Lu YT, Cheng P, Chen ZG. Design and implementation of the Tianhe-2 data storage and management system. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 35(1): 27–46 Jan. 2020. DOI 10.1007/s11390-020-9799-4

Design and Implementation of the Tianhe-2 Data Storage and Management System

Yu-Tong Lu[1], *Distinguished Member*, *CCF*, Peng Cheng[2], and Zhi-Guang Chen[1], *Member*, *CCF*

[1] *National Supercomputer Center in Guangzhou, Sun Yat-sen University, Guangzhou 510000, China*
[2] *College of Computer, National University of Defense Technology, Changsha 410073, China*

E-mail: {yutong.lu, peng.cheng, zhiguang.chen}@nscc-gz.cn

Received July 15, 2019; revised October 14, 2019.

Abstract With the convergence of high-performance computing (HPC), big data and artificial intelligence (AI), the HPC community is pushing for "triple use" systems to expedite scientific discoveries. However, supporting these converged applications on HPC systems presents formidable challenges in terms of storage and data management due to the explosive growth of scientific data and the fundamental differences in I/O characteristics among HPC, big data and AI workloads. In this paper, we discuss the driving force behind the converging trend, highlight three data management challenges, and summarize our efforts in addressing these data management challenges on a typical HPC system at the parallel file system, data management middleware, and user application levels. As HPC systems are approaching the border of exascale computing, this paper sheds light on how to enable application-driven data management as a preliminary step toward the deep convergence of exascale computing ecosystems, big data, and AI.

Keywords high-performance computing (HPC), data management, converged application, parallel file system

1 Introduction

As the high-performance computing (HPC) community moves toward the exascale, big data and artificial intelligence (AI) are being hailed as the fourth paradigm of science and have gained widespread success in many fields, including astronomical image processing [1], weather prediction [2] and biological data analysis [3]. Driven by applications and algorithms, HPC, big data and AI are converging to expedite scientific discoveries. On the one hand, scientists demand converged applications to obtain new insights into various scientific domains. The fusion of HPC and big data emanates high-performance data analytics (HPDA) to extract values from massive scientific datasets via extreme data analytics at scale [4]. The fusion of HPC and AI emanates AI-enhanced HPC, which aims to improve traditional HPC models by optimizing the parameter selections or training an AI model as an alternative component [5]. On the other hand, classical HPC algorithms shared by big data and AI are the internal impetus for convergence. For example, dense linear algebra [6] and backtrack and branch-and-bound algorithms [7] are commonly used in all three scenarios. Since these algorithms have been highly optimized in HPC systems, supporting higher-level big data and AI applications on HPC systems could lead to superior performance improvements.

Nevertheless, system architectures are designed to best support the typical workloads running on clusters. Traditional HPC systems aim to provide the maximum computational density and local bandwidth with a given power/cost constraint for compute-intensive workloads. In aspects of storage and I/O, parallel file systems (PFSs, e.g., Lustre [8]) are prevalent to support HPC applications, which are characterized by high bandwidth and burst I/O requests, larger volumes of outputs relative to inputs, and well-structured data

Regular Paper
Special Section on Selected I/O Technologies for High-Performance Computing and Data Analytics
This work is supported by the National Key Research and Development Program of China under Grant No. 2016YFB1000302, the National Natural Science Foundation of China under Grant Nos. U1611261 and 61872392, and the Program for Guangdong Introducing Innovative and Entrepreneurial Teams under Grant No. 2016ZT06D211.

commonly stored in self-descriptive data formats (e.g., HDF5① and NetCDF [9]). In contrast, big data applications are supported by distributed file systems (e.g., HDFS [10]) and characterized by larger volumes of inputs relative to outputs. Most files are modified by appending new data rather than overwriting existing data, and data are typically unstructured or semistructured. The storage and I/O features of AI applications are similar to those of big data applications.

The largely different I/O characteristics among HPC, big data and AI workloads lead to considerable complexity of the converged applications. Moreover, massive heterogeneous scientific data, close cooperation between geo-distributed HPC clusters [11], and diverse storage systems (e.g., Lustre, HDFS, and MongoDB②) that suit different scenarios add complexity to data management, causing PFSs to experience serious challenges in supporting converged applications.

First, the dramatically increased data volume and the complexity of the converged applications exacerbate the I/O bottleneck. Whereas computing infrastructure innovation is driven by Moore's law, I/O and storage have lagged far behind computing. For example, compute core concurrencies of exascale machines are expected to increase approximately 4 000 times compared with early PFlops machines, but storage performance in the same time period is predicted to improve only 100 times [12]. This imbalanced development between computing and storage leads to the I/O bottleneck that limits system utility and applicability. Because converged applications couple simulations, data analytics, and learning workloads to process TB- or even PB-scale datasets [13, 14], the complex data access patterns and frequent data sharing requirements between coupled modules present serious challenges in I/O performance.

Second, HPC systems require adaptive or intelligent data management optimizations to accommodate diverse application requirements. The convergence of HPC, big data and AI has introduced variety in aspects of I/O patterns, data types, and storage systems. However, the existing one-fits-all data management strategy fails to realize the hidden benefits of HPC systems. As an example, many modern HPC systems generally contain multiple storage tiers to improve the I/O performance, such as the shared burst buffer in the Cori system [15] and the node-local SSD in the Summit system③. Although different data placement strategies could lead to widely varying I/O performance, the one-fits-all strategy that uses the top storage tier (e.g., SSD tier) as the performance tier and uses a lower storage tier (e.g., HDD tier) as the capacity tier might lead to an inefficient use of the hierarchical storage architecture because of load imbalance [16] and resource contention [17].

Third, there is an increasing need for unified data management mechanisms to support the analysis, processing, and storage of massive heterogeneous scientific data. In HPC systems, most scientific datasets are managed by PFSs [18]. As scientific data are rapidly increasing in data size and variety, PFS alone is difficult for matching the diverse use cases, such as query processing [19] and efficient data reduction [20–22]. Providing a unified data management mechanism that not only enriches the utility of PFSs but also reconciles the divergence of different data models is vital to accelerate scientific discoveries. However, such unified data management is extremely challenging and has a wide range of issues, including distributed and scalable metadata management, unified data access interface across different storage systems, efficient index and data locating services at the granularity of both a file and a record, and the management of diverse global namespaces stemming from geo-distributed HPC clusters.

In this paper, we summarize our efforts in solving these data management challenges to support the converged applications on the Tianhe-2 system [23], which has captured the number one spot on the TOP500 list④ 6 times. Specifically, we look back at the initial design of the hybrid hierarchy file system (H2FS) [24] and discuss how we optimize metadata management and small file management to embrace converged workloads. We also propose a spectrum of data management optimizations to fit the hierarchical storage architecture and diverse application demands. Together with application-specific optimizations, converged applications are able to run efficiently on the Tianhe-2 system.

Although our previous studies have addressed a piece of the puzzle, each of them fails to meet all the requirements of converged applications alone. For example, while Pream [25] accelerates metadata operations by preallocating metadata and initializing a proxy server

①https://www.hdfgroup.org/solutions/hdf5, Jun. 2019.

②https://www.mongodb.com/white-papers, Feb. 2019.

③https://www.olcf.ornl.gov/for-users/system-user-guides/summit, May 2019.

④https://www.top500.org/, Jun. 2019.

to serve metadata requests, the I/O bottleneck remains unsolved. Although tiered data management [26] takes advantage of the hierarchical storage architecture and workflow data access patterns to optimize the I/O performance, locality-agnostic task placement results in redundant data movement. Compared with our earlier publications, this paper extends our earlier designs and proposes complementary optimizations to build a more comprehensive runtime environment for diverse workloads. Our contributions in this paper are summarized as follows.

• We reveal the differences among HPC, big data, and AI workloads and detail three data management challenges when supporting converged applications on HPC systems.

• We propose file system optimizations in terms of metadata and small files to enable high-throughput metadata processing and low-latency file access.

• We propose data management optimizations and application-specific optimizations to provide better support for converged applications.

The remainder of this paper is organized as follows. We discuss some related studies in Section 2 and review the design of H2FS in Section 3. Sections 4–6 present our efforts in solving the aforementioned data management challenges at the levels of underlying file system, data management middleware and user application, respectively. We conclude the paper and discuss future work in Section 7.

2 Related Work

With the convergence of HPC, big data and AI, data management on HPC systems has received increased attention. We detail works related to I/O optimization, adaptive and intelligent storage optimization, and unified data management in the following subsections.

2.1 Hardware and Software Consolidated I/O Optimizations

Hierarchical staging frameworks are increasingly used in HPC systems to improve the I/O performance. Hermes [27] is a multi-tiered I/O buffering system that provides three data placement policies to efficiently utilize all storage tiers. UniviStor [28] provides an integrated storage subsystem with multiple layers of storage. All data are first written to memory-mapped log-structured files for performance and then spilled to lower storage tiers once their size exceeds a predefined threshold. ARCHIE [29] is a hierarchical caching framework for array data. It utilizes array data access patterns to prefetch data, thereby optimizing the I/O performance. Nevertheless, the potential of heterogeneous devices is restrained by the redundant data movement caused by locality-agnostic task placement. In contrast to these studies, we customize data management strategies for common data access patterns of scientific workflows and propose complementary data-aware task scheduling to make full use of different storage tiers. Many efforts have been made to provide better support for the converged applications, such as integrating PFS and HDFS to prevent repetitive data movement between different storage systems [30], optimizing shuffle strategies for big data analytics frameworks to fully utilize HPC systems [31], preventing redundant data access and applying speculative parallel prefetch operations to optimize the I/O access of a deep learning framework [32]. Whereas these studies propose optimizations for a specific framework (e.g., Hadoop[5] and Caffe [33]), we optimize the underlying PFS in terms of metadata management and small files to provide general support for big data and AI scenarios.

2.2 Emerging Adaptive and Intelligent Storage Optimization

Due to the largely different I/O patterns of diverse applications, providing a general data management strategy that suits all scenarios is challenging. One of the potential solutions is to enable adaptive and intelligent storage optimization [34, 35]. Stacker [36] leverages hierarchical n-grams models to predict upcoming read requests and prefetch data from burst buffer to memory spaces intelligently. To optimize data placement on a hierarchical storage architecture, OctopusFS [16] explores the idea of using multiobjective optimization techniques for making intelligent data management decisions based on the requirements of fault tolerance, data and load balancing, and throughput maximization. Tahoe [37] characterizes memory access patterns of tasks and uses this information to decide the optimal data placement for multiple tasks. Compared with these studies, we treat selecting the optimal storage tier as a multiclassification problem and use machine learning techniques to make smart data placement strategies under varied data access patterns and runtime system statuses.

[5] http://hadoop.apache.org/, Aug. 2019.

2.3 Unified Data Management for Diverse Use Cases

Several data management systems have been used to manage massive heterogeneous scientific data, such as iRODS⑥, SciDB [38] and ArrayUDF [39]. Although these data management systems provide data locating services, scientific data must be explicitly imported into these systems before they can be queried. In contrast, the index and query module (namely, UniIndex) presented in this paper prevents data movement cost by extracting metadata from existing files and applying an in-situ indexing strategy. To enable record locating services, FastQuery [40] applies bitmap indexing techniques and provides parallel query frameworks to accelerate data selection on HDF5 and NetCDF data formats. Chiu *et al.* [41] developed an in-memory version of FastQuery, which combines distributed shared memory, spatial data layout reorganization, and location-aware parallel execution to provide better support for in-situ processing scenarios. Dong *et al.* [42] proposed multidimensional binning and a spatially clustered join algorithm to enable query processing with less I/O cost and fast query response. Gu *et al.* [43] designed a query interface for a high-level I/O library to allow arbitrary combinations of range conditions on known variables. Wu *et al.* [44] used the idea of the block index to enable efficient point and small-range queries. However, these studies store the indexes in a separate index file, which may lead to potential write contention and extra index load costs when processing query requests [45]. Compared with these studies, UniIndex allows users to enable in-situ indexing and organizes the indexes as key-value (KV) pairs to accelerate both index building and query processing.

3 Design of H2FS

With increasing data scale, the conventional centrally shared storage architecture suffers from I/O request contention since a large number of I/O clients often request the shared I/O resources of one storage server. In addition, applications with bursty I/O patterns exacerbate the shortcomings [46]. For this reason, the Tianhe-2 system adopts a novel hybrid hierarchy storage architecture to support the high scalability of I/O clients and improve the burst I/O bandwidth. Specifically, the upper layer is composed of I/O nodes (IONs), where each of them is equipped with two 1 TB PCIe SSDs. The bottom layer is composed of storage servers, including metadata servers (MDSs) and object storage servers (OSSs). The upper layer is the local storage that provides sufficient burst performance for applications. The bottom layer is the shared storage that enables a large storage capacity. To fit this architecture and optimize the I/O performance, a user-level virtualized file system named H2FS was designed and implemented.

Fig.1 illustrates the architecture of the H2FS file system. H2FS introduces the data processing unit (DPU) and hybrid virtual namespace (HVN) to combine the local storage and the shared storage into a dynamic single namespace. Each DPU tightly couples an ION with its local storage to provide service for N computing nodes, where N is determined by deployment of the system. HVN provides a virtualized storage space by aggregating several DPUs and shared storage in a unified single namespace. Each HVN is assigned to an application instance exclusively and can be created/released dynamically. The file data in this unified namespace can be located in DPUs, shared storage or duplicated in both, which is determined by the application at runtime. For the interface, H2FS keeps the POSIX interface for compatibility and uses the layout API, policy API, and HVN management API jointly for performance optimization.

H2FS benefits both data-intensive applications and typical HPC applications by providing three predefined I/O modes: local I/O, global I/O, and hybrid I/O. In local I/O mode, H2FS uses the local storage layer to serve all the I/O requests and asynchronously moves data to the shared storage layer. In global I/O mode, H2FS directly forwards I/O requests to the shared storage layer without depositing data into local storage. Specifically, local storage is completely transparent to applications for the above two scenarios. In hybrid I/O mode, H2FS combines the local storage layer and the shared storage layer into a single namespace. With the support of the extended layout APIs, data placement can be explicitly controlled by users. For example, users are allowed to use the local storage layer as the burst buffer to stage all the intermediate data generated from simulation workloads. Subsequent data analysis workloads can read data from the local storage layer directly and write the output to the shared storage layer for persistence.

⑥https://irods.org/, Jun. 2019.

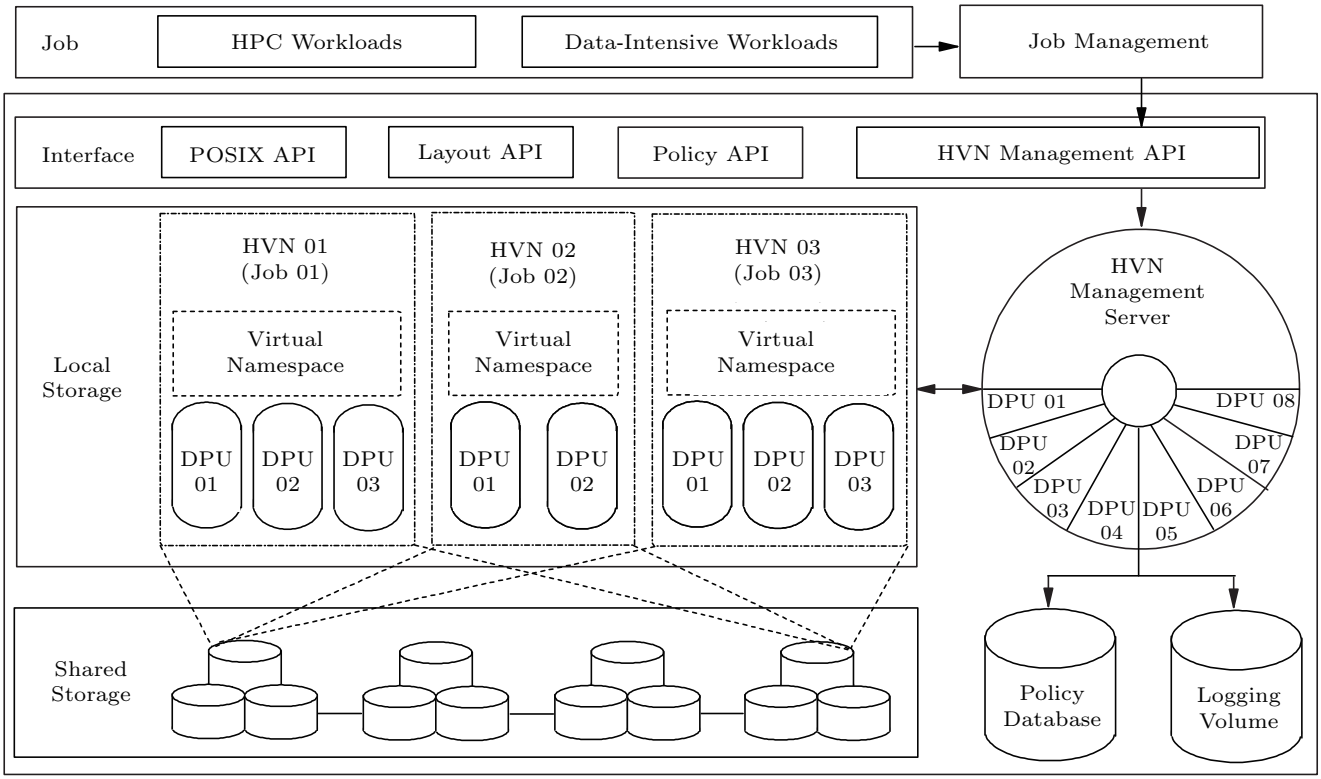

Fig.1. Architecture of H2FS.

4 Further Optimizations to H2FS

H2FS was originally designed for scalable HPC applications; however, because HPC, big data and AI are poised to converge, the file system must be optimized further to adapt to hybrid workloads. In contrast to traditional HPC applications, big data and AI applications are more likely to generate and access a large number of relatively small files. This characteristic imposes a serious negative impact on the metadata management and access latency of small files. Accordingly, we further optimize H2FS in terms of metadata management and small files to support the convergence of HPC, big data and AI.

4.1 Optimizations to Metadata Management

Traditional PFSs target compute-intensive workloads and use striping for placing data over a multitude of storage servers to provide aggregate I/O bandwidth. With the use of a hierarchical storage architecture and predefined I/O modes, H2FS makes a further step to benefit data-intensive workloads that involve large sequential read/write requests. Unfortunately, as with other PFSs, expensive metadata operations handicap hybrid workloads that require concurrent and high-performance metadata operations. One such scenario is processing a large volume of scientific datasets via big data processing frameworks (e.g., Hadoop and Spark[⑦]), which will generate many temporary files during the shuffle phase. Due to the diskless architecture of compute nodes, all these temporary files are staged into the underlying PFS. For each temporary file, the client (e.g., a map task) sends the file create request to the metadata servers of the PFS, which in turn initialize a metadata object and return the result to the client. With the increasing scale of big data workloads, the metadata service needs to handle a vast number of file create requests in a short period. The concurrent metadata requests cause the metadata service to become a serious performance bottleneck.

Accordingly, we propose Pream, a lightweight metadata management framework that aims to address this challenge. Pream targets scenarios of supporting data-intensive workloads that generate a vast number of temporary files on diskless compute nodes. It uses the preallocation strategy [25, 47] and responds to metadata requests in a proxy manner to provide a high-throughput

⑦http://spark.apache.org/, Jun. 2019.

metadata service. As illustrated in Fig.2, Pream is mainly composed of the client and the proxy server. When Pream is initialized, the proxy server sends file create requests to the metadata servers of the underlying PFS in advance and manages these preallocated file metadata locally to accelerate metadata operations. The client intercepts metadata requests from user applications and redirects them to the proxy server. While newly created temporary files keep residing in PFSs, open/create requests of temporary files can be handled by Pream locally without connecting with PFSs. Our evaluation demonstrates that Pream can outperform Lustre in many workloads and efficiently reduces the latency of metadata operations.

4.2 Optimizations to Small Files

Applications that involve a large number of small files are generally sensitive to access latency. The latency of accessing a small file is mostly introduced by two important operations: searching a given file in a directory containing many subfiles/subdirectories and reading the file from storage media. Accordingly, we propose using the cuckoo hash [48] and the KV data structure to accelerate the two operations, respectively.

Traditionally, the subfiles/subdirectories contained in a directory are indexed by a hierarchical data structure such as HTree [49] or B+ Tree [50]. However, as the total number of subfiles/subdirectories in a directory increases, the hierarchical data structure expands and has more levels. Searching a file in such a data structure requires multiple I/O requests issued to storage media. Specifically, in each level of the tree, the file system must generate a read request to fetch the node of the corresponding level and then determine which node should be fetched from storage media in the next level. The dependency between two subsequent levels indicates that the read requests from different levels cannot be issued in parallel. Consequently, searching a file in a large directory is likely to experience a long latency since the multiple read requests must be issued one by one.

To reduce the latency of searching files in large directories, we propose keeping the subfiles/subdirectories belonging to a parent directory in a flat data structure, such as a hash table [51, 52], where the entire hash table can be fetched from storage media via exactly only one read request. The inspiration behind our new proposal is that the latency of a large I/O request is mostly shorter than the sum of latencies of multiple small I/O requests since the prevalent storage devices, such as SSDs, are prone to provide higher bandwidth, while the access latency is hard to reduce due to the complex software stack. We propose fetching the entire parent directory from storage devices

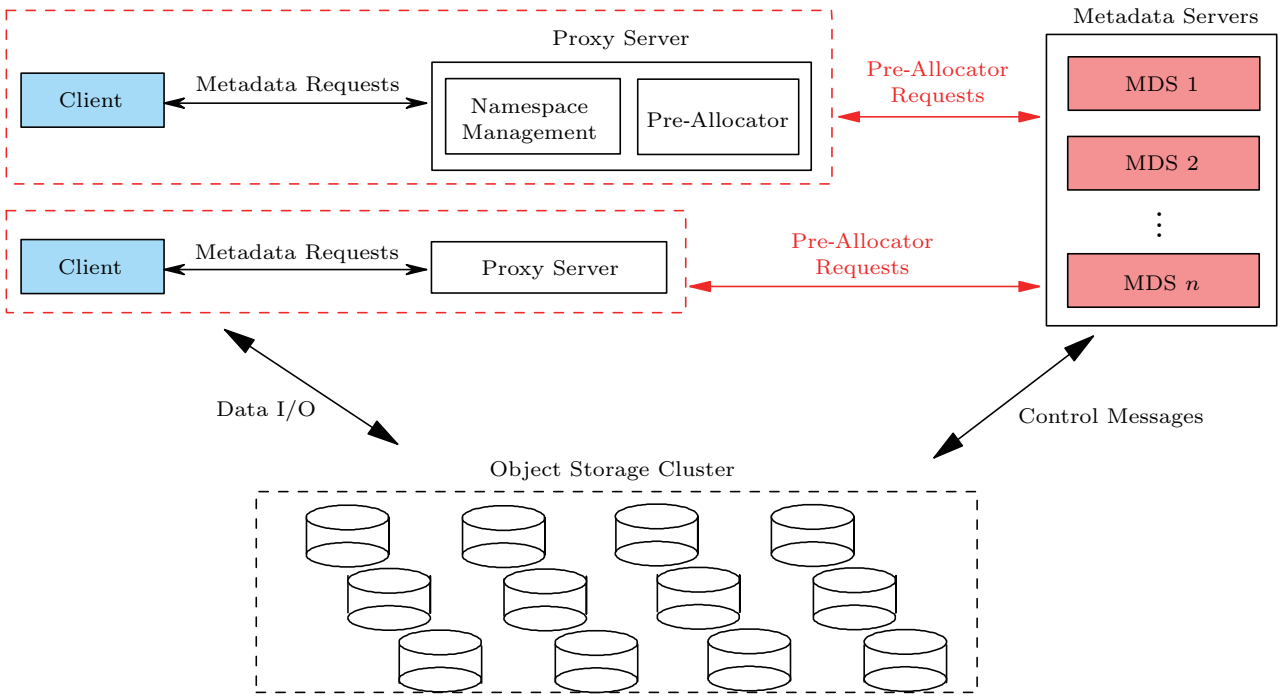

Fig.2. Architecture of Pream.

through only one large read request and then performing an in-memory search, rather than searching a subfile/subdirectory from an on-disk data structure that will inevitably introduce multiple small read requests. Consequently, our method is likely to achieve lower latency in searching a subfile/subdirectory in a parent directory, which is important for accessing a small file. Furthermore, it is possible that the distribution of items in a traditional hash table is unbalanced. To handle this situation, we adopt the cuckoo hash, which employs multiple hash functions to map each item to multiple buckets. When inserting an item into the hash table and the target bucket determined by one hash function is full, the item can be directed to another bucket using the next candidate hash function. In this way, the items will be evenly distributed in all buckets.

Another optimization to small files is that we design a KV store to organize data on disks. Traditional file systems mostly organize data on disks via indirect addressing. The multiple levels of indirect addressing also introduce additional read requests when fetching file data from disks. Taking the Ext4 file system [53] as an example, if the disk is formatted into 1 KB blocks, a file larger than 12 KB requires the first level of indirect addressing; accordingly, fetching the file data from disk requires at least two read requests. Furthermore, a file larger than 256 KB requires a second level of indirect addressing, and fetching the file data from a disk requires at least three read requests. For the same reason analyzed above, the multiple read requests will significantly increase the latency of accessing a small file. In this work, we develop a KV store to organize small files on disks, where the key is the inode number of a file and the value with varied length contains the file content. Specifically, the backend storage device is managed by the proposed KV store directly. It segments the disk space into multiple zones, where each of them consists of many fix-sized blocks (e.g., 8 KB–64 MB). Since each block holds the value of a specific key, the proposed KV store guarantees that a small file can be fetched from disk by only one read request via the GET operation.

In conclusion, we optimize the performance of small files in two aspects, i.e., the metadata and file content. Both optimizations share the same principle, i.e., fetching a small file from disk by one large read request, rather than by multiple small requests, helps to remarkably reduce the access latency.

5 Data Management Optimizations

We propose a data management middleware to provide better support for converged applications on HPC systems. Currently, this middleware contains the following components:

• tiered data management: manage data on hierarchical storage architecture and customize data management strategies to fit workflow data access patterns;

• data-aware task scheduling: cooperate with resource management software to bring computations to the data;

• indexing and query processing: enhance data locating services at the granularity of both a file and a record;

• intelligent storage optimization: learn the data access patterns between components of converged applications, and use the learned model to make intelligent data management decisions.

5.1 Tiered Data Management

Storage hierarchies are becoming deeper on modern HPC systems, as the emerging memory-based staging solutions use the memory space of compute nodes to stage intermediate data. However, the burden of making data management decisions among different storage tiers is left to users since the memory storage tier is managed by the I/O library or in-memory file system separately. To provide unified management of different storage tiers, we propose a tiered data management system (TDMS). As illustrated in Fig.3, TDMS acts as a shared file system and provides both the POSIX interface and encapsulated API that allows applications to stage data on heterogeneous storage devices. It employs a master-slave architecture, where the master is primarily responsible for managing the global metadata of the system. Each worker stores data as blocks in various available storage tiers, including local memory, local SSD of ION and the shared storage layer.

To make full use of heterogeneous storage devices, TDMS customizes different data management strategies for common workflow data access patterns, which are categorized into *pipeline*, *scatter*, *gather*, *multicast* and *reduce* patterns [54, 55]. These customized strategies focus on data placement on different storage tiers, including the horizontal data distribution strategy (which worker to store the data), vertical data distribution strategy (which storage tier to use), the block size of files, and cross-tier load balance strategy. Considering

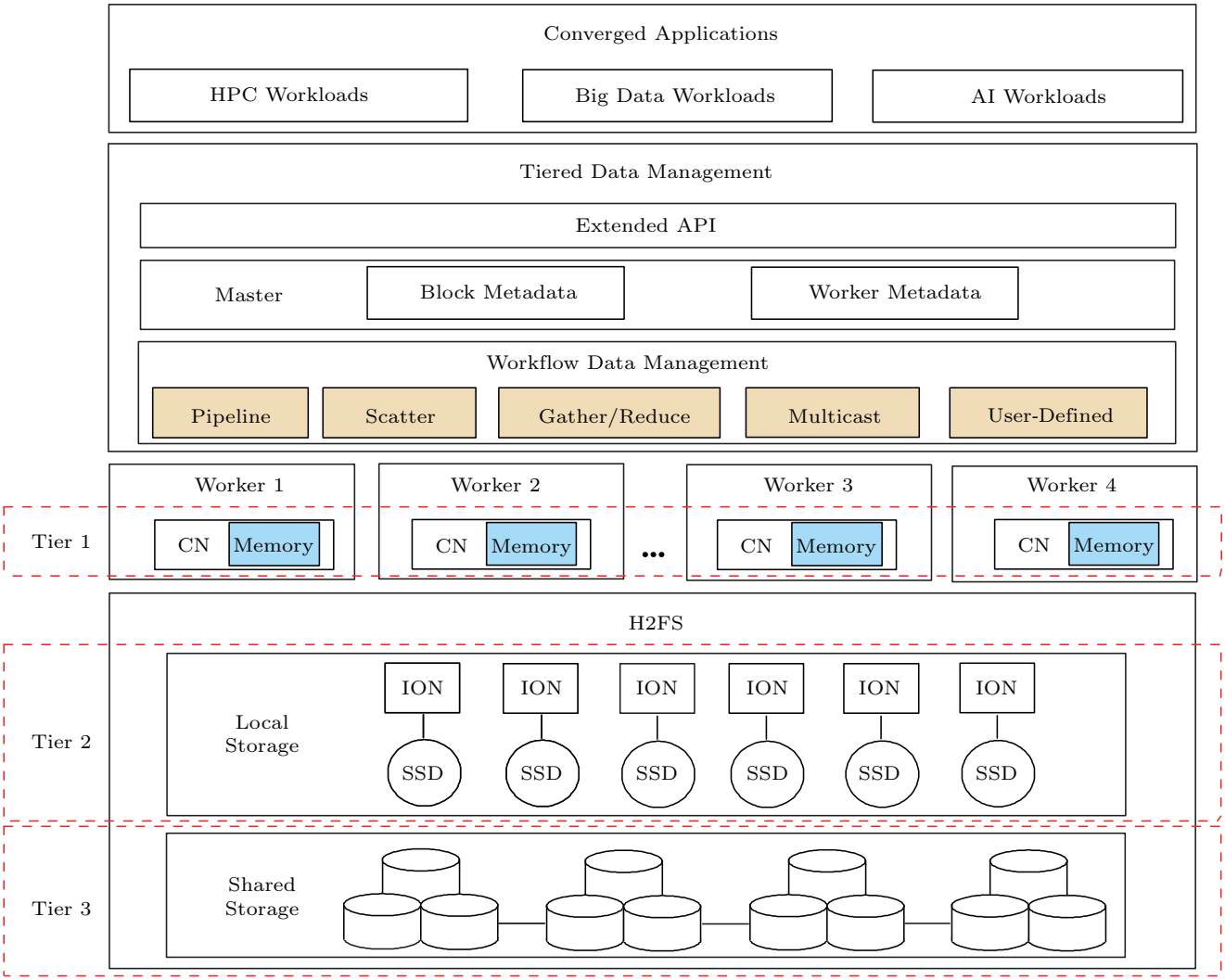

Fig.3. Overview of TDMS.

the performance diversity of different storage architectures and the variety of scientific workflows, TDMS also provides extended APIs that allow users to customize their own data management decisions for user-defined data access patterns. More details about the customized data management strategies can be found in our previous work [26].

5.2 Data-Aware Task Scheduling

When supporting converged applications on HPC systems, a task scheduling strategy should bring computations to the data rather than bring data to the computations. Nevertheless, the default task scheduling scheme in HPC resource management systems (e.g., Slurm [56]) is data locality agnostic since each compute node has the same distance to the underlying PFS. With the use of a hierarchical storage architecture, intermediate data generated from converged applications can be cached in compute nodes. The data locality-agnostic scheduling in these use cases, however, would result in a substantial amount of network traffic since data-dependent read tasks may be launched on different nodes from the write tasks.

For this reason, we propose data-aware task scheduling (DATS) to take advantage of data locality and to bring computations to the data. The key idea of DATS is to add data locality labels for each task dynamically and rank available resources based on these labels. The more the data that a compute node holds, the higher the priority that it has. The following task will be scheduled to the node that holds the most required data. As illustrated in Fig.4, DATS can be summarized in the following steps.

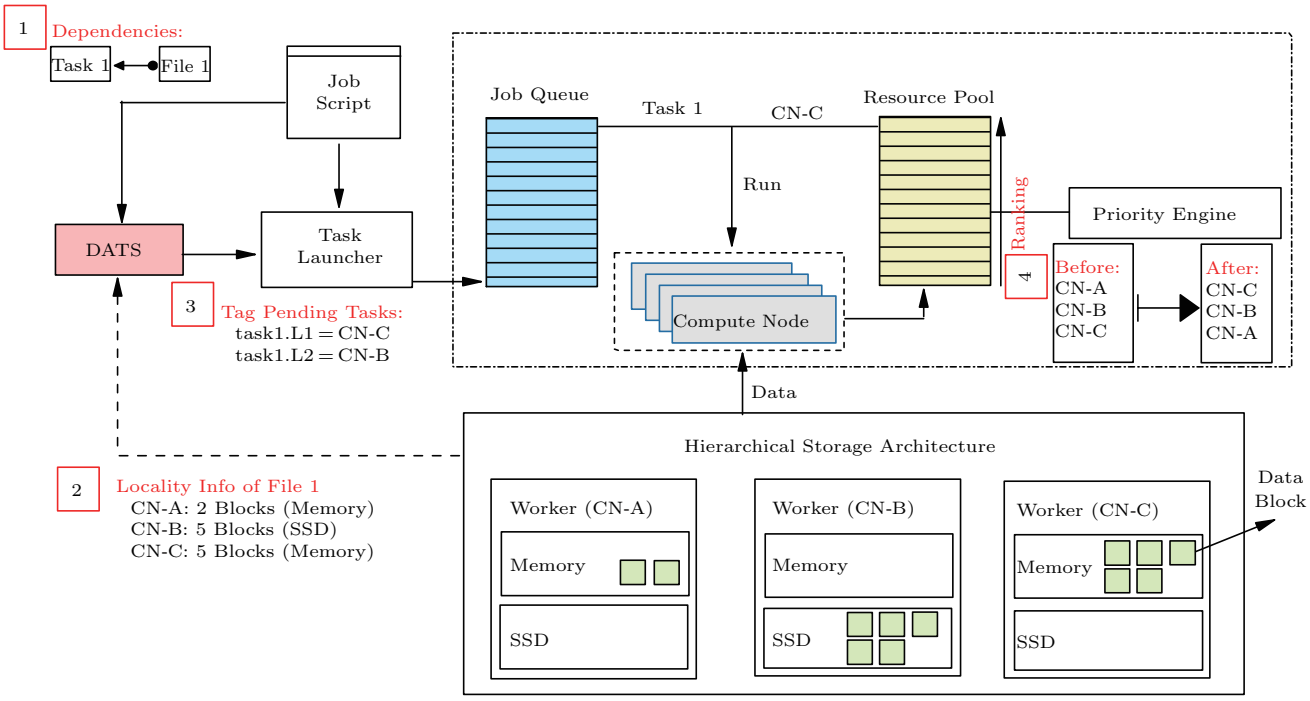

Fig.4. Data-aware task scheduling.

• *Step* 1. Extract data dependencies from the job script (e.g., workflow description file in scientific workflows) and obtain the input file list of each pending task.

• *Step* 2. Retrieve data locality information of input files and identify the compute nodes that hold most of the data.

• *Step* 3. Tag pending tasks with locality labels that stand for the hostnames of compute nodes received from step 2.

• *Step* 4. Guide the resource management system to rank available resources and then schedule the pending task to the resource with the highest score.

5.3 Indexing and Query Processing

As data analysis scenarios continue to increase on HPC systems, the ability to select a small fraction of data from a large volume of scientific datasets is vital for accelerating scientific discoveries. However, PFSs alone cannot provide efficient data locating services at the granularity of both a file and a record. User-defined metadata (UDM), such as event type and resolution in meteorology scenarios, are commonly used to annotate scientific datasets. Although UDM can be implemented as extended attributes in inode structures [19], UDM-based queries are not supported by PFSs out of the box. In addition, the increasing data scale leads to individual files with millions of records. Although building indexes based on the records of each file has been proven to be an efficient way to accelerate record-level query operations [57], PFSs lack the ability to do this directly. Consequently, crawling the file system to locate target files or retrieving all the data belonging to a specific file to locate target records is inefficient and costly.

We design an indexing and query processing module named UniIndex, which integrates parallel metadata extraction, in-situ indexing, lightweight bitmap-range index structure [58], in-memory cache layer, and two-level query processing to provide efficient data locating services. As illustrated in Fig.5, the UniIndex service can be set up on dedicated compute nodes and provides both command-line utilities and encapsulated APIs for interaction. For UDM-based file locating services, UniIndex provides a mounting API to attach an underlying PFS to its own namespace. During the mounting process, it extracts UDM from existing files and reconstructs them as KV pairs. For files in HDF5 and NetCDF data formats, all datasets and their corresponding attributes within a file will also be extracted. For record locating services, UniIndex allows users to enable in-situ indexing when applications are writing data to the underlying PFS. Specifically, we propose the bitmap-range index, which combines the virtues of

the Bitmap index [57] and MinMax index [59], to provide an efficient and lightweight index structure.

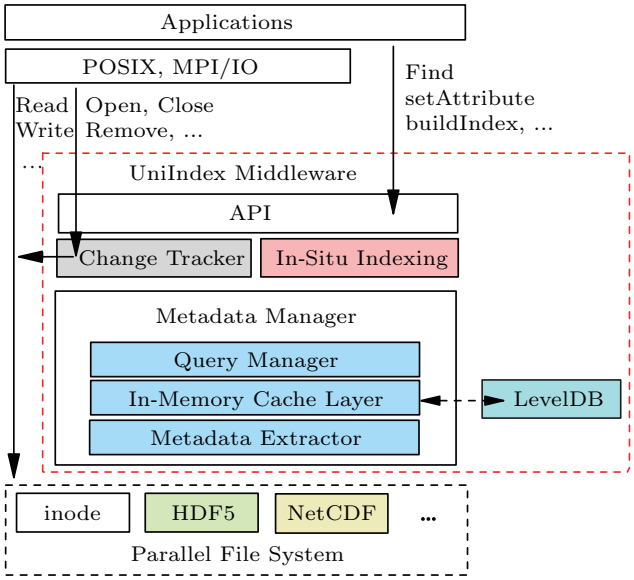

Fig.5. UniIndex architecture overview.

The metadata manager manages the extracted UDM and the in-situ generated bitmap-range indexes as KV pairs and maintains multiple in-memory data structures to accelerate query processing. To survive unexpected server failures, updates to the in-memory cache layer should be persisted to the disk. We choose LevelDB[8] as the backend database since it is optimized for write-intensive workloads and therefore accelerates the metadata extraction process and in-situ indexing. Updates to the in-memory cache layer trigger synchronized updates to the LevelDB, which store the database file on the underlying PFS. To keep the consistency between external KV pairs and files residing in the underlying PFS, the UniIndex client service captures the metadata requests (e.g., open, remove) from the application and updates the locating metadata objects asynchronously. By applying the in-memory cache layer and implementing a parallel query processing framework, the time to build indexes and locate target files or records can be dramatically reduced.

5.4 Intelligent Storage Optimization

In a hierarchical storage architecture, different data placement strategies could lead to widely varying I/O performance due to the largely different latencies, bandwidths and capacities of heterogeneous storage devices [16, 17]. In contrast to the one-fits-all strategy, we believe that both data access patterns and real-time system status should be taken into consideration to make the optimal data placement decision. For example, if the top storage layer has sufficient space to stage all the intermediate data during the execution of converged applications, then choosing the top storage layer to serve every write request will provide superior I/O performance. Otherwise, only data that will be immediately accessed by the subsequent task can be written into the top storage layer, and other data should be written to the lower storage layer to prevent resource contention. Although setting these rules manually may perform well for some applications, providing a general solution that can be applied in different storage architectures and diverse applications is challenging.

The intelligent storage optimization module is designed to leverage machine learning techniques to mine the relationship between data placement strategies and I/O performance under various data access patterns and system statuses and to use the learned model to choose the optimal storage tier intelligently. We propose adaptive storage learner (ASL) as an example to provide better support for scientific workflows. We identify 11 parameters that affect the I/O performance and collect 3 810 I/O records from 58 workflows with varying scales and I/O characteristics. A gradient boosting algorithm with classification and regression tree (CART)[9] as base learners is used to train the prediction model. The final prediction model can be treated as an ensemble of CART models.

Fig.6 presents the architecture overview of ASL. ASL acts as a middleware integrated with an existing data management system that can manage data on a hierarchical storage architecture. During the workflow execution, ASL extracts workflow characteristics (e.g., control-flow dependencies and data-flow dependencies between workflow tasks) from the workflow description file and collects real-time system status (e.g., the performance metrics and the remaining capacity of different storage tiers) dynamically. For each file create request, ASL constructs variables related to the workflow characteristics and system status and uses the prediction model to intelligently choose the target storage tier. As in the case of new workflows, ASL can also predict the result since none of the input variables depend on historical information.

[8] https://github.com/google/leveldb, Oct. 2019.
[9] https://github.com/catboost/catboost, Sept. 2019.

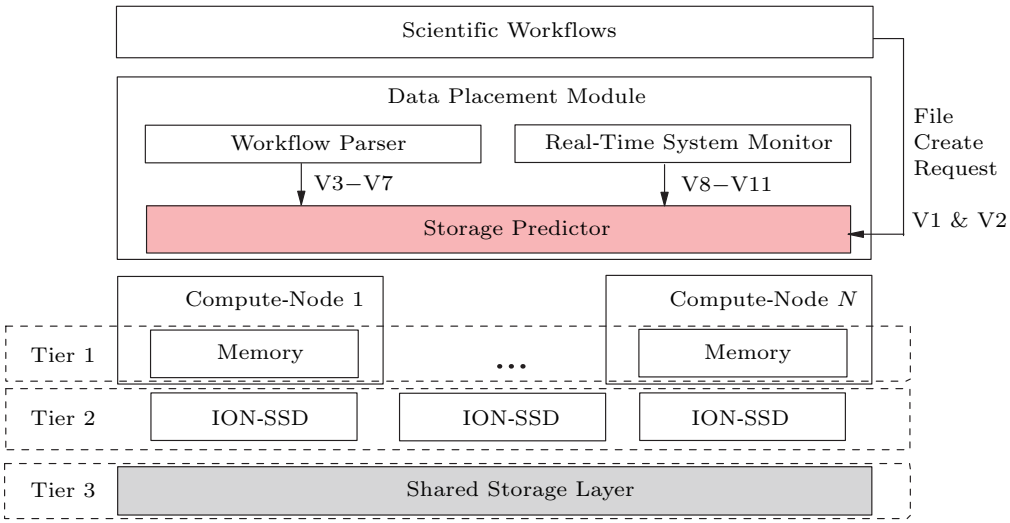

Fig.6. ASL architecture overview.

6 Application Optimizations

In addition to file system and data management optimizations, we leverage application-specific optimizations to accelerate diverse workloads, including traditional HPC applications, typical big data applications, and compounding workflow applications.

6.1 Experimental Setup

We ran the evaluated applications on the Tianhe-2 System, which consists of 16 000 compute nodes. Each compute node is equipped with two 2.20 GHz Intel® Xeon® E5-2600 processors and 64 GB of DRAM. All these nodes are connected using a dedicated TH Express-2 network switch [23], the customized internal high-speed interconnect of the Tianhe-2 system. The storage subsystem contains 256 IONs and 64 storage servers with a total capacity of 12.4 PB. Specifically, each I/O node is configured with two PCIe SSDs that provide a total capacity of 2 TB. We use H2FS as the underlying file system during the evaluation.

6.2 HPC Application: Grapes

The Global/Regional Assimilation Prediction System (Grapes) [60] is a middle-scale weather forecast system that has been developed by the China Meteorological Administration since 2001. Grapes, as well as the other earth science's simulation software, employs a collected I/O model. The root processor collects all output data from other processors, stores the data in arrays in sequence and writes the arrays to disk while other processors are waiting. Unfortunately, Grapes must output data frequently during the simulation. Consequently, Grapes spends increasingly more time for I/O with increasing parallel scale.

The physics limit of the I/O wall is hard to break through, but we can find some methods to step around it if we view the problem from the perspective of the whole application. The method that we implement to optimize the I/O problem is splitting the processes into two different parts, one for the main calculation and the other only for I/O, to overlap the I/O and calculation processes. After splitting, we subdivide the calculation processors into different groups, and each group is assigned one I/O processor. Each I/O processor collects the output data from the processors of its group during the output phase, which can effectively reduce the gather communication cost and call the parallel I/O after the data arrangement.

As shown in Fig.7, we evaluate the wall clock time of Grapes under various numbers of computation tasks. When the number of tasks is less than 768, up to 2 I/O tasks are used in the optimized version because of the limited scale. The elapsed time of the original version, the optimized version with 2 I/O tasks and the optimized version with 24 I/O tasks are presented in three bars from left to right. The performance of Grapes increases almost 300% in 3 072 CPU cores after our optimization. The optimized version has been used in the 1 km model of the South China High Precision Weather Forecast System since 2017. It enables the system to accomplish the six-hour forecast in eight minutes with 5 300 CPU cores.

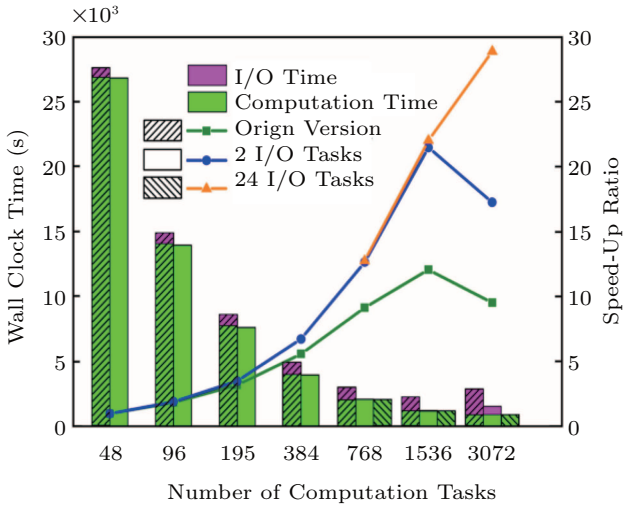

Fig.7. Grapes I/O optimization.

6.3 Big Data Workloads

As detailed in Section 4, we improve the H2FS in terms of metadata management and small files to support big data applications. To evaluate the efficiency of our improvement, we resort to genome-wide association studies (GWAS)[61] to demonstrate that the improved H2FS is able to provide more optimization options for users. A typical GWAS job processes the genetic data of millions of populations, where each population corresponds to a file that is several gigabytes. The job is divided into a large number of individual tasks that can be executed in parallel. The I/O patterns exhibited among these tasks are shown in Fig.8. Specifically, each file will be accessed by all these tasks, where each task demands a segment of data from every file, and the segments demanded by a given task from different files share the same offset and length. For this I/O pattern, the parallel tasks introduce serious contention on all these files and inevitably degrade the overall performance.

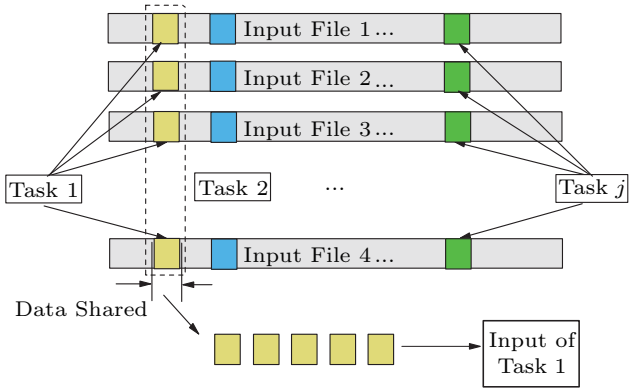

Fig.8. The I/O patterns of GWAS.

In our work, we reorganize the data segments involved in each task to produce a more I/O-friendly workload. Specifically, we divide each file into a configurable number of segments (e.g., N), where the i-th segment of each file has the same offset within the corresponding file. The set of files to be analyzed is taken as a matrix, where each file corresponds to a row, and the segments from different files sharing the same offset constitute a column. Accordingly, we propose converting the traditional row-based organization (i.e., the raw data belonging to a given population are stored as a file) to the column-based organization (i.e., the segments from different files sharing the same offset are stored as a file). In this way, different tasks will not access the same file, and the contention will be significantly alleviated. However, this optimization presented above is likely to produce a large number (e.g., N) of small files, which will introduce challenges into the metadata management of file systems. Fortunately, our improved H2FS is capable of handling a large number of files and can efficiently support this optimization.

To demonstrate the efficiency of our proposal, we analyze the genetic data of as many as 40 000 populations. Fifty compute nodes are used during the evaluation. The experimental result is shown in Fig.9. As shown in this figure, the total time consumed by an analysis job is increased exponentially with respect to the increase in populations for the original method (denoted by origin), while for our method (denoted by gwasin), the time just increases linearly.

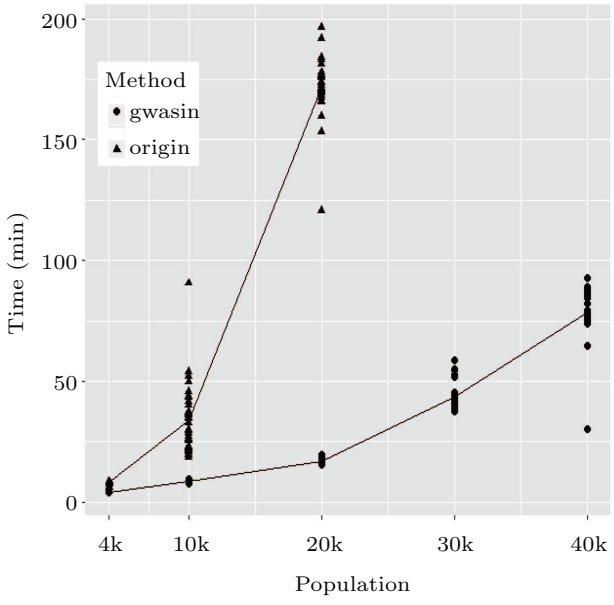

Fig.9. Performance of genetic data analysis.

DenseIO[10] is a data-intensive application that runs on top of the Tianhe-2 system. It applies a MapReduce-like framework to handle word count workloads and generates many temporary files during its execution. Depending on the type of words to be counted, the number of temporary files changes dramatically. Due to the expensive metadata operations of PFSs, concurrent file create requests lead to a serious performance bottleneck. As we discussed in Subsection 4.1, we propose Pream to accelerate metadata operations by preallocating file metadata and serving metadata requests in a proxy manner. Fig.10 illustrates the performance improvement of DenseIO under various numbers of temporary files. Serving the metadata requests by the underlying PFS is denoted as *Original-Version*, and serving the metadata requests by the Pream middleware is denoted as *Pream*. We also compare the performance of Pream with the lustremount strategy [62], where a local file system is mounted and backed by an H2FS file. For 1k temporary files, *Pream* outperforms *Original-Version* with a 5x speedup since the open/create requests of these temporary files can be handled by proxy servers locally without connecting with the underlying PFS. The *lustremount* strategy shows similar performance since it aggregates massive temporary files together and therefore alleviates the metadata bottleneck. As the number of temporary files increases, *Pream* shows a clear performance improvement over the *lustremount* strategy. This improvement is because the *lustremount* strategy creates file metadata as needed and leads to bursty metadata requests. In comparison, *Pream* applies the preallocation strategy and serves the metadata requests more efficiently.

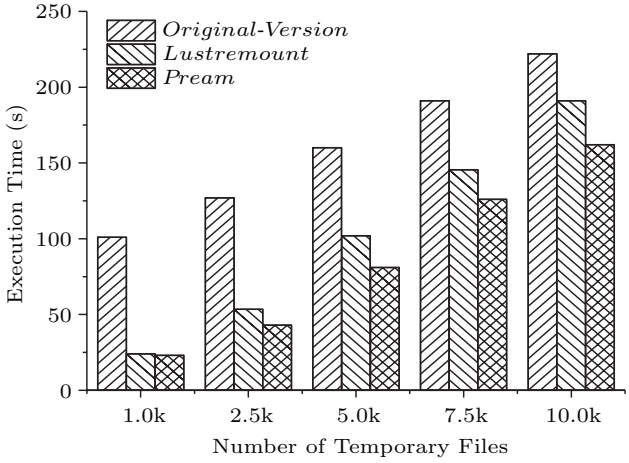

Fig.10. Performance of DenseIO workload.

We evaluate the record locating performance of UniIndex using the GenBase [63] dataset. The GenBase dataset is a two-dimensional array that records the gene expression values of different patients. We compare the performance of UniIndex with that of an existing tool, FastQuery [40]. Although there are several improvements [41, 42] to FastQuery, we choose FastQuery as the baseline since it can easily be obtained and installed. UniIndex applies in-situ indexing and calculates bitmap-range indexes when the GenBase data generator is writing data to PFS. In comparison, FastQuery applies a post-indexing strategy and needs to read data from disks before building bitmap indexes. As illustrated in Fig.11, UniIndex shows much better performance than FastQuery in all cases. For a 144 GB data size, UniIndex (16 cores) achieves a 37.2x speedup relative to FastQuery (16 cores). UniIndex outperforms FastQuery by applying a lightweight bitmap-range index and overlapping the index computing time with the data writing time. In comparison, FastQuery takes a significant amount of time to compute the bitmap index. Moreover, instead of writing indexes into a separate file as FastQuery does, UniIndex implements the index as KV pairs and reduces the index writing overhead.

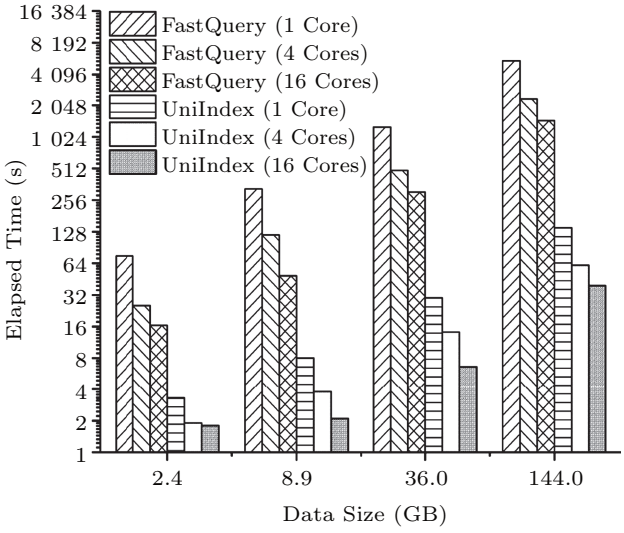

Fig.11. Buliding index under varied cores.

We also compare the query performance of UniIndex with that of FastQuery and the traverse strategy. Fig.12 shows the single-core query performance under varied query selectivity, which controls the percentage of elements that would be selected from the original datasets. We fixed the data size of the GenBase dataset

[10] https://github.com/hb-lee/DenseIOTest, Sept. 2019.

to 144 GB. The elapsed time of the traverse strategy remains stable since it reads the entire dataset despite the query selectivity. FastQuery performs slightly better than UniIndex when the selectivity is set to 1% and 0.5% because FastQuery can answer the queries by performing bitwise AND/OR operations on bitmap indexes. As the selectivity decreases, UniIndex outperforms FastQuery since only a few virtual index blocks are filtered to perform the fine-grained selection. For 0.001% selectivity, UniIndex shows a 1.16x speedup and 190x speedup relative to FastQuery and Traverse, respectively.

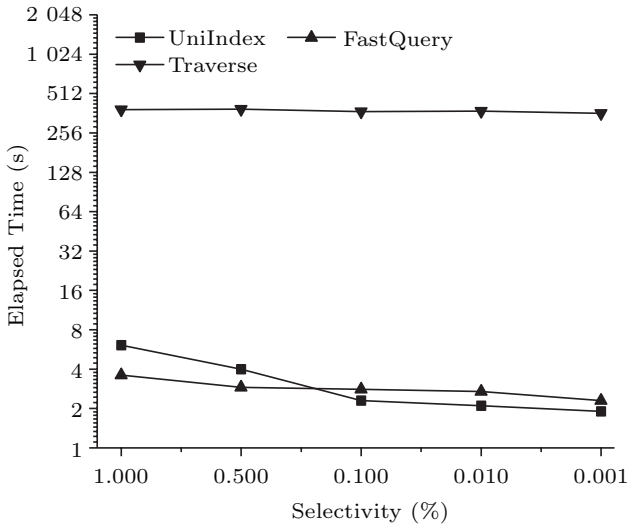

Fig.12. Query performance under varied selectivity.

6.4 AI Workloads

Although deep learning has emerged as a new technology for analyzing large volumes of scientific data, training a deep neural network is nontrivial. Such training involves processing a tremendous amount of training data iteratively to converge to the desired network accuracy. To accelerate reading the inputs and prevent the I/O bottleneck caused by many small files, raw training samples are often reformatted to large files (e.g., LMDB database files[11]) before training. Such preprocessing is extremely time-consuming since it incurs frequent metadata requests to load raw data from persistent storage. As an example, the ImageNet dataset [64] contains more than 1.3 million images, and each image file is 10 KB–150 KB in size. It takes more than 12 hours to preprocess the entire dataset on the H2FS. As scientific data are rapidly increasing in data size, preprocessing the raw data exposes new bottlenecks in the I/O system.

As our improved H2FS is optimized in terms of metadata and small files, there is no need to take special measures to address the large number of small files. To demonstrate the effectiveness of small file optimizations as discussed in Subsection 4.2, we first compare the throughput of metadata operations using the mdtest benchmark[12]. A 1 TB PCIe SSD is used as the backend storage device of the evaluated metadata server. Managing metadata objects on top of the Ext-4 file system with the B+-Tree index structure is denoted as *Original-Version*. Managing metadata objects on top of the proposed KV store with the cuckoo hash index structure is denoted as *Optimized-Version*. Fig.13 illustrates the result when a single client is used to perform *creation*, *stat* and *read* operations. During the evaluation, the mdtest benchmark generates 100 000 files where each of them is 1 KB in size. For the *creation* and the *stat* operation, the *Optimized-Version* outperforms the *Original-Version* with a 2.7x and 2.9x speedup, respectively. This is because all subfiles belonging to a parent directory are organized in a flat data structure and can be located in constant time. For the *read* operation, the *Optimized-Version* shows similar throughput to the *Original-Version* since the size of each file is set to 1 KB, which means it contains only one block and incurs one read request no matter how data are organized on the backend storage device.

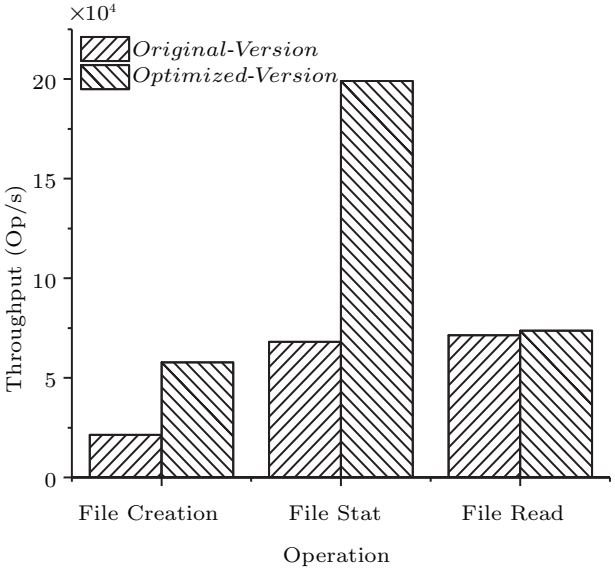

Fig.13. Throughput comparison of metadata operations.

[11] https://lmdb.readthedocs.io/en/release/, Aug. 2019.

[12] https://sourceforge.net/projects/mdtest, Sept. 2019.

Fig.14 illustrates the performance improvement of our strategy over three datasets. The Galaxy dataset[13] contains more than 43 513 astronomy images with a total size of 1.4 GB. The Amazon reviews dataset[14] contains the data of 82.8 million reviews with a total size of 14.3 GB. Pre-processing of these datasets on top of the original H2FS and the optimized H2FS are denoted as *Original-Version* and *Optimized-Version*, respectively. As the size of the dataset increases, *Optimized-Version* shows a more obvious performance improvement. For the ImageNet dataset, *Optimized-Version* achieves a 3.5x speedup relative to *Original-Version*.

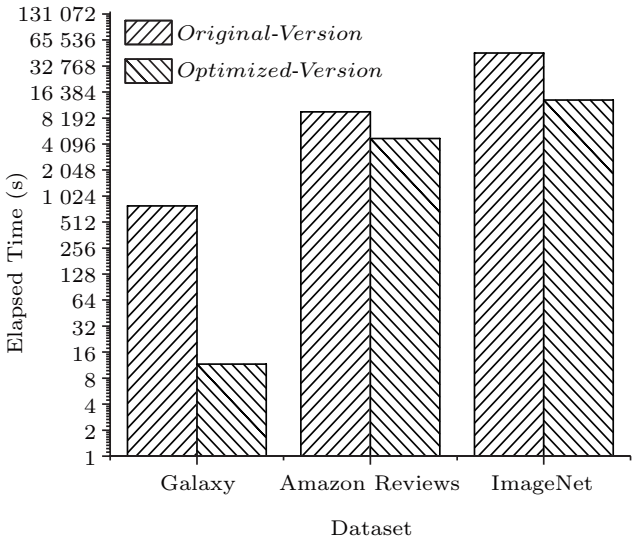

Fig.14. Pre-processing of the raw training data.

6.5 Scientific Workflows

A scientific workflow is the assembly of complex sets of scientific data processing activities with data dependencies between them [65]. Due to the repetitive nature of scientific discovery, scientific workflows are increasingly used in HPC environments to manage complex simulations and analyses. For example, the Montage workflow [66] is an astronomical image processing workflow that assembles the Flexible Image Transport System (FITS) images into custom mosaics of the sky. The Binary-Tree workflow [67] is a synthetic benchmark that evaluates the I/O performance of the underlying file system. While these workflows exhibit complex data dependencies, frequent data sharing requirements between loosely coupled tasks become an impediment to the overall performance.

Targeting scientific workflows, we identify coarse-grained data access patterns of workflow tasks and leverage customized data management (CDM) strategies to make full use of the tiered storage architecture on the Tianhe-2 system. As an example, in the Montage workflow, the input FITS images are scattered to available compute nodes (scatter pattern) and are processed in a pipeline (pipeline pattern). During the execution, one of the compute nodes will collect the intermediate data (gather pattern) to determine the background adjustment parameter and multicast it to other compute nodes (multicast pattern). Finally, the adjusted data will be collected to obtain the final output (gather pattern). For the pipeline pattern, CDM chooses the local memory of each compute node as the primary storage tier to minimize the data access time for subsequent tasks. The generated intermediate files will be stored in the same node where the task corresponding to that stage runs. In comparison, in the gather pattern, one subsequent task consumes multiple output data of previous tasks. Therefore, CDM allows users to define a set of compute nodes to collect these output data in all available storage tiers. When the subsequent gather or reduce tasks are scheduled on these nodes, they can obtain all the required data without retrieving data from distributed compute nodes. The DATS module is responsible for bringing computations close to data without user involvement. To provide better support for scientific workflows, we implemented DATS in HTCondor[15], a commonly used resource management system that is coupled with workflow management systems. Once DATS is enabled, data-dependent tasks are scheduled to compute nodes that hold the input data automatically.

Fig.15 and Fig.16 present the performance improvements in the Montage workflow and the Binary-Tree workflow when CDM and DATS are enabled, respectively. Thirty-two compute nodes are used during the evaluation. We allocate 20 GB DRAM of each compute node to constitute the memory storage tier when CDM is enabled. For the Montage workflow, we fix the scale of the input data to 10×10 degrees and break down the I/O time of each phase. CDM+DATS provides a 1.6x speedup in overall performance compared with the original-version base case. Specifically, *mImgtbl* and *mAdd* enjoy the most benefits of pattern-specific optimizations and show 3.7x speedups in data

[13] https://galaxyproject.org/data-libraries, Sept. 2019.

[14] http://jmcauley.ucsd.edu/data/amazon, Aug. 2019.

[15] https://research.cs.wisc.edu/htcondor/index.html, Aug. 2019.

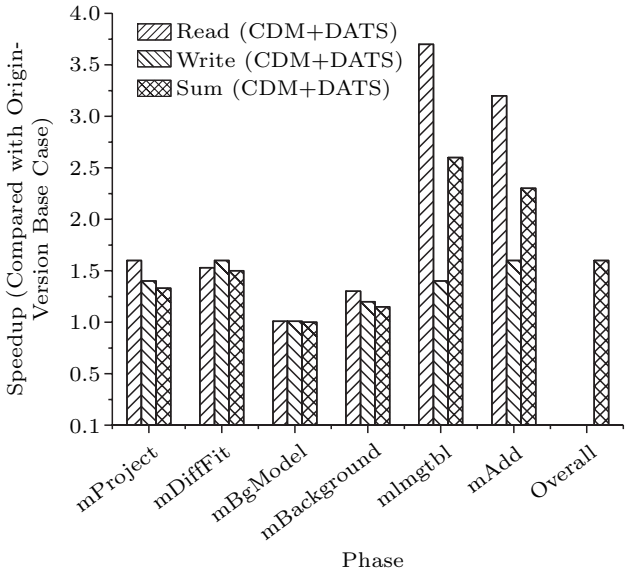

Fig.15. Performance of Montage workflow.

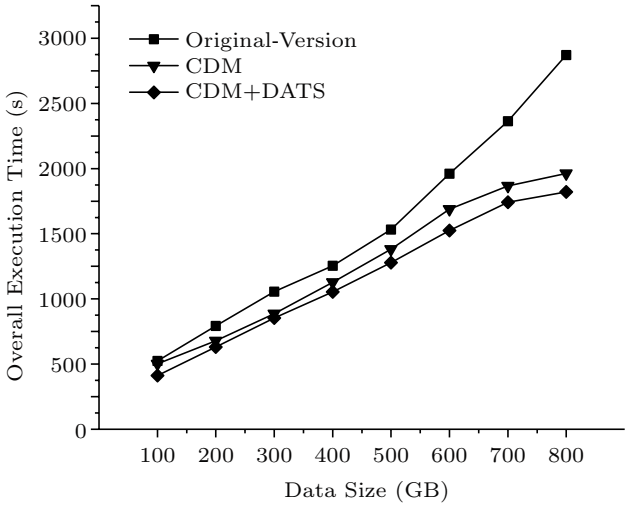

Fig.16. Performance of Binary-Tree workflow.

reading since all data are precollected in one node under the gather data access pattern. The *mProject*, *mDiffFit* and *mBackground* phases also show obvious I/O speedups because of the efficient data sharing provided by the pipeline data management optimizations. For the Binary-Tree workflow, we vary the data scales from 100 GB to 800 GB. Compared with the original-version base case, CDM provides better performance by utilizing the tiered storage architecture for efficient data sharing. Although CDM optimizations alleviate the data transfer between memory and disks, the data transfer cost between compute nodes still exists. With DATS optimization, dependent tasks can be scheduled to the worker node where the input data are located. Clearly, CDM+DATS performs further optimization and achieves a 1.54x speedup over the original version for 800 GB data size.

In addition to CDM and DATS, we also enable the ASL to intelligently choose the optimal storage tier for scientific workflows, as discussed in Subsection 5.4. Fig.17 presents the results of the GenBase workflow [63] under various data scales. Specifically, for 80k × 80k input data scales, 120 GB of raw data are processed and will generate more than 400 GB of intermediate data. Choosing the memory storage tier and the SSD storage tier manually to stage all intermediate data are denoted as *Memory-strategy* and *SSD-strategy*, respectively. Enabling ASL to intelligently make data placement decisions is denoted as *ASL-strategy*. When the data size is smaller than 40k × 40k, *Memory-strategy* performs better than *SSD-strategy* since the memory storage layer has sufficient space to stage all the intermediate data. As data size continues to increase, *Memory-strategy* migrates data from the memory tier to the SSD tier to make room for the newly created file. The resource contention between regular write requests and backend data migration requests leads to performance degradation. Since the SSD tier has sufficient space to stage all intermediate data during our evaluations, the resource contention problem does not occur in *SSD-strategy*. In contrast to these strategies, *ASL-strategy* combines information including the available space of each storage layer, the input size of the current task, and the distance of dependent task to choose the optimal storage tier for each intermediate file. By leveraging the workflow characteristics and preventing resource contention, *ASL-strategy* shows the best performance in all cases.

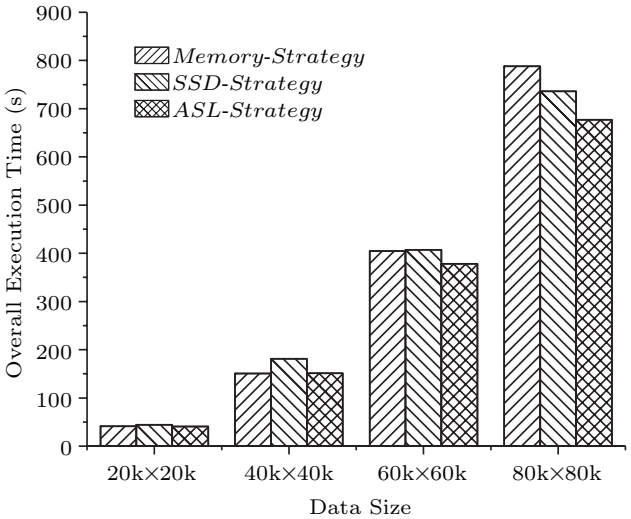

Fig.17. Performance of GenBase workflow.

7 Conclusions

The explosive growth in scientific data and new application requirements are driving the need for converged data management at the intersection of HPC, big data and AI. In this paper, we highlighted three data management challenges in supporting converged applications on HPC systems, including the exacerbated I/O bottleneck, the emerging adaptive and intelligent storage optimizations, and the unified management of massive heterogeneous scientific data. Centering around the objective of accommodating the requirements of converged applications, we summarized our experience in addressing these challenges on the Tianhe-2 system. At the file system level, we optimized H2FS further in terms of metadata management and small files to provide better support for concurrent metadata requests and small file scenarios. At the data management middleware level, we proposed tiered data management, DATS, indexing and query processing, and an intelligent storage optimization module to enhance data management services for PFS. At the user application level, application-specific optimizations are adopted to expedite the execution, including overlapping the I/O and the calculation, reorganizing data structures, and identifying workflow data access patterns.

Although we have explored some solutions to adapt to the ever-changing application demands, many unprecedented challenges still remain to be researched. These challenges include the followings:

• innovative storage architecture that can best utilize the emerging non-volatile storage devices (e.g., Intel Optane DC Persistent Memory[16]);

• high-performance, scalable, and fault-tolerant PFS for next-generation exascale system;

• data-centric programming models that are able to express massive parallelism, data locality, and resilience.

In the future, we will focus on these challenges and devote our efforts to establishing a prosperous ecosystem for exascale computing.

References

[1] Zhang Z, Barbary K, Nothaft F *et al*. Scientific computing meets big data technology: An astronomy use case. In *Proc. the 2015 IEEE International Conference on Big Data*, October 29–November 1, 2015, pp.918-927.

[2] Yang X, Liu N, Feng B, Sun X H, Zhou S. PortHadoop: Support direct HPC data processing in Hadoop. In *Proc. the 2015 IEEE International Conference on Big Data*, October 29–November 1, 2015, pp.223-232.

[3] Klein M, Sharma R, Bohrer C, Avelis C, Roberts E. Biospark: Scalable analysis of large numerical datasets from biological simulations and experiments using Hadoop and Spark. *Bioinformatics*, 2017, 33(2): 303-305.

[4] Usman S, Mehmood R, Katib I. Big data and HPC convergence: The cutting edge and outlook. In *Proc. the 1st International Conference on Smart Societies, Infrastructure, Technologies and Applications*, November 2017, pp.11-26.

[5] Kurth T, Treichler S, Romero J *et al*. Exascale deep learning for climate analytics. In *Proc. the 2018 International Conference for High Performance Computing, Networking, Storage, and Analysis*, November 2018, Article No. 51.

[6] Song F G, Dongarra J J. A scalable approach to solving dense linear algebra problems on hybrid CPU-GPU systems. *Concurrency and Computation: Practice and Experience*, 2015, 27(14): 3702-3723.

[7] Karp R M, Zhang Y J. Randomized parallel algorithms for backtrack search and branch-and-bound computation. *J. ACM*, 1993, 40(3): 765-789.

[8] Schwan P. Lustre: Building a file system for 1,000-node clusters. In *Proc. the 2013 Linux Symposium*, July 2003, pp.380-386.

[9] Li J W, Liao W K, Choudhary A N *et al*. Parallel netCDF: A high-performance scientific I/O interface. In *Proc. the 2003 ACM/IEEE Conference on High Performance Networking and Computing*, November 2003, Article No. 39.

[10] Shvachko K, Kuang H, Radia S, Chansler R. The Hadoop distributed file system. In *Proc. the 26th IEEE Symposium on Mass Storage Systems and Technologies*, May 2010, Article No. 9.

[11] Barisits M, Beermann T, Berghaus F *et al*. Rucio — Scientific data management. arXiv:1902.09857, 2019. https://arxiv.org/abs/1902.09857, Oct. 2019.

[12] Narasimhamurthy S, Danilov N, Wu S, Umanesan G, Markidis S, Gomez S R, Peng I B, Laure E, Pleiter D, Witt S D. SAGE: Percipient storage for exascale data centric computing. *Parallel Computing*, 2019, 83: 22-33.

[13] Sewell C M, Heitmann K, Finkel H *et al*. Large-scale compute-intensive analysis via a combined in-situ and co-scheduling workflow approach. In *Proc. the 2015 International Conference for High Performance Computing, Networking, Storage and Analysis*, November 2015, Article No. 50.

[14] Miyoshi T, Lien G Y, Satoh S *et al*. "Big data assimilation" toward post-petascale severe weather prediction: An overview and progress. *Proceedings of the IEEE*, 2016, 104(11): 2155-2179.

[15] Bhimji W, Bard D, Romanus M. Accelerating science with the NERSC burst buffer early user program. In *Proc. the 2016 Cray User Group Meeting*, May 2016.

[16] Kakoulli E, Herodotou H. OctopusFS: A distributed file system with tiered storage management. In *Proc. the 2017 ACM International Conference on Management of Data*, May 2017, pp.65-78.

[16] https://www.intel.com/content/www/us/en/products/memory-storage/optane-dc-persistent-memory.html, Aug. 2019.

[17] Dong B, Byna S, Wu K S, Prabhat, Johansen H, Johnson J N, Keen N. Data elevator: Low-contention data movement in hierarchical storage system. In *Proc. the 23rd IEEE International Conference on High Performance Computing*, December 2016, pp.152-161.

[18] Lim S H, Sim H, Gunasekaran R, Vazhkudai S S. Scientific user behavior and data-sharing trends in a petascale file system. In *Proc. the 2017 International Conference for High Performance Computing, Networking, Storage and Analysis*, November 2017, Article No. 46.

[19] Sim H, Kim Y, Vazhkudai S S, Vallée G R, Lim S H, Butt A R. Tagit: An integrated indexing and search service for file systems. In *Proc. the 2017 International Conference for High Performance Computing, Networking, Storage and Analysis*, November 2017, Article No. 5.

[20] Jenkins J, Arkatkar I, Lakshminarasimhan S, Boyuka-II D A, Schendel E R, Shah N, Ethier S, Chang C S, Chen J, Kolla H, Klasky S, Ross R B, Samatova N F. ALACRITY: Analytics-driven lossless data compression for rapid in-situ indexing, storing, and querying. *Trans. Large-Scale Data- and Knowledge-Centered Systems*, 2013, 10: 95-114.

[21] Lu T, Suchyta E, Pugmire D, Choi J, Klasky S, Liu Q, Podhorszki N, Ainsworth M, Wolf M. Canopus: A paradigm shift towards elastic extreme-scale data analytics on HPC storage. In *Proc. the 2017 IEEE International Conference on Cluster Computing*, September 2017, pp.58-69.

[22] Foster I T, Ainsworth M, Allen B *et al*. Computing just what you need: Online data analysis and reduction at extreme scales. In *Proc. the 23rd International Conference on Parallel and Distributed Computing*, August 2017, pp.3-19.

[23] Liao X K, Xiao L Q, Yang C Q, Lu Y T. MilkyWay-2 supercomputer: System and application. *Frontiers Comput. Sci.*, 2014, 8(3): 345-356.

[24] Xu W X, Lu Y T, Li Q *et al*. Hybrid hierarchy storage system in MilkyWay-2 supercomputer. *Frontiers Comput. Sci.*, 2014, 8(3): 367-377.

[25] Li H B, Cheng P, Chen Z G, Xiao N. Pream: Enhancing HPC storage system performance with pre-allocated metadata management mechanism. In *Proc. the 21st IEEE International Conference on High Performance Computing and Communications*, August 2019, pp.413-420.

[26] Cheng P, Lu Y T, Du Y F, Chen Z G. Accelerating scientific workflows with tiered data management system. In *Proc. the 20th IEEE International Conference on High Performance Computing and Communications*, June 2018, pp.75-82.

[27] Kougkas A, Devarajan H, Sun X H. Hermes: A heterogeneous-aware multi-tiered distributed I/O buffering system. In *Proc. the 27th International Symposium on High-Performance Parallel and Distributed Computing*, June 2018, pp.219-230.

[28] Wang T, Byna S, Dong B, Tang H J. UniviStor: Integrated hierarchical and distributed storage for HPC. In *Proc. IEEE International Conference on Cluster Computing*, September 2018, pp.134-144.

[29] Dong B, Wang T, Tang H J, Koziol Q, Wu K S, Byna S. ARCHIE: Data analysis acceleration with array caching in hierarchical storage. In *Proc. the 2018 IEEE International Conference on Big Data*, December 2018, pp.211-220.

[30] Feng K, Sun X H, Yang X, Zhou S J. SciDP: Support HPC and big data applications via integrated scientific data processing. In *Proc. the 2018 IEEE International Conference on Cluster Computing*, September 2018, pp.114-123.

[31] Wasi-ur-Rahman M, Lu X Y, Islam N S, Rajachandrasekar R, Panda D K. High-performance design of YARN MapReduce on modern HPC clusters with Lustre and RDMA. In *Proc. the 2015 IEEE International Parallel and Distributed Processing Symposium*, May 2015, pp.291-300.

[32] Pumma S, Si M, Feng W C, Balaji P. Parallel I/O optimizations for scalable deep learning. In *Proc. the 23rd IEEE International Conference on Parallel and Distributed Systems*, December 2017, pp.720-729.

[33] Jia Y Q, Shelhamer E, Donahue J, Karayev S, Long J, Girshick R B, Guadarrama S, Darrell T. Caffe: Convolutional architecture for fast feature embedding. In *Proc. the ACM International Conference on Multimedia*, November 2014, pp.675-678.

[34] Tomes E, Rush E N, Altiparmak N. Towards adaptive parallel storage systems. *IEEE Trans. Computers*, 2018, 67(12): 1840-1848.

[35] He S B, Sun X H, Wang Y, Xu C Z. A migratory heterogeneity-aware data layout scheme for parallel file systems. In *Proc. the 2018 IEEE International Parallel and Distributed Processing Symposium*, May 2018, pp.1133-1142.

[36] Subedi P, Davis P E, Duan S H, Klasky S, Kolla H, Parashar M. Stacker: An autonomic data movement engine for extreme-scale data staging-based in-situ workflows. In *Proc. the 2018 International Conference for High Performance Computing, Networking, Storage, and Analysis*, November 2018, Article No. 73.

[37] Wu K, Ren J, Li D. Runtime data management on non-volatile memory-based heterogeneous memory for task-parallel programs. In *Proc. the International Conference for High Performance Computing, Networking, Storage, and Analysis*, November 2018, Article No. 31.

[38] Stonebraker M, Brown P, Zhang D H, Becla J. SciDB: A database management system for applications with complex analytics. *Computing in Science and Engineering*, 2013, 15(3): 54-62.

[39] Dong B, Wu K S, Byna S, Liu J L, Zhao W J, Rusu F. ArrayUDF: User-defined scientific data analysis on arrays. In *Proc. the 26th International Symposium on High-Performance Parallel and Distributed Computing*, June 2017, pp.53-64.

[40] Chou J, Howison M, Austin B, Wu K S, Qiang J, Bethel E W, Shoshani A, Rübel O, Prabhat, Ryne R D. Parallel index and query for large scale data analysis. In *Proc. the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, November 2011, Article No. 30.

[41] Chiu H T, Chou J, Vishwanath V, Wu K S. In-memory query system for scientific dataseis. In *Proc. the 21st IEEE International Conference on Parallel and Distributed Systems*, December 2015, pp.362-371.

[42] Dong B, Byna S, Wu K S. Spatially clustered join on heterogeneous scientific data sets. In *Proc. the 2015 IEEE International Conference on Big Data*, October 29–November 1, 2015, pp.371-380.

[43] Gu J M, Klasky S, Podhorszki N, Qiang J, Wu K S. Querying large scientific data sets with adaptable IO system ADIOS. In *Proc. the 4th Asian Conference on Supercomputing Frontiers*, March 2018, pp.51-69.

[44] Wu T H, Chou J, Hao S, Dong B, Klasky S, Wu K S. Optimizing the query performance of block index through data analysis and I/O modeling. In *Proc. the 2017 International Conference for High Performance Computing, Networking, Storage and Analysis*, November 2017, Article No. 12.

[45] Kim J, Abbasi H, Chacón L, Docan C, Klasky S, Liu Q, Podhorszki N, Shoshani A, Wu K S. Parallel in situ indexing for data-intensive computing. In *Proc. the IEEE Symposium on Large Data Analysis and Visualization*, October 2011, pp.65-72.

[46] Liu N, Cope J, Carns P H *et al*. On the role of burst buffers in leadership-class storage systems. In *Proc. the 28th IEEE Symposium on Mass Storage Systems and Technologies*, April 2012, Article No. 5.

[47] Lee J Y, Lee J H. Pre-allocated duplicate name prefix detection mechanism using naming-pool in mobile content-centric network. In *Proc. the 7th International Conference on Ubiquitous and Future Networks*, July 2015, pp.115-117.

[48] Pagh R, Rodler F F. Cuckoo hashing. In *Proc. the 9th Annual European Symposium*, August 2001, pp.121-133.

[49] Phillips D. A directory index for EXT2. In *Proc. the 5th Annual Linux Showcase & Conference*, November 2001.

[50] Sweeney A, Doucette D, Hu W, Anderson C, Nishimoto M, Peck G. Scalability in the XFS file system. In *Proc. the 1996 USENIX Annual Technical Conference*, January 1996, pp.1-14.

[51] Lensing P H, Cortes T, Brinkmann A. Direct lookup and hash-based metadata placement for local file systems. In *Proc. the 6th Annual International Systems and Storage Conference*, July 2013, Article No. 5.

[52] Lensing P, Meister D, Brinkmann A. hashFS: Applying hashing to optimize file systems for small file reads. In *Proc. the 2010 International Workshop on Storage Network Architecture and Parallel I/Os*, May 2010, pp.33-42.

[53] Mathur A, Cao M M, Bhattacharya S, Dilger A, Tomas A, Vivier L. The new ext4 filesystem: Current status and future plans. In *Proc. the 2007 Linux Symposium*, June 2007, pp.21-33.

[54] Shibata T, Choi S J, Taura K. File-access characteristics of data-intensive workflow applications. In *Proc. the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, May 2010, pp.522-525.

[55] Katz D S, Armstrong T G, Zhang Z, Wilde M, Wozniak J M. Many-task computing and blue waters. arXiv:1202.3943, 2012. https://arxiv.org/abs/1202.3943, Oct. 2019.

[56] Yoo A B, Jette M A, Grondona M. SLURM: Simple Linux utility for resource management. In *Proc. the 9th International Workshop on Job Scheduling Strategies for Parallel Processing*, June 2003, pp.44-60.

[57] Wu K S, Ahern S, Bethel E W *et al*. FastBit: Interactively searching massive data. *Journal of Physics: Conference Series*, 2009, 180(1): Article No. 012053.

[58] Cheng P, Wang Y, Lu Y T, Du Y F, Chen Z G. IndexIt: Enhancing data locating services for parallel file systems. In *Proc. the 21st IEEE International Conference on High Performance Computing and Communications*, August 2019, pp.1011-1019.

[59] Wu T H, Chou J, Podhorszki N, Gu J M, Tian Y, Klasky S, Wu K S. Apply block index technique to scientific data analysis and I/O systems. In *Proc. the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, May 2017, pp.865-871.

[60] Chen D H, Xue J S, Yang X S *et al*. New generation of multi-scale NWP system (GRAPES): General scientific design. *Chinese Science Bulletin*, 2008, 53(22): 3433-3445.

[61] Bush W S, Moore J H. Chapter 11: Genome-wide association studies. *PLoS Computational Biology*, 2012, 8(12): Article No. e1002822.

[62] Chaimov N, Malony A D, Canon S, Iancu C, Ibrahim K Z, Srinivasan J. Scaling spark on HPC systems. In *Proc. the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, May 2016, pp.97-110.

[63] Taft R, Vartak M, Satish N R, Sundaram N, Madden S, Stonebraker M. GenBase: A complex analytics genomics benchmark. In *Proc. the 2014 ACM SIGMOD International Conference on Management of Data*, June 2014, pp.177-188.

[64] Deng J, Dong W, Socher R, Li L J, Li K, Li F F. ImageNet: A large-scale hierarchical image database. In *Proc. the 2009 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, June 2009, pp.248-255.

[65] Deelman E, Gannon D, Shields M S, Taylor I J. Workflows and e-science: An overview of workflow system features and capabilities. *Future Generation Comp. Syst.*, 2009, 25(5): 528-540.

[66] Berriman B G, Good J C, Laity A C *et al*. Chapter 19: Web-based Tools — Montage: An astronomical image mosaic engine. In *The National Virtual Observatory: Tools and Techniques for Astronomical Aesearch,* Graham M J, Fitzpatrick M J, McGlynn T A (eds.), Astronomical Society of the Pacific, 2007, pp.179-189.

[67] Hazekamp N, Kremer-Herman N, Tovar B *et al*. Combining static and dynamic storage management for data intensive scientific workflows. *IEEE Transactions on Parallel and Distributed Systems*, 2018, 29(2): 338-350.

Yu-Tong Lu is a professor in the School of Data and Computer Science, Sun Yat-sen University, Guangzhou. She is also the Director of National Supercomputer Center in Guangzhou. Her research interests include high-performance computing, parallel file system, and advanced programming environment.

Peng Cheng is a Ph.D. candidate in the College of Computer, National University of Defense Technology, Changsha. He received his Master's degree in computer science and technology from the same university in 2015. His research interests include high-performance computing, big data processing and programming models.

Zhi-Guang Chen is an associate professor in the School of Data and Computer Science, Sun Yat-sen University, Guangzhou, and National Supercomputer Center in Guangzhou. He received his Ph.D. degree in computer science and technology from National University of Defense Technology, Changsha, in 2013. His research interests include parallel operating system, global file system and non-volatile storage.