Parallel Computing 000 (2016) 1–17

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Accelerating big data analytics on HPC clusters using two-level storage

Pengfei Xuan[a], Walter B. Ligon[b], Pradip K. Srimani[a], Rong Ge[a], Feng Luo[a,*]

[a] School of Computing, Clemson University, Clemson, SC, 29634, United States
[b] Electrical and Computer Engineering, Clemson University, Clemson, SC, 29634, United States

ARTICLE INFO

Article history:
Received 30 January 2016
Revised 30 July 2016
Accepted 1 August 2016
Available online xxx

Keywords:
Two-level storage
In-memory file system
Parallel file system
Data-intensive computing

ABSTRACT

Data-intensive applications that are inherently I/O bound have become a major workload on traditional high-performance computing (HPC) clusters. Simply employing data-intensive computing storage such as HDFS or using parallel file systems available on HPC clusters to serve such applications incurs performance and scalability issues. In this paper, we present a novel two-level storage system that integrates an upper-level in-memory file system with a lower-level parallel file system. The former renders memory-speed high I/O performance and the latter renders consistent storage with large capacity. We build a two-level storage system prototype with Tachyon and OrangeFS, and analyze the resulting I/O throughput for typical MapReduce operations. Theoretical modeling and experiments show that the proposed two-level storage delivers higher aggregate I/O throughput than HDFS and OrangeFS and achieves scalable performance for both read and write. We expect this two-level storage approach to provide insights on system design for big data analytics on HPC clusters.

1. Introduction

Computer clusters consisting of a large number of compute nodes provide an indispensable computing infrastructure for scientific and engineering modeling and simulations. Traditionally, a computer cluster has a network-attached storage that is managed by parallel file systems. Such clusters have been widely employed in industry and academia to run diverse compute-intensive HPC applications [1–3]. HPC system software, including parallel file systems and storage [4,5], plays an essential role to deliver high computational throughput to scientific and engineering applications.

An emergent and prevalent workload on computer clusters is big data applications that process large volumes of data, often of size in terabytes or petabytes [6–8]. To meet the unique requirements of these applications for data access and storage, numerous packages at each layer of the system software stack are developed, including HDFS [9] and Espresso [10] distributed data storage, cluster resource management systems [11,12], data-parallel programming frameworks [13–16], and high-level application oriented libraries [17–22].

Data-intensive computing presents a new challenge when its applications migrate to computer clusters geared for traditional HPC applications [5,23–25], mainly due to the different workload characteristics and optimization objectives. Data storage and I/O access are among the causal issues. To support the emergent data-intensive applications, previous studies

* Corresponding author. Fax: +0 864 656 0145.
E-mail address: luofeng@clemson.edu (F. Luo).

http://dx.doi.org/10.1016/j.parco.2016.08.001
0167-8191/© 2016 Elsevier B.V. All rights reserved.

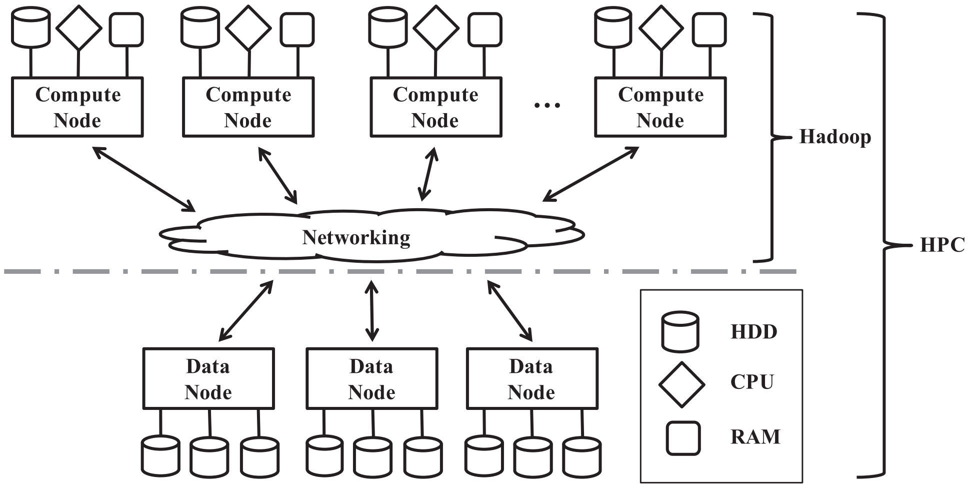

Fig. 1. Architectural abstractions for Hadoop and HPC infrastructures.

either directly use HPC parallel file systems and storage [26–28] or deploy distributed file systems such as HDFS on compute nodes [29,30]. Using HPC file systems and storage provides high capacity with low-cost fault tolerance, but suffers poor performance limited by network and disk I/O bandwidth of storage nodes. On the other hand, deploying data-intensive file system on compute nodes takes advantage of data locality and results in high aggregate I/O throughput, but incurs costly data fault tolerance and low storage capacity.

In this paper, we explore a novel approach to accelerate data-intensive computing on HPC clusters. We develop a two-level storage system by integrating an in-memory file system with a parallel file system, and build a prototype with Tachyon [31] and OrangeFS [32]. Both theoretical modelling and experimental measurements show that the two-level storage system can increase the aggregate I/O throughput while maintaining low cost data fault tolerance and achieving a high storage capacity. We expect this two-level storage approach to provide insights on system design for big data analytics on HPC clusters.

2. Background

2.1. HPC and Hadoop architectures

System architectures are designed to best support the typical workloads running on the clusters and optimize the dominating operations of these workloads. Traditional HPC systems are aimed to deliver high CPU compute rate to compute-intensive workloads with high ratios of compute to data access. In contrast, data-intensive computing frameworks provide high data access throughput, especially for disk I/O accesses, to big data analytics with very low ratios of compute to data access. Consequently, data-intensive computing frameworks are different, as shown in Fig. 1.

The HPC architecture physically separates data nodes from compute nodes and connects the nodes with high speed networks. The data nodes are typically connected to large volume storage devices. As such, HPC systems consist of two types of data storage services: global parallel file system on data nodes and local file system on compute nodes. The former has a large capacity and stores consistent user input and output, and the latter has a small capacity and stores temporal data. During execution, user data is transferred from global storage on data nodes to compute nodes that perform computation. Data hosted on local storage devices is ephemeral and purged when jobs complete or storage is full. Physically separating compute nodes from data nodes provides easy data sharing, but fails to provide spatial data locality for computation tasks, unable to achieve scalable data-intensive computing.

To deliver high I/O throughput, data-intensive computing frameworks such as Hadoop co-locate compute nodes and data nodes on the same physical machines. The local storage device on each compute node accounts for a part of the persistent system storage. Computation tasks are launched on physical machines where the required data are to maximally leverage data locality. Hadoop systems allow multiple computation tasks accessing data from different nodes simultaneously and achieve higher aggregate I/O throughput with more nodes.

2.2. Data access performance

We select several national HPC clusters to demonstrate the importance of in-memory access for improving I/O performance. The clusters in Table 1 include four national HPC clusters [33] as well as the Palmetto cluster [34] in Clemson University. A HPC compute node often uses single hard disk drives (HDDs) as the local storage and deploys high-end DRAM

Table 1
Storage capacity and network bandwidth of computer nodes on several national HPC clusters.

HPC	Disk (GB)	DRAM (GB)	Parallel FS (PB)[a]	CPU (Core)	Network
Stampede	80	32	14	16	56 Gbit/s FDR InfiniBand
Maverick	240	256	20	20	56 Gbit/s FDR InfiniBand
Gordon	280	64	1.6	16	40 Gbit/s QDR InfiniBand
Trestles	50	64	1.4	32	40 Gbit/s QDR InfiniBand
Palmetto	900	128	0.2	20	10 Gbit/s Ethernet
Avg.	310	109	7.4	21	40 Gbit/s

[a] 1 PB = 10^{15} bytes; 1 TB = 10^{12} bytes; 1 GB = 10^{9} bytes; 1 MB = 10^{6} bytes; 1 KB = 10^{3} bytes.

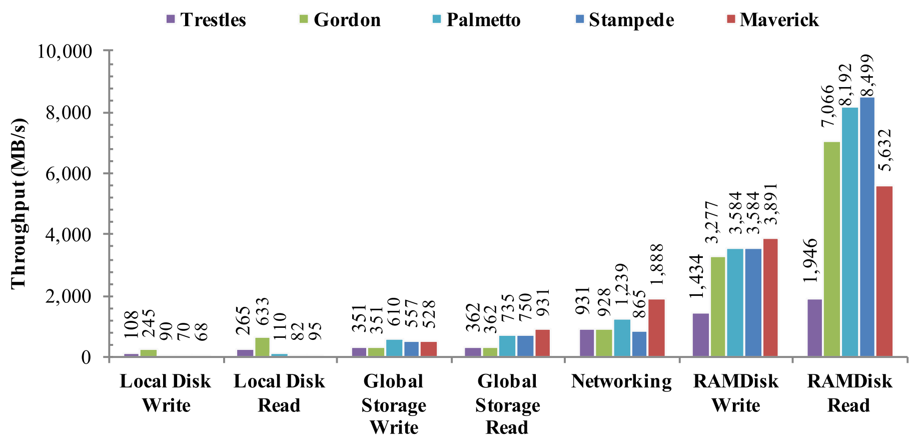

Fig. 2. I/O throughput of a single compute node on national HPC clusters.

modules. The memory capacity is comparable to the local storage disk capacity on each node. These nodes are connected via high speed Ethernet or Infiniband. As these clusters have been existent for some time, the memory modules are DDR2/DDR3 DRAMs. The system-wide storages are located on dedicated data nodes and managed by parallel file systems, such as Lustre and OrangeFS.

We empirically evaluate the I/O performance of a program on a compute node when the program reads from and writes to different storage locations. These locations include local memory and local disk drive, remote memory on a different compute node across the network, and the global disk storage through the data node across the network. We run the Linux built-in tool "*dd*" in the sequential mode and use the *direct I/O* option to bypass buffer caches. We measure local and remote disk I/O performances by consecutively reading and writing 16 different 1GB files, and the memory performance with files of 10 GB. This latter larger size can saturate the memory bandwidth and assure measurement accuracy. Each measurement is repeated five times on three different compute nodes.

As shown in Fig. 2, the I/O performance is the highest when local memory devices serve as the storage, and drops to the lowest when local disks serve as the storage. For read, I/O performance is improved about 10× when storage moves from local disk to local memory and 2.65× to global storage. Similarly, for write, I/O performance is improved 6.57× and 4× respectively. The remote memory throughput, denoted by network, is measured using "*Iperf*" based-on IPoIB link-layer. We get a reduced TCP throughput on high-performance Infiniband network due to the low MTU value (2044) hard coded on the HPC compute nodes. Higher MTU values should be able to deliver a higher throughput.

3. The design and implementation of the two-level storage system

Simply employing HDFS over compute nodes or parallel system on global storage makes data-intensive computing suffer poor scalability or poor throughput. On the other hand, deploying HDFS over local disk storage on the compute nodes results small aggregated storage capacity. If Hadoop uses the global parallel file system as storage, the average I/O throughput of each compute node decreases as more compute nodes participate in computation, namely, the I/O throughput does not scale.

We propose a two-level storage system shown in Fig. 3 to support high-performance and scalable data-intensive computing on HPC infrastructure using Hadoop. This two-level storage system combines an in-memory file system on the compute nodes and a parallel file system on the data nodes. As the compute nodes in HPC clusters are often equipped with large memory, the in-memory file system can have a storage capacity comparable to local storage-based HDFS. In addition, the I/O throughput of in-memory file system is significantly larger than local disk I/O throughput, and scales with system size.

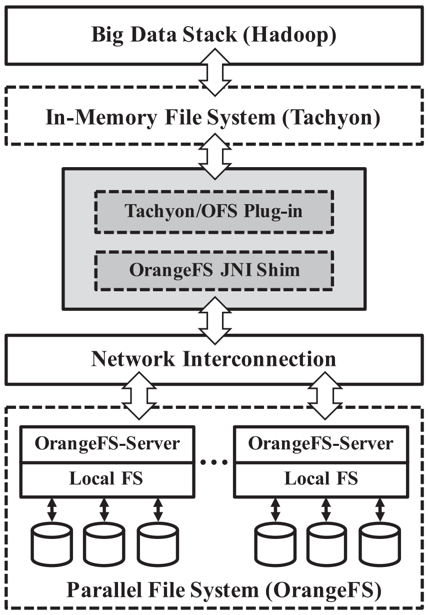

Fig. 3. System architecture of two-level storage.

At the same time, the parallel file system provides relatively low cost data-fault tolerance and large storage capacity. Thus, the two-level storage scheme exploits advantages of both in-memory file system and parallel file system.

We implement a prototype of the two-level storage system by integrating the in-memory file system Tachyon-0.6.0 with the parallel file system OrangeFS-2.9.0. Tachyon is implemented in Java and the OrangeFS is implemented in C. These two levels are tightly integrated with following two modules indicated in shadowed area in Fig. 3.

- **Tachyon/OFS Plug-in**: A Java plug-in that provides an interface to translate the functionalities of Tachyon in-memory file system to the functionalities of OrangeFS parallel file system. The plug-in also provides *hints* with storage layout support to allow deep tuning between the two file systems.
- **OrangeFS JNI Shim**: A Java API that forwards all function calls from Tachyon/OFS Plug-in to the OrangeFS Direct Interface. To deliver a high bandwidth, the shim layer uses Java Native Interface (JNI) with Non-blocking I/O (NIO) APIs and optimized buffer size to minimize overheads introduced in JVM.

While designing and implementing our prototype, we have added new features for OrangeFS and Tachyon projects and contributed our work back to the open source communities. Related patches have been merged into both of OrangeFS trunk and Tachyon master branch.

3.1. Data layout mapping

In our two-level storage system, OrangeFS and Tachyon have different data layouts. As shown in Fig. 4, an input file is transparently stored in Tachyon as a set of fixed size logical blocks and each block is materialized into a Tachyon data file cached in local in-memory space. The block size controls data-parallel granularity and can be predefined in configuration. In contrast, the data file is stored in OrangeFS as stripes. Each OrangeFS data file is then striped at the disk level, which is usually performed by hardware RAID (redundant array of independent disks) built in each data node. Data fault tolerance of the two-level storage system is ensured by the low-level erasure coding inside each data node.

The mapping between Tachyon and OrangeFS data layouts affects the load balance among data nodes and the aggregate I/O throughput. To conserve the designs of each layer, we organize the data with the layout of the destination file system whenever there is a data transfer between Tachyon and OrangeFS. In our current design, we use two parameters to control the mapping. The first parameter, *stripe_size*, determines the data granularity across the list of data nodes and has a direct impact on the aggregate I/O throughput. The second parameter, *layout*, decides the ordered set of data nodes to store the distributed OrangeFS data file and manipulates the I/O load patterns between Tachyon data files and remote data nodes. To achieve optimal performance, we tune several parameters including Tachyon's block size and OrangeFS' stripe size. The Tachyon parameters are decided before runtime while the OrangeFS parameters are dynamically adjustable through hints implemented in our plug-in.

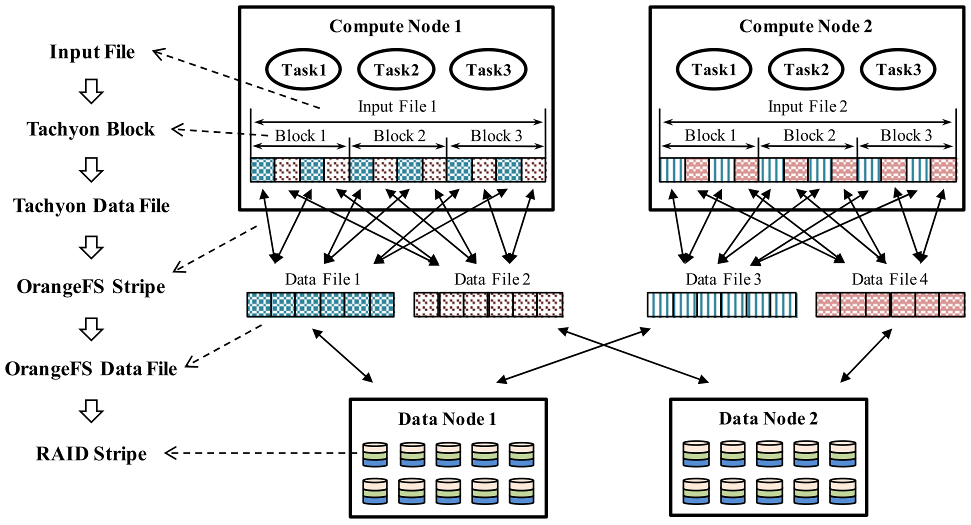

Fig. 4. Input file partition and checkpointing data file striping on the two-level storage system.

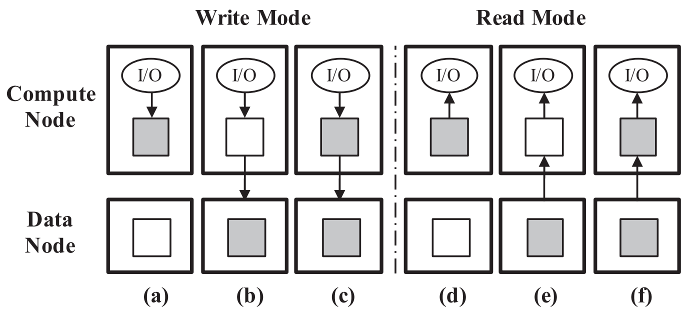

Fig. 5. I/O operation modes of two-level storage.

3.2. I/O modes of the two-level storage

Currently, we have implemented synchronous I/O on the prototype two-level storage system. The prototype provides three write modes and three read modes. The three write modes are: (1) data is stored only in Tachyon, (2) data bypasses Tachyon and is written to OrangeFS, and (3) data is synchronously written to OrangeFS when data is created or updated in Tachyon. Fig. 5(a–c) shows these three modes respectively. The read modes are similar and shown in Fig. 5(d–f): (1) data is read from Tachyon only, (2) data is read from OrangeFS directly without being cached in Tachyon, and (3) data is read partially from Tachyon and partially from OrangeFS. The read mode in Fig. 5(f) is the primary usage pattern in data-intensive computing. It improves read performance by caching reusable data and adopting a proper data replacement policy such as LRU.

Reading data from remote data nodes, especially from overloaded data nodes, is very expensive. To minimize I/O congestion and contention, we apply distance-based read policy over local and remote storage. Data access pipeline has two adjustable I/O buffers, one (upper buffer) between application and Tachyon and the other (lower buffer) between Tachyon and OrangeFS, as illustrated in Fig. 6. The read I/O request always goes to the next closest available storage device hosting the target data. Since Hadoop schedules computing tasks based on data locality, most of the computing tasks always first take step 1 to fetch data from the upper buffer. If it fails, it takes step 2 to fetch data from Tachyon storage. If both attempts fail (the data is not present in Tachyon), the read request takes step 3 and 4 to the lower level to load the block from OrangeFS persistent storage layer. We use 1 MB as the upper buffer size and 4 MB as the lower buffer size. These sizes are experimentally determined as they deliver good I/O throughput and latency.

4. I/O throughput analysis

4.1. I/O modeling of different storage systems

Considering a HPC system consisting of N compute nodes and M data nodes, we make the following system assumptions to simplify the modeling effort:

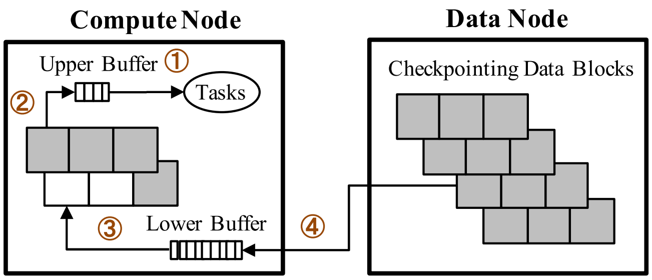

Fig. 6. Read I/O flow of two-level storage. The numbers indicate the order of distance-based read policy.

Table 2
List of notations.

Symbol	Meaning
D	The size of data each node processes
N	Number of compute nodes
M	Number of data nodes
f	The ratio of the size of data in Tachyon to the total size of data
Φ	Bandwidth of switch backplane, bisection bandwidth of network (Mbps)
ρ	Bandwidth of network interface of compute and data nodes (Mbps)
μ	I/O throughput of local hard drives on compute nodes (Mbps)
μ'	I/O throughput of local hard drives on data nodes (Mbps)
ν	I/O throughput of local memory (Mbps)
q	Average I/O throughput received on compute nodes (Mbps)

- All nodes have identical hardware configurations and are connected to each other via non-blocking switches.
- Workload is uniformly distributed among compute nodes, i.e., each node has the same amount of I/O load.
- The centralized switch and the bisection bandwidth of network provide a uniform non-blocking backplane throughput Φ, and each node is connected by a full-duplex network interface with bandwidth ρ.
- There is no network-level interference, such as TCP congestion, and Incast/Outcast.

We focus on I/O bound computational tasks that are evenly distributed on compute nodes without data skew. Hadoop can use four different types of storages: HDFS, OrangeFS, Tachyon and the two-level storage system. We model the I/O throughput of each compute node with these four types of storages when it reads/writes a fixed size D of data from/to storage. Table 2 lists the notations used in the models.

4.1.1. I/O modeling of HDFS

With HDFS, Hadoop reads from the local hard drives. In this case, the I/O throughput μ of local hard drive determines the read throughput q^{HDFS}_{read} of each compute. If data is not available on local hard drive, Hadoop reads from other nodes across the network. In this case, q^{HDFS}_{read} is determined by the minimum of three factors: ρ, bandwidth throughput of network interface of each node, Φ/N, shared backplane throughput, and μ, I/O throughput to local hard drive. Thus, the read throughput of each node, q^{HDFS}_{read}, is

$$q^{HDFS}_{read} = \begin{cases} \mu, & \textit{local access} \\ min\left(\rho, \frac{1}{N}\Phi, \mu\right), & \textit{remote access} \end{cases} \tag{1}$$

To maintain the fault-tolerance of data, by default, Hadoop synchronously writes one copy of data to local hard drive and two replicas to two other nodes by streaming through network. Again, the write throughput q^{HDFS}_{write} of each node is given by the minimum of three factors: bandwidth of network interface of each node, shared backplane throughput and the I/O throughput of local hard drive. Considering the entire cluster, all nodes write three copies of data to local storage. Then, the maximum write throughput of each node to local hard drive is $\frac{1}{3}\mu$. Each node writes two copies of data to network. Thus, throughput of network interface of each node is limited by $\frac{1}{2}\rho$ and average throughput of bisection backplane is bounded by $\frac{1}{2N}\Phi$. Thus, the write throughput of each node, q^{HDFS}_{write}, can be estimated as

$$q^{HDFS}_{write} = min\left(\frac{1}{2}\rho, \frac{1}{2N}\Phi, \frac{1}{3}\mu\right) \tag{2}$$

4.1.2. I/O modeling of OrangeFS

With OrangeFS as storage for Hadoop, N compute nodes read/write data from M data nodes. All read and write traffic must pass through the network. Thus, both write and read throughput are determined by the throughput of following four resources: (1) bandwidth of network interface of a compute node, ρ. (2) the shared throughput on bisection backplane; since all nodes share the bandwidth of the switch backplane, the average throughput received by each compute node is $\frac{1}{N}\Phi$ assuming $N > M$, (3) the shared throughput of network interface of data nodes; aggregated throughput of network interface of M data nodes is $M \times \rho$, shared by N compute nodes. Thus, the average network interface throughput of data nodes that each compute node receives is $\frac{M}{N}\rho$, (4) the shared I/O throughput to local hard drive in data nodes; aggregate local hard drive I/O throughput of M data nodes is $M \times \mu'$, shared by N compute nodes. Thus, the average I/O throughput to local hard drive on data nodes is $\frac{M}{N}\mu'$. Together, the read throughput, q^{OFS}_{read}, and write throughput, q^{OFS}_{write}, of each compute node are

$$q^{OFS}_{write} = q^{OFS}_{read} = min\left(\rho, \frac{1}{N}\Phi, \frac{M}{N}\rho, \frac{M}{N}\mu'\right) \tag{3}$$

4.1.3. I/O modeling of Tachyon

The architecture of the Tachyon system is similar to that of HDFS except that: (1) it uses DRAM, rather than local hard drive, to store data; (2) it uses lineage-based recovery, rather than data replication, to achieve data fault-tolerance. Tachyon can potentially improve the write throughput significantly.

With Tachyon as storage, Hadoop reads from DRAM of each compute node. The read throughput $q^{Tachyon}_{read}$ of each compute node is determined by the I/O throughput ν of DRAM. If the data is not available from local DRAM, Hadoop reads from DRAM of other nodes in the network. In this case, the $q^{Tachyon}_{read}$ is determined by the minimum of three throughput: network interface bandwidth, ρ; shared backplane bandwidth, Φ/N; and the I/O throughput ν to DRAM. Namely, the read throughput of each node, $q^{Tachyon}_{read}$, is:

$$q^{Tachyon}_{read} = \begin{cases} \nu, & local\ access \\ min\left(\rho, \frac{1}{N}\Phi, \nu\right), & remote\ access \end{cases} \tag{4}$$

With Tachyon as storage**,** Hadoop write the data to DRAM of each compute node. Then, the write throughput of each compute node, $q^{Tachyon}_{write}$ is limited by the throughput to memory:

$$q^{Tachyon}_{write} = \nu \tag{5}$$

4.1.4. I/O modeling of the two-level storage

Our analysis is focused on the third write mode in Fig. 5(c) and the third read mode in Fig. 5(f) of the two-level storage system where the Hadoop reads/writes data from/to both Tachyon and OrangeFS. For the third write mode, the data is synchronously written to Tachyon and OrangeFS at the same time. As the write throughput to Tachyon is much higher than those to OrangeFS, the write throughput of each compute node on two-level storage, q^{TLS}_{write}, is bounded by the write throughput to OrangeFS:

$$q^{TLS}_{write} = min(q^{Tachyon}_{write},\ q^{OFS}_{write}) = q^{OFS}_{write} \tag{6}$$

Let f be the ratio of the size of data in Tachyon over the total size of data, D. Then, the size of data in Tachyon is $f \times D$ and the size of data in OrangeFS is $(1 - f) \times D$. The Hadoop read $f \times D$ data from Tachyon with throughput ν (Tachyon in the two-level storage do not read the data from other compute nodes) and $(1 - f) \times D$ data from OrangeFS with throughput q^{OFS}_{read}. Combined together, the read throughput of each compute node is:

$$q^{TLS}_{read} = 1/\left(\frac{f}{\nu} + \frac{1-f}{q^{OFS}_{read}}\right) \tag{7}$$

If $f = 1$, data is read from Tachyon only and if $f = 0$, data is read from OrangeFS only. The higher the value of f is, the larger the read throughput provided by the two-level system is.

4.1.5. Aggregate I/O throughput comparison

The aggregate read/write throughput of HDFS can linearly scale up with the number of compute nodes. On the other hand, the aggregate read/write throughput of parallel file systems (such as OrangeFS) and the two-level storage are bounded by the network bandwidth and aggregate throughput of local disks on data nodes. To understand the aggregate I/O throughput of parallel file systems and two-level storage compared to those of HDFS, we have done a case study using the average I/O throughput of HPC clusters (Fig. 2). The network bandwidth is set to 1170 MB/s per node. The local disk read throughput is 237 MB/s and the local disk write throughput is 116 MB/s. The local memory throughput is 6267 MB/s. We have tested two parallel file systems aggregate throughput: 10 GB/s and 50 GB/s. We assume the HDFS is deployed on single hard disk of compute nodes of HPC. We don't consider the storage capacity that systems can support, but focus only on the throughput.

We observe the aggregate read/write throughput of three different values of *f*. As shown in Fig. 7, at 10 GB/s aggregate bandwidth of parallel file system, HDFS needs 43 nodes to achieve a higher aggregate read bandwidth than that of parallel file system and needs 53 nodes ($f = 0.2$) and 83 nodes ($f = 0.5$) to achieve higher read aggregate bandwidths than that of two-level storage. At 50 GB/s aggregate bandwidth of parallel file system, the HDFS needs 211 nodes to have a higher

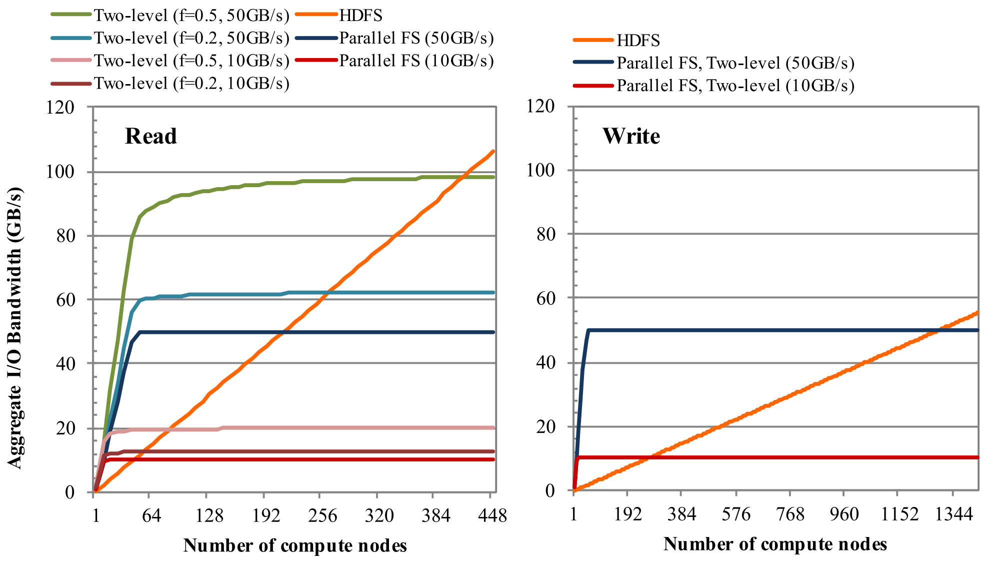

Fig. 7. Aggregate read throughput (left) and write throughput (right) of HDFS, parallel file system and two-level storage.

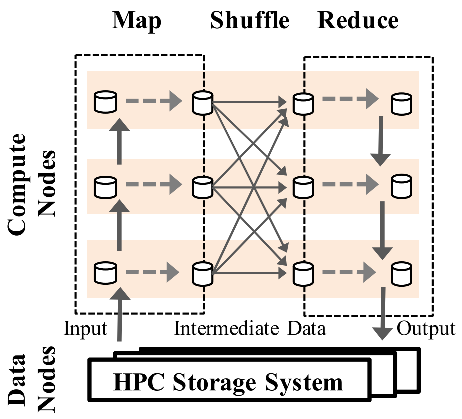

Fig. 8. Storage I/O accesses in MapReduce job.

aggregate read bandwidth than that of parallel file system has and needs 262 nodes ($f = 0.2$) and 414 nodes ($f = 0.5$) to have higher aggregate read bandwidths than that of the two-level storage. Our results show that the two-level storage increases the aggregate read bandwidth by about 25% at $f = 0.2$ (from 10 GB/s to 12.5 GB/s or from 50 GB/s to 62 GB/s) and about 95% at $f = 0.5$ (from 10 GB/s to 19.6 GB/s or from 50 GB/s to 98 GB/s). Thus, use of the two-level storage can increase the number of compute nodes to deploy Hadoop without sacrificing read performance.

At 10 GB/s aggregate bandwidth of parallel file system, HDFS needs 259 nodes to have higher aggregate write bandwidth than those parallel file system and the two-level storage have, and at 50 GB/s aggregate bandwidth of parallel file system, HDFS needs 1294 nodes to have higher aggregate write bandwidth than those parallel file system and the two-level storage have. The write throughput of HDFS is significantly smaller than the read throughput since Hadoop needs to write two copies of data through network. Thus, write throughput is usually not the constraint to use Hadoop on HPC with parallel or two-level storages.

4.2. MapReduce I/O cost analysis

The main idea of MapReduce programming model is to partition a large problem into sub-parts, compute partial solutions on sub-parts independently, and then reassemble the partial solutions into the final solution. Standard MapReduce programming model includes three major phases: Map, Shuffle and Reduce as shown in Fig. 8. Here, we analyze the I/O cost of each phase in MapReduce when different underlying storage systems are used.

4.2.1. *I/O cost in Map phase*

In the Map phase, the tasks load input data from storage systems, and spill intermediate data on local storage. When HDFS and OrangeFS are used, the task output the intermediate data to local disk. Assuming the size of the spilled on-disk intermediate data in Map phase is I_M, the I/O cost on Map phase when HDFS and OrangeFS are used can be given as follows:

$$T_{map} = T_{read}^{input} + T_{write}^{I_M} = \left(\frac{D}{q_{read}} + \frac{I_M}{\mu} \right) \tag{8}$$

When Tachyon and two-level storage are used, the tasks output the intermediate data to in-memory file system. So the I/O cost on Map phase when Tachyon and two-level storage are used can be given as (assuming in-memory file system can store all intermediate data):

$$T_{map} = T_{read}^{input} + T_{write}^{I_M} = \left(\frac{D}{q_{read}} + \frac{I_M}{\nu} \right) \tag{9}$$

4.2.2. *I/O cost in Shuffle phase*

In the Shuffle phase, the tasks pull Map output from its local storage, sort it and send it to corresponding local storage at Reduce side. When HDFS and OrangeFS are used, the tasks read the Map output from local disk and write the intermediate data to local disk. We define the size of materialized intermediate data in Shuffle phase as I_S. Thus, when HDFS and OrangeFS are used, the I/O cost on Shuffle phase is:

$$T_{shuffle} = T_{read}^{I_M} + T_{write}^{I_S} = \frac{I_M + I_S}{\mu} \tag{10}$$

When Tachyon and two-level storage are used, the tasks read the Map output from local memory and write the intermediate data to local memory. So, when Tachyon and two-level storage are used, the I/O cost in Shuffle phase is:

$$T_{shuffle} = T_{read}^{I_M} + T_{write}^{I_S} = \frac{I_M + I_S}{\nu} \tag{11}$$

4.2.3. *I/O cost in Reduce phase*

In Reduce phase, the tasks load the sorted intermediate data from Shuffle phase from local storage, process it and output back to underlying storage systems. When HDFS and OrangeFS are used, the tasks read the Shuffle output from local disk. Assuming the size of output data generated in Reduce phase is O_R. Then, when HDFS and OrangeFS are used, the I/O cost in Reduce phase is

$$T_{reduce} = T_{read}^{I_S} + T_{write}^{output} = \frac{I_S}{\mu} + \frac{O_R}{q_{write}} \tag{12}$$

When Tachyon and two-level storage are used, the tasks read the Shuffle output from local memory. Then, the I/O cost in Reduce phase is

$$T_{reduce} = T_{read}^{I_S} + T_{write}^{output} = \frac{I_S}{\nu} + \frac{O_R}{q_{write}} \tag{13}$$

5. Evaluation

In this section, we evaluate our two-level storage system using three experiments. We first characterize I/O throughput behavior of the two-level storage. Then, we compare performance of CPU, disk and network I/O utilizations of each compute node and the performance of disk and network I/O utilizations of each data node using TeraSort benchmark program when Hadoop is deployed on HDFS, OrangeFS and the two-level storage, respectively. We then use K-means to investigate the performance of iterative workflow on three storage systems.

5.1. *Experimental setup*

All experiments are performed on Palmetto HPC cluster hosted at Clemson University. We select nodes with the same hardware configuration (Table 3) for our experiments. Each compute node is attached with a single SATA hard disk, and each data node is attached to a 12 TB disk array. Although we cannot control the bandwidth of switch backplane, the backplane bandwidth is much higher than the network interface bandwidth in our experiments and is not the bottleneck resource in our experiments.

For the first experiment, we use Tachyon built-in performance evaluation program as the benchmark tool to measure the average read throughput received from two-level storage under a range of *data sizes* with different *skip sizes*. In the experiment, we conduct our measurements between one compute node and one data node. We allocate 16 GB for Tachyon storage space on compute node and the data node has a 12 TB OrangeFS file system. The skip size is defined as a fragment of data skipped per MB access. Since OrangeFS has much higher access latency than Tachyon has, a large skip size has larger

Table 3
Hardware configurations of selected nodes on Palmetto cluster.

Component	Specification
CPU	Intel Xeon E5-2670 v2 20 × 2.50 GHz
HDD	1 TB 7200RPM SATA
RAID	12 TB LSI Logic MegaRAID SAS
DRAM	128 GB DDR3-1600
Network	Intel 10 Gigabit Ethernet
Switch	Brocade MLXe-32 with 6.4 Tbps backplane

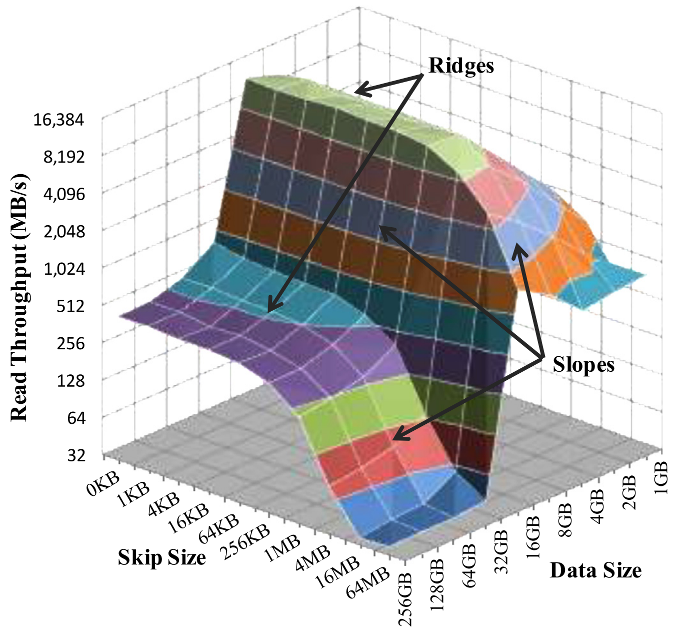

Fig. 9. The storage mountain of two-level storage system.

impact on the I/O throughput for OrangeFS than for Tachyon. The data size is varied from 1 GB to 256 GB. For each data size, we test a range of skip sizes from 0 KB to 64 MB.

For second and third experiments, we run Terasort benchmark and K-means on 256 GB datasets using 17-node Hadoop cluster with 2-node OrangeFS as back-end storage system. In Hadoop cluster, one machine is used as head node to host YARN's *ResourceManager* (RM) and Tachyon's Master Service, and the rest of 16 compute nodes are used to ingest MapReduce workloads. On each compute node, we assign 16 containers to occupy 16 CPU slots and leave the rest of 4 CPU slots to handle extra system overhead. Thus, we can run 256 Mappers or Reducers and the workload can achieve full system utilization if CPU utilization reaches 80%.

The capacity of Tachyon storage on each compute node is 32 GB. Then, the total capacity of Tachyon is 512 GB. The Tachyon block size is set to 512 MB. Each block is striped into 8 chunks with a strip size of 64 MB that are evenly distributed across 2 data nodes in a round-robin fashion. Before each test, we empty OS page caches to measure actual I/O costs. The concurrent write and read throughput on local disk for each of compute nodes are about 65 MB/s. Concurrent write throughput on RAID for each of OrangeFS data nodes is about 200 MB/s, and read is close to 400 MB/s.

5.2. I/O characterization for two-level storage

As illustrated in Fig. 9, we generate a two dimensional function of read throughput versus data size and skip size. This function is similar to the memory mountain that characterizes the capabilities of memory system. We call this function *the storage mountain* of two-level storage system. The storage mountain reveals the performance characteristics of our prototype two-level storage system. There are two ridges on the storage mountain. The high ridge corresponds to throughput of Tachyon and the low ridge reflects the throughput of OrangeFS. The I/O buffer size between applications and Tachyon is set to 1 MB (Section 3.2).

There is a sharp slope between the two ridges when the data size is larger than 16 GB, which is the size of Tachyon storage. If we take a sliding window through the mountain with a fixed skip size as in Fig. 10, we can see the impact of in-memory file system size on the read throughput. For data sizes smaller than 16 GB, the reads are from Tachyon only and

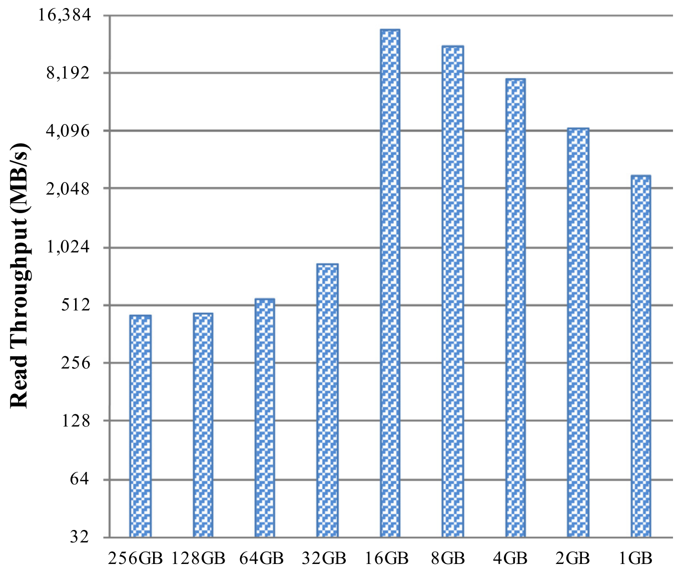

Fig. 10. The read throughput of two-level storage shows the data locality. The graph shows a slice of Fig. 9 when skip size is 0 KB.

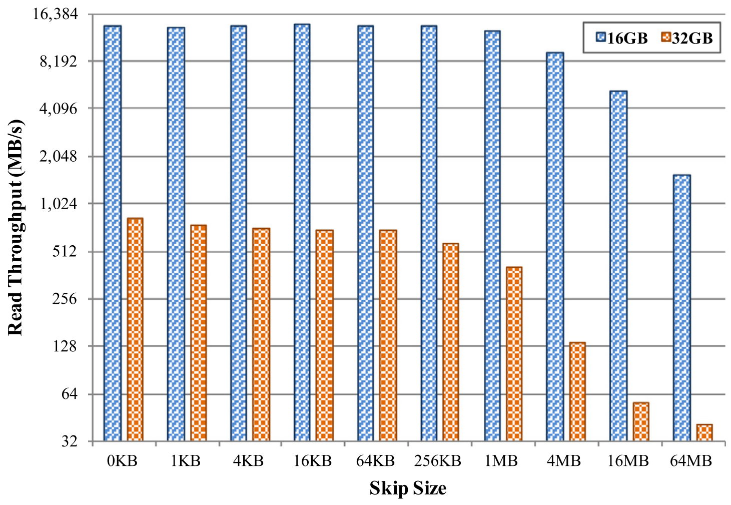

Fig. 11. The impact of read skip size on read throughput.

the read throughput reaches the peak throughput of about 13.2 GB/s. For data sizes greater than 16 GB, namely, the data cannot fit in-memory space, the read throughput are reduced sharply to less than 1 GB. The higher the data size is more than 16 GB, the lower the read throughput is. For data of size 256 GB, the read throughput is reduced to less than 512 MB.

Slicing through the storage mountain along an opposite direction gives us an insight into the impact of data access pattern on the read throughput. Fig. 11 shows the read performance of two fixed data sizes: 16 GB and 32 GB. For the skip sizes up to 1 MB, every read request is guaranteed to have at least one hit on either of the two I/O buffers or Tachyon in-memory space, and the read throughput remains at high level. Once the request size reaches the secondary I/O buffer size, 4 MB, every read request misses in I/O buffers and must spend extra cycles to fetch the data. As shown in Fig. 11, the large skip sizes have led to more significant performance degradation of read throughput on data of size 32 GB than those on data of size 16 GB.

In addition, there are two slopes on both ridges when the skip sizes are larger than 1 MB. The read throughput decreases when the data size is small. This is because the extra overheads, such as scheduling cost, data serialization, become noticeable when the I/O cost of small data is low.

The storage mountain shows that the performances of the two-level storage is affected by multiple factors, such as data size and skip size. Since the ridge of Tachyon is much higher than that of OrangeFS, we need to keep frequently used data in Tachyon to achieve better performance.

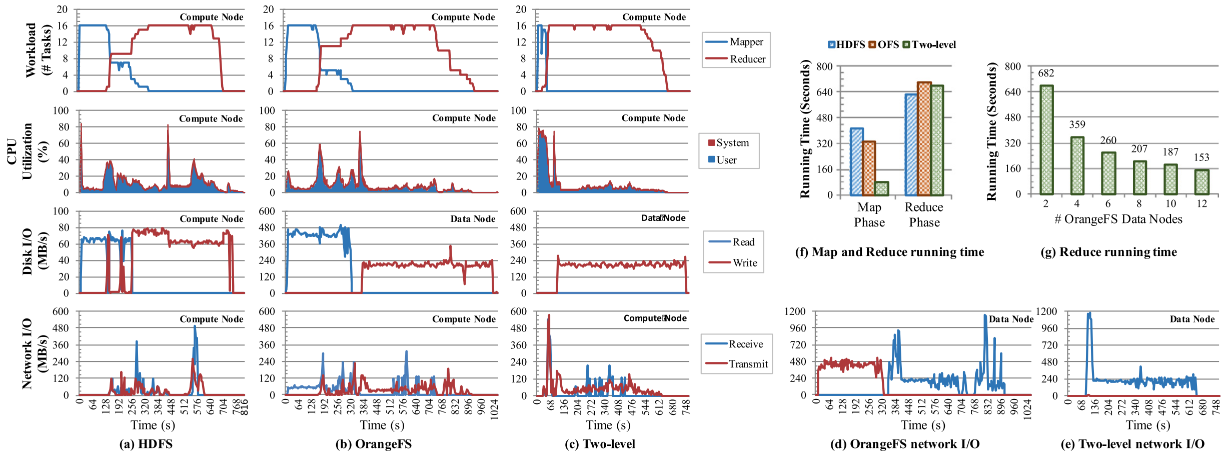

Fig. 12. Performance profiling metrics for TeraSort benchmark suit on three storage systems.

5.3. Performance evaluation using TeraSort

In this experiment, we profile the detailed performance metrics with the TeraSort benchmark workload. The TeraSort benchmark has three stages: *TeraGen* stage generates and writes input data to storage; *TeraSort* stage loads input data, sorts and writes output data to storage; and *TeraValidate* stage reads and validates the sorted output data. Since the TeraSort stage reads once and writes once and is an I/O bounded task, we use this stage to evaluate I/O performance of three storages: HDFS, OrangeFS and two-level storage.

We first run the TeraGen stage using a Map-only job to generate 256 GB data and store in three storages: HDFS, OrangeFS and two-level storage (one copy in Tachyon and one copy in OrangeFS). We then run the TeraSort stage using one MapReduce cycle. Mapper reads the data from storage and Reducer writes the sorted data back to storage. We profile the performance of CPU, disk and network I/O utilizations of each compute node and the performance of disk and network I/O utilizations of each data node (Fig. 12(a–e)).

With HDFS, the Mapper reads from and the Reducer writes to local disks on compute nodes. With OrangeFS, the Mapper reads from and the Reducer writes to OrangeFS on data nodes. With the two-level storage, the Mapper reads from Tachyon (RAM) on compute nodes and the Reducer writes to OrangeFS on data nodes.

Since we can store all data in Tachyon of two-level storage in our experiments, the Mapper can achieve peak read throughput (Tachyon ridge in storage mountain) as shown Fig. 12(f). The Mapper on two-level storage is able to achieve about 5.4× and 4.2× speedup comparing to the Mapper on HDFS and OrangeFS, respectively (Fig. 12(f)). The high read throughput even pushes the Mapper reaching full CPU usage (Fig. 12(c)). Keeping part of data in Tachyon of two-level storage also reduces the network traffic. In our extreme case, there is no network traffic from data nodes for Mappers using two-level storage (Fig. 12(e)).

Writing to OrangeFS through Tachyon can also slightly improve the performance compared to directly writing to OrangeFS (Fig. 12(b, c, f)). It benefits from unidirectional I/O access from Tachyon to OrangeFS, in which OS page caches of data nodes can fully engage in optimizing write loads. As a comparison, the data nodes are involved for handling both read and write loads when only OrangeFS is used. The Reducer running time on OrangeFS and two-level storage is slightly longer than that using HDFS (Fig. 12(f)) when we use only two data nodes. However, the write throughput of OrangeFS and two-level storage can be steadily improved by scaling the data node. For example, when a new data node to our testing system is added, roughly an extra 200 MB/s concurrent write throughput can be achieved. Running time of TeraSort reduce phase decreases by 1.9× and 4.5× when the number of data nodes increases from 2 to 4 and 12 respectively (Fig. 12(g)). In all tests, performance is bounded by either aggregate disk throughput or CPU FLOPs of compute nodes, rather than networking bandwidth. As shown in Fig. 12(a, b, c, d, e), the network throughput never reaches its limit.

The data used in our current experiments is relatively small and can be completely stored in Tachyon of two-level storage. If we use large data that need to be stored in both Tachyon and OrangeFS of two-level storage, the performance of TeraSort using two-level storage is degraded gracefully. However, according to our theoretical analysis (Fig. 7), we still expect that the two-level storage could always provide better performance than OrangeFS. Finally, the two-level storage has the added advantage to deliver higher I/O throughput and larger storage capacities than HDFS when number of compute nodes is limited.

5.4. Performance evaluation using K-means

Next, we use K-means to examine the performance of iterative workflows on the three storage systems. The K-means clustering algorithm partitions *n* data points into *K* clusters. It has two steps: assignment and update. In assignment step, the

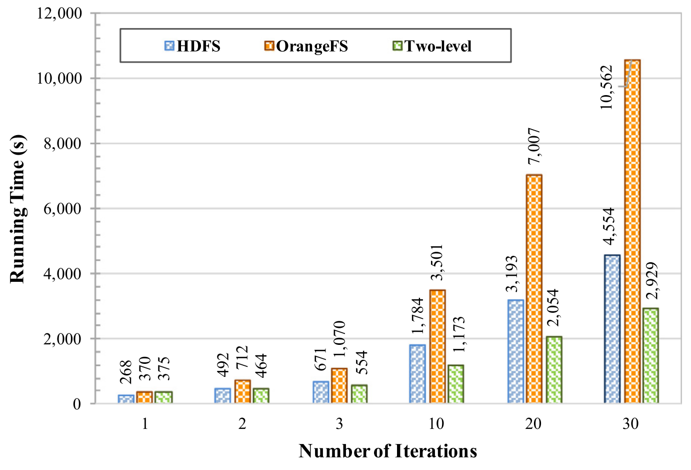

Fig. 13. Accumulated running time of K-means algorithm with three different storage systems.

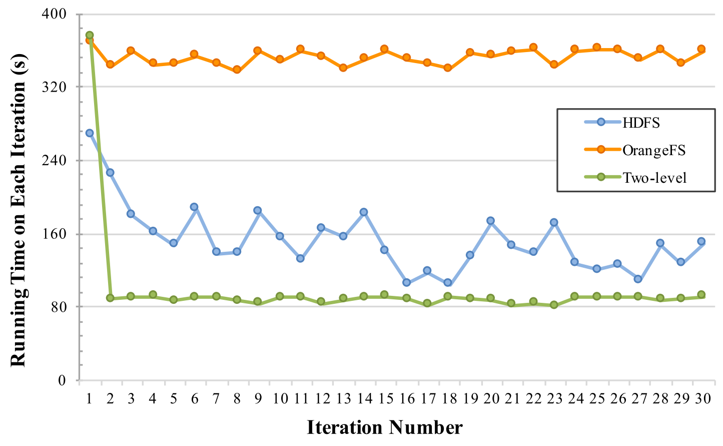

Fig. 14. Running time on each K-Means iteration.

algorithm assigns each data point to a cluster whose centroid is the closest to it. In update step, the algorithm recalculates the centroid of each cluster based on new assignment. The K-means algorithm iterates these two steps until it converges.

In our experiments, we sample 10 clusters of 27 dimension data points using HiBench [24]. The size of total input data is about 256 GB. The data is initially stored in HDFS and OrangeFS. For the two-level storage, the data is stored in Tachyon after the first iteration. We run the K-means algorithm from Apache Mahout on the Hadoop with three different storage systems for 30 iterations. In each iterative step, the Mappers load all input data and centroids, and emit the nearest centroid for each data point; the Reducers collect the emitted data, calculate the new centroids, and save them onto disk.

Fig. 13 shows the running times of first three iterations and 10th, 20th and 30th iterations, respectively. In the first iteration, input data are fully loaded from HDFS and OrangeFS directly. Thus, the running times of the first iteration are mainly dominated by the I/O cost. The cost of HDFS is lower than those of OrangeFS and two-level storage in the first iteration. This is because the aggregate read bandwidth of HDFS is about 1040 MB/s (Eq. (1)) and is higher than the aggregate bandwidth (800 MB/s) of OrangeFS. The cost of the two-level storage in first iteration is slightly higher than that of OrangeFS, which indicates a small overhead after adding Tachyon storage layer. In subsequent iterations, I/O throughput of the two-level storage becomes significantly higher since the data is kept in Tachyon. Thus, the running time of K-means on two-level storage becomes significantly shorter. After 30 iterations, the running time of K-means on the two-level storage is 3.6× and 1.6× shorter than those on OrangeFS and HDFS. The K-means workload becomes CPU-bound (after the first I/O-bounded iteration) as the data is fully cached in memory.

Fig. 14 shows the running times for K-mean iterations on three different storages. The running times of K-means iterations on HDFS and OrangeFS is reduced after first iteration because both HDFS and OrangeFS implement cache to improve performance. The running times of K-means iterations on OrangeFS only slightly reduce after the first iteration because the OrangeFS client-side cache is either too small or uses an unmatched cache eviction policy (e.g. LRU), leading to a poor cache hit rate. In contrast, the running times of K-means iterations on HDFS is reduced significantly due to large distributed page caches on each compute node. The running times of K-means iterations on the two-level storage are stable after first

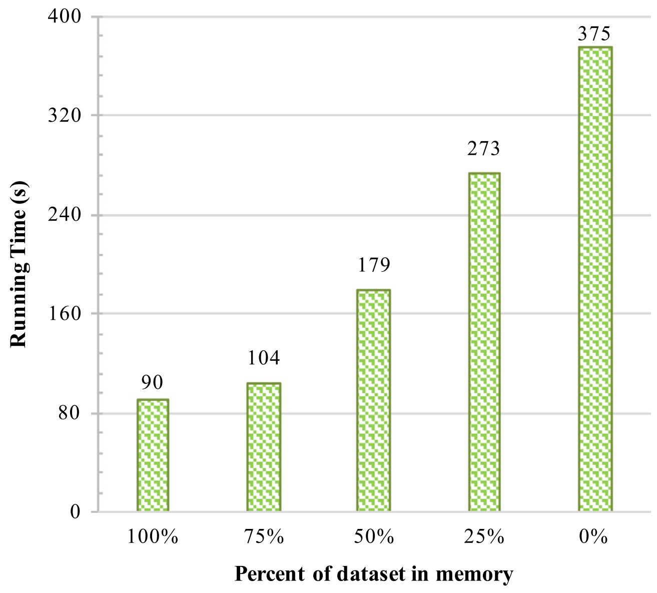

Fig. 15. Average running times of K-means using 256 GB data on 16 machines with varying amounts of data in Tachyon (first iteration running time excluded).

iteration. The capacity of Tachyon in our test is larger than the data size. Thus, all input and intermediate data is kept in Tachyon, providing stable cache hit rates for data.

The running time of K-means iterations on HDFS has relatively wide variations for two reasons. First, K-means iteration workload leads repeated reads of input datasets and results in all three replicas of data blocks being fetched into the page cache. Eventually, the size of occupied data blocks can be much larger than the size of page cache allocated in compute nodes, and a portion of reusable input datasets are evicted. Second, compared to the two-level storage, the use of page cache in HDFS does not provide the fine-grain control that application workload actually needs. The OS controlled cache size and eviction policy could become problematic when Hadoop runtime tasks or other applications have a heavy I/O load on local disk during the period of K-means execution. The page cache pollution and interference suffered in HDFS result in an unstable hit rate on the input datasets.

If the in-memory file system is not large enough to hold all data, part of the data has to be stored in the OrangeFS of two-level storage. To understand how the capacity of Tachyon affects the performance, we run the K-means with only part of input data stored in Tachyon. Fig. 15 shows average running times of K-means with various amounts of input data kept in Tachyon. Overall, the performance is reduced when less input data is kept in Tachyon. However, the running time only gets a 15.5% degradation when 25% input datasets is moved out of Tachyon. This performance maintenance comes from the convergence of aggregate I/O bandwidth over individual compute nodes and reduction of I/O contention across the underlying parallel file system. In this case, local in-memory access to 75% of input data has reduced more than 75% overhead on both network and OrangeFS. The local in-memory access to input data greatly reduces the probability of network congestion as well as improves the throughput of concurrent disk access on remote data nodes. Thus, the higher in-memory hit is not only able to increase I/O bandwidth, but also improves the compute efficiency when storage bandwidth gets saturated. Once the data kept in Tachyon is less than 75%, the average running times decrease linearly with the percentage of in-memory data.

To further investigate the scalability of K-means on three storage systems, we conduct experiments using different combinations of compute nodes and data nodes. The ratio of numbers of compute nodes and data nodes is fixed at 8:1. The size of input data is linearly increased from 128 GB for 8 computer nodes and 1 data node to 1024 GB for 64 compute nodes and 8 data nodes. For each experiment, we run K-means on two stages: the stage of 10 iterations and the stage that writes the output to the storage system.

Fig. 16 shows the running times of two stages for different configurations. For the iteration stage, with the benefit of data locality, the running times of K-means on both HDFS and two-level storage show no performance degradation. On the other hand, the K-means running times of iteration stage on OrangeFS have gradually increased as the compute/data nodes and input data size increase, which indicates that the overhead for reading data from OrangeFS has increased as the number of compute/data nodes increase. For the write stage, the running times of K-means on both OrangeFS and two-level storage show no performance degradation. The K-means running time of write stage on OrangeFS is much higher than that of K-means on two-level storage. This is due to the implementation characteristics of K-means algorithm in Mahout. The K-means application in Mahout first reads a copy of input data, and then writes the output back to storage. For the two-level storage, it reads the data from Tachyon while it reads data directly from OrangeFS if only OrangeFS is used. The K-means

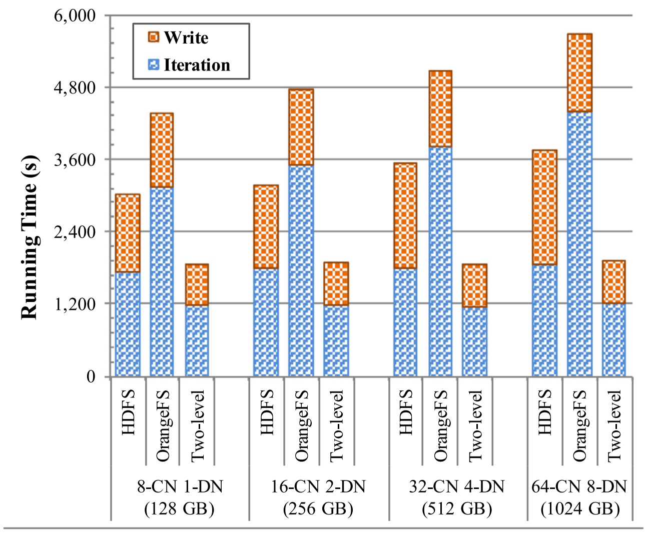

Fig. 16. Weak scalability of three storage systems. CN: compute nodes, DN: data nodes.

running time of write stage on HDFS has gradually increased as compute/data nodes and input data size increase. Since HDFS uses a synchronous pipeline to duplicate two additional replicates of output data on remote machines, it is possible that some machines get overwhelmed and the network gets congested, which may lower the performance at the write stage. This phenomenon becomes more significant when the number of compute nodes increases. Overall, the two-level storage achieves weak scalability on both iteration and write stages.

6. Related work

There are three major research directions to integrating Hadoop with HPC infrastructure. Previous work has explored directly deploying Hadoop atop of existing parallel file systems, such as GPFS [35], Ceph [26], Lustre [36]. These efforts mainly focus on showing the performance enhancement by exploring suitable mapping between parallel file systems and Hadoop, such as increasing the size of stripe unit, using different layout distribution, and applying optimal data prefetching. However, the performance of data-intensive workload is still tightly coupled with available I/O bandwidth of parallel file systems.

Instead of using dedicated data servers, some previous studies deploy Hadoop on compute nodes only. Tantisiriroj et al. [37] explore the I/O performance benefit by migrating data server to compute nodes with emulated HDFS-style data layout, replication and consistency semantics. In their experiments, the performance of PVFS (v2.8.2) is very close and even higher than that of HDFS (v0.20.1) on 51-node OpenCloud cluster when using optimized I/O buffer size, data mapping and layout. Other researches have deployed parallel file system, Gfarm [38] and GlusterFS [39] as well as QFS [27], on compute nodes in their production cluster. However, the capacity, performance and in consistence of local disk in traditional HPC clusters limit the usability of deploying Hadoop on compute nodes.

Third approach deploys Hadoop on data nodes. Xu et al. [40] have studied performance enhancement by employing MapReduce on the storage sever of HPC. This deployment solution can access data on persistent storage natively. It works on a small size of workloads but it could have scalability issues when the job has mixed CPU and data-intensive workloads, the reason being that data nodes on HPC are usually equipped with relatively slow and limited computing units. This is especially true for many of CPU-bound data analysis workloads [41].

One project [42], is similar in spirit to our project, but from a different direction. It uses two optimized schedule techniques (Enhanced Load Balancer and the Congestion-Aware Task Dispatching) to improve the I/O performance of local disk. Our solution is focusing on integration of two storage systems.

Wang et al. [43] have also utilized memory to increase I/O performance of parallel file system. They introduce a dedicated buffer layer deployed at the front-end of data nodes of HPC to buffer the burst I/O. In our system, we use the memory of compute nodes as part of storage.

PortHadoop [44] and Triple-H [45] are two closest research works as the proposed two-level storage for accelerating Hadoop/Spark workloads on HPC clusters. These solutions share the same goal but use different building blocks and interception techniques. However, the former two solutions rely on HDFS interface and services (NameNode service, DataNode service or both) and thus have an isolated naming space between parallel file system and HDFS. Two-level storage can decouple this dependence using the unified namespace between Tachyon and parallel file system. For example, if in-memory files have been synchronized to underlying parallel file system, HPC applications can still access those files using the namespace from parallel file system transparently. In addition, Tachyon also provides additional lineage APIs to ensure data fault tolerance, which can significantly improve the write throughput for two-level storage if further integration has been implemented.

Finally, the search engine, Baidu, reported use of Tachyon as a transparent layer for data exchange between Baidu file system (BFS) hosted in data centers in China and those in USA research center [46]. Depending on the workload type, overall improvement was 30× to 60× speedups.

7. Conclusions

In this paper, we develop a prototype of the two-level storage by integrating the in-memory file system, Tachyon, and the parallel file system, OrangeFS. In the two-level storage, Tachyon is deployed on compute nodes and OrangeFS is deployed on data nodes. Tachyon provides the mechanism to exploit temporal locality of data that does not need to be retrieved from data nodes via network. Our theoretical modeling and experimental evaluation show that the current version of the two-level storage can increase read throughput. Since write throughput is usually not a bottleneck for running Hadoop on HPC, higher read throughput of the two-level storage scales up with the number of compute nodes for Hadoop.

Although running Hadoop on Tachyon alone can also take advantage of high I/O throughput and data locality, it has two issues. First, the capacity of Tachyon is limited compared to large storage capacity on data nodes. Second, Tachyon uses lineage to recover data when there is a fault. This recovery incurs computing cost. In our two-level storage, local data always has a copy in OrangeFS; thus, OrangeFS provides fault-tolerance for Tachyon.

Public HPC clusters are usually shared by a large number of users. Each user is usually allocated a limited number of compute nodes. The two-level storage can provide higher read and write throughput with limited number of compute nodes. Thus, running Hadoop with the two-level storage may provide a better performance solution for big data analytics on traditional HPC infrastructures.

Acknowledgements

We are thankful to Haoyuan Li from Alluxio, Inc. for providing helpful feedbacks. HPC resources used in this research are supported by Omnibond Systems, LLC, the Clemson Computing and Information Technology (CCIT) and the Extreme Science and Engineering Discovery Environment (XSEDE). Ge's work was partially supported by the NSF under Grant No. CCF-1551511.

References

[1] D.J. Becker, T. Sterling, D. Savarese, J.E. Dorband, U.A. Ranawak, C.V. Packer, BEOWULF: a parallel workstation for scientific computation, in: Proceedings, International Conference on Parallel Processing, 1995.
[2] G. Bell, J. Gray, What's next in high-performance computing? Commun. ACM 45 (2002) 91–95.
[3] H. Meuer, E. Strohmaier, J. Dongarra, H.D. Simon, Top500 supercomputer sites, in: Proceedings of SC, 2001, pp. 10–16.
[4] V.R. Basili, J.C. Carver, D. Cruzes, L.M. Hochstein, J.K. Hollingsworth, F. Shull, M.V. Zelkowitz, Understanding the high-performance-computing community: a software engineer's perspective, IEEE Softw. 25 (2008) 29.
[5] D.A. Reed, J. Dongarra, Exascale computing and big data, Commun. ACM 58 (2015) 56–68.
[6] H. Jagadish, J. Gehrke, A. Labrinidis, Y. Papakonstantinou, J.M. Patel, R. Ramakrishnan, C. Shahabi, Big data and its technical challenges, Commun. ACM 57 (2014) 86–94.
[7] R.T. Kouzes, G.A. Anderson, S.T. Elbert, I. Gorton, D.K. Gracio, The changing paradigm of data-intensive computing, Computer (2009) 26–34.
[8] J. Manyika, M. Chui, B. Brown, J. Bughin, R. Dobbs, C. Roxburgh, and A.H. Byers, "Big data: the next frontier for innovation, competition, and productivity," 2011.
[9] K. Shvachko, H. Kuang, S. Radia, R. Chansler, The hadoop distributed file system, in: Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on, 2010, pp. 1–10.
[10] L. Qiao, K. Surlaker, S. Das, T. Quiggle, B. Schulman, B. Ghosh, A. Curtis, O. Seeliger, Z. Zhang, A. Auradar, On brewing fresh Espresso: Linkedin's distributed data serving platform, in: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, 2013, pp. 1135–1146.
[11] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A.D. Joseph, R.H. Katz, S. Shenker, I. Stoica, Mesos: a platform for fine-grained resource sharing in the data center, in: NSDI, 2011, p. 22.
[12] V.K. Vavilapalli, A.C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, Apache Hadoop yarn: yet another resource negotiator, in: Proceedings of the 4th annual Symposium on Cloud Computing, 2013, p. 5.
[13] J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters, Commun. ACM 51 (2008) 107–113.
[14] S. Ewen, "Programming abstractions, compilation, and execution techniques for massively parallel data analysis," 2015.
[15] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M.J. Franklin, S. Shenker, 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, 2012, p. 2.
[16] M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker, I. Stoica, Spark: cluster computing with working sets, HotCloud 10 (2010) 10.
[17] M. Armbrust, R.S. Xin, C. Lian, Y. Huai, D. Liu, J.K. Bradley, X. Meng, T. Kaftan, M.J. Franklin, A. Ghodsi, Spark sql: relational data processing in spark, in: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, 2015, pp. 1383–1394.
[18] C. Avery, Giraph: Large-scale graph processing infrastructure on hadoop, in: Proceedings of the Hadoop Summit, Santa Clara, 2011.
[19] R.S. Xin, J.E. Gonzalez, M.J. Franklin, I. Stoica, Graphx: a resilient distributed graph system on spark, in: First International Workshop on Graph Data Management Experiences and Systems, 2013, p. 2.
[20] G. Ingersoll, Introducing apache mahout, Scalable, Commercialfriendly Mach. Learn. Building Intell. Appl. (2009).
[21] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, and S. Owen, "Mllib: machine learning in apache spark," arXiv preprint arXiv:1505.06807, 2015.
[22] T. Kraska, A. Talwalkar, J.C. Duchi, R. Griffith, M.J. Franklin, M.I. Jordan, MLbase: a distributed machine-learning system, in: CIDR, 2013, p. 2.1.
[23] A.J. Hey, S. Tansley, K.M. Tolle, The Fourth Paradigm: Data-Intensive Scientific Discovery, vol. 1, Microsoft research Redmond, WA, 2009.
[24] S. Huang, J. Huang, J. Dai, T. Xie, B. Huang, The HiBench benchmark suite: Characterization of the MapReduce-based data analysis, in: New Frontiers in Information and Software as Services, Springer, 2011, pp. 209–228.

[25] P. Xuan, Y. Zheng, S. Sarupria, A. Apon, SciFlow: a dataflow-driven model architecture for scientific computing using Hadoop, in: Big Data, 2013 IEEE International Conference on, 2013, pp. 36–44.
[26] C. Maltzahn, E. Molina-Estolano, A. Khurana, A.J. Nelson, S.A. Brandt, S. Weil, Ceph as a scalable alternative to the Hadoop Distributed File System, login: The USENIX Magazine 35 (2010) 38–49.
[27] M. Ovsiannikov, S. Rus, D. Reeves, P. Sutter, S. Rao, J. Kelly, The quantcast file system, in: Proceedings of the VLDB Endowment, 6, 2013, pp. 1092–1101.
[28] E.H. Wilson, M.T. Kandemir, G. Gibson, Will they blend? exploring big data computation atop traditional hpc nas storage, in: Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on, 2014, pp. 524–534.
[29] S. Krishnan, M. Tatineni, and C. Baru, “myHadoop-Hadoop-on-Demand on Traditional HPC Resources,” San Diego Supercomputer Center University of California Technical Report TR-2011-2, San Diego, 2011.
[30] TACC Hadoop Cluster, http://www.tacc.utexas.edu/tacc-projects/hadoop-cluster/.
[31] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, I. Stoica, Tachyon: reliable, memory speed storage for cluster computing frameworks, in: Proceedings of the ACM Symposium on Cloud Computing, 2014, pp. 1–15.
[32] M. Moore, D. Bonnie, B. Ligon, M. Marshall, W. Ligon, N. Mills, E. Quarles, S. Sampson, S. Yang, B. Wilson, OrangeFS: advancing PVFS, FAST Poster Session (2011).
[33] The Extreme Science and Engineering Discovery Environment. http://www.xsede.org/.
[34] The Palmetto Cluster. http://citi.clemson.edu/palmetto/.
[35] Z. Fadika, E. Dede, M. Govindaraju, L. Ramakrishnan, Mariane: mapreduce implementation adapted for hpc environments, in: Grid Computing (GRID), 2011 12th IEEE/ACM International Conference on, 2011, pp. 82–89.
[36] N. Rutman, “Map/reduce on lustre,” white paper. Technical report, Xyratex Technology Limited, Havant, Hampshire, UK 2011.
[37] W. Tantisiriroj, S.W. Son, S. Patil, S.J. Lang, G. Gibson, 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, 2011, p. 67.
[38] O. Tatebe, K. Hiraga, N. Soda, Gfarm grid file system, in: New Generation Computing, 28, 2010, pp. 257–275.
[39] A. Davies, A. Orsaria, Scale out with GlusterFS, Linux Journal 2013 (2013) 1.
[40] C. Xu, R. Goldstone, Z. Liu, H. Chen, B. Neitzel, W. Yu, Exploiting analytics shipping with virtualized mapreduce on HPC backend storage servers, Parallel Distrib. Syst. IEEE Trans. 27 (2016) 185–196.
[41] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, B.-G. Chun, Making sense of performance in data analytics frameworks, in: 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15), 2015, pp. 293–307.
[42] T. Wang, S. Oral, Y. Wang, B. Settlemyer, S. Atchley, W. Yu, Burstmem: a high-performance burst buffer system for scientific applications, in: Big Data (Big Data), 2014 IEEE International Conference on, 2014, pp. 71–79.
[43] Y. Wang, R. Goldstone, W. Yu, T. Wang, Characterization and optimization of memory-resident mapreduce on HPC systems, in: Parallel and Distributed Processing Symposium, 2014 IEEE 28th International, 2014, pp. 799–808.
[44] X. Yang, N. Liu, B. Feng, X.-H. Sun, S. Zhou, PortHadoop: support direct HPC data processing in Hadoop, in: Big Data (Big Data), 2015 IEEE International Conference on, 2015, pp. 223–232.
[45] N.S. Islam, X. Lu, M. Wasi-ur-Rahman, D. Shankar, D.K. Panda, Triple-H: a hybrid approach to accelerate HDFS on HPC clusters with heterogeneous storage architecture, in: Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on, 2015, pp. 101–110.
[46] S. Liu, “Fast big data analytics with Spark on Tachyon in Baidu,” 2015.