2014 IEEE 28th International Parallel & Distributed Processing Symposium

# Characterization and Optimization of Memory-Resident MapReduce on HPC Systems

Yandong Wang* Robin Goldstone† Weikuan Yu* Teng Wang*

Auburn University* Lawrence Livermore National Laboratory†
{wangyd,wkyu,tzw0019}@auburn.edu {goldstone1}@llnl.gov

***Abstract*—MapReduce is a widely accepted framework for addressing big data challenges. Recently, it has also gained broad attention from scientists at the U.S. leadership computing facilities as a promising solution to process gigantic simulation results. However, conventional high-end computing systems are constructed based on the compute-centric paradigm while big data analytics applications prefer a data-centric paradigm such as MapReduce. This work characterizes the performance impact of key differences between compute- and data-centric paradigms and then provides optimizations to enable a dual-purpose HPC system that can efficiently support conventional HPC applications and new data analytics applications. Using a state-of-the-art MapReduce implementation *Spark* and the Hyperion system at Lawrence Livermore National Laboratory, we have examined the impact of storage architectures, data locality and task scheduling to the memory-resident MapReduce jobs. Based on our characterization and findings of the performance behaviors, we have introduced two optimization techniques, namely *Enhanced Load Balancer* and *Congestion-Aware Task Dispatching*, to improve the performance of Spark applications.**

## I. Introduction

One grand challenge faced by our society is a deluge of digital data, so called *Big Data*. According to the 2011 IDC report [1], a total of 1,220 exabytes of data was created and replicated on earth in 2010. IDC estimated that the volume of digital data will continue to grow at an annual rate of 50%, *i.e.*, the amount of data is expected to reach more than 8 zettabytes by 2015. To cope with the data deluge challenge, the past few years have witnessed rapid development of big data analytics frameworks [2], [3], [4], [5], [6]. Among them, MapReduce [2] has achieved widespread success.

Many organizations have been embracing MapReduce and deploying its different implementations such as Hadoop [3], Dryad [4], and Spark [5] to meet their needs of massive computation and analysis of enormous datasets, thereby mining critical knowledge for their business.

In this modern rush for gold from data, different organizations are facing very different considerations when it comes to a decision on their data analytics systems. With the prevalence of cloud platforms and commercial computing services, many customers can leave that decision to their system providers. But the system providers really have to juggle between two choices: should they construct from scratch dedicated systems for data analytics, or should they evolve their systems to meet the demands of data analytics applications while continuing to support existing applications and customers? The latter is a particularly perplexing situation faced by the users and administrators at the leadership computing facilities who have been relying on traditional HPC (High-Performance Computing) systems for their scientific applications. Along with this dilemma is that there is a hidden paradigm shift along with the emergent focus on big data. For the first few decades of computer history, computing power has been a scarce resource. Thus conventional systems are constructed based on a compute-centric paradigm while the grand objective is to aggregate as much computing power as possible in terms of the number of floating-point operations per second. The need to analyze big data has actually pushed the transition of computer systems into a data-centric paradigm for which the grand objective is to attain the fastest analytics power in terms of the number of bytes and records processed per second.

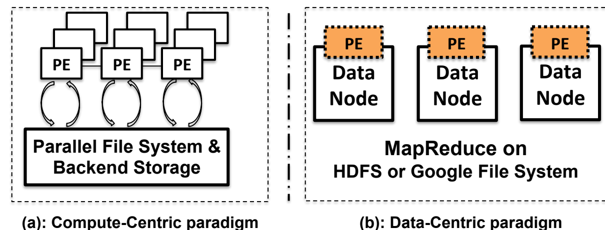



Fig. 1: Data-centric and compute-centric paradigms.

Fig. 1 shows a comparison between compute- and data-centric paradigms. There are two key distinctions between these paradigms. First, there is a key difference on the placement of compute and storage resources. The conventional compute-centric paradigm has separated compute and storage resources in the form of a computer cluster and a parallel file system that are connected via high speed networks. In contrast, the data-centric paradigm provides co-located compute and storage resources on the same node. Second, there is a key difference in terms of the impact of task scheduling and data placement. In the compute-centric paradigm, tasks on compute nodes are, in general, equally distant from the backend storage system. HPC systems are Typical manifestations of this paradigm. In the data-centric paradigm, tasks have strong affinity to the nodes containing their datasets. (Note that we are aware of some hierarchical compute-centric systems such as BlueGene series on which physical locations of compute nodes affect the speed of storage access.) Because of these distinctions, on compute-centric paradigm, applications sharing

1530-2075/14 $31.00 © 2014 IEEE
DOI 10.1109/IPDPS.2014.87



IEEE computer society

the same data often involve repetitive data movement between the computing resource and the storage backend. In contrast, the data-centric paradigm provides co-located compute and storage resources on the same node to facilitate locality-oriented task scheduling. By scheduling computing tasks to where data resides, data movement can be minimized for applications sharing the data.

These distinctions between compute- and data-centric paradigms have significant performance implications to different types of application workloads. For system providers who are eager to support more MapReduce-based analytics applications on HPC platforms, it is imperative to characterize the performance of key architectural components in these two different paradigms. Particularly, how does the configuration of storage resources such as parallel file systems affect job scalability and throughput? What is the impact of data placement and task scheduling? And how to reconcile and converge the architectural differences between the two paradigms so that one system can be configured and tuned for productive sharing by both conventional HPC applications and the emergent MapReduce-based analytics applications.

In this paper, we undertake an effort with intensive experiments to characterize the performance, identify the inefficiency of a MapReduce-based framework on the compute-centric paradigm, and compare its performance with that on the data-centric paradigm. Accordingly, we also introduce several optimizations targeting at compute-centric HPC systems.

Among many MapReduce frameworks, we have chosen Spark [5], which is a memory-resident implementation shown to outperform Hadoop for many applications by orders of magnitude [5], [7]. We leverage the Hyperion [8] system at Lawrence Livermore National Laboratory with two distinct configurations: one under the compute-centric paradigm and the other under the data-centric paradigm.

In summary, we conduct a comprehensive investigation to characterize the performance critical aspects of compute- and data-centric paradigms and shed light on how to build a dual-purpose HPC system to enable fast data analytics. We have made the following contributions in this research.

- We have studied the impact of storage architecture to the performance of different types of MapReduce jobs, and revealed that their performance on HPC systems is highly dependent on their computation intensity.
- We have characterized the importance of intermediate data placement and the benefits of hierarchical storage media to Spark applications. Particularly, we show that MapReduce applications need to be aware of the performance implications of storage consistency mechanisms on HPC systems and avoid the cascading effects of lock contention from HPC file systems such as Lustre.
- We have evaluated the impact of locality-oriented scheduling techniques for MapReduce jobs on compute-centric HPC systems. We show that maximizing data locality is not so critical, and delay scheduling [9], a popular strategy to delay tasks for data locality can even cause performance degradation.
- We have introduced two optimization techniques: Enhanced Load Balancer and Congestion-Aware Task Dispatching. The former takes into account performance variation and imbalanced data distribution when scheduling tasks, resulting an improvement of 26% on job execution time. The latter recognizes the existing oblivity of Spark to new storage devices such as SSD, throttles the launch of Spark tasks and mitigates the congestion, thereby achieving a performance gain up to 41.2%.

## II. Comparison Between Compute-Centric and Data-Centric Paradigms

In this section, we provide a direct comparison between the compute-centric and data-centric processing paradigms.

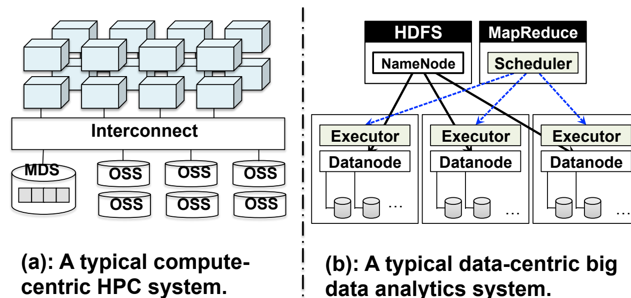



Fig. 2: Detailed comparison between compute- and data-centric paradigms.

### A. HPC Systems Representing the Compute-Centric Paradigm

Fig. 2(a) shows a diagram of typical compute-centric HPC systems. The core of such systems consists of a large collection of compute nodes, *i.e.,* processing elements (PEs), which offer the bulk of computing power. Via a high-speed interconnect, these PEs are connected to a parallel file system from the storage backend for data I/O. Lustre is a typical file system used on HPC systems. It is a POSIX-compliant, object-based parallel file system, offering parallel I/O services to the clients (PEs) through a MetaData Server (MDS) and many Object Storage Servers (OSSes).

Lustre provides fine-grained parallel file services with its distributed lock management. To guarantee file consistency, it serializes data accesses to a file or file extents using a distributed lock management mechanism. Because of the need for maintaining file consistency, all processes first have to acquire locks before they can update a shared file or an overlapped file block. Thus, when all processes are accessing the same file, their I/O performance is dependent not only on the aggregated physical bandwidth from the storage devices, but on the amount of lock contention among them as well.

### B. Spark – A Representative of Data-Centric Paradigm

MapReduce frameworks distribute computation map and reduce tasks among a number of slave nodes. Reduce tasks consolidate and transform the intermediate data generated by tasks from the previous map phase. Spark is a recent, highly popular MapReduce implementation. It consists of two categories of components: a scheduler and many executors. The

scheduler is in charge of scheduling tasks, monitoring their progress, and fault handling through task re-execution. The executors are responsible for executing the actual computing and data processing tasks. As many MapReduce implementations, Spark usually works together with distributed file systems that are designed to co-locate the storage resource (*i.e.,* DataNode) with the compute resources (*i.e.,* tasks launched by Executors) as shown in Fig. 2(b). For example, Spark relies on the HDFS [10] to manage the flow of data. HDFS is composed of a master NameNode and many slave DataNodes. Google's MapReduce has a similar reliance on the Colossus, the latest version of Google file system. Such co-localization of DataNodes and Executors realizes a data-centric computing model to minimize data movement between computation tasks and the storage system.

### *C. Memory-Resident Resilient Distributed Datasets in Spark*

Compared to other MapReduce implementations such as Hadoop [3], Spark provides two key features. First, Spark leverages the distributed memory from all slave nodes to store most intermediate data during job execution and the final execution results at job completion. By doing so, it avoids the file system, retaining most data resident in distributed memory across phases in the same job and/or different jobs. Such *memory-resident* feature benefits many applications such as *machine learning* or *iterative algorithms* that require extensive reuse of results among multiple MapReduce jobs. Second, Spark introduces *resilient distributed datasets* (RDDs) to facilitate the programming of parallel applications. Each RDD represents a collection of data partitions that spread across the cluster. A rich set of operations are provided to manipulate RDDs (*e.g.*, *map*, *flatMap*, *groupBy*, and *reduce*, *etc*). Overall, those operations can be categorized into two types, which are *transformation* and *action*, respectively.

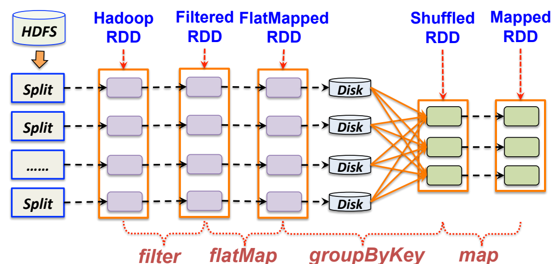



Fig. 3: MapReduce processing pipeline via using RDDs.

A transformation converts a source RDD to a destination RDD by applying *User-Defined Functions* (UDF) to each partition contained in the former. Fig. 3 illustrates an example of a Spark MapReduce job, in which an HDFS file is transformed to the final *MappedRDD* through four transformations: *filter*, *flatMap*, *groupByKey* and *map*. When Spark is deployed on a cluster featuring compute-centric paradigm, *HadoopRDD* can be replaced by system dependent RDD, such as *LustreRDD*, to retrieve input from HPC parallel file system.

Spark's actions include *reduce*, *count*, *collect*, *etc*. An action triggers Spark to construct an execution plan represented internally as a *directed acyclic graph* (DAG) that consists of multiple stages. Each stage includes many transformations that can be pipelined. Stages are connected through the shuffle operations for intermediate data shuffling. An implicit stage is embedded into the DAG for every shuffle operation. For example, *filter* and *flatMap* in Fig. 3 are grouped into a same stage, while the *groupByKey* is in an independent stage. Sparks launches stages within the DAG in a serialized manner.

The shuffling of intermediate data is a major performance bottleneck of MapReduce implementations, including Spark. However, such shuffle operation widely exists in many critical operations, such as *join*, *reduceByKey*, and *groupBy*, *etc*. To avoid substantial overhead and provide reliable job execution, Spark materializes partitions onto the local file system. When a shuffle operation is encountered, Spark will undertake two phases for moving intermediate data: storing and shuffling. In the storing phase, Spark schedules a round of *ShuffleMapTasks* to flush in-memory output from the previous stage to the file system. Then in the shuffling phase, a *ShuffledRDD* is introduced to transfer the intermediate data across the network.

## III. Methodology

### *A. Experimental Testbed*

TABLE I: List of key Spark configuration parameters.

| Parameter Name | Value |
|---|---|
| spark.reducer.maxMbInFlight | 1GB |
| spark.rdd.compress | false |
| spark.shuffle.compress | true |
| spark.buffer.size | 8MB |
| spark.default.parallelism | application dependent |

Unless otherwise specified, our experiments are carried out on the Hyperion cluster [8] with 101 compute nodes at Lawrence Livermore National Laboratory. One node serves as the master of the Spark and the NameNode of HDFS. Each compute node is equipped with two 2.60*GHz* Intel E5-2670 processors (16 cores per node) and 64 GB of RAM. We allocate 30 GB per node for Spark jobs and reserve 32 GB for RAMDisk. On each node, there is one SATA-based SSD of 128 GB storage space mounted via *ext4* file system. Its peak sequential write and read bandwidths reach 387 MB/sec and 507 MB/sec, respectively. All compute nodes span across two racks and are fully connected through InfiniBand QDR which delivers up to 32 Gpbs link bandwidth. A centralized Lustre file system providing 47 GB/sec aggregated bandwidth is mounted on all the compute nodes.

All compute nodes run Linux 2.6.32 kernels. Spark 0.7.0, along with Scala 2.9.2 and Oracle Java 1.7.0 are used. The HDFS block size is set as 128 MB. We have also carefully tuned Spark on Hyperion. Table I summarizes main parameters that have noticeable performance impact. For all tests, we report the median of five test runs.

### *B. Benchmarks*

We have selected three representative benchmarks including *GroupBy*, *Grep*, and *Logistic Regression* (LR). They are described as follows.

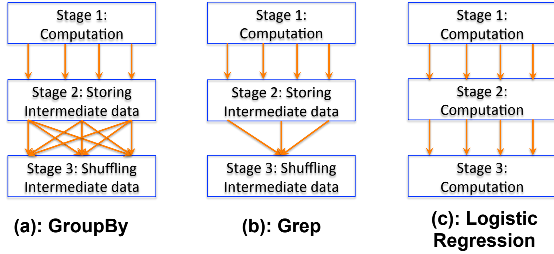
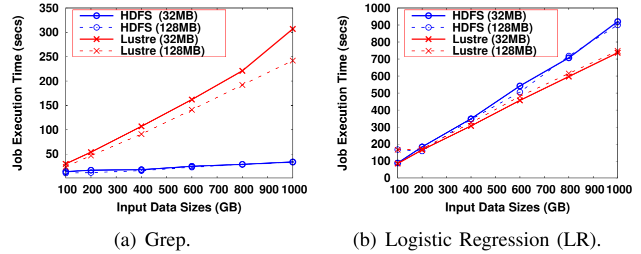



Fig. 4: Execution plans of three representative benchmarks.




(a) Grep. (b) Logistic Regression (LR).

Fig. 5: Performance of retrieving inputs from HDFS and Lustre.

***GroupBy*** is a critical operation used by many applications, including *kMeans*, *wordcount*, and *calculating transitive closure of a graph*, *etc*. It helps reveal the pattern of shuffle operations. Fig. 4(a) depicts the execution plan of GroupBy. It consists of three stages. In the first computation stage, each task generates <key, value> pairs in memory. In the second stage, Spark schedules ShuffleMapTasks to partition the intermediate data and store them into the file systems. In the last stage, fetching tasks shuffle intermediate data over the network. Across such data processing pipeline, the intermediate data size is equal to the input size.

***Grep*** searches a string that matches a regular expression from a set of documents. It represents a wide range of data analytics applications, such as *logQuery* and *select*, *etc*. Grep's execution plan as shown in Fig. 4(b) bears some similarity to that of GroupBy. However, it generates much less intermediate data, requiring very little shuffling of data. Its intermediate data size ranges from 1 MB to 200 MB in our test cases.

***Logistic Regression*** **(LR)** is an iterative application that predicts the value of a vector according to a qualitative response model. It can leverage the strength of Spark in caching job results in memory. As shown in Fig. 4(c), we run three iterations for LR. Every iteration is translated into one Spark job that is executed in one stage. Multiple stages are not pipelined in this benchmark.

## IV. The Impact of Storage Architecture

As discussed in the introduction, the storage architecture is a key distinction between data- and compute-centric paradigms. In the data-centric paradigm computation tasks are co-located with the storage resources, while in compute-centric paradigm tasks need to access a separate storage subsystem via interconnect. In this section, we characterize the impact of storage architecture on MapReduce jobs. To have a storage architecture for the data-centric paradigm, we configure an HDFS file system with 32 GB RAMDisk as the storage for each DataNode on Hyperion. For the storage architecture of the compute-centric paradigm, we directly use the Lustre file system of Hyperion.

### A. Location of Data Source

Among the three benchmarks, both Grep and LR work with a varying amount of input data. But they differ significantly in terms of their analytics computation. Grep generates a small amount of intermediate data, for which shuffling is required; and LR does mostly computation. We run both benchmarks with their input coming from the compute-centric *Lustre*-based configuration and the data-centric *HDFS*-based configuration.

Fig. 5 shows the comparison of the job execution time of Grep and LR benchmarks for both configurations. Overall, we have observed that the extent of impact is highly dependent on the computational intensity of MapReduce tasks. For Grep jobs with low computation, such as simply scanning of the input, the Lustre configuration results in severe performance penalty. Fig. 5(a) shows that, with 32 MB split size, the compute-centric Lustre configuration performs up to 5.7× worse than HDFS on average. For the Lustre configuration, increasing the split size from 32 MB to 128 MB reduces the job execution time by 15.9% due to less scheduling overhead. But there is still a significant performance loss when running Grep on the compute-centric Lustre configuration.

On the contrary, for the computation-intensive jobs, such as multidimensional vector multiplication in LR, the cost of retrieving input from Lustre is not as significant as shown in Fig. 5(b). Furthermore, as shown in the figure, the Lustre configuration outperforms HDFS by 12.7% on average for a 32 MB split size. This improvement is consistent across different split sizes. The performance difference is caused by delay scheduling policy [9] adopted by Spark, which will be further analyzed in Section V-A.

Taken together, the impact of the storage architecture to MapReduce applications depends on the characteristic of the applications' computation tasks. For LR-type computation-intensive jobs, the impact is negligible. But for Grep-based jobs with low computation requirements, the compute-centric Lustre configuration negatively affects the performance.

### B. Location of Intermediate Data

The location of intermediate data is another critical issue. It directly determines the performance of intermediate data shuffling. To investigate this factor, we use the GroupBy benchmark that allows flexible tuning of the intermediate data size. During the evaluation, we run GroupBy and store the intermediate data to the two different storage configurations.

Fig. 7(a) illustrates the performance of GroupBy when intermediate data resides in different storage architectures. Overall, the data-centric HDFS configuration exhibits significant advantage over the compute-centric alternative. It outperforms the optimal Lustre case (Lustre-local) by up to 6.5× on average, and the improvement ratio increases linearly

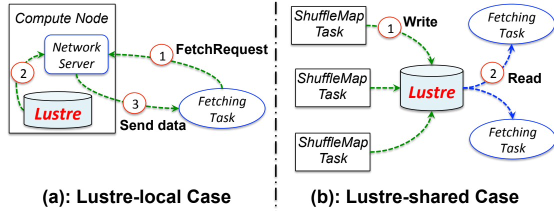
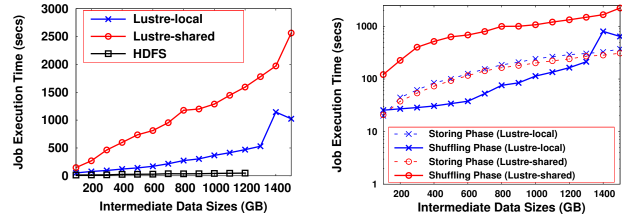



Fig. 6: Two approaches to use Lustre to conduct intermediate data shuffling.




(a) Job Execution Time of GroupBy. (b) Dissection of Lustre Cases.

Fig. 7: Performance when intermediate data resides in Lustre.

with the size of intermediate data. However, due to the limited storage spaces, HDFS can only support a maximum of 1.2 TB intermediate data size.

However, in many scenarios, compute nodes in HPC clusters are not equipped with any local persistent storage systems, for which placing intermediate data on the compute-centric Lustre-based storage is the only choice.

*Lustre-local* and *Lustre-shared*, as shown in Fig. 6, illustrate two approaches to use Lustre for intermediate data shuffling. In the Lustre-local case, fetching tasks that need to shuffle the intermediate data are unaware of the existence of the Lustre. Thus they initiate FetchRequests to the remote servers which in turn retrieve the data from their local Lustre directories and send them back across the network. However, since data retrieval from Lustre requires a movement over the network, Lustre-local can cause repetitive data movements, wasting the network bandwidth.

We have examined the alternative Lustre-shared approach, in which each fetching task directly retrieves intermediate data from Lustre. Although this approach seemingly addresses the issue of repetitive data movement within Lustre-local, it suffers from tremendous performance degradation due to file consistency ensured by Lustre.

Fig. 7(a) illustrates that Lustre-shared performs worse than Lustre-local by up to 3.8× with GroupBy benchmark. The detailed dissection in Fig. 7(b) further reveals that, although the two approaches perform comparably in the data storing phase, the shuffling phase of Lustre-shared is inferior to that of Lustre-local by up to one order of magnitude. The main reason for the inferiority of Lustre-shared is that retrieving intermediate data written by remote servers incurs costly metadata operations at the OSSes due to the need to maintain the storage consistency.

In the Lustre-local approach, the server that handles the FetchRequests simply retrieves the intermediate data written by the tasks on the same node. Meanwhile, due to the effect of large buffer cache in a compute node, it is likely that those intermediate data and corresponding metadata, such as *write locks*, still reside in the local memory. Thus, they can be quickly retrieved to serve the FetchRequests without involving expensive internal operations of Lustre for maintaining the data consistency.

On the contrary, in the Lustre-shared case, each fetching task accesses Lustre to retrieve the data written by remote nodes. Such design requires the *Distributed Lock Manager* of Lustre to revoke the write locks. After lock revocation, intermediate data cached remotely is forced to be flushed to the OSSes before they become available to fetching tasks. This sequence of internal operations substantially delays the intermediate data movement. Furthermore, current Spark launches fetching tasks of a job simultaneously during the shuffling phase, forcing all the intermediate data to be flushed to the OSSes around the same time. As a result, such behavior can cause serious contention at Lustre, significantly degrading the performance of the shuffling phase.

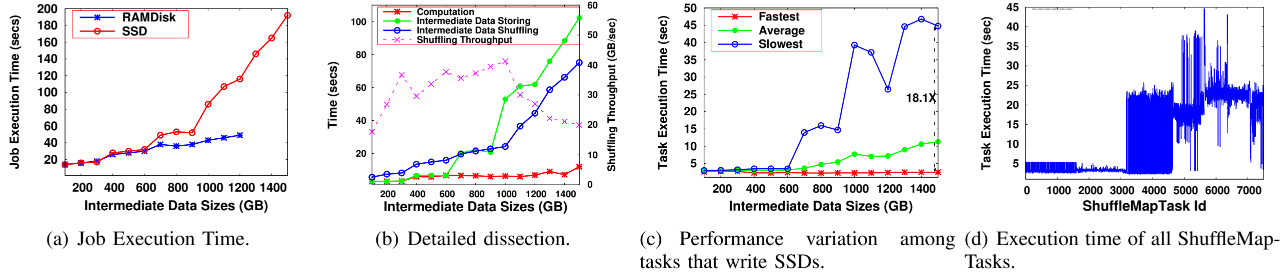
In summary, the data-centric HDFS configuration shows dramatic advantage over the compute-centric configuration when used for storing the intermediate data. In the compute-centric case with a shared file system such as Lustre, fetching tasks can avoid costly metadata operation for better performance if they are oblivious to the features of the shared file system.

### *C. Leveraging Solid State Disks for Intermediate Data*

Many HPC systems are embracing a hierarchical stack of different storage devices in order to support both data-centric and compute-centric paradigms, so that they can support both traditional HPC applications and emerging data analytics programs. A major effort to achieve such goal is the trend to integrate high-performance Solid State Drives (SSD) to the compute nodes. An immediate impact to MapReduce is that they can efficiently facilitate the processing of intermediate data. To understand such performance implication, we have conducted a set of experiments to study the performance impact of SSD on MapReduce jobs with similar data-centric HDFS configuration as that in Section IV-B. The performance of using RAMDisk as the local persistent storage is employed for performance comparison. We continue to use GroupBy as the benchmark for this study.

Fig. 8(a) presents the job execution time of GroupBy when intermediate data is stored on RAMDisk and SDD, respectively. Overall, using SSD for intermediate data achieves comparable performance as RAMDisk when the data size ranges from 100 GB to 600 GB due to the caching effects from the file system. Once the data size exceeds 700 GB, RAMDisk performs substantially better than SSD. Note that SSD can support jobs with much larger intermediate data sizes than RAMDisk due to the capacity advantage of SSD.

Fig. 8(b) further shows a detailed dissection of job execution time when SSD is employed. Data shuffling is shown as




(a) Job Execution Time.

(b) Detailed dissection.

(c) Performance variation among tasks that write SSDs.

(d) Execution time of all ShuffleMapTasks.

Fig. 8: Performance when SSD is used for storing the intermediate data and detailed analysis of tasks that write to and shuffle data from SSDs.

the key bottleneck when data size $\leq$ 600 GB, in which the throughput is bounded by the network bandwidth. When the data size is between 700 GB and 900 GB, the cache can no longer satisfy all the write operations during the storing phase. As a result, both storing and shuffling of intermediate data contribute equally to the job execution. When the data size increases further beyond 900 GB, we observe sharp drops on the performance of storing and shuffling phases due to the degraded performance of SSD write and read operations. In addition, the write performance falls more drastically than that of read. When the storing phase of intermediate data becomes the major bottleneck of job execution, the throughput of data shuffling then becomes SSD-bound.

### D. Inefficiency in Utilizing SSD

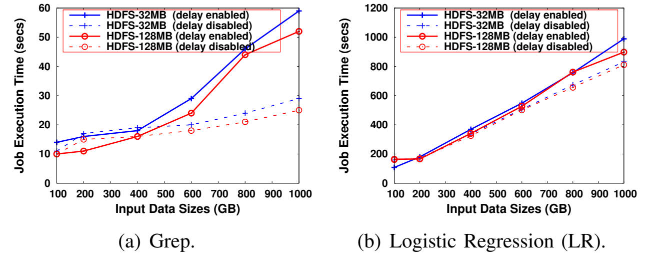
In our experiments with SSD, there is a significant performance variation among *ShuffleMapTasks* writing intermediate data to SSDs as shown in Fig. 8(c). The performance gap between the fastest and the slowest tasks can be as wide as 18$\times$ when the data size reaches 1.5 TB. On the contrary, the performance variation among shuffling tasks is moderate (not shown for brevity), indicating a mild interference among SSD read operations.

The dramatic variations among *ShuffleMapTasks* is because Spark aggressively launches tasks as they arrive in order to reduce the latency. This is oblivious to the congestion of underlying SSDs. When multiple data-intensive tasks are running and issuing a large number of write requests, such oblivity can result in substantial interference amony tasks. To gain insight into this issue, we have profiled the execution times of all ShuffleMapTasks in the 1.5 TB test case. We plot the execution times of these tasks based on the order of their launch time in Fig.8 (d). As shown in the figure, early tasks can take advantage of write buffer and clean blocks on SSDs. They can quickly complete their work. When the buffer gradually fills up and clean SSD blocks are depleted, internal operations for delayed write and garbage collection are activated. These operations start to interfere with the execution of ShuffleMapTasks. Thus we observe a degraded performance for Tasks ranging from 3100 to 4500. However, Spark is unaware of such interference and continues to insert tasks. This behavior further exacerbates the contention on the SSDs and leads to severer interference among Tasks from 4800 to 6400.

In summary, our study reveals that the lack of awareness on the unique features of SSD can lead to inefficient utilization of resource when in the storage of intermediate data. Fortunately, the inefficiency of congestion-oblivious write has also been documented by many prior studies on SSD [11], [12], [13]. In Section VI-B, we will demonstrate that an optimization using a throttling mechanism can effectively mitigate the interference and improve the storing phase by 41.2%.

## V. The Impact of Data Locality and Task Scheduling

### A. Locality-Oriented Scheduling




(a) Grep.

(b) Logistic Regression (LR).

Fig. 9: Performance degradation caused by delay scheduling.

Maximizing data locality has been a critical objective of MapReduce schedulers [9], [14], [15]. *Delay scheduling* [9], adopted by Spark, is a notable effort for obtaining high data locality for MapReduce frameworks in the environments where network bandwidth is a scarce resource. Using the same compute- and data-centric configurations as described in Section IV, we conduct an experiment to characterize the importance of locality-oriented scheduling.

Fig. 9 shows the experiment results when we activate delay scheduling for the data-centric HDFS configuration. When the split size is equal to 32 MB, job execution time degrades by 42.7% and 9.9% on average for Grep and LR, respectively. Similar degradation occurs for other split sizes as well. In contrast, with the compute-centric Lustre configuration, tasks can be immediately launched on available compute nodes since there is no locality constraint. All the computation tasks are roughly at the same distance from storage resources. Thus, compared to the data-centric configuration that favors the use of delay scheduling for better data locality of tasks, this setting can benefit the computation-intensive MapReduce jobs as shown in Fig. 5(b),

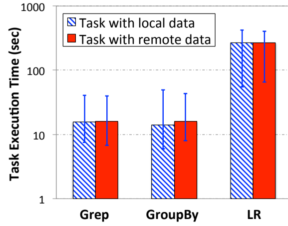



Fig. 10: Task execution time of three benchmarks.

In addition, Spark pipelines computation with data input, further diminishing any benefit of data locality. Fig. 10 demonstrates such argument. It shows the comparison of average task execution times along with maximum and minimum values of three different benchmarks. *"Task with local data"* denotes that the data input is obtained locally, while *"Task with remote data"* indicates the data input from remote servers. As shown in the figure, enforcing tasks to achieve 100% locality provides little performance gain for all three benchmarks.

Taken together, our evaluation and characterization of locality-oriented scheduling for the compute- and data-centric configurations suggest that (1) scheduling for good locality may not be effective in improving the performance of MapReduce jobs in HPC environments, and (2) introducing delays for better task locality is even detrimental on compute-centric systems because of the uniform reachability of storage resources to all computation tasks.

### *B. Load Balance of MapReduce Tasks*

Although the compute nodes in a compute-centric environment are homogeneous, there exist performance variations among compute nodes due to the skew of workloads over time. As a result, fast nodes tend to be assigned with more tasks by the scheduler. When each of these tasks deposits a unit of intermediate data, fast nodes end up with much more data to shuffle or move. This leads to imbalanced distribution of intermediate data. When a shuffle operation is needed, such imbalanced distribution can cause straggler issue [16] that prolongs the ensuing I/O-intensive data storing and shuffling phases as depicted in Fig. 11.

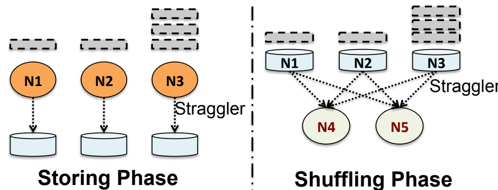
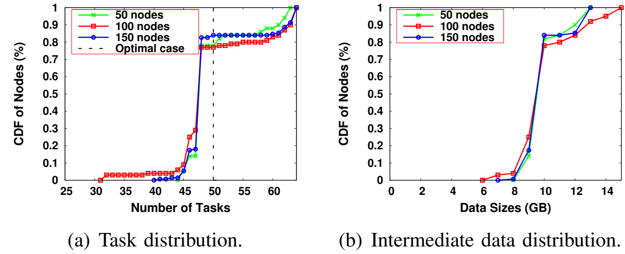



Fig. 11: Straggler issue caused by imbalanced intermediate data distribution during I/O intensive shuffle operation.

To investigate this issue, we use GroupBy as the benchmark with a split size of 256 MB. Three sets of experiments are conducted to run 2500 tasks on 50 nodes, 5000 tasks on 100 nodes, and 7500 tasks on 150 nodes, respectively. Fig. 12 (a) and (b) illustrate the *cumulative distribution functions* (CDF) of task and intermediate data distributions.




(a) Task distribution. (b) Intermediate data distribution.

Fig. 12: Unbalanced task assignment leads to unbalanced intermediate data distribution.

As shown in Fig. 12 (a), the workload among compute nodes varies substantially. In the case of 100 nodes, for the first 3% nodes at the head of the distribution, each machine only hosts 7 GB of intermediate data. While for the last 10% nodes at the tail of the distribution, each node accommodates more than 14 GB, i.e., $2\times$ of workload difference. Because the execution time of storing and shuffling phases are directly determined by the slowest tasks, those nodes with the most intermediate data can severely drag down the performance regardless of how fast other tasks have achieved.

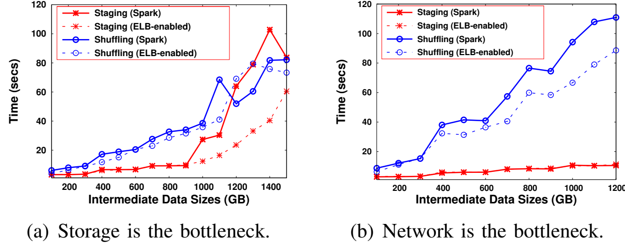
In summary, performance variations and workload skews on compute-centric systems can lead to imbalanced distribution of both MapReduce tasks and their intermediate data. Without an appropriate solution, such issue can hinder MapReduce systems from achieving the best performance on compute-centric HPC systems. We will demonstrate in Section VI-A that, by taking into account of the intermediate data size, the shuffle operation can be effectively accelerated.

## VI. Optimizations for Spark on Compute-Centric HPC Systems

Based on the characterization from Sections IV and V, we have shown that there are two performance issues that need to be addressed for the memory-resident Spark framework in order for it to be effectively supported by the compute-centric HPC systems. Firstly, the scheduler should take into account of the need to balance the intermediate data among compute nodes and mitigate the variations of task execution, thereby avoiding stragglers. Secondly, the MapReduce workers should be aware of the unique features of hierarchical storage devices such as SSDs to effectively utilize them. Accordingly, we introduce two optimizations: namely *Enhanced Load Balancer* (ELB) and *Congestion-Aware task Dispatching* (CAD), to address these issues.

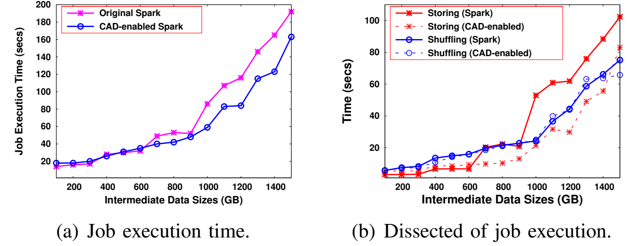
### *A. Enhanced Load Balancer (ELB)*

We design ELB to address the issue of imbalanced distribution of intermediate data. It considers the size of intermediate data generated by tasks before making further task assignment decision. When a job starts, ELB-enabled scheduler assigns tasks to the workers in a round-robin manner. During the job execution, ELB records the amount of intermediate data generated by each completed task and monitors the average data size among all nodes. When the size on a node goes beyond the average by a threshold (25% currently), ELB




(a) Storage is the bottleneck. (b) Network is the bottleneck.

Fig. 13: Dissection of GroupBy job execution time.




(a) Job execution time. (b) Dissected of job execution.

Fig. 14: Performance of Congestion-Aware task Dispatching.

notifies the scheduler to stop assigning more tasks to that worker node. Instead, it picks the nodes hosting the least amount of intermediate data to execute the pending tasks. Once the average size goes up, ELB resumes to assign more tasks to the original heavily loaded worker.

Although ELB can balance the size of intermediate data among compute nodes, two issues arise under such design. Firstly, ELB may conflict with the data locality, since the nodes hosting the least amount of intermediate data may not possess the input for the tasks. However, as shown in Section V-A, enforcing data locality has negligible impacts on the task execution time in the HPC environment. Thus it is desirable to trade off the locality of task scheduling for a balance of data distribution. Secondly, ELB can cause the idling of certain workers when they have completed their share of computation tasks, and the entire computation phase can be consequently delayed due to the slowest task. However, we have observed that the cost of waiting for the slowest computation task is much less than the cost of waiting for the slowest I/O tasks.

In this context, to demonstrate the performance improvement of ELB-enabled scheduler to communication and storage bottlenecks during the shuffle operations, the GroupBy benchmark is used. To create a scenario of storage bottlenecks, SSD is used as the local storage device. In Hyperion, we are not allowed to use other networks other than InfiniBand. So to create a scenario of network bottlenecks, we reduce the data size set in FetchRequest from 1 GB to 128 KB. Thus, many more requests are needed to shuffle the same amount of data, and the network bandwidth is consequently narrowed (due to space constraint, we only present the dissection of job execution and omit execution time of computation phases for clearness).

**Storage Bottlenecks**: Fig. 13(a) shows that when the data size $\leq$ 900 GB, Spark and ELB perform similarly. However, ELB outperforms Spark by 26% on average in terms of job execution time when the data size is between 1 TB and 1.5 TB. Such improvement is mainly attributed to the accelerated staging phase introduced by the ELB. When data sizes reach beyond 1 TB, Spark performs worse than ELB by 2.2$\times$ on average in the staging phase. On the contrary, computation phases from both remain nearly the same.

**Network Bottlenecks**: Spark performs 14.8% worse than ELB on average in terms of job execution time. Moreover, when the network is the bottleneck, unbalanced distribution has severe impact on small datasets, showing up to 17.5% degradation when data size is 400 GB. Such difference is strongly determined by the shuffling phases as shown in Fig. 13(b). On average, Spark shuffles data slower than ELB by 29.1% when the input size ranges from 400 GB to 1.2 TB.

Taken together, our ELB demonstrates that unbalanced distribution of intermediate data can prevent memory-resident Spark from achieving the optimal performance in the HPC environment.

### B. *Congestion-Aware Dispatching (CAD) of Tasks*

We design CAD as a feedback control algorithm that aims to mitigate the task interference when SSD is used as the storage device for intermediate data. CAD speculates the congestion status of SSD devices by monitoring the task execution time of completed ShuffleMapTasks. When a significant jump of execution time is detected, it throttles the dispatching of tasks by introducing a delay interval before each dispatching step. In the current design, we increase the interval by 50 ms whenever the average execution time increases by 2$\times$ (these are empirically chosen during our tuning process). Conversely, we reduce the interval accordingly when the average task execution time drops by half. Though simple, we have observed that such mechanism is effective in optimizing the SSD writes. This is because such delay interval allows more time for outstanding operations inside SSD to complete their work without worsening the congestion. In addition, it also provides more opportunity to group many small writes, which are harmful to SSD, thus further reducing the interference.

Fig. 14 compares the performance of the original Spark with our CAD-enabled Spark by using the GroupBy benchmark with different input sizes. Overall, CAD effectively accelerates the intermediate data storing phase once the data size goes beyond 600 GB. It achieves this without affecting another two phases as shown in Fig. 14(b). On average, CAD reduces the storing phase by up to 41.2% when the data size ranges from 700 GB to 1.5 TB. Such acceleration is reflected in the job execution time as shown in Fig. 14(a). The average improvement ratio reaches 19.8%.

## VII. Discussion

In this section, we summarize our major findings and discuss their implications to the design of future systems.

**The Impact of Storage Architecture**: Computation intensity of MapReduce tasks determines how much impact the storage architecture of HPC systems will have on the

job execution. For computation-intensive applications, there is little impact between the storage architectures of data- and compute-centric paradigms. In addition, there is no locality to the storage for compute nodes on compute-centric systems; tasks can be launched on any node with little loss of performance, or even better performance compared to the data-centric environment. However the data-centric paradigm still exhibits superior performance for applications with low computation intensity and high data intensity. Therefore, it is critical to consider the characteristics of MapReduce jobs before making data placement decisions. This is important for system providers in planning the evolvement of their compute-centric HPC systems for data-centric analytics applications.

In addition, the storage architecture may use distributed locking mechanism for maintaining file consistency, which can severely degrade the performance of intermediate data movement. So we show that designing shuffling mechanisms can avoid the cascading effects of locking contention and keep the efficiency of intermediate data shuffling. Users need to avoid a pitfall to use traditional HPC parallel file system as a bridge for fast storage of intermediate data.

When SSDs are used as the storage device for intermediate data, our analysis shows that Spark is currently incapable of utilizing them efficiently. Uncoordinated resource utilization can cause severe congestion on the device, leading to significant task interference, as also shown in [17]. Our findings suggest that comprehensive examinations are needed to assure the performance of MapReduce applications while evolving the underlying storage of a system to SSDs. Optimization strategies, such as *task throttling* as shown by our study, can be leveraged to improve the efficiency of SSD device utilization.

**The Effectiveness of Locality-Oriented MapReduce Schedulers in HPC Environment**: Our characterization reveals that, when a data-centric storage architecture is configured for computer nodes of an HPC system, MapReduce schedulers that strive for maximum data locality is not critical. Moreover, they may even hurt the performance by forcing a task delay for future opportunistic locality. We have also revealed that while HPC systems generally have homogeneous computer nodes, load imbalance can still arise. The current scheduler is oblivious to the size of intermediate data generated by computation tasks, leading to imbalanced data distribution that can cause many stragglers during shuffle operations. Our study demonstrates that such imbalanced distribution can cause suboptimal performance to MapReduce jobs. MapReduce applications on HPC systems shall not focus on locality-oriented task scheduling but other critical factors such as balancing distribution of intermediate data.

## VIII. Related Work

Spark is a critical cornerstone of *Berkeley Data Analytics Stack* (BDAS) [18] that aims to compete with the open-source Hadoop. It plays a pivotal role in many industry and academia projects [7], [19], [20], [21] *etc.* Shark [7] is a query processing framework on top of Spark. It compiles user-submitted SQL queries into Spark jobs and leverages optimization strategies commonly used in database systems to optimize the execution plan. Also coupled with Spark, BlinkDB [20] is another approximate query engine that trades query accuracy for response time so that it can delivery near instant response for interactive queries over massive scale datasets. Spark streaming [19] exploits the potential of Spark to process real-time streaming data. It partitions streaming computations into small-sized deterministic batch jobs to fit the computation model of Spark. Sparkler [21] optimizes the Spark to support large-scale matrix factorization more efficiently. It identifies a major inefficiency existing in current Spark's broadcast variable and introduces a *Carousel Maps* to spread large dataset via using distributed hash table. Our work is orthogonal to those efforts. In addition, Zaharia *et al.* have introduced LATE [22]. Ananthanarayanan *et al.* have introduced Mantri [16] and small job cloning [23] to mitigate the impact of stragglers. However, none of them considers the imbalanced intermediate data distribution issue.

Many parties have tried to incorporate MapReduce frameworks with distributed file systems for compute-centric paradigm. Ananthanarayanan *et al.* [24] evaluated MapReduce when it runs with HDFS and GPFS. Maltzahn *et al.* [25] studied the combination of Hadoop with Ceph file system. Panasas [26] is also delivering the support for Hadoop. Our analysis in this work provides researchers with the first hand data about absorbing MapReduce into compute-centric HPC paradigm that relies on above high-performance file systems.

Many efforts have been conducted to investigate the performance of HPC applications on data-centric cloud. Evangelinos *et al.,* [27] analyzed a scientific HPC application on Amazon EC2 and revealed that the performance of network in cloud is worse than that of HPC by one to two orders of magnitude. Gupta *et al.,* [28] observed similar performance on different cloud platforms. Though raw performance difference between compute-centric HPC and data-centric cloud is pronounced, Marathe *et al.,* [29] pointed out that queue wait time is another critical factor to consider when choosing which environment is the best for the applications. Our work stands on the other side of the spectrum by investigating the data-centric analytics framework on compute-centric paradigm.

## IX. Conclusions

While many existing HPC facilities are evolving new capabilities to support efficient analytics of big data, this research addresses an important question on how to support the traditional compute-centric paradigm for HPC applications and the emerging data-centric paradigm for big data analytics applications on the same HPC systems. We have examined the design and architecture of a state-of-the-art MapReduce framework – Spark – on HPC systems. Our work sheds light on the performance issues and design inefficiency when running Spark jobs on HPC systems with distinct data-centric and compute-centric configurations: In particular, we have investigated the impact of storage architecture, locality-oriented scheduling, and emerging storage devices to memory-resident MapReduce applications on HPC systems. Based on the

experimental results, our optimization techniques, including the Enhanced Load Balancer and the Congestion-Aware Task Dispatching, can efficiently improve the performance of Spark applications on HPC systems.

### Acknowledgments

We are very thankful to the anonymous reviewers for their insightful comments. This work is funded in part by an Intel grant, an Alabama Innovation Award, and by National Science Foundation awards 1059376, 1320016 and 1340947. This work was also performed under the auspices of the US Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-647813).

## REFERENCES

[1] "The 2011 Digital Universe Study: Extracting Value from Chaos." http://www.emc.com/collateral/demos/microsites/emc-digital-universe-2011/index.htm.

[2] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, pp. 107–113, Jan. 2008.

[3] "Apache Hadoop Project." http://hadoop.apache.org/.

[4] Michael Isard and Mihai Budiu and Yuan Yu and Andrew Birrell and Dennis Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *EuroSys* (P. Ferreira, T. R. Gross, and L. Veiga, eds.), pp. 59–72, ACM, 2007.

[5] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, NSDI'12, (Berkeley, CA, USA), pp. 2–2, USENIX Association, 2012.

[6] Y. Wang, X. Que, W. Yu, D. Goldenberg, and D. Sehgal, "Hadoop acceleration through network levitated merge," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, (New York, NY, USA), pp. 57:1–57:10, ACM, 2011.

[7] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica, "Shark: Sql and rich analytics at scale," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, (New York, NY, USA), pp. 13–24, ACM, 2013.

[8] "Hyperion Project." https://hyperionproject.llnl.gov.

[9] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling," in *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, (New York, NY, USA), pp. 265–278, ACM, 2010.

[10] 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, (Washington, DC, USA), pp. 1–10, IEEE Computer Society, 2010.

[11] "Bcache." http://bcache.evilpiepirate.org/.

[12] F. Chen, D. A. Koufaty, and X. Zhang, "Understanding intrinsic characteristics and system implications of flash memory based solid state drives," in *Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*, SIGMETRICS '09, (New York, NY, USA), pp. 181–192, ACM, 2009.

[13] K. Shen and S. Park, "Flashfq: A fair queueing i/o scheduler for flash-based ssds," in *Proceedings of the USENIX Annual Technical Conference*, USENIX ATC'12, (Berkeley, CA, USA), USENIX Association, 2013.

[14] W. Wang, K. Zhu, L. Ying, J. Tan, and L. Zhang, "A throughput optimal algorithm for map task scheduling in mapreduce with data locality," *SIGMETRICS Performance Evaluation Review*, vol. 40, no. 4, pp. 33–42, 2013.

[15] Y. Wang, J. Tan, W. Yu, X. Meng, and L. Zhang, "Preemptive reducetask scheduling for fair and fast job completion," in *Proceedings of the 10th International Conference on Autonomic Computing*, ICAC'13, June 2013.

[16] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris, "Reining in the outliers in map-reduce clusters using mantri," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, (Berkeley, CA, USA), pp. 1–16, USENIX Association, 2010.

[17] X. Li, Y. Wang, Y. Jiao, C. Xu, and W. Yu, "Coomr: Cross-task coordination for efficient data management in mapreduce programs," in *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '13, (New York, NY, USA), pp. 42:1–42:11, ACM, 2013.

[18] "Berkeley Data Analytics Stack." https://amplab.cs.berkeley.edu/software/.

[19] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters," in *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*, HotCloud'12, (Berkeley, CA, USA), pp. 10–10, USENIX Association, 2012.

[20] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica, "Blinkdb: queries with bounded errors and bounded response times on very large data," in *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, (New York, NY, USA), pp. 29–42, ACM, 2013.

[21] B. Li, S. Tata, and Y. Sismanis, "Sparkler: supporting large-scale matrix factorization," in *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT '13, (New York, NY, USA), pp. 625–636, ACM, 2013.

[22] 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, (Berkeley, CA, USA), pp. 29–42, USENIX Association, 2008.

[23] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Effective straggler mitigation: attack of the clones," in *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*, nsdi'13, (Berkeley, CA, USA), pp. 185–198, USENIX Association, 2013.

[24] R. Ananthanarayanan, K. Gupta, P. Pandey, H. Pucha, P. Sarkar, M. Shah, and R. Tewari, "Cloud analytics: do we really need to reinvent the storage stack?," in *Proceedings of the 2009 conference on Hot topics in cloud computing*, HotCloud'09, (Berkeley, CA, USA), USENIX Association, 2009.

[25] M. Carlos, M.-E. Esteban, K. Amandeep, J. N. Alex, S. A. Brandt, and W. Sage, "Ceph as a scalable alternative to the hadoop distributed file system," ;login' 10, USENIX Association, 2010.

[26] "Accelerating and Simplifying Apache Hadoop with Panasas ActiveStor." https://www.panasas.com/sites/default/files/uploads/docs/hadoop_wp_lr_1096.pdf.

[27] C. Evangelinos and C. N. Hill, "Cloud computing for parallel scientific hpc applications: Feasibility of running coupled atmosphere-ocean climate models on amazons ec2," in *In The 1st Workshop on Cloud Computing and its Applications (CCA)*, 2008.

[28] A. Gupta and D. Milojicic, "Evaluation of hpc applications on cloud," *Open Cirrus Summit*, vol. 0, pp. 22–26, 2011.

[29] A. Marathe, R. Harris, D. K. Lowenthal, B. R. de Supinski, B. Rountree, M. Schulz, and X. Yuan, "A comparative study of high-performance computing on the cloud," in *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, HPDC '13, (New York, NY, USA), pp. 239–250, ACM, 2013.