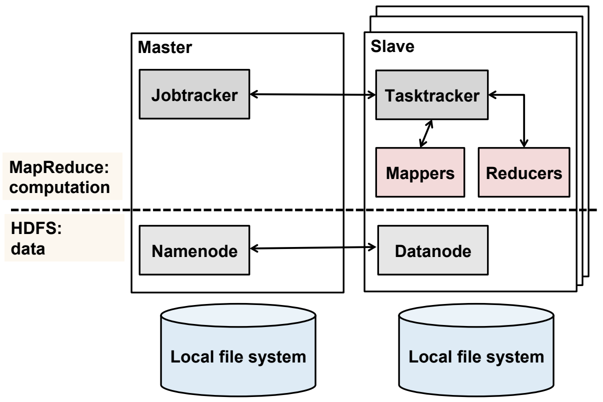


Figure 2: Hadoop overview: the data storage layer can be configured with different implementations, including Amazon S3 cloud storage and local file systems.

## 2. BACKGROUND

Here we give a brief overview of Hadoop along with other systems that have been used for large-scale data processing.

In this study, we employ Hadoop, a popular open-source system for processing and storing unstructured data. Since its open-source debut, Hadoop has evolved into a general-purpose, large-scale data processing framework. Recently this has included extensions to operate in cloud computing environments (Amazon Elastic MapReduce [1]) over shared memory [14, 38], and over shared storage [22]. For example, users are deploying Hadoop over network-attached storage devices (e.g., EMC Isilon) and virtual storage (i.e., Amazon S3 [2], VMWare) [21]. While we explore the same overall concept of utilizing shared storage, our approach is fundamentally different since we employ a shared, parallel filesystem.

As shown in Figure 2, Hadoop consists of two layers: a computational processing layer inspired by Google's MapReduce [12] and a data storage layer (HDFS [30]) inspired by the Google Filesystem [15]. Within the computational layer, nodes are assigned to be either a *Jobtracker* or a *Tasktracker*. Typically, the Jobtacker executes on a single, master node and schedules individual computational tasks. Taskstrackers execute on multiple, worker nodes and actually executes these tasks. In addition, the Jobtracker monitors the health of Tasktrackers and checks the progress of tasks in order to manage stragglers as well as provide fault-tolerence [39].

The computational layer works closely with the Hadoop Filesystem (HDFS), a distributed POSIX-like (but not compliant) filesystem. As with other parallel filesystems, HDFS assigns metadata responsibilities to a single server running the *Namenode*. Actual data blocks are stored across multiple *Datanodes*. Because HDFS was designed to work in tandem with the MapReduce processing layer, HDFS explicitly exposes block locality to reduce data movement across the cluster. Although HDFS is the primary storage layer for Hadoop, Hadoop is designed to support additional filesystem backends [32]. We take advantage of this feature to explore the use of alternative POSIX-compliant, shared filesystems such as Lustre, PVFS2 [6], or Ceph [35].

## 2.1 Related Works

Our approach is fundamentally different from approaches that re-implement the MapReduce model. For example there have been at least two implementations that utilize MPI [18] [28]. However, it is unclear to exactly the extant these implementations are comparable to Hadoop. For instance, the authors in [18] point out the difficulty of supporting fault-tolerance and arbitrary MapReduce operations in the current version of the MPI standard. Moreover, we believe a greater challenge is the inability for these implementations to take advantage of a growing number of Hadoop related tools. For example, the Hadoop ecosystem currently includes machine learning libraries (Mahout), graph processing capabilities (Giraph), SQL features (Hive), and various scripting languages (Pig).

Other tools similar to ours include *myHadoop*, a tool to provision Hadoop clusters in an HPC environment available from the San Diego Supercomputing Center [20]. While our current user-facing scripts are similar, our work focuses on the rigorous evaluation of operating Hadoop on a shared, parallel filesystem across a variety of parameters. To our knowledge, such an evaluation has not been conducted before. We expect that both set of user-facing tools will continue to evolve to meet the demands of specific user communities.

It should be noted that we are not the first to run Hadoop over a parallel filesystem [11] [10]. Argonne National Lab attempted to integrate PVFS2 into Hadoop [32]. Those works, however, focused on integrating parallel file systems into the Hadoop framework, while maintaining the use of local storage. The contribution of our work is that we evaluate the behavior of native Hadoop when we run Hadoop over a production, *shared* filesystem. We believe this is a more practical option for institutions that already have a shared filesystem.

## 3. DESIGN AND IMPLEMENTATION

The design and implementation of our system was motivated by several factors. This includes the desire to re-use ORNL's shared storage resources, to simplify the overall data pipeline from simulations to data analysis, and to leverage the broad set of tools developed by the Hadoop community. It has been our experience that Hadoop often constitutes a single element of a larger data analysis pipeline that may include other parallel tools, such as HPC simulations, relational databases, and visualization tools (refer to Figure 1.) Unfortunately, most of these tools rely on their own internal storage mechanism. To connect these elements, traditional tools must explicitly copy data from one tool to another. Although this can be automated, the process can be very fragile and may not scale well as the number of data pipeline elements increases.

## 3.1 Technical Approach

Our approach replaces HDFS with Lustre, a POSIX-compliant parallel filesystem. Re-using a single, shared POSIX-compatible filesystem alleviates many of the issues outlined, since data can be logically shared across many applications (including legacy applications). This is especially useful for scientific computing institutions, such as ORNL. Most scientific simulation software assumes the availability of a POSIX-compatible filesystem. This will also allow us to construct dynamic, data pipelines that can process the output of simulations using our on-demand Hadoop service. By doing so, users will be able to leverage a wide range of data analytic libraries, including machine learning (Mahout [4]), relational data warehousing (Hive [3]), and high level scripting tools (Pig [5]). This is in contrast to alternative implementations of the MapReduce model, which are unable to fully leverage these community-developed resources [28].

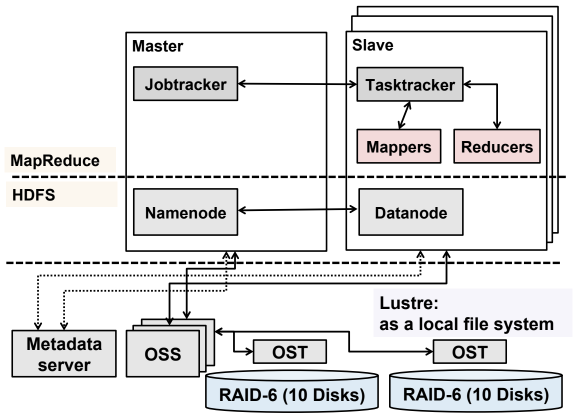
Besides being POSIX-compliant, Lustre also enables users to customize the performance behavior by specifying the degree of I/O parallelism (via *stripe count*) and block size (via *stripe size*). We exploit both these parameters in our work. In our current implementation, we leveraged the *RawLocalFilesystem* provided by the default Hadoop installation. The shared Lustre filesystem is mounted on every node, and is exposed as a normal filesystem. Before users start their jobs, data directories are automatically created and initialized. Optimizing the Hadoop filesystem backend for Lustre-specific capabilities is left to future work. On typical shared computing resources, users can only reserve the computing resources for a finite amount of time. Thus, it is necessary that a service component enables users to dynamically instantiate a Hadoop cluster. We implemented such a service component for this study. This component performs three major functions: (1) allocates the necessary resources via Torque [8]; (2) dynamically creates the necessary configuration files such as network information of allocated nodes; and (3) configures and starts the Hadoop cluster. Users, by providing a simple job description file, can then immediately launch jobs on this cluster.

## 3.2 Storage subsystem

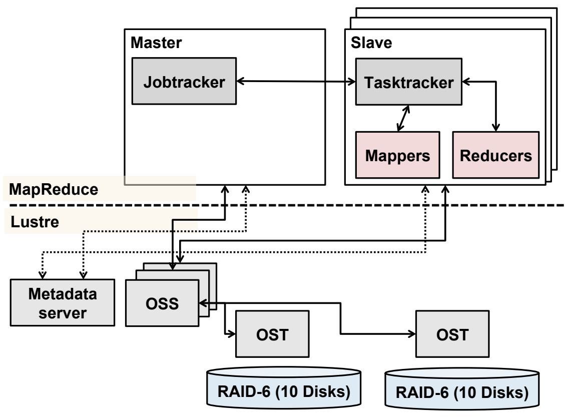
Our on-demand Hadoop tool resides on top of the Spider filesystem. Spider is a Lustre-based storage cluster of 96 DDN S2A9900 RAID controllers with an aggregate bandwidth of 240 GB/s and over 10 petabytes of storage. Storage is spread over 13,440 1-terabyte SATA drives. Each controller has 10 SAS channels through which the backend disk drives are connected, and the drives are RAID 6 formatted in an $8 + 2$ configuration, constituting an Object Storage Target (OST). In addition, each controller has two dual-port 4x DDR IB HCAs for host side connectivity.

Access to the storage is through the 192 Lustre Object Storage Servers (OSS) connected to the controllers over InfiniBand. Each OSS is a Dell dual-socket quad-core server with 16 GB of memory. All the compute platforms (i.e., Smoky, Titan, etc.) connect to the storage infrastructure over a multistage InfiniBand network, referred to as SION (Scalable I/O network) [13]. A visual overview of the Spider filesystem is as shown in Figure 3. In this study, we used approximately one-quarter of Spider's capacity (3360 disks).

## 3.3 Hadoop with Lustre

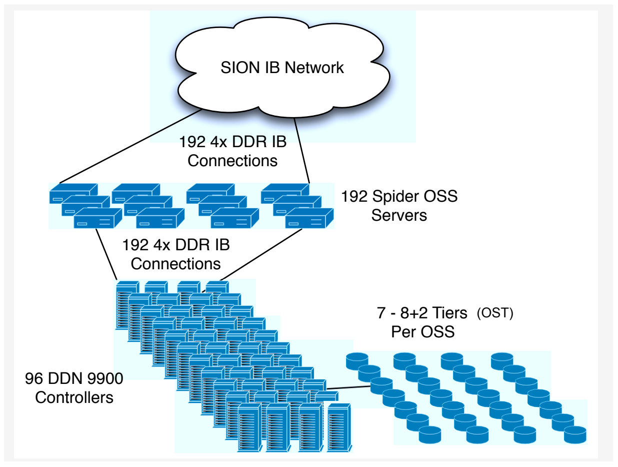


(a) HDFS with Lustre



(b) Lustre as an alternative of HDFS

Figure 4: Two alternative implementation options for integrating shared storage in Hadoop. We can run HDFS on top of Lustre (treating Lustre as a simple local volume) or by completely replacing HDFS with Lustre (treatig Lustre as a parallel filesystem). The second option reduces functional redundancies and simplifies the overall architecture.



SION IB Network
192 4x DDR IB Connections
192 Spider OSS Servers
192 4x DDR IB Connections
7 - 8+2 Tiers (OST) Per OSS
96 DDN 9900 Controllers

Figure 3: Overview of the Spider filesystem. Spider is one of the largest Lustre installations with 192 object storage servers (OSS) and 7 OSTs per OSS. An OST consists of 10 disks with RAID-6 configuration.

There are two main implementation strategies to consider when running Hadoop over shared storage (Figure 4). The first option is to use HDFS over Lustre. Since HDFS uses the local file system of each slave node, we could simply use Lustre to serve local directories as shown in Figure 4a. However, this approach unnecessarily duplicates functionality and introduces additional overheads. For example, processing a single data block requires two cascaded metadata lookups (one for HDFS and another for Lustre). In addition, HDFS performs its own block checksum calculation since local filesystems do not guarantee data integrity. In our environment, Lustre already provides this functionality. Hence, in order to maximize efficiency, we chose to use the *RawLocalFilesystem* implementation as illustrated in Figure 4b. This bypasses HDFS while still enabling distributed computation. Evaluating using TestDFSIO (a Hadoop MapReduce I/O benchmark), this option provided more than 10 times faster performance for both read and write operations. However, due to space constraints, we omitted detailed performance comparisons between these two implementations.

### 3.4 Hadoop configuration

Hadoop has extensive configuration options for users to optimize their particular workloads. Because our tool is used in an on-demand fashion, all configuration changes must be made in the absence of pre-existing workload information. Also, due to the complexity of selecting optimal configuration values [37], we chose to modify the Hadoop configuration using best practice heuristics to ensure proper execution. The modified Hadoop configuration parameters fall into two categories: to use a remote file system instead of HDFS; and to tolerate wider variance of network and I/O latency. These changes are summarized in Table 1. Further configuration options are detailed in § 4. These configuration values are not intended to optimize performance, though § 5 suggests some directions to optimize the performance of Hadoop clusters on parallel filesystems.

## 4. EVALUATION RESULTS

We conducted experimental evaluation on a 80-node development cluster (Smoky [25]). Smoky is housed in two racks, and each rack is equipped with a 32Gbps (2XQDR) Infiniband (IB) switch. These switches are, in turn, connected to the shared storage system, Spider [26]. Nodes on Smoky are also equipped with a 1 GE interconnect for processor communication, without local storage. Table 2 details the key technical features of the system. Although there are larger clusters connected to Spider, we chose Smoky due to the simplicity of development and initial evaluation. We expect to port our system software to other ORNL clusters (including Titan) in the near future.

In order to characterize Hadoop performance over a shared storage system, we ran several benchmarks while varying Lustre and Hadoop-specific parameters. In addition to this

Table 1: Hadoop configuration

| Parameter | Value | Comments | File |
|---|---|---|---|
| basic configuration | | | |
| Java heap size | 2GB | 1GB is the default. | hadoop-env.sh |
| Child Java VM heap size | 2GB | 200 MB is the default. | mapred-site.xml |
| to use Lustre as local file system | | | |
| fs.default.name | rawfile:/// | Use customized file system | core-site.xml |
| fs.rawfile.imple | org.apache.hadoop.fs.RawLocalFileSystem | use RawLocalFileSystem to avoid checksum calculation | core-site.xml |
| to tolerate wider variance of network or I/O latency | | | |
| mapred.reduce.shuffle.maxfetchfailures | 20 | 10 is the default. | mapred-site.xml |
| mapred.reduce.shuffle.connect.timeout | 72000000 | the maximum amount of time in *ms* for a reduce tasks to spend in trying to connect to a tasktracker for getting map outputs | mapred-site.xml |
| mapred.reduce.shuffle.read.timeout | 72000000 | the maximum amount of time in *ms* for a reduce tasks to wait for reading map outputs after establishing connection. | mapred-site.xml |
| mapred.task.timeout | 7200000 | The number of *ms* before a task will be terminated | mapred-site.xml |
| mapred.tasktracker.expiry.interval | 7200000 | The time-interval in *ms*, after which a tasktracker is decalred as lost if it does not send heartbeats | mapred-site.xml |
| tasktracker.http.threads | 50 | The number of http threads in tasktracker to be used for fetching map output | mapred-site.xml |

we also measured scalability results by varying the number of nodes allocated to the Hadoop cluster. As shown in Table 3, we used three representative Hadoop workloads (available from the HiBench [19] Hadoop benchmark suite). Both Smoky and Spider were used non-exclusively during production hours. During this time other users, primarily running scientific simulations written in MPI, accessed these systems. However, individual nodes allocated to the Hadoop cluster were used exclusively, to make sure individual machines were not shared. Because all results were collected from production systems, results often exhibited a wide variance. We ran multiple executions for all experiments and used error bars to represent the 95% of confidence intervals for the sample means.

Table 2: System configuration

| Parameter | Value | Comments |
|---|---|---|
| Cluster size | 80 nodes | |
| Cores/node | 16 | 4 quad-core processors |
| Memory/node | 32GB | |
| Interconnect | 2xQDR IB | 2 switches |
| File System | Lustre | Spider [26] |
| # of OSSs | 48 | 7 OSTs/OSS |
| # of OSTs | 336 | |
| # of disks/OST | 10 | RAID-6 |
| Stripe width | 1MB | the default. |

## 4.1 Microbenchmark

Since our evaluation focuses on the storage subsystem, we first characterized the I/O overhead associated with Hadoop. To generate baseline data, we ran the `dd` Linux command in parallel with each command generating one 1GB file and measured the overall throughput. We observed that the aggregate throughput peaked approximately at 2GB/s. Although Spider can support up to 240GB/s peak bandwidth, the network bandwidth between the development cluster and storage is limited by a 32Gbps IB network.

In order to identify the overheads from using Hadoop, we ran `TestDFSIO`, a standard benchmark used to stress-test the I/O subsystem. These overheads can stem from two factors: the additional software layers in MapReduce, and differences between the MapReduce task scheduler and the Linux process scheduler for the `dd` instances. Like `dd` in `TestDFSIO`, a single map task generated one 1GB file. In order to minimize extraneous performance impact, we used a Lustre stripe count of 1 and varied the number of files from 4 to 128. Note that we used 1MB of buffer for both `dd` and `TestDFSIO` workloads. For this particular experiment, we used a small number of slave nodes (one and four) to minimize the effect of the distribution of tasks across slave nodes.

### *4.1.1 Overheads from MapReduce*

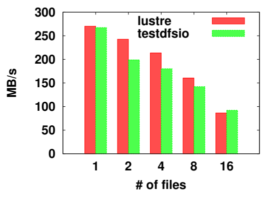
Figure 5 illustrates the individual process throughput of these two tests as the number of files are varied. For *writes*, Hadoop does very well with both a single slave and four slave nodes, often within an 18% margin of `dd`. For a large number of files (16 files per node), Hadoop actually outperforms the naive `dd` test. This is likely due to better process scheduling.

On the other hand, the read throughput of the MapReduce workloads was relatively low compared to `dd` (often one-fifth the throughput for a low number of concurrent files in the system). However, as the number of concurrent files per node increased, the performance of the raw Lustre workloads sharply degraded while the MapReduce workloads decreased only marginally. This indicates that per map task read performance is relatively low. We are currently investigating why this might be.

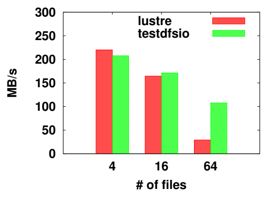
Along with throughput, we also measured the overall running time of workloads in order to show the impact from the Hadoop task scheduler (Figure 5). Although the individual write throughput is similar to native Lustre, the running time was more than 10 times slower. Although surprising, this is partly because of the set-up time of tasks in MapRe-

Table 3: Workloads: unless otherwise specified 8 mappers and 5 reducers per node was used.

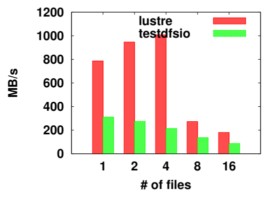
| Name | Description | Data set | Comments |
|---|---|---|---|
| Raw Lustre | run DD command instances in parallel | 1GB per process | all proceesses are either read or write. Stripe counts are varied: 1, 4, 16, and 64. 16 processes per node. |
| TestDFSIO | Test IO throughput and Average IO rate | 8, 16, 64, and 128 files | Each file sizes 1GB. 1 map task per file. 16 mappers/node is configured. |
| TeraSort | Transforming data representation | 50GB for 16 slaves; 100GB for 32 slaves; 500GB for 48 slaves | stripe count: 16. |
| K-means | Machine learning algorithm (Mahout) | 580MB | 8, 16, and 32 slaves; auto-generated by HiBench [19] |



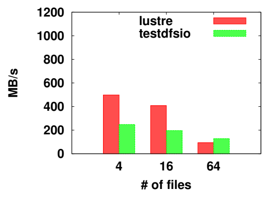
(a) write/1 slave



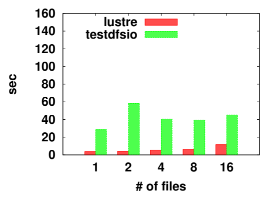
(b) write/4 slaves



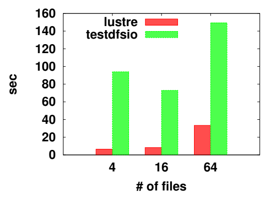
(c) read/1 slave



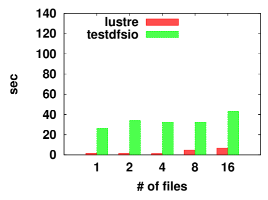
(d) read/4 slaves



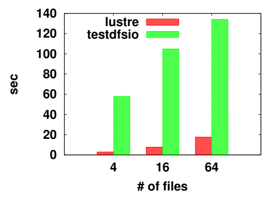
(e) write/1 slave



(f) write/4 slaves



(g) read/1 slave



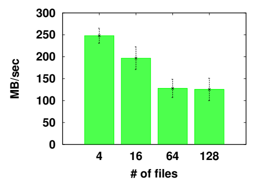
(h) read/4 slaves

Figure 5: Throughputs of raw Lustre and TestDFSIO (a), (b), (c), and (d). Running times of raw Lustre and TestDFSIO (e), (f), (g), and (h). For each workload, one 1 GB file is accessed by one process (one mapper in TestDFSIO). The Lustre stripe count was set to 1.

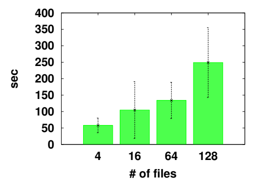
duce. For instance, to read four files with four slaves, the first task launched, on average, 6.8 seconds after starting the workloads (which was longer than the duration of raw Lustre workloads at 2.91 seconds). This can be attributed to both Hadoop-specific overheads and possibly Java startup time. Another possible factor is the task schedule across the slave nodes. Specifically, the task scheduler may not evenly distribute tasks to each host or fully use all the available resource slots. We explore this issue further in § 5.

### 4.1.2 Scalability

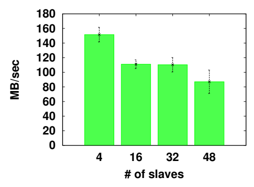
Next we considered the effect of workload and system size on the overall system performance. The system size was fixed while the data set size increased from 4GB to 128GB (Figures 6a and 6b). As Figure 6 illustrates, the average running times of workloads increased sub-linearly while individual throughputs decreased sub-linearly. We note that when the number of files was increased from 64 to 128, the average individual throughput was not statistically distinguishable, even though the running time increased by a factor of 1.86. This reflects the fact that the number of actual concurrent map tasks was limited by a maximum number of tasks per node (a set of 16 per node), which results in the same number of concurrent tasks for both 64 and 128 files. The wide variance exhibited by the data reflects the fact that experiments were conducted at different times on



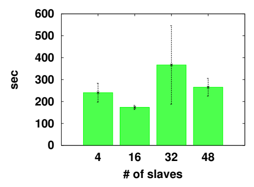
(a) throughput/files



(b) running time/files

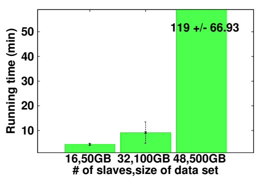


(c) throughput/slaves

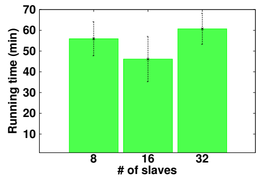


(d) running time/slaves

Figure 6: Average individual read throughputs and running times of TestDFSIO workloads. Each process accessed one 1GB file. For (a) and (b), the stripe count was set 1 and the number of slave nodes to 4. For (c) and (d), the stripe count was set 16 and the number of files to 128.



(a) Terasort



(b) Kmean

Figure 7: Average running time of Terasort and Kmean: error bars represent 95% of confindence interval.

production systems. Consequently, there were times when the number of concurrent users were higher than others, potentially leading to poorer performance.

In the next experiment (Figure 6c and Figure 6d) we increased the system size while keeping the workloads constant at 128 1GB files. In addition, as the number of slave nodes was increased, the number of maximum concurrent tasks was also increased. For instance, 4 slave nodes had up to 64 concurrent tasks, while 16 slave nodes had up to 256. While individual throughput decreased slightly (from approximately 150 to 100 MB/s), overall throughput did not change significantly with more than 16 nodes. This indicates that overall the I/O subsystem was able to scale relatively well with an increased number of concurrent accesses. With respect to average running time 16 slave nodes performed the best of all tested (Figure 6d). However, all the tests performed similarly (approximately 200 seconds), though we did observe a great deal of variance with 32 nodes, ranging from 200 to 500 seconds. This variance suggests that significant challenges still exist with respect to scheduling tasks over heterogeneous hardware (caused by multi-user contention).

## 4.2 Application benchmarks

To gain an understanding of how well our system performed on more realistic applications, we chose to benchmark `Terasort` and `Kmean` (Kmean is supplied by the Apache Mahout library). Terasort is designed to sort a set of input data from storage and output the sorted results back to storage. The actual input consists of a set of 100 byte records. This is a useful operation for a variety of data analytic applications that may need to sort data for pre-processing (i.e., extract-transform-load), or for business intelligence. In addition, Terasort combines both I/O and compute-bound processes. Kmean is a popular machine learning algorithm that iteratively clusters a set of sample data. Like Terasort, this benchmark also reads and writes to storage. The Kmean benchmark, however, tends to be more computationally intensive.

Figure 7a illustrates the average running time of Terasort for three different sizes of data sets. As the input sizes were increased, the number of nodes was also increased. Ideally, the running time would be proportional to the ratio of worker nodes to input size. However, this did not appear to be the case for our set of results. Indeed, overall running time approximately doubled from 50 to 100 GB even though the number of nodes was proportionally doubled from 16 to 32. The running time drastically increased as the data set size was increased to 500 GB (approximately 7x for the lower bound) using 48 nodes. This increase is partly explained by the large number of concurrent map tasks for this application (1562 for 50 GB, 3125 for 100 GB, and 15625 for 500 GB). As we have seen before (Figure 6), the process rate is bounded by the maximum number of concurrent tasks. An observation to note, however, is that overall execution time across all three input sizes were reasonable.

Another noticeable trend was the increase in variation as the number of slave nodes was increased. For example, with 48 slave nodes, the average running time was 119 minutes with a standard deviation of 30% of the mean. We hypothesize that this is caused by increased contention for shared storage resources. As the data set size increases, the number of disks used also increases, leading to additional contention with other users. In contrast, in local storage environments, the degree of disk contention is bound by the number of local tasks. Thus, choosing the right number of workers may be more important in shared storage environments.

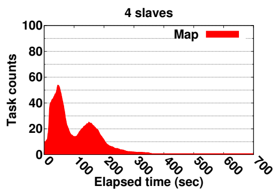
The Kmean benchmark is iterative in nature (each iteration refines the cluster membership). For our experiments, each run consisted of five iterations clustering 580 MB of input data. Figure 7b shows the average running times of Kmean clustering while varying the number of nodes (8, 16, and 32 nodes). Surprisingly, the overall running time was not affected by the increased number of nodes (all averaged approximately 50 - 60 minutes). This is because the Kmean benchmark is primarily memory-bound. So with 580 MB of input, 8 nodes provided sufficient memory capacity. Increasing the capacity did not necessarily help. This leads to an interesting observation; with shared-nothing architectures, organizations are often forced to increase the number of nodes to increase the overall storage capacity. If these nodes are used, this might lead to increased communication with limited benefit to users. However, in a shared storage environment, we are free to provision the number of compute nodes independent of storage. For example, in this experiment, we increased the number of slave nodes to provide a larger memory footprint, not to increase storage capacity.

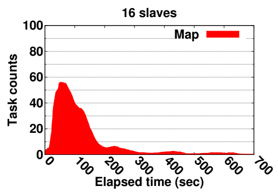
## 5. DISCUSSION

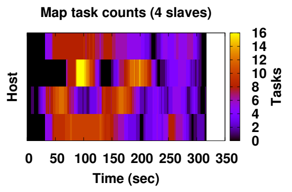
As the results in § 4 have shown, the overall performance of Hadoop over shared storage is promising but also indicate some potential performance issues. In this section, we examine some of these issues in closer detail. We examine the effects of task scheduling and interaction between tasks. We also examine the effect of applying some Lustre-specific optimizations.

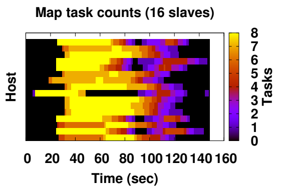
## 5.1 Performance impact from task schedules

In order to inspect the behavior of the Hadoop scheduler, we ran TestDFSIO benchmarks on 4 and 16 worker nodes reading 128 files in parallel. We show the number of map tasks per time step for the entire cluster and for each slave node in Figure 8. Since TestDFSIO consists primarily of a single map stage, we graphed the number of concurrent map tasks over time (Figure 8a and 8b). As the maximum number of map tasks per node is limited to 16, the four-slave case consisted of two distinct "waves" of map tasks. Using more nodes collapsed the two waves into a single wave. However, due to the scheduling startup delay, not all of the









(a) read 128 files/ 4 slaves (b) read 128 files/ 16 slaves (c) read 128 files/ 4 slaves (d) read 128 files/ 16 slaves

Figure 8: Task flow of TestDFSIO: when processing 128 map tasks, with a total of 64 task slots (16 per slave node), the four-slave cases showed two distinctive waves of mappers in (a) and (b). The 16-slave cases had one wave of mappers, but did not launch all 128 mappers at the same time, shown in (c).

128 files were accessed at the same time. Although $16 \times 16 = 196$ slots are available for map tasks, less than 60 files were accessed at the peak.

We also examined the distribution of tasks across hosts (Figures 8c and 8d). Overall, most hosts were synchronized their tasks (thus leading to fewer stragglers). However, the delayed startup time is clearly visible (as black areas in the figures). We argue that synchronizing the tasks (either by synchronized starts or dynamic scheduling) will lead to better overall performance. Also, the use of shared storage leads to fairly balanced task distributions, since tasks no longer must be tightly coupled to local data.

## 5.2 Interactions among tasks

We also performed a similar analysis with Terasort, which contains shuffle and reduce tasks. For the most part the map tasks were fairly well synchronized across nodes (Figure 9). We also observe that the running time of the application primarily depended on the running time of the map tasks. This is illustrated by the few (approximately 15) map tasks that took longer to complete.

Although the effects of these “stragglers” was partially mitigated by the overlapping reduce tasks, solving the straggler problem has been an active area of research for some time [39]. Several techniques have been proposed, but operating in a shared storage environment poses some unique challenges. In a shared-nothing architecture, the degree of sharing secondary storage among map tasks is limited by the number of tasks on a single node. However, in a shared storage environment, all the tasks in the cluster will access the same storage. Thus, a long running map task may interfere with all successive map tasks, making the application more vulnerable to straggler tasks. Simultaneously, however, shared storage presents some opportunities to proactively mitigate stragglers. For example, by starting additional tasks on other nodes to access the same data blocks (if the straggler is caused by computational interference) or even by virtually splitting a block (by seeking into a file).

## 5.3 Lustre optimization

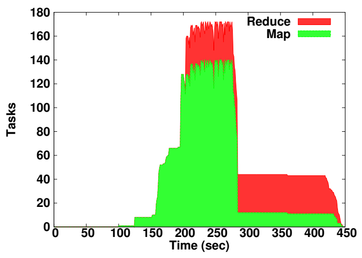
Lustre provides the ability to vary the number of associated storage targets, using the *stripe count*. It is important to note that on our Spider filesystem, a single OST consists of multiple, parallel SAS tracks of disks, instead of a single disk. Consequently, even a stripe count of 1 may target multiple disks on a single OST. Although stripe count is a Lustre-specific parameter, the concept can be generalized to represent the degree of parallelism when accessing remote storage devices. For this experiment, we ran Terasort for a 150 GB data set on 16 slave nodes while varying the stripe count. We used the same settings with prior Terasort experiments, the input consisting of 77 files. After running the workload, the reduce tasks outputted 70, 2 GB files. During the execution, Hadoop launched a total of 4466 map tasks (32 MB per a map task and 60 map tasks per input file). The overall running times and 95% confidence intervals are shown in Figure 10b.

We observe that performance variability increased drastically with increased stripe count. Indeed, the best performance for this particular appliation was with a stripe count of 1. This behavior results from the interference within the Terasort application and between applications in the system, as illustrated in Figure 10a. Even though increasing stripe count increases bandwidth to the entire shared storage, the probability of interference also increases. In addition, since the block size from a map task is small (32 MB), one OST in our system already provides sufficient bandwidth. We expect that the optimal stripe count value will be tightly dependent on the conjunction of the system hardware, the workload, and other workloads in the system. We expect to explore this issue in further detail in future work.

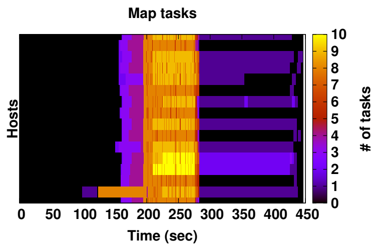
We also note that these parameters (i.e., stripe count) may vary on a per-application basis, and that HPC users typically configure their software to maximize I/O throughput (using a combination of stripe count, stripe size, and data layout). Because these parameters may also affect the data analysis time, users will also need to consider data analysis needs when configuring their applications. Exploring this issue further is a topic for future work.

## 6. CONCLUSIONS

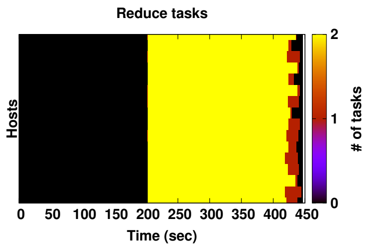
We presented the design and evaluation of an on-demand Hadoop service running over a shared, parallel filesystem. Our filesystem is used simultaneously by many users across several compute clusters of varying sizes. We have shown that running Hadoop in this environment produces strong results with some benchmarks, mostly I/O-bound workloads,



(a) total

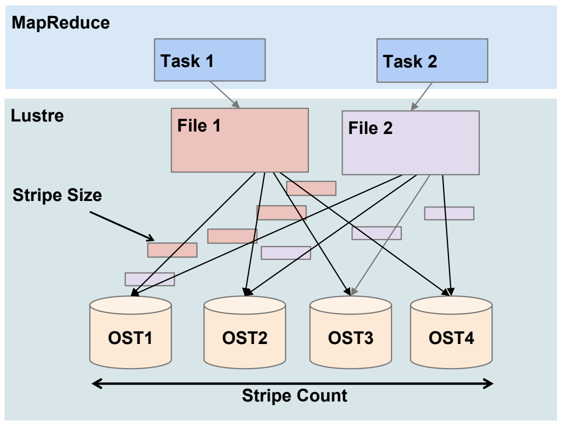


(b) map

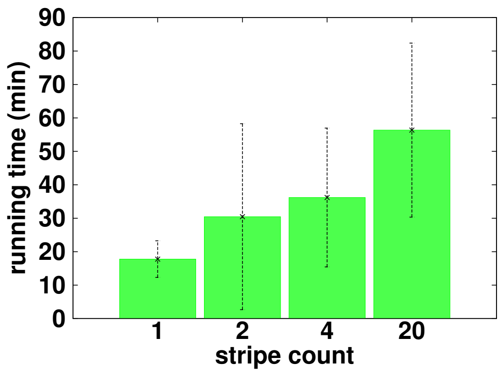


(c) reduce

Figure 9: Distribution of tasks over time: Terasort 50 GB (16 slaves), (a) shows total tasks in the cluster, (b) and (c) show tasks per host.



(a) File accesses in MapReduce over Lustre



(b) Terasort: 150GB/16 slaves

Figure 10: For MapReduce over Lustre, interference in storage accesses will be connected to the stripe count (a), which may contribute to the increase in the average running time and the variance(b).

performing very well. Besides the potential performance benefits, operating Hadoop in a shared storage environment simplifies data management, since users no longer have to worry about transferring data across different storage systems. In addition, we have also shown that there are real challenges of operating Hadoop in this environment. Specifically, variability in the filesystem from resource contention for shared resources can severely degrade overall performance. We believe that these initial experiences with Hadoop over a shared filesystem demonstrate that high-performance computing environments can be used for large-scale data analytics in an efficient manner. Indeed, we expect our tool to become widely used for this purpose at Oak Ridge National Laboratory.

## 7. ACKNOWLEDGMENTS

This manuscript has been authored by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes.

## 8. REFERENCES

[1] Amazon elastic mapreduce. `http://aws.amazon.com/elasticmapreduce`.
[2] Amazon simple storage service. `http://s3.amazonaws.com`.
[3] Apache hive: a data warehouse system for hadoop. `http://hive.apache.org`.
[4] Apache mahout: scalable machine learning and data mining. `http://mahout.apache.org`.
[5] Apache pig: a platform for analyzing large data sets. `http://pig.apache.org`.
[6] Pvfs2: Parallel virtual file system, version 2. `http://www.pvfs.org`.
[7] Titan. `http://www.olcf.ornl.gov/titan`.
[8] Torque resource manager. `http://www.adaptivecomputing.com/products/open-source/torque/`.
[9] Virtual hadoop. `http://wiki.apache.org/hadoop/VirtualHadoop`.
[10] Grid engine hadoop integration, 2011. `http://gridscheduler.sourceforge.net/howto/GridEngineHadoop.html`.

[11] Running hadoop with lustre, 2011. http://wiki.lustre.org/index.php/Running_Hadoop_with_Lustre.

[12] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, Jan. 2008.

[13] D. A. Dillow, G. M. Shipman, S. Oral, Z. Zhang, and Y. Kim. Enhancing i/o throughput via efficient routing and placement for large-scale parallel file systems. In *Proceedings of the 30th IEEE International Performance Computing and Communications Conference*, PCCC '11, 2011.

[14] C. Engle, A. Lupher, R. Xin, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: fast data analysis using coarse-grained distributed memory. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, 2012.

[15] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. 2003.

[16] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. Vl2: a scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, SIGCOMM '09, 2009.

[17] A. Gulati, G. Shanmuganathan, I. Ahmad, C. Waldspurger, and M. Uysal. Pesto: online storage performance management in virtualized datacenters. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, 2011.

[18] T. Hoefler, A. Lumsdaine, and J. Dongarra. Towards efficient mapreduce using mpi. In *Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 2009.

[19] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *IEEE 26th International Conference on Data Engineering Workshops (ICDEW),*, 2010.

[20] S. Krishnan, M. Tatineni, and C. Baru. myhadoop – hadoop-on-demand on traditional hpc resources. Technical Report SDSC-TR-2011–2, San Diego Supercomputer Center, 2011.

[21] R. McDougall. Towards an elastic elephant: Enabling hadoop for the cloud, 2012. http://cto.vmware.com/towards-an-elastic-elephant-enabling-hadoop-for-the-cloud.

[22] M. Mihailescu, G. Soundararajan, and C. Amza. Mixapart: decoupled analytics for shared storage systems. In *Proceedings of the 4th USENIX conference on Hot Topics in Storage and File Systems*, HotStorage'12, 2012.

[23] E. B. Nightingale, J. Elson, J. Fan, O. Hofmann, J. Howell, and Y. Suzue. Flat datacenter storage. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, OSDI'12, 2012.

[24] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. Portland: a scalable fault-tolerant layer 2 data center network fabric. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, SIGCOMM '09, 2009.

[25] ORNL. Smoky. http://www.olcf.ornl.gov/computing-resources/smoky.

[26] ORNL. Spider. http://www.olcf.ornl.gov/kb_articles/spider.

[27] B. Palanisamy, A. Singh, L. Liu, and B. Jain. Purlieus: locality-aware resource allocation for mapreduce in a cloud. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, 2011.

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

[29] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. Network file system (nfs) version 4 protocol, 2003.

[30] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, 2010.

[31] A. Singh, M. Korupolu, and D. Mohapatra. Server-storage virtualization: Integration and load balancing in data centers. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, SC 2008, 2008.

[32] W. Tantisiriroj, S. W. Son, S. Patil, S. J. Lang, G. Gibson, and R. B. Ross. On the duality of data-intensive file system design: reconciling hdfs and pvfs. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, 2011.

[33] VMware. vstorage vmfs, 2011. http://www.vmware.com/files/pdf/VMware-vStorage-VMFS-DS-EN.pdf.

[34] F. Wang, S. Oral, G. Shipman, O. Drokin, T. Wang, and I. Huang. Understanding lustre filesystem internals. Technical Report ORNL/TM-2009/117, Oak Ridge National Lab, 2009.

[35] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th USENIX conference on Operating systems design and implementation*, OSDI'08, 2006.

[36] J. Wilkes. Traveling to rome: Qos specifications for automated storage system management. In *Proceedings of the 9th International Workshop on Quality of Service*, IWQoS '01, 2001.

[37] H. Yang, Z. Luan, W. Li, and D. Qian. Mapreduce workload modeling with statistical approach. *Journal of Grid Computing*, 10(2), June 2012.

[38] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, HotCloud'10, 2010.

[39] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, 2008.