Experiences of Converging Big Data Analytics Frameworks with High Performance Computing Systems

Peng Cheng[1,2(✉)], Yutong Lu[3], Yunfei Du[3], and Zhiguang Chen[1,2]

[1] College of Computer, National University of Defense Technology, Changsha, China
peng_cheng_l3@l63.com
[2] State Key Laboratory of High Performance Computing, Changsha, China
[3] National Supercomputer Center in Guangzhou (NSCC-GZ), Guangzhou, China

Abstract. With the rapid development of big data analytics frameworks, many existing high performance computing (HPC) facilities are evolving new capabilities to support big data analytics workloads. However, due to the different workload characteristics and optimization objectives of system architectures, migrating data-intensive applications to HPC systems that are geared for traditional compute-intensive applications presents a new challenge. In this paper, we address a critical question on how to accelerate complex application that contains both data-intensive and compute-intensive workloads on the Tianhe-2 system by deploying an in-memory file system as data access middleware; we characterize the impact of storage architecture on data-intensive MapReduce workloads when using Lustre as the underlying file system. Based on our characterization and findings of the performance behaviors, we propose shared map output shuffle strategy and file metadata cache layer to alleviate the impact of metadata bottleneck. The evaluation of these optimization techniques shows up to 17% performance benefit for data-intensive workloads.

Keywords: High performance computing · Big data · Convergence
File system · Hadoop

1 Introduction

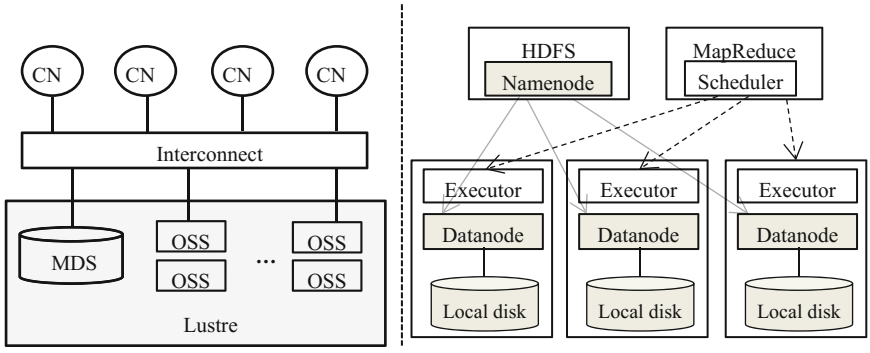
The strong need for increased computational performance has led to the rapid development of high-performance computing (HPC) systems, including Sunway TaihuLight [1], Tianhe-2 [2], Titan [3], etc. These HPC systems provide an indispensable computing infrastructure for scientific and engineering modeling and simulations [4–6]. While HPC systems mostly focus on large computational workloads, the emerging big data analytics frameworks target applications that need to handle very large and complex data sets on commodity machines. Hadoop MapReduce [7] and Spark [8] are the most commonly used frameworks for distributed large-scale data processing and gained wide success in many fields over the past few years.

© The Author(s) 2018
R. Yokota and W. Wu (Eds.): SCFA 2018, LNCS 10776, pp. 90–106, 2018.
https://doi.org/10.1007/978-3-319-69953-0_6

Recently, many researchers have predicted the trend of converging HPC and big data analytics frameworks to address the requirements of complex applications that contains both compute-intensive and data-intensive workloads [9, 10]. The motivation behind this converging trend is twofold. Firstly, traditional data analytics applications need to process more data in given time, but the dynamics of the network environment and cloud services result in a performance bottleneck. Compared with normal machines, HPC systems that equipped with better hardware and high performance network can provide much higher capacity. Secondly, scientific applications are more complex to fully utilize the computing capacity of the HPC systems and the high resolution data from advanced sensors. For example, in the NASA Center for Climate Simulation (NCCS), climate and weather simulations can create a few terabytes of simulation data [11]. To visualize the data of interesting events such as hurricane center and thunderstorms, part of these data need to be processed by a visualization tool under Hadoop environment. These complex applications contain both compute-intensive and data-intensive jobs and need to be processed in HPC environment.

However, the converging trend presents a new challenge when data-intensive applications migrate to HPC systems that are geared for traditional compute-intensive applications, mainly due to the different workload characteristics and optimization objectives.

System architectures are designed to best support the typical workloads running on the clusters. Traditional HPC systems are invented to solve compute-intensive workloads, such as scientific simulation, with the optimization goal of providing maximum computational density and local bandwidth for given power/cost constraint. In contrast, big data analytics systems aim to solve data-intensive workloads with the optimization goal of providing maximum data capacity and global bandwidth for given power/cost constraint. Consequently, big data analytics systems are different to HPC systems, as shown in Fig. 1.



a. System architecture of typical HPC systems

b. System architecture of big data analytics systems

Fig. 1. System architecture comparison

Typical HPC systems consist of a large collection of compute nodes (CN), which are connected through high-speed, low-latency interconnects (such as InfiniBand [12]). Parallel file system (e.g. Lustre [13]) on top of disk array are used for persistent data storage. In most HPC systems, the compute node is diskless and performs well for compute-intensive workloads with high ratios of compute to data access. Parallel file systems simplify data sharing between compute nodes, but its performance is bottlenecked by metadata operations and fails to provide spatial data locality for computation tasks.

Big data processing frameworks like Hadoop and Spark utilize low-cost commodity machines to solve data-intensive problems, where each machine co-locate processing unit and local disks together. Hadoop distributed file system (HDFS [14]) is built on top of these local disks to provide the ability of persistent storage. Computation tasks are launched on physical machines where the data locality of required data can be leveraged maximally.

On the one hand, these distinctions between HPC and big data analytics systems have significant performance implications for different types of application workloads. On the other hand, the converging trend of HPC and big data is imperative and provides a lot of chance for researchers to make their attempt.

In this paper, we try to figure out three problems:

1. How to accelerate the complex application that contains both simulation and analytics jobs? The output data of HPC workloads are stored in parallel file systems, while traditional big data analytics frameworks rely on HDFS to read or write data. Hence, it is highly desirable to utilize a middleware that allows applications to access data stored in different data source without redundant data movement.
2. What is the impact of using Lustre parallel file system as the underlying file system of big data analytics frameworks since compute nodes in most HPC systems are diskless?
3. How to reconcile and converge the architectural differences between the two paradigms so that data-intensive MapReduce applications can be accelerated in HPC environments?

Previous works in [15, 16] have analyzed the performance differences when deploying Hadoop and Spark on HPC systems, but they did not provide optimizations for complex applications. Many efforts have explored directly deploying Hadoop atop of existing parallel file systems, such as Ceph [17], PVFS [18] and GPFS [19], but these works are limited to the specific version of Hadoop. Compared with these works, we deploy an in-memory file system, Alluxio [20], as data access middleware to accelerate complex applications. We analyze its performance with intensive experiments on the Tianhe-2 system and introduce shared map output file shuffle strategy and file metadata cache layer targeting at compute-centric HPC systems to accelerate data-intensive Hadoop applications.

Our contributions in this paper can be summarized as follows.

(1) We have utilized an in-memory file system on Tianhe-2 system to reconcile the architectural differences and accelerate complex applications.
(2) We have evaluated the feasibility and performance impacts of in-memory file system through different workloads.

(3) We proposed advanced acceleration techniques, including shared map output file shuffle strategy and file metadata cache layer, to accelerate data-intensive MapReduce applications on HPC systems.

Section 2 gives a brief background of this paper. Section 3 explore the feasibility of data access middleware and analyze the performance impact. We propose our design of shared map output file shuffle strategy and file metadata cache layer in Sect. 4. Section 5 provides some related studies currently existing in the literature. We conclude the paper and talk about the future work in Sect. 6.

2 Background

In this section, we give a direct comparison between HDFS and Lustre file system and review the design of Alluxio.

2.1 HDFS vs Lustre

HDFS and Lustre file system is the representative underlying file system for big data analytics frameworks and HPC applications, respectively. Table 1 summarizes their key differences.

Table 1. Comparison between HDFS and Lustre

	POSIX compliant	Access model	Data locality information
HDFS	No	Write once read many	Yes
Lustre	Yes	Many write many read	No

HDFS is a distributed file system designed to run on commodity hardware. HDFS has a master/slave architecture, the namenode is the master that manages the file system namespace and regulates access to files by clients, datanodes are the slaves that stores data blocks and are responsible for serving read and write requests from the file system's clients. HDFS is highly fault tolerant since every data block is replicated three times by default to minimize the impact of a data center catastrophe. All the data block info are stored in the master node, which allow the computation tasks to launch on physical machines where the data locality of required data can be leveraged maximally. Because of the nature of big data analytics workloads that most data won't be modified once it is generated, HDFS relaxes consistency and provide a write once read many access model to improve file access concurrency.

Lustre is a POSIX-compliant, open-source distributed parallel file system, and is deployed in most modern HPC clusters due to the extremely scalable architecture. Lustre is composed of three components: metadata servers (MDSs) manage file metadata, such as file names, directory structures, and access permissions. Object storage servers (OSSs) expose block devices and serves data. The clients issue the I/O requests. To access a file, a client must send a request to MDS first to get the file

metadata, including file attributes, file permissions, and the layout of file objects in the form of extended attributes (EA). With the EA, the client can communicate with corresponding OSSs to get the file data.

2.2 Alluxio

Due to the growing I/O bandwidth gap between main memory and disk, the storage layer became the bottleneck that limits the application scalability. To alleviate the storage pressure, a variety of in-memory file systems that act as a fast, distributed cache are developed to enhance I/O performance [21–23].

Alluxio is an open source memory speed virtual distributed storage system that sits between the underlying storage systems and processing framework layers. It has the same architecture as HDFS where a master is primarily responsible for managing the global metadata of the system, Alluxio workers store data as blocks in local resources. These resources could be local memory, SSD, or hard disk and are user configurable. Alluxio provides Hadoop API, Java API and POSIX interface for user applications and computation frameworks while it can connect with different underlying storage systems such as Amazon S3, Apache HDFS through encapsulated adapters. Because of the memory-centric design and the outstanding compatibility with different computation frameworks and storage systems, Alluxio plays an important role in the big data ecosystem.

We choose Alluxio as data access middleware to converge big data analytics frameworks with HPC systems because of the compatibility to different kinds of underlying file system and the POSIX-compliant interface it provides.

3 Experiment and Analysis

To explore an efficient way of accelerating complex applications on HPC systems and analyze the performance impact of different architecture between big data analytics and HPC, we use three benchmarks to cover the performance space: (1) IOZone benchmark [24] and Mdtest benchmark [25] tests the basic file system performance of HDFS and Lustre; (2) HiBench benchmark [26] evaluates the efficiency of big data analytics frameworks through micro workloads, machine learning workloads, etc. (3) Simulation of complex applications where RandomWriter simulates HPC workload and generates 10 GB data per node, Sorter simulates data analytic workload that analyzes the output data of RandomWriter in Hadoop environment.

3.1 Experiment Setup

We used 64 nodes to conduct our experiments on the data analytics cluster of the Tianhe-2 system at national supercomputer center in Guangzhou (NSCC-GZ), where each node is equipped with two 2.20 GHz Intel Xeon E5-2692 processors (24 cores per node) and 64 GB of RAM. We allocate 32 GB RAM for MapReduce jobs and reserve 32 GB for ramdisk. Besides, there is one HDD of 1 TB storage space mounted on each node as the local disk. All nodes run Linux 3.10.0 kernels and are connected via Tianhe high performance interconnects.

We used 32 nodes to simulate the typical data analytics environment where HDFS is set on top of the local disk. Spark-2.1.0, Hadoop-2.7.3 and Oracle Java 1.8.0 are used. The HDFS block size is set to 128 MB.

Another 32 nodes are used to simulate the typical HPC environment. We did not utilize the local disk as underlying storage but mounted Lustre file system on these nodes. In our test environments, the mounted Lustre file system contains 48 OSTs. To enable Hadoop and Spark to read/write data from/to Lustre, we deployed Alluxio-1.4.0 on these nodes.

As mentioned, we evaluate IOZone, Mdtest, HiBench benchmark and a simulated complex application on HPC and data analytics environment, where the underlying file system is Lustre and HDFS respectively. Each benchmark has been executed at least three times and we report the mean performance.

3.2 Data Access Middleware

As discussed in the introduction, the converging trend of big data analytics and HPC is imperative, it's important to give a feasible solution that can allow Hadoop or Spark to access data stored in Lustre. A straightforward way to use Hadoop or Spark in HPC environment is to copy data from Lustre to HDFS. However, this will result in tremendous data movement cost and is a waste of storage space.

After considering these issues, we deploy Alluxio in HPC systems and compare its performance against Hadoop adapter for Lustre (HAL) [27]. HAL, which is developed by the Intel high performance data division, is used to make Hadoop work on Lustre, instead of HDFS, without any Lustre changes. HAL modifies Hadoop's built-in LocalFileSystem class to add the Lustre file system support by extending and overriding default file system behavior. Moreover, HAL optimizes the shuffle strategy by avoiding repetitive data movements. In the default implementation of Hadoop shuffle strategy, reduce tasks send the data fetch requests to remote servers which in turn retrieve the data from their local directories and send them back across the network. Because of Lustre can provide a shared file system view for reduce tasks, default shuffle strategy will result in repetitive data movements, HAL reimplements the shuffle phase so that each reduce task can accesses Lustre to retrieve the data written by remote nodes directly.

HAL is packaged as a single Java library, Hadoop and Spark applications can access Lustre without modifying their code after HAL library is added to the Hadoop classpath and the configuration files are updated. What's important for users to know is that HAL-Lustre is not compatible with HDFS, which means the applications can access only Lustre file system after changing the configuration files. Besides, HAL cannot provide data locality information for map/reduce tasks.

We allocate 20 GB ramdisk for Alluxio worker to store data blocks on each node. Alluxio acts as an in-memory HDFS while communicating with different underlying storage systems like S3 or Lustre simultaneously. Compared with HAL, Alluxio provides data locality information if data were stored in Alluxio worker. Moreover, it allows the applications to access HDFS or Lustre file system by mounting them to Alluxio namespace without modifying configuration files.

We compare the performance of HAL and Alluxio via HiBench benchmark. All input data are stored in Lustre and we run each workload in data analytics environment where each node can write the intermediate data to local disk. We did not use the shuffle strategy implemented in HAL because of the existence of local disk. Figure 2 shows the performance comparison of different data access strategy where hdfs-put means copy data from Lustre to HDFS before running the workload, hadoop-hdfs represents Hadoop workload read input data from HDFS and hadoop-alluxio represents Hadoop workload read input data from Lustre directly via Alluxio by such analogy. All the intermediate data are stored in the local disk. Obviously, both Alluxio and HAL provide an efficient way for Hadoop and Spark workloads to read data stored in Lustre without redundant data movement.

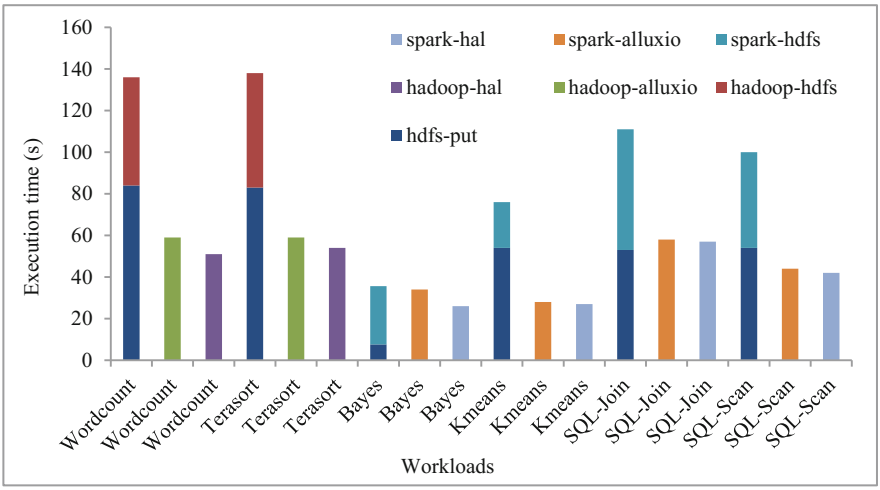


Fig. 2. HiBench benchmark performance

It is worth mentioning that Hadoop/Sprak over HAL run faster than over Alluxio. The reason behind this phenomenon is that Alluxio provides HDFS view for user application which in turn makes a more complicated data access process than HAL. In Fig. 3, to access a file that stored in underlying file system, a client requests the Alluxio master to get file metadata. After receiving the metadata request, the master will create an inode object and generate file metadata in HDFS metadata format according to the information from underlying file system and sent back to the client. According to the metadata, the client will construct the underlying file system info and sent to Alluxio worker if the file is not stored in Alluxio and the block locations info is null. After receiving the data requests from the client, Alluxio worker will create a packet reader and act as a client of underlying file system to read file data and sent them back. Data access in HAL is much simpler because each client sent data access request to underlying Lustre file system directly without providing HDFS function for user application.

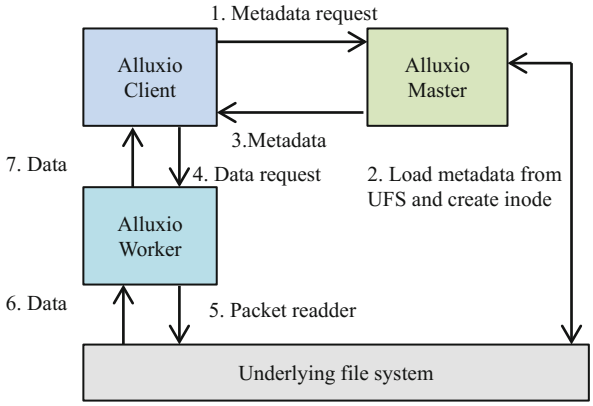


Fig. 3. Reading a file stored in underlying file system in Alluxio

Although the data access performance of HAL is better, Alluxio can provide a memory-centric distributed storage space and is compatible with different underlying file system simultaneously. We believe that Alluxio is a better choice for complicated applications. To validate our assumption, we define a complex application that consists of RandomWriter and Sorter. RandomWriter simulates HPC workload, which is executed in HPC environment and generates 10 GB data per node. Sorter simulates data analytic workload that analyzes the output data of RandomWriter in Hadoop environment. The results are shown in Fig. 4. We scale the number of nodes from 1 to 32, RandomWriter-HAL indicates that RandomWriter writes data into Lustre and Sort-HAL read data from Lustre via HAL. RandomWriter-Alluxio represents the output data of RandomWriter are stored in Alluxio and Sort-Alluxio can read data from Alluxio directly. Obviously, Sort-Alluxio spent less time than Sort-HAL since the output data are stored in memory and can be accessed directly while Sort-HAL needs to read data from underlying Lustre file system.

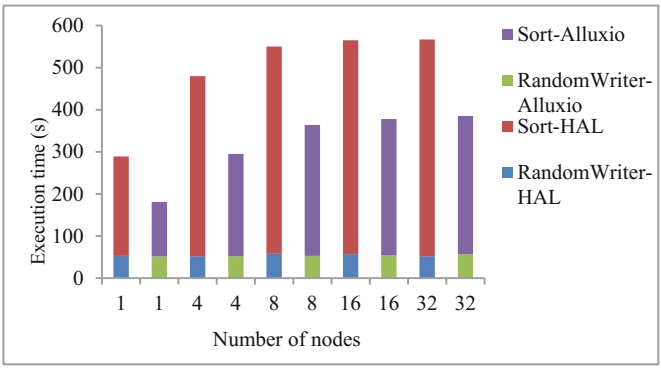


Fig. 4. Simulation of complex application

3.3 The Impact of Storage Architecture

Alluxio provides a feasible solution for big data analytics frameworks like Hadoop and Spark to access data stored in Lustre rather than HDFS. However, the key distinction between HPC and data analytics environment is the storage architecture. In data analytics environment, the intermediate data generated during shuffle phase can be stored in the local file system on top of the local disk, while in HPC environments, these data need to be stored in underlying parallel file system. The location of intermediate data is a critical factor that influences shuffling performance, which will further dominate a MapReduce job execution time. In this section, we characterize the impact of storage architecture on data-intensive MapReduce jobs when using Lustre as the underlying file system.

First of all, we run IOZone benchmark on Lustre and local disk to analyze the basic file system performance. The results are shown in Fig. 5: In our test environments, Lustre provides a much higher read or write bandwidth than local disk because files stored in Lustre are broken into stripes, which are typically stored on multiple object storage targets (OSTs), allowing parallel read and write access to different parts of the file. In contrast, the bandwidth of local disk is limited and may become the bottleneck of HDFS that build on it.

Secondly, to investigate the influence of metadata latency, we run Mdtest benchmark, a MPI-coordinated metadata benchmark that performs open/stat/close operations on files and directories and then reports the performance. Figure 6 shows the result of Mdtest, local file system performs much better than Lustre as expected. Local file system can perform 32397 file create operations and 240534 file read operations per second while Lustre can perform only 880 file create operations and 1578 file read operations per second. Lustre file system uses a limited number of MDSs to manage the file metadata, the centralized metadata management is a potential bottleneck and will result in serious performance loss when gigantic metadata requests need to be processed during shuffle phase.

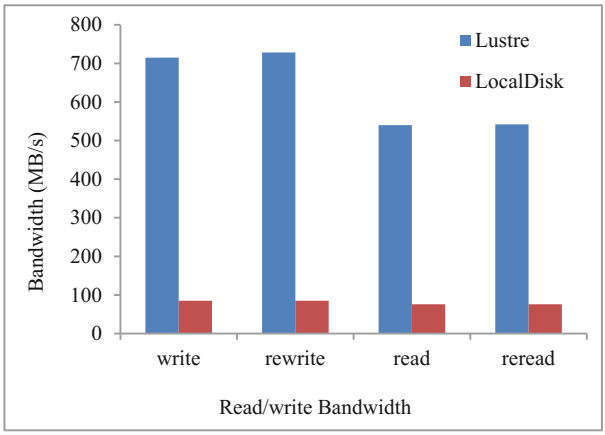


Fig. 5. IOZone benchmark performance

To validate the speculation, we run the terasort workload of HiBench benchmark in both HPC and data analytics environment where the intermediate data are stored in Lustre file system and local disk respectively. During the evaluation, terasort workload reads the input data from Lustre via HAL or Alluxio and writes the output back to Lustre, the generated intermediate data size is equal to the input size. Figure 7 illustrates the performance of terasort when intermediate data resides in different storage architectures. HAL-Local disk means that terasort workload read data from Lustre via HAL and stored intermediate data in local disk, while HAL-Lustre means intermediate data are stored in Lustre. In general, as data size grows, Lustre-based intermediate data storage results in serious performance loss and HAL-Lustre performs even worse than Alluxio-Lustre. The reason of performance differences can be ascribed to the characteristic of shuffle phase and the metadata operation bottleneck of Lustre.

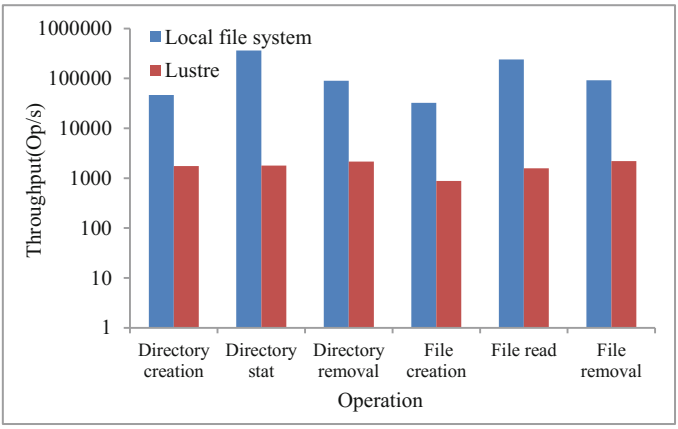


Fig. 6. Metadata operation throughput of Lustre and local file system

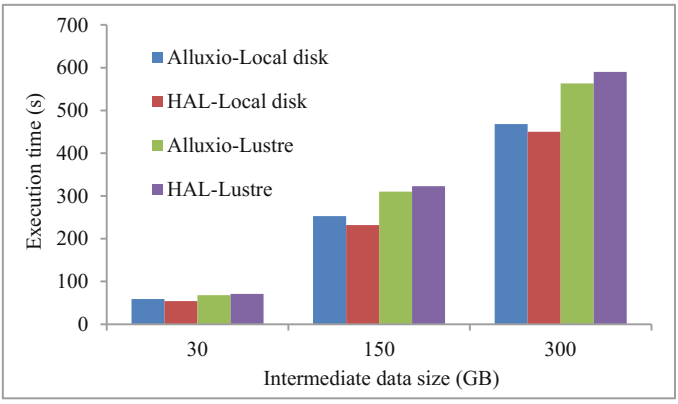


Fig. 7. Impact of intermediate data location

The shuffle phase of MapReduce jobs is shown in Fig. 8: (1) each map task is assigned a portion of the input file and applies the user-defined map function on each key-value pair. The processed key-value data are stored in a memory buffer named kvbuffer first and will be spill to the intermediate data directory every time the available space of kvbuffer is less than 20%. These spill files will be sorted and merged into one output file before each map task finished. (2) Reduce tasks starts fetching these intermediate data that stored in local disk form each node after all map tasks finished. These data from different map output files are sorted and merged again to generate one final input data for each reduce task. Overall, the shuffle phase contains gigantic file create and read/write operations and is sensitive to network latency and disk bandwidth.

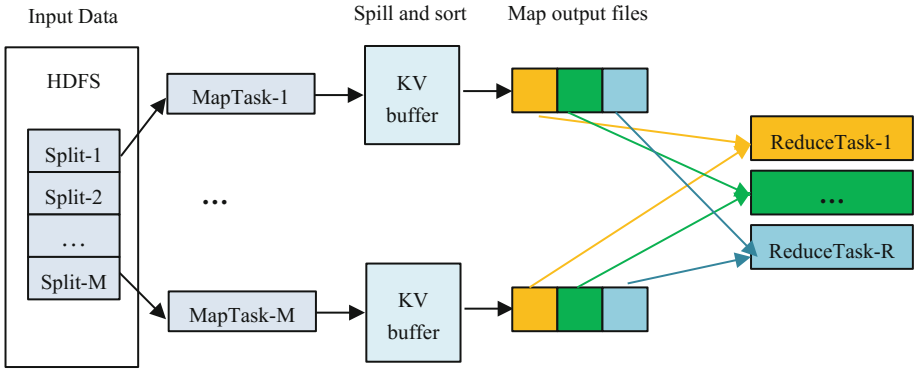


Fig. 8. The shuffle phase of MapReduce jobs

During the shuffle phase, each map/reduce task sent file create/open requests to local file system to write/read intermediate data in data analytics environment with local disk, and its performance is subject to network latency and disk latency. In HPC environment, however, these requests will be sent to the underlying parallel file system, and its performance is subject to metadata latency along with network latency and disk latency.

The reason why HAL-Lustre performs worse than Alluxio-Lustre is that the shuffle strategy in HAL will generate more intermediate data. To prevent repetitive data movements, HAL reimplements the shuffle phase to allow each reduce task retrieve data from Lustre directly. In default shuffle strategy, each map task will generate one intermediate file for all reduce tasks every time the kvbuffer spill the data to local disk. These intermediate files generated from the same map task will be merged into one final output file and is fetched by reduce tasks. The total number of intermediate files will be n * M, where n represents the number of spill operations and M represents the number of map tasks. However, in HAL shuffle strategy, each map task will generate one intermediate file for each reduce task and all the intermediate files that belong to one reduce task are stored in one directory. The total number of intermediate files will be n * M * R, where R represents the number of reduce tasks. HAL shuffle strategy can avoid the merge phase of map tasks and prevent the repetitive data movements cost, but the metadata operation cost to gigantic intermediate files result in the performance loss.

In summary, when using Lustre as the underlying file system of big data analytics frameworks, Lustre can provide higher aggregate bandwidth than traditional HDFS that build on top of the local disk, but the costly metadata operation may result in serious performance loss if massive intermediate data were stored in Lustre.

4 Optimization and Evaluation

To support data analytics frameworks on compute-centric HPC systems effectively, there are two performance issues that need to be addressed based on the characterization of Sect. 3.3. Firstly, the total number of intermediate files that generated in shuffle phase need to be reduced. Secondly, the costly metadata operations need to be accelerated. Accordingly, we propose two optimizations: shared map output shuffle strategy and file metadata cache layer.

4.1 Shared Map Output Shuffle Strategy

Shared map output shuffle strategy is intended to reduce the total number of intermediate files while utilizing the shared file system view provided by Lustre. The design of this shuffle strategy is shown in Fig. 9.

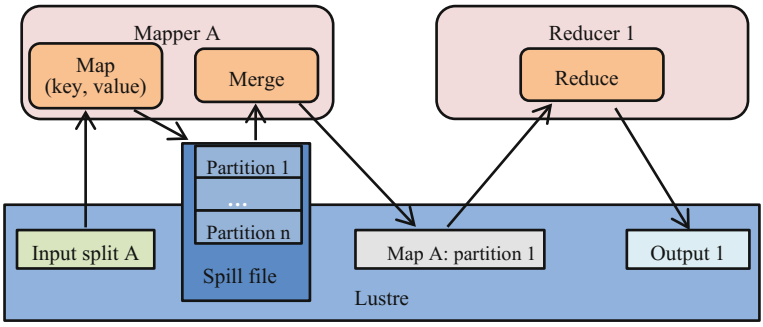


Fig. 9. Shared map output shuffle strategy

Firstly, each map task generates one intermediate file every time the kvbuffer spill the data to underlying Lustre file system. Each spill file contains multiple partitions and every partition stores the data that corresponding to one reduce task. Secondly, all the spill files generated by each map task are retrieved and sorted. Due to the effect of large memory space in a compute node, it is likely that those spill files still reside in local memory and can be retrieved quickly. Finally, the sorted map output data are stored in multiple intermediate files where each intermediate file contains the data that belongs to one reduce task. In other words, each map task generates R intermediate files no matter how many spill operations it went through, where R is the number of reduce tasks. The intermediate files that will be processed by the same reduce task are stored in the same directory.

The proposed shuffle strategy has several advantages: (1) It can utilize the effect of large memory space in a compute node and reduce the time of retrieving spill files. (2) Compared with the shuffle strategy of HAL described in Sect. 3.3, the proposed shuffle strategy can reduce the total number of intermediate files form n * M * R to M * R, where n represents the number of spill operations, M represents the number of map tasks and R represents the number of reduce tasks. Therefore, the number of costly metadata operations can be reduced. (3) Compared with default shuffle strategy, each reduce task can fetch the map output files from the corresponding directory via Lustre directly without repetitive data movements cost.

4.2 File Metadata Cache Layer

We introduce file metadata cache layer to facilitate metadata operations. During the shuffle phase, each map task creates multiple segments to fetch data partition by partition, where a segment represents a partition of a spill file. The spill file will be opened every time a segment is created and closed after each segment is closed. As a result, each spill file will be opened and read multiple times. In data analytics environments, file open and read requests can be processed quickly, while in HPC systems that deployed a parallel file system, these requests would take more time because of the centralized metadata bottleneck.

To alleviate the stress of metadata server, we implement a file metadata cache layer. Each map task initiates a key-value data structure, where the key is the path of the file and the value is the corresponding file descriptor. Once a file is opened, the file descriptor is cached in the pool and subsequent file open requests can retrieve the file descriptor and create a new file input stream. To keep the original file read operations unchanged, we initiate the file metadata cache layer only in shuffle phase and disable it as soon as the merging of spill files is finished. If file metadata cache layer was enabled, file open operation will query this pool to get corresponding file descriptor based on file name. If the response is not null, a new file input stream will be created based on the retrieved file descriptor. If the response is null, which means this file has not been opened before, this file will be opened via Java FileInputStream and the file descriptor will be cached.

In the current implementation, the capacity of the key-value data structure is set to 20 based on experiments. Besides, we use the first in first out (FIFO) eviction policies to solve capacity conflicts.

4.3 Evaluation

To validate the effectiveness of our proposed optimizations, we run the terasort workload of HiBench benchmark in the HPC environment, and the results are shown in Fig. 10. Default strategy represents the default shuffle strategy of Hadoop. HAL represents the original HAL shuffle strategy without optimizations. Shared shuffle represents the proposed shared map output shuffle strategy and file metadata cache represents using file metadata cache layer with shared map output shuffle strategy together. We vary the data size from 300 GB to 1500 GB and all the intermediate data are stored in Lustre file system.

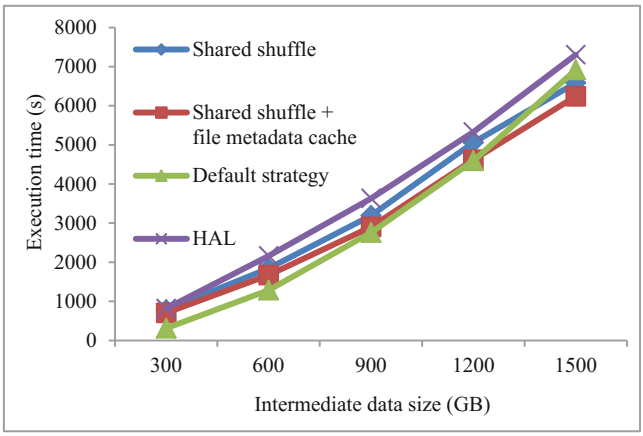


Fig. 10. Performance of HiBench-Terasort

When intermediate data size is less than 1200 GB, the default shuffle strategy performs the best since the data server that serves the requests of reduce tasks can fetch the intermediate data from local memory due to the effect of large buffer cache in a compute node. As intermediate data size grows, the memory space of compute node is insufficient to store all the intermediate data, data servers need to fetch data from Lustre and sent them back to reduce tasks. The repetitive data movement cost in default shuffle strategy results in the performance loss.

In contrast, our proposed optimizations allow reduce tasks to fetch data from underlying Lustre file system directly without repetitive data movement cost. Moreover, it reduces the total number of intermediate files and shows obvious performance benefit compared to original HAL shuffle strategy. For 1500 GB data size, shared map output shuffle strategy has a performance benefit of 11% compared to HAL and it can provide 17% benefit when file metadata cache layer is used together.

5 Related Work

In recent years, big data analytics has gained great success in different fields. Previous work [9, 28–30] has identified and analyzed the characteristic of bid data and discussed the big data challenges, including data storage and transfer, data security, scalability of data analytics systems etc.

Many efforts have been conducted to integrate big data analytics frameworks with HPC infrastructure. Chaimov et al. [15] ported Spark on Cray XC systems and evaluate a configuration with SSDs attached closer to compute nodes for I/O acceleration. Wang et al. [16] characterizes the performance impact of key differences between compute-centric 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. Wasi-ur-Rahman et al. [31] proposed a high-performance design for running YARN MapReduce on HPC clusters by utilizing

Lustre as the storage provider for intermediate data and introduced RDMA-based shuffle approach. These works analyzed the performance differences when deployed Hadoop and Spark on HPC systems, but they were lacking in providing optimizations for complex applications.

In-memory file systems like MemFS [21], FusionFS [22] and AMFS [23] are developed to alleviate the storage pressure, but they provide limited compatibility with underlying file systems. Two-level storage [32] is the closest research work that integrates an upper-level in-memory file system with a lower-level parallel file system for accelerating Hadoop/Spark workloads on HPC clusters. However, it lacks an in-depth discussion on the performance impact of data-intensive analytics workloads when using Lustre as underlying file system. In this paper, we make a detailed comparison of system architectures and provide two optimizations to alleviate the metadata bottleneck of Lustre.

There are also many research works that directly deployed big data analytics frameworks atop of existing parallel file systems. Maltzahn et al. [17] describe Ceph and its elements and provide instructions for installing a demonstration system that can be used with Hadoop. Yang et al. [18] propose PortHadoop, an enhanced Hadoop architecture that enables MapReduce applications reading data directly from HPC parallel file systems. Xuan et al. [32] present a two-level storage system that integrates an upper-level in-memory file system with a lower-level parallel file system. Comparing with previous works, this paper presents our experiences of converging big data analytics frameworks with the Tianhe-2 system. We aim at the growing need of complex applications and provide a feasible solution to accelerate it by utilizing an in-memory file system.

6 Conclusion

In this paper, we deployed an in-memory file system as data access middleware to reconcile the differences of system architectures between HPC systems and big data analytics systems. We characterized the impact of storage architecture on big data analytics frameworks when using Lustre as underlying file system. The result of experiments shows that the centralized metadata management of Lustre is a potential bottleneck and can result in serious performance loss when gigantic intermediate data are stored in Lustre. To alleviate the impact of metadata bottleneck and accelerate data-intensive MapReduce applications in HPC environments, we proposed shared map output shuffle strategy and file metadata cache layer. Our results ensure up to 17% performance benefit for data-intensive workloads.

Overall, it's critical to find a solution that can accelerate complex applications under the converging trend of HPC and big data analytics. Our work provides useful experience and gives a feasible solution. In the future, we plan to investigate the performance impact of big data analytics frameworks when high-capacity NVM is equipped in compute nodes and provide better data management strategy based on the results.

Acknowledgment. This work was supported by National Nature Science Foundation of China under Grant No. U1611261 and No. 61433019, the National Key R&D Program of China 2017YFB0202201, and the Program for Guangdong Introducing Innovative and Entrepreneurial Teams under Grant No. 2016ZT06D211.

References

1. Fu, H.H., Liao, J.F., Yang, J.Z., Wang, L.N., Song, Z.Y., Huang, X.M., et al.: The Sunway TaihuLight supercomputer: system and applications. Sci. China Inf. Sci. **59**(7), 1–16 (2016)
2. Liao, X.K., Xiao, L.Q., Yang, C.Q., Lu, Y.T.: Milkyway-2 supercomputer: system and application. Front. Comput. Sci. **8**(3), 345–356 (2014)
3. Titan - Cray XK7 (2017). https://www.olcf.ornl.gov/titan/
4. Wang, F., Yang, C.Q., Du, Y.F., Chen, J., Yi, H.Z., Xu, W.X.: Optimizing Linpack benchmark on GPU-accelerated petascale supercomputer. J. Comput. Sci. Technol. **26**(5), 854–865 (2011)
5. Yang, C., Wu, Q., Tang, T., Wang, F., Xue, J.: Programming for scientific computing on peta-scale heterogeneous parallel systems. J. Cent. South Univ. **20**(5), 1189–1203 (2013)
6. French, S., Zheng, Y., Romanowicz, B., Yelick, K.: Parallel Hessian assembly for seismic waveform inversion using global updates. In: IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 753–762. IEEE (2015)
7. Bhandarkar, M.: MapReduce programming with apache Hadoop. In: IEEE International Symposium on Parallel and Distributed Processing (IPDPS), p. 1 (2010)
8. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., Mccauley, M.: Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: USENIX Conference on Networked Systems Design and Implementation, p. 2 (2012)
9. Kambatla, K., Kollias, G., Kumar, V., Grama, A.: Trends in big data analytics. J. Parallel Distrib. Comput. **74**(7), 2561–2573 (2014)
10. Reed, D.A., Dongarra, J.: Exascale computing and big data. Commun. ACM **58**(7), 56–68 (2015)
11. NASA Center for Climate Simulation (2017). http://www.nasa.gov/topics/earth/features/climate-sim-center.html
12. InfiniBand Homepage (2017). http://www.infinibandta.org/
13. Donovan, S., Kleen, A., Wilcox, M., Huizenga, G., Hutton, A.J.: Lustre: building a file system for 1,000-node clusters. In: Proceedings of the Linux Symposium, p. 9 (2003)
14. Shvachko, K., Kuang, H., Radia, S., Chansler, R.: The Hadoop distributed file system. In: MASS Storage Systems and Technologies, pp. 1–10 (2010)
15. Chaimov, N., Malony, A., Canon, S., Iancu, C., Ibrahim, K.Z., Srinivasan, J.: Scaling Spark on HPC systems. In: Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC), pp. 97–110 (2016)
16. Wang, Y., Goldstone, R., Yu, W., Wang, T.: Characterization and optimization of memory-resident MapReduce on HPC systems. In: IEEE International Symposium on Parallel and Distributed Processing (IPDPS), pp. 799–808 (2014)
17. Maltzahn, C., Molinaestolano, E., Khurana, A., Nelson, A.J., Brandt, S.A., Weil, S.: Ceph as a scalable alternative to the Hadoop distributed file system. The Magazine of USENIX and SAGE, pp. 38–49 (2010)
18. Yang, X., Liu, N., Feng, B., Sun, X.H., Zhou, S.: PortHadoop: support direct HPC data processing in Hadoop. In: IEEE International Conference on Big Data, pp. 223–232 (2015)

19. Fadika, Z., Dede, E., Govindaraju, M., Ramakrishnan, L.: MARIANE: MApReduce implementation adapted for HPC environments. In: International Conference on Grid Computing, pp. 82–89 (2011)
20. Li, H., Ghodsi, A., Zaharia, M., Shenker, S., Stoica, I.: Tachyon: reliable, memory speed storage for cluster computing frameworks. In: Proceedings of the ACM Symposium on Cloud Computing, pp. 1–15. (2014)
21. Uta, A., Sandu, A., Costache, S., Kielmann, T.: Scalable in-memory computing. In: International Symposium on Cluster, Cloud and Grid Computing, pp. 805–810 (2015)
22. Zhao, D., Zhang, Z., Zhou, X., Li, T.: FusionFS: toward supporting data-intensive scientific applications on extreme-scale high-performance computing systems. In: IEEE International Conference on Big Data, pp. 61–70 (2014)
23. Zhang, Z., Katz, D.S., Wozniak, J.M., Espinosa, A.: Design and analysis of data management in scalable parallel scripting. In: International Conference on High PERFORMANCE Computing, Networking, Storage and Analysis, pp. 1–11 (2012)
24. IOzone Filesystem Benchmark (2017). http://www.iozone.org/
25. MDTest Metadata Benchmark (2017). https://github.com/MDTEST-LANL/mdtest
26. Huang, S., Huang, J., Dai, J., Xie, T., Huang, B.: The HiBench benchmark suite: characterization of the MapReduce-based data analysis. In: International Conference on Data Engineering Workshops, pp. 41–51 (2010)
27. Hadoop Adapter for Lustre (HAL) (2017). https://github.com/intel-hpdd/lustre-connector-for-hadoop
28. Hu, H., Wen, Y., Chua, T.S., Li, X.: Toward scalable systems for big data analytics: a technology tutorial. IEEE Access **2**(1), 652–687 (2017)
29. Brohi, S.N., Bamiah, M.A., Brohi, M.N.: Identifying and analyzing the transient and permanent barriers for big data. J. Eng. Sci. Technol. **11**(12), 1793–1807 (2016)
30. Tolle, K.M., Tansley, D.S.W., Hey, A.J.G.: The fourth paradigm: data-intensive scientific discovery [point of view]. Proc. IEEE **99**(8), 1334–1337 (2011)
31. Wasi-ur-Rahman, M., Lu, X., Islam, N.S., Rajachandrasekar, R., Panda, D.K.: High-performance design of YARN MapReduce on modern HPC clusters with Lustre and RDMA. In: IEEE International Symposium on Parallel and Distributed Processing (IPDPS), pp. 291–300 (2015)
32. Xuan, P., Ligon, W.B., Srimani, P.K., Ge, R., Luo, F.: Accelerating big data analytics on HPC clusters using two-level storage. Parallel Comput. **61**, 18–34 (2016)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.