HadaFS: A File System Bridging the Local and Shared Burst Buffer for Exascale Supercomputers

Xiaobin He[1], Bin Yang[2 1*], Jie Gao[1], Wei Xiao[1], Qi Chen[2], Shupeng Shi[1],
Dexun Chen[1], Weiguo Liu[4], Wei Xue[2 3 1], Zuo-ning Chen[5†]
[1]*National Supercomputing Center in Wuxi,* [2]*Tsinghua University, Dept. of C.S,*
[3]*Tsinghua University, BNRist.,*[4]*Shandong University,* [5]*Chinese Academy of Engineering*

Abstract

Current supercomputers introduce SSDs to form a Burst Buffer (BB) layer to meet the HPC application's growing I/O requirements. BBs can be divided into two types by deployment location. One is the local BB, which is known for its scalability and performance. The other is the shared BB, which has the advantage of data sharing and deployment costs. How to unify the advantages of the local BB and the shared BB is a key issue in the HPC community.

We propose a novel BB file system named HadaFS that provides the advantages of local BB deployments to shared BB deployments. First, HadaFS offers a new Localized Triage Architecture (LTA) to solve the problem of ultra-scale expansion and data sharing. Then, HadaFS proposes a full-path indexing approach with three metadata synchronization strategies to solve the problem of complex metadata management of traditional file systems and mismatch with the application I/O behaviors. Moreover, HadaFS integrates a data management tool named Hadash, which supports efficient data query in the BB and accelerates data migration between the BB and traditional HPC storage. HadaFS has been deployed on the Sunway New-generation Supercomputer (SNS), serving hundreds of applications and supporting a maximum of 600,000-client scaling.

1 Introduction

High Performance Computing (HPC) is experiencing an era of explosive growth in computing scale and data. In order to meet the growing I/O demands of HPC applications, researchers propose Burst Buffer (BB) [35] to build a data acceleration layer through new storage media such as SSDs to serve applications' I/O quickly. Since 2016, more and more supercomputers have introduced Burst Buffer, such as Frontier [44], Fugaku [18], LUMI [15], Summit [43], Tianhe-2 [63], etc. Depending on the deployment location of SSDs, BBs can be classified into two types [10]: 1) local BB, which means SSDs are deployed on each computing node as local disks;2) shared BB, which means SSDs are deployed on dedicated nodes that can be accessed by computing nodes, such as I/O forwarding nodes [5] to support shared data access.

Since each BB node in the local BB is dedicated to serving one computing node, the local BB can achieve good scalability, and its performance can grow linearly with the number of computing nodes. However, it still has some limitations: 1) The local BB is not suitable for scenarios such as N-1 I/O mode (all processes share one file) and workflow due to the difficulty of data sharing. 2) The local BB architecture results in significant resource waste due to the large variance in I/O load between HPC applications and the relatively low percentage of data-intensive applications [67]. 3) The deployment cost of the local BB will rise sharply in the future as supercomputers scale up rapidly.

In contrast, the shared BB has the advantage of data sharing and deployment costs compared to the local BB. But it is challenging to support ultra-scale supercomputers with hundreds of thousands of clients [71], and existing work has many limitations. For example, Qian et al. [48] proposed LPCC, a caching technique that integrates SSDs in the Lustre [8] clients to improve read/write performance. However, LPCC is inefficient for data sharing and metadata-intensive access because data stored on the Lustre clients' SSDs must be flushed to the Lustre server before being shared. Herold et al. [25] proposed BeeOND, which functions similarly to LPCC but inherits the scalability and cache sharing limitations of BeeGFS.

Currently, we have entered the era of exascale supercomputers, which leads to a sharp increase in concurrent I/O. At the same time, the I/O requirements of HPC applications vary widely. How to unify the advantages of local BB and shared BB to meet the various application requirements and reduce the cost of building BB is an urgent problem to solve. Moreover, both the local BB and the shared BB have the advantage of high performance compared with the traditional global

*First Author and Second Author contribute equally to this work.
†Zuo-ning Chen is the corresponding author, email: chenzuoning@vip.163.com.

file system (e.g., Lustre, and we will use "GFS" to represent "global file system" in this paper.) but have the disadvantage of small capacity. So, BBs must work in conjunction with the GFS to meet capacity requirements. But the existing BBs either run in a static data migration mode [16, 43, 48] or require applications to migrate data through computing nodes [24, 51, 52], which has low migration efficiency and leads to a waste of computing resources. Large-scale BB data management and migration is also a problem that needs to be solved.

To solve these problems, we propose a novel BB file system, HadaFS, building on the shared BB deployment, which combines the scalability and performance advantages of local BB with the data sharing and deployment costs advantages of shared BB. HadaFS proposes a new architecture named Localized Triage Architecture (LTA) to solve the problem of insufficient scalability of the shared BB. LTA constructs all HadaFS servers as a shared storage pool, flexibly controlling the concurrency scale between clients and servers to ensure convenient data sharing. Additionally, HadaFS proposes a runtime user-level interface to ensure that I/O requests can be processed on the nearest server, helping clients use the BB in a manner that approximates the local BB. To solve the performance problems caused by the strong POSIX consistency, HadaFS proposes a full-path indexing approach, using the K-V approach instead of the traditional directory tree, supporting three-category metadata management policies. What's more, HadaFS integrates a data management tool to help users manage data in the BB, and migrate data between the BB and the GFS quickly and efficiently.

HadaFS has been deployed on the Sunway New-generation Supercomputer (SNS) [36], serving hundreds of applications, supporting a maximum of 600,000-client scaling, with an I/O aggregation bandwidth of 3.1 TB/s. The main contributions of this paper include:

- This paper describes a novel BB file system named HadaFS and performs a comprehensive experimental study on the SNS to evaluate its effectiveness.
- This paper proposes the LTA architecture, which enables the application-oriented data layout, achieves scalability on par with node-local BBs, and reduces interference caused by a large number of connections on a single server.
- HadaFS proposes three metadata synchronization strategies to address the mismatch between traditional file systems' complex metadata management and HPC applications' various consistent semantics requirements.
- This paper proposes a localized data management method that enables all BB nodes to execute data management commands in parallel via a pipeline, enabling efficient data query and fast data migration between the BBs and the GFS.

2 Motivation and Background

2.1 Motivation

As I/O requirements for HPC applications continue to grow, BBs have been introduced to many cutting-edge supercomputers. However, the existing major types of BB technologies still have many limitations.

2.1.1 The contradiction between BBs' scalibility and application behaviors

With the barrier to exascale computing being broken, the I/O concurrency of cutting-edge supercomputers can reach hundreds of thousands, which stresses the scalability of BBs. At the same time, the increase in the proportion of data sharing applications such as AI and workflow has led to changes in I/O requirements, and high-speed sharing of large-scale data has become much more important [45].

Building a more flexible BB architecture to meet the new changes in supercomputer systems and application requirements has become a challenge for the design of exascale supercomputers. Currently, some cutting-edge supercomputers use different solutions. For example, Frontier is an exascale supercomputer and uses independent hardware to build the local BB and the shared BB, respectively [44]. But this method requires many acceleration devices (SSDs) and high construction and maintenance costs. Fugaku deploys the shared BB and uses software to provide storage services similar to the local BB and the shared BB with different name spaces [21]. But their implementation is static and is challenging to control performance contention during large-scale I/O concurrency. Summit deploys the local BB and supports data sharing through the software [43]. But this method requires data sharing through GFS storage, which is inefficient.

In summary, the above methods have obvious advantages and disadvantages. Since shared BB can also be deployed on computing or data forwarding nodes [21, 66], from the perspective of cost control, we believe the shared BB deployment is more suitable for future ultra-large-scale computing node systems. To this end, in this paper we investigate how, starting from a shared BB model, we can attain the benefits of the local BB model to better address the requirements of HPC applications at exascale and beyond.

2.1.2 Complex metadata management mismatches application behaviors

Traditional file systems are designed for generality, so their file management is implemented in the directory tree structure and strictly follows the POSIX protocol. However, in HPC, computing nodes are generally responsible for reading and writing data and rarely perform directory tree access [32]. So relaxation of POSIX has become a common choice for many file systems [6, 12, 43, 59] to improve the performance.

However, due to the wide variety of HPC applications, how to relax POSIX remains a huge challenge.

Table 1: Applications and their suitable consistency semantics

Consistency Semantics	Applications
Strong consistency	–
Commit consistency	FLASH-HDF5 [60]
Session consistency	NWChem [58], QMCPACK [29], VASP [55] LBANN [20], Chombo [4], VPIC-IO [64]
Eventual consistency	ENZO [9], pF3D-IO [31], HACC-IO [38]

Wang et al. [60] studied the requirements of some typical HPC applications and classified the consistency semantics of HPC file systems into strong consistency semantics, commit consistency semantics, session consistency semantics, and eventual consistency semantics, as shown in Table 1. The higher the degree of consistency a system supports, the more adaptable it is, but at the cost of higher overhead. And different HPC applications have different requirements for consistency. Therefore, it is a big challenge to choose consistency semantics flexibly to balance the application's requirements and exploit the BB performance.

2.1.3 Inefficiencies in data management

A recent study found that although most applications on Summit and Cori can use the BB to speed up I/O performance, the BB utilization is low, and it is necessary to develop flexible data management tools for users [7]. Besides, the BB is not a place for applications to persistent store data in most cases. On the one hand, the BB capacity is smaller than the GFS capacity, e.g., Summit's BB capacity is 7.4PB while its GFS capacity is 250PB [26], Fugaku's BB capacity is 16PB while its GFS capacity is 100PB [21]. On the other hand, some typical HPC applications require hundreds of terabytes of data to input or output, e.g., NICAM-LETKF [69] has nearly 300,000 files and over 400 terabytes, Tokmark [65] has 32,768 files and nearly 100 terabytes. So, the BB system needs to consider efficiently and conveniently migrating data between the BB and the GFS.

Data migration between the BB and the GFS can be divided into two types: transparent and non-transparent. In transparent data migration, software automatically migrates the BB data to the GFS in blocks or files [16, 43, 48], which may cause a large amount of unnecessary data migration. In non-transparent migration, data migration often needs computing nodes to participate, leading to the computing resource being idle during the data migration process [24, 51, 52] and wasting resources. Both of the above types support loading data from the GFS to the BB asynchronously statically in advance, which can satisfy the data readahead requirements. However, neither of the above two types could support users to dynamically manage the BB data migration during the application running, which is very unfavorable for efficient utilization of the BB.

2.2 Background

The SNS is built on Sunway's new-generation heterogeneous high-performance many-core processors and interconnection network chips and adopts a similar architecture to Sunway TaihuLight [13]. The supercomputer consists of a computing system, interconnection network system, software system, storage system, maintenance and diagnosis system, power supply system, and cooling system. Figure 1 shows the overall architecture.

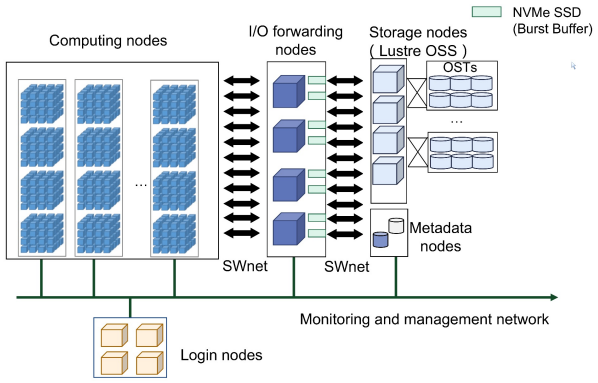


Figure 1: Architecture of the SNS

Each computing node contains one Sunway new-generation many-core processor *SW26010P*, which adopts a heterogeneous architecture similar to SW26010 and has 6 CGs (Core Groups [36]) with 390 computing cores. These components are interconnected through a ring network. The whole system is composed of more than 100,000 SW26010P processors, which are interconnected by a fat-tree network called *SWnet*.

The computing nodes are connected to the I/O forwarding nodes through the interconnection network, and the I/O forwarding nodes provide I/O request forwarding or storage. When providing I/O request forwarding, the SNS adopts a similar software architecture to TaihuLight (LWFS+Lustre [13]) and connects to the storage nodes through a separate storage network. When providing storage services, the SNS adopts a new software architecture and deploys a burst buffer file system, HadaFS, which is proposed in this paper. The I/O forwarding nodes serve as the HadaFS servers and use NVMe SSDs to handle users I/O requests.

3 Design and Implementation

3.1 Overview of HadaFS

Figure 2 shows the overall architecture of HadaFS, including the HadaFS client, HadaFS server, and data management tool. HadaFS serves as a shared burst buffer file system and can provide a global view for each client. The HadaFS client runs on the computing nodes and serves as a static/dynamic library that intercepts and redirects the POSIX I/O requests from applications to the HadaFS server, which means the

lifecycle of the HadaFS client is entirely dependent on applications. Note that HadaFS does not support the move, rename, or link operation as recent studies have demonstrated that these functions are rarely or not used at all during parallel application running [32]. The HadaFS server runs on the dedicated burst buffer nodes where NVMe SSDs are deployed, providing global data and metadata storage services. Each file in HadaFS is associated with two types of servers. One type is the data storage server that stores the HadaFS file's data through the basic file system on NVMe SSDs, and the other type is the metadata storage server that stores the HadaFS file's metadata through a high-performance database (RocksDB [17]). The data management tool, named *Hadash*, runs on the user login nodes and is used to manage the data migration between the global file system and HadaFS. More details can be seen in Section 3.7.

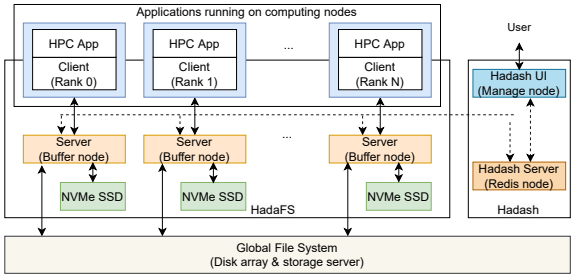


Figure 2: Architecture of HadaFS

3.2 Localized Triage Architecture

The traditional kernel file system handles the application's I/O requests by mounting the file system through the operating system, requiring to implement the full POSIX semantics and introducing the kernel's overhead of I/O requests stage-in and stage-out. An alternative method is to mount the file system through the application and bypass the kernel. Although this method can help clients avoid many rules that the kernel imposes on a file system and reduce the overhead of I/O requests stage-in and stage-out of the kernel, too many links are not conducive to large-scale expansion and may lead to service instability [71]. For example, for a computing node with 24 CPU cores, a file system client running in the kernel mode can be accessed by all processes running on the node only after mounting once. In contrast, each application process has to mount a client for a file system running in the user space. Obviously, both methods have certain limitations. HadaFS combines both advantages in a new approach named Localized Triage Architecture (LTA).

HadaFS follows the idea of bypassing the kernel and uses it by directly mounting the client into the application. In order to control the number of clients served by a single server at the same time and avoid the performance bottleneck caused by too many clients connected to a single server, HadaFS adopts the method of connecting only one server per client. For a HadaFS client, we call the HadaFS server connected to it the *bridge server*. The bridge server is responsible for handling all I/O requests generated by the client and writes data to the underlying file based on the offset and size of the I/O request initiated by the client. Each file corresponds to an independent file in the underlying file system(ext4) on the bridge server. When the client needs to access data on another server, it must be forwarded through the bridge server. Therefore, servers are a fully connected structure. Note that if the storage space of one bridge server is filled up, all the clients connected to it will automatically switch to another HadaFS server.

Considering that the number of clients (computing node processes) in an ultra-scale supercomputer will be much larger than the number of servers (storage nodes), it is a better way to perform the necessary I/O forwarding through the full connection of the server. In order to ensure that most I/O requests are processed on the bridge server and reduce the forwarding ratio of requests, HadaFS proposes an interface ($mount(mount\_point, Seq)$) to allow applications to control the selection of bridge servers when this is advantageous. *mount_point* stands for the mount point and is a prefix for a file path in HadaFS. *Seq* can be set flexibly according to changes in the application data sharing mode, network topology differences, and other factors to adapt to the application's data parallelism and system architecture parallelism. Currently, HadaFS supports three types of settings:

- *Seq* is set to the *MPI_RANK* of the application, which will connect the server for the client in a round-robin manner. This setting is suitable when the application is submitted to different computing nodes multiple times to ensure that each application process can connect precisely to the original bridge server, thus reducing the data forwarding during data access.
- *Seq* is set to the computing node ID, which can be used to match the topology between the specific computing node and the BB node, and helps ensure that the computing node can store data in the nearest BB node in the network.
- *Seq* is set according to the application's actual data distribution and sharing requirements, which means each client can specify any server it wants to connect to. So applications can improve the efficiency of data access by flexibly controlling the mapping between the clients and the servers.

LTA not only provides each computing node with a bridge server that runs as the local BB but also supports the sharing of all clients through the full interconnection between all bridge servers, combining the advantages of the local BB and the shared BB. Moreover, *mount_point* and *Seq* can be controlled by the environment variables, so HadaFS can support transparent mounting for users by loading the HadaFS library to read environment variables in advance before the application starts. However, to fully exploit the high performance

of HadaFS, especially for read performance, we advise applications to change their code to specify the client-to-server mapping, which can help reduce the data forwarding. After HadaFS mounted, applications can perform I/O with the interface, which is exactly the same as POSIX file operations.

3.3 Namespace and metadata handling

In order to improve the scalability and performance, HadaFS abandons the idea of directory trees and employs a full-path indexing approach like CHFS [57] and Vesta [14]. For a file in HadaFS, its data is stored on the bridge server of the HadaFS client that generated the file, and its metadata storage location is determined by the path hash. Files' metadata are stored by key-value, and the file path is a globally unique ID (key). The HadaFS client performs compliance checking on the absolute path of files based on its specific prefix mount point instead of checking layer by layer in the form of a directory tree. When multiple files need to be accessed, the load can be distributed to various servers, thus significantly improving metadata performance [57].

The metadata of HadaFS is compatible with the metadata items under the stat structure in Linux, including *name*, *ino*, *owner*, *mode*, *timestamp*, etc. HadaFS divides its metadata information into four categories:

- The first category is maintained during the file creation, including *name*, *owner*, *mode*, etc.
- The second category is maintained during the file access, including *file size*, *modification time*, *access time*, etc.
- The third category is information that HadaFS does not need to maintain, such as *ino*, *stdev*, etc. Since HadaFS adopts the file path as the globally unique ID, this information has no meaning in HadaFS.
- The fourth category is an ordered list of the location information of the file segments. Each item in the list is sorted by offset, consisting of server name, fragment offset, size, writing time, and other information.

Two kinds of metadata databases are maintained on each HadaFS server, and their data structures are shown in Figure 3. One is the local metadata database (*LMDB*), which stores the first and fourth category metadata information of the file locally, and the file's local identification (LID) is the local path corresponding to the file. The other is the global metadata database (*GMDB*), which stores the first two and the fourth categories of metadata information. The metadata of a HadaFS file is stored in a unique GMDB on a HadaFS server located by hashing the full path of the file.

Both metadata databases are built based on the RocksDB [50], which is also used to maintain metadata by many other famous file systems, such as GekkoFS [59], MadFS [28], etc. Although RocksDB does not support multi-threaded shared writing, it doesn't constitute a bottleneck, and this is demonstrated by both production-run and test scenarios. The keys of the two metadata databases are composed of the user's *UID*, *GID*, and *PATH*. The GID and UID are used to control the range of string retrieval because HadaFS uses string prefix matching to retrieve files. For the N-N I/O mode, each client writes the independent file, and the metadata stored in the LMDB matches the category one and four metadata stored in the GMDB. For the N-1 I/O mode, multiple clients share the same file and may use different HadaFS bridge servers. At this time, GMDB is responsible for merging file metadata from multiple LMDBs.

During the file reading and writing, LMDB records the change of its metadata, maintains an ordered list of local data segment locations, and sends the data to the GMDB to which the HadaFS file belongs. GMDB is responsible for maintaining a global list of data segment locations for files to support the global sharing of data between HadaFS servers. More details can be seen in Section 3.6.2.

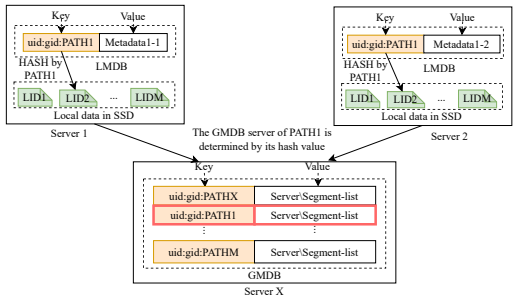


Figure 3: Two K-V tables on the HadaFS server

3.4 HadaFS I/O control and data flow

Here, we discuss the control and data flow details of HadaFS. Figure 4 shows an example with three HadaFS clients and three HadaFS servers:

- Client A performs an I/O request to create a file $F1$ and write 100-MB data. The data will be written directly to client A's bridge server, X.
- Server X writes the metadata information and location information of $F1$ to the LMDB.
- Based on the file path of $F1$, the metadata of $F1$ is calculated to be stored on server Y. Then, server X writes the metadata information and location information of $F1$ to the GMDB on server Y.
- Client C performs an I/O request to read a file $F1$.
- Client C's bridge server, Z, receives the I/O request and gets the metadata and location information of F1 from server Y based on the path of $F1$.
- Server Z reads data from server X and forwards it to client C.

Note that ensuring local writes and global readability of data streams is advantageous, especially for scenarios where the application needs to output checkpoints frequently. Besides, read-intensive applications can also achieve high performance through the mount interface, which can control the mapping relationship between the client and the bridge server to reduce the probability of forwarding as much as possible and improve read performance. In summary, approaches proposed by HadaFS not only help constrain the number of clients undertaken by each server and reduce performance jitter but also lay the foundation for the storage system to support the application's parallelism fully.

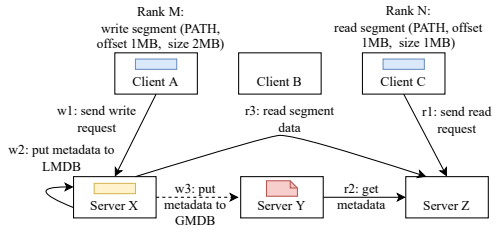


Figure 4: An example of HadaFS I/O control and data flow

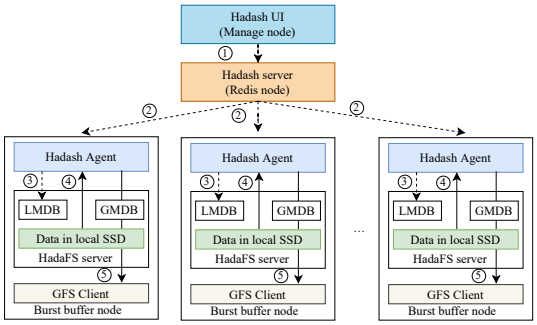


Figure 5: The stage-out flow of Hadash

3.5 Data management tool

We seek to overcome some disadvantages of existing BB approaches, such as LPCC [48] and Datawarp [24]. LPCC may result in the migration of large amounts of temporary data. Datawarp requires the application to specify the migration between the BB and GFS in their source code or job submission scripts, which is usually a static migration approach and requires computing nodes to participate in the migration. HadaFS provides a data management tool named *Hadash* to support users in retrieving and managing files in the directory tree view, which is divided into two categories according to functions: metadata information query and data migration.

The metadata information query mainly provides commands such as *ls*, *du*, *find*, *grep*, etc. Among them, *ls* and *find* support file information query with directory tree view. Hadash obtains information from the metadata database for these query-type operations and presents them in commands commonly used in the Linux shell. The other commands involve data migration, such as *rm*, *get*, *put*, etc. Hadash sends the commands to the data management modules on the HadaFS servers through a specific Redis [49] pipeline. Then, the data management module on each HadaFS server uses LMDB for local data location and executes these commands in parallel.

Figure 5 shows an example of the data migration flow from HadaFS to the GFS. Firstly, the user sends a data management command to the Hadash server via Hadash UI. Secondly, the Hadash server receives and forwards the command to all Hadash agents on the BB nodes. Thirdly, the Hadash agents parse the command, obtain the list of files specified by the command from LMDB, and then read these files from the local SSDs. Finally, the Hadash agents write these files to the GFS. When all files have finished writing, the Hadash agents will return success via another Redis pipeline, and then Hadash will tell the user that the stage-out has been completed. If the file to be migrated is continuously appended, Hadash will also continuously copy the newly written data. This behavior is the same as that of Linux's default data copying tool, *cp*. It is worth mentioning that Hadash uses a prefix-matching approach to present a virtual directory tree in data management, and the prefix-matching approach could be executed locally through the LMDB, thus reducing the impact on the GMDB.

Hadash uses a distributed management method to support data localization management. During the data management process, there is no need to know all the data views on the BB nodes, so its performance can grow linearly with the number of BB nodes (the main bottleneck is the GFS client).

3.6 Optimizations on HadaFS

3.6.1 Consistency semantics and metadata optimization

HadaFS adopts the idea of relaxed consistency semantics as other file systems [59]. HadaFS does not support cache data on the client and the server. It relies on the cache mechanism of the basic file system (ext4) to increase performance, and its consistency semantics mainly depend on metadata synchronization. Thus, HadaFS proposes three metadata synchronization strategies for different application scenarios to avoid the over-designing of traditional file systems.

The first strategy (called *mode1*) is to update all metadata asynchronously (corresponding to eventual consistency semantics). All operations are executed locally on the bridge servers first during file opening, deletion, reading, and writing, and metadata will be updated asynchronously from the LMDB to the GMDB later, which is the highest performance metadata update mode. This strategy is equivalent to providing node-local storage for computing nodes and is suitable for scenarios without data dependencies.

The second strategy (called *mode2*) is to update part of the metadata synchronously and part of the metadata asynchronously (corresponding to session consistency semantics and commit consistency semantics). The first metadata cat-

egory mentioned above (such as *name*, *owner*, etc.) will be updated synchronously when the file is created. The second metadata category (such as $file\ size$, $modify\ time$, etc.) will be updated asynchronously during file reading and writing or be updated synchronously by the flush operation. The second strategy is the default way to use and beneficial to improving file reading and writing performance.

The third strategy (called *mode3*) is to synchronize the metadata in all open, read, and write operations during the file access processes (slightly weaker than strong consistency semantics since HadaFS does not support overlapping writes). All servers need to get the file location first to ensure the synchronization of the first metadata category, and all operations such as open, write, read and flush need to synchronize the second metadata category.

HadaFS has special data location rules and does not use a distributed lock mechanism, so it isn't easy to ensure data consistency by HadaFS itself. If the application uses the N-N I/O mode (also known as File Per Process), then there is no data conflict because there is no file sharing. However, for the N-1 write scenario, since HadaFS requires that the data be written to the bridge server locally, it cannot support the overlap write in the N-1 Mode. In addition, atomic write is only supported under the third metadata synchronization strategy. To ensure data consistency, users must at least understand the file-sharing mode of the application, which could be obtained through Darshan [11], Beacon [67], etc.

3.6.2 Optimization on the shared file

The LTA architecture is suitable for N-N I/O mode, which can realize multi-file parallelism and help fully utilize the performance of the BBs. For N-1 I/O mode, HadaFS proposes a management method similar to ADIOS BP file layout [40] to improve the performance, where each client writes its own data in an independent file (corresponding to BP's process group), and the GMDB maintains the file's metadata(corresponding to BP's group index). In this way, a shared file can be stored on multiple servers, and the reading and writing of the shared file can be converted into concurrent reading and writing. However, since the HadaFS server stores data in file format, the data layout of each process is completely unknown, so the amount of fragment information managed by the GMDB may be high in extreme scenarios, which affects system performance. For example, suppose there are 100,000 processes concurrently writing a shared file, and each process writes 6 times consecutively. In a completely random case (no fragment information to merge), it may produce 600,000 file fragments.

To this end, HadaFS uses a list sorted by offset to store segment location information and merges location information for adjacent segments of the same bridge server to improve the performance of the segment management. The average time complexity of segment insertion and retrieval during write and read is $O(logN)$, where N is the number of segments in the file. All three metadata synchronization strategies support N-1 mode, and the metadata of a file will only be stored in one GMDB. Each GMDB uses one thread to access the RocksDB, and the peak IOPS of a GMDB is the peak IOPS for accessing a single file when using the third metadata strategy.

3.6.3 Interference avoidance

As we all know, there are many jobs running simultaneously on supercomputers. These jobs tend to compete for shared resources, resulting in I/O interference. I/O interference is a serious problem known to modern supercomputer users [19, 30, 33]. Many studies have also proved that dynamically mapping the client to the server is also helpful in improving application performance [27, 72]. Since the shared BBs support data sharing for many applications, the clients belonging to different jobs may share the same server, resulting in resource competition and performance degradation. Thanks to the flexible design of HadaFS, users can dynamically formulate the connection relationship from the HadaFS clients to the HadaFS servers, which can effectively help isolate the BB resources for different applications to solve the I/O interference between jobs. Section 4.4.3 demonstrates the effectiveness of HadaFS in avoiding interference with five real-world applications.

Moreover, we have also noticed that the flexible design of HadaFS has also led to high requirements for users who want to fully utilize HadaFS, as mentioned in Section 3.6.1 and Section 3.2. In order to reduce the burden on users, the HadaFS team is developing an automatic server assignment tool based on the monitoring tool [67] and adaptive I/O optimization framework [68]. This tool can automatically assign underlying BB resources, set the mount environment variables, and select the metadata synchronization strategy for applications, helping users isolate the underlying BB resources and improve the performance of their applications.

3.7 HadaFS on the SNS

HadaFS has been deployed on the SNS for over a year and supports hundreds of applications, including the 2021 Gordon Bell Award finalist application (Tokamak Plasma Simulation) [65] that scales to 480,000 processes (32,768 I/O processes) via HadaFS with an I/O aggregation bandwidth of 700 GB/s. Figure 6 shows the deployment of HadaFS. There are two HadaFS server on every I/O forwarding node, and each HadaFS server uses an NVMe SSD to support the storage of the HadaFS file's data (with ext4) and metadata (with LMDB and GMDB).

As we all know, the overhead of achieving fault tolerance can be significant. Therefore, HPC storage systems often transfer high availability to the application layer for implementation to pursue higher performance. Most HPC applications

generally use periodic write checkpoints [6] to reduce the cost of restoring applications after failures occur. So, HadaFS is positioned as a temporary high-performance BB system, similar to the BB system on Frontier [44], Summit [43], etc. The Sunway new generation supercomputer doesn't adopt erasure code or data redundancy to handle node failures. If a BB node fails, HadaFS can be available after the BB node recovery as long as the SSD is not damaged. Moreover, in order to reduce the cost of recovering data in case of failure, HadaFS supports applications to periodically back up key data to the GFS. There have been 15 BB node failures but no SSD corruptions for over a year of deployment.

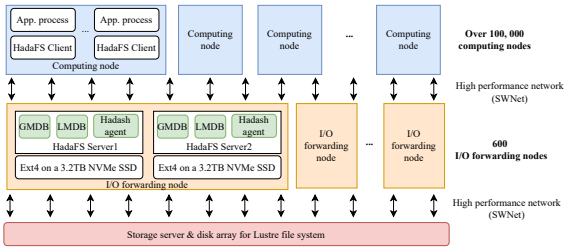


Figure 6: The deployment of HadaFS

4 Evaluation

We carry out the evaluation on the SNS to test the performance of HadaFS. The SNS contains more than 100,000 computing nodes, and each node can start up to 6 MPI processes and 6 HadaFS clients. That means the whole machine can support more than 600,000 MPI processes and 600,000 HadaFS clients. It has a total of 600 I/O forwarding nodes, and each I/O forwarding node is configured with two 3.2TB NVMe SSDs (Each NVMe SSD corresponds to a HadaFS server). All nodes are interconnected using the SWnet network, and HadaFS uses the SWnet-based RDMA protocol to transfer data. We compare the performance of HadaFS with BeeGFS (a popular parallel file system that many supercomputers have used to manage the BB [3,25,41]) and GFS (the traditional parallel file system used by the SNS based on LWFS [67] and Lustre [8]). To ensure the evaluation fairness, we slightly modify BeeGFS (version: 7.2.5) to make it better suited to the SNS, including using the SWnet to provide the high-speed RDMA communication and increasing the number of processes for managing services to achieve high-speed mounting performance. And, BeeGFS is configured with the same number of storage servers and metadata servers as HadaFS. For GFS, it uses a similar forwarding architecture as Sunway TaihuLight [13], consisting of 132 OSSs and 4 MDSs.

4.1 Metadata performance evalutaion

In order to improve the metadata performance, HadaFS proposes three metadata management strategies. Here, we use *mode1*, *mode2*, and *mode3* to represent these three strategies as mentioned in Section 3.6.1.

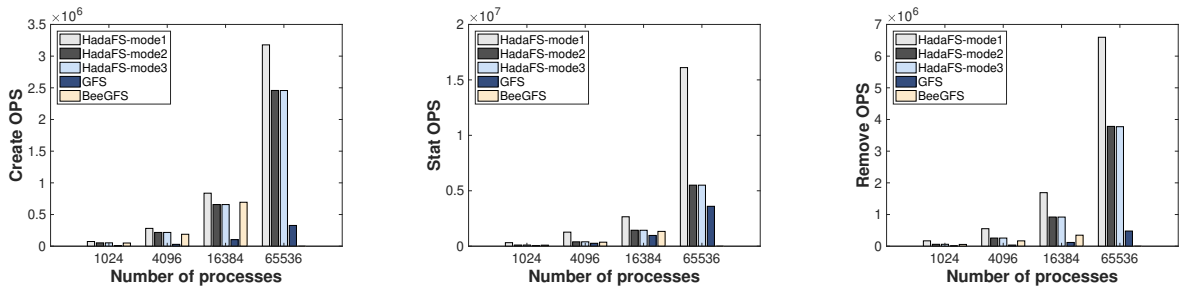
We first use MDTest [2] (A benchmark for metadata performance evaluation) to compare the metadata performance differences of HadaFS, GFS, and BeeGFS with parallel scales of 1024, 4096, 16384, and 65536 processes. 4, 16, 64, and 256 I/O forwarding nodes are used for HadaFS and BeeGFS, each running a data server and a metadata server on a common SSD. Note that the number of the GFS metadata servers in this experiment is 4 due to the limited metadata servers of the Lustre file system.

Figure 7(a), 7(b), and 7(c) show the OPS comparison of *Create*, *Stat*, and *Remove*, respectively. Mode1 has the highest performance. Mode2 has comparable performance to mode3 because there is no read/write operation in the MDTest setting. BeeGFS metadata performance is similar to HadaFS's mode2 and mode3 for 1024 processes. When the number of test processes increases, both HadaFS and BeeGFS obtain the higher performance, but the performance of BeeGFS is slightly slower than HadaFS. Besides, BeeGFS can not scale up to 65,536 processes. The main reason is that BeeGFS needs to mount 16384 clients to support 65,536 processes on the SNS, but it cannot mount successfully at such a large scale due to the limitation of centralized management service (It isn't easy to successfully mount clients in batches after exceeding 10,000 nodes). Unsurprisingly, the traditional file system GFS has the lowest performance due to the performance overhead caused by data forwarding software LWFS and the limited metadata servers of Lustre.

4.2 Data performance evaluation

Here, we use IOR [53] (A benchmark for data performance evaluation) to compare the I/O bandwidth differences between HadaFS, GFS, and BeeGFS with parallel scales of 1024, 4096, 16384, and 65536 processes. The request size is set to 8 KB for random read/write and 1 MB for sequential read/write. 4, 16, 64, and 256 I/O forwarding nodes are used for HadaFS and BeeGFS, each running a data server and a metadata server on a common SSD. Specifically, for GFS, the data server is the Lustre OSS, and the metadata server is the Lustre MDS. All 132 OSSs (located on the storage nodes) and 4 MDS are used in the experiment.

Figure 8 shows the results. For HadaFS, mode1 has the highest performance, followed by mode2, and finally mode3. HadaFS does not show a significant performance advantage at smaller scales, but as the scale reaches 65,536 processes, HadaFS performs much better than other file systems. For read operations, HadaFS can approach the theoretical performance limit of SSDs. For write operations, random writes are not conducive to the performance of HadaFS due to the inability to utilize the kernel caching mechanism. For BeeGFS, it can perform close to mode1 and mode2 sometimes but still cannot scale to 65,536 processes. Expectantly, GFS has the lowest performance again due to the forwarding overhead (see



(a) create OPS (b) stat OPS (c) remove OPS

Figure 7: Metadata performance comparison

Section 4.1) and the storage medium (OSTs are constructed by HDDs).

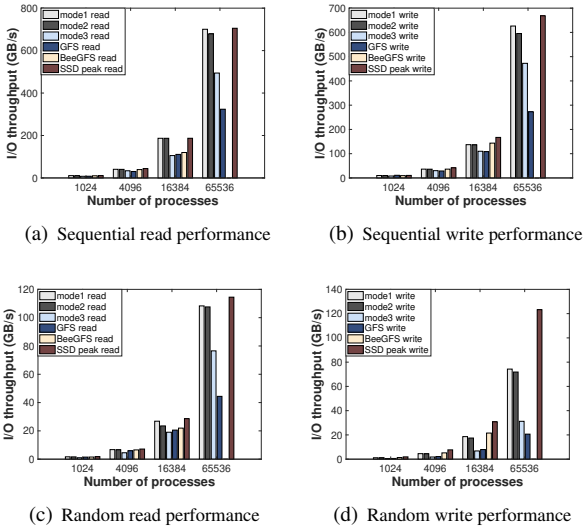
In addition, we also scale HadaFS to 600,000 clients and 1200 servers, and Figure 9 shows the experiment results. Since mode2 is a default metadata management strategy of HadaFS, all tests are done based on mode2, and the request size is set to 1 MB. Comparing the theoretical performance of 1200 NVMe SSDs (3.4 TB/s and 3.1 TB/s for read and write, respectively), the bandwidth utilization of the SSDs is close to 90% under ultra-large-scale concurrent data access.

4.3 Data migration evaluation

In order to manage data migration between the BB system and the GFS quickly and efficiently, HadaFS provides a data management tool named Hadash. Here, we evaluate Hadash in terms of its I/O throughput and its ability to migrate small files compared with Datawarp [24] simulated by Slurm-LUA [52] (Datawarp and LUA are the only two BB plugins supported by Slurm). And like BeeGFS, we construct a new LUA script based on a BB-LUA-example provided by Slurm to fit the experimental environment to ensure fairness. HadaFS is configured with 256 data servers and 256 metadata servers, and Datawrap is configured with 4096 processes for data migration.

First, we use 4096 files for the data stage-in and stage-out experiments, and the total data volume of these files ranges from 256 MB to 64 TB. Figure 11 shows the results of the experiment. When the total volume of the files to be migrated is relatively small (less than 64 GB for stage-in and less than 16 GB for stag-out), Hadash obtains a slightly worse performance than Datawarp. This is because when the total volume is small, the size of the individual files is also small, resulting in the command distribution and result acquisition mechanism based on the Redis pipeline occupying a larger proportion of the time. However, as the total volume and the individual file size get larger, the I/O throughput of Hadash stabilizes around 100 GB/s (for stage-in) and 140 GB/s (for stage-out), which is much higher than Datawarp.

Additionally, we found that the stage-out performance is significantly better than the stage-in performance. The write performance of the GFS and the read performance of the BB determines the stage-out performance, while the read performance of the GFS and the write performance of the BB determines the stage-in performance. In our test, the GFS (Lustre) client has a write cache, so the write performance of the GFS is higher than the read performance, and the read performance of the BB is also higher than the write performance, which leads to the higher stage-out performance.



(a) Sequential read performance (b) Sequential write performance

(c) Random read performance (d) Random write performance

Figure 8: I/O throughput comparison

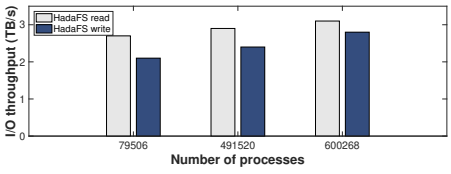


Figure 9: Ultra-scale performance

We also evaluated Hadash's ability to handle large amounts of small files. Figure 12 shows the result of the experiment using different numbers of 4-KB small files for the data stage-in and stage-out. For stage-in, Hadash outperforms Datawarp significantly when the number of small files exceeds 10,000

while Datawarp's performance varies less. For stage-out, Hadash outperforms Datawarp significantly when the number of small files exceeds 100,000.

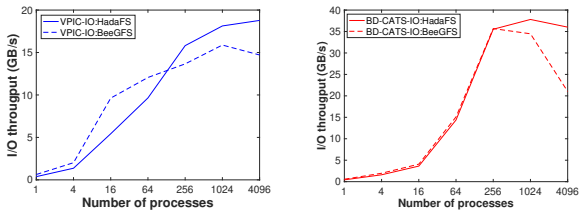
Again, the performance of stage-out is better. One of the reasons is as stated above, and another reason is as follows. In the stage-in flow, Hadash needs to read all files in a single directory from the GFS, and this process takes longer as the number of files in a single directory increases. In contrast, Hadash does not need to read any files in the directory from the GFS in the stage-out flow and only needs to create files.

4.4 Evaluation with real-world applications

4.4.1 Performance evaluation on the shared files

For the shared file access pattern, HadaFS adopts the idea of BP files similar to ADIOS [39] and further improves the shared file access performance by merging adjacent segments through ordered lists. This subsection compares the performance differences between HadaFS and BeeGFS on shared file access using several applications. Both HadaFS and BeeGFS are configured with 16 servers, each running on a common SSD. Figure 10 shows the results.

First, we use VPIC-IO [64] (provides scalable writing HDF5 data by VPIC) to evaluate the performance of HadaFS when writing shared files. Applications' parallelism scales from 1 to 4096, and each process writes about 1.1-GB data to a shared file containing 8 variables. Figure 10(a) shows the results. BeeGFS performs better than HadaFS when the application's parallelism is less than 64. This is because BeeGFS clients can use the kernel's cache, and the striping technique used by BeeGFS can ensure a low probability of conflict at small scales. However, as the parallelism gets larger, HadaFS outperforms BeeGFS significantly due to its good scalability.



(a) Write performance (b) Read performance

Figure 10: Performance evaluation on the shared file

Then, we use BD-CATS-IO [46] (provides scalable reading HDF5 data by the VPIC) to evaluate the performance of HadaFS when reading shared files. Applications' parallelism also scales from 1 to 4096, and each process reads about 1.1-GB data from a shared HDF5 file. Figure 10(b) shows the results. BeeGFS and HadaFS have almost the same performance when the parallelism of applications is less than 256. Similar to the write performance, when the application scale gets larger, the performance of HadaFS will be significantly better than that of BeeGFS, as HadaFS uses the LTA architecture to better isolate I/O conflicts between different clients.

4.4.2 Performance evaluation on the mount policy

Compared to the traditional fully connected mount approach, HadaFScannot guarantee that the data demanded by the client is always on its bridged server, so we first evaluate the performance impact of I/O forwarding on HadaFS. We distribute files evenly and regularly on the server according to the RANK number in advance and then accurately control the forwarding of generated data between servers through the mount interface.

We use one process to evaluate the latency variation of different block sizes due to the I/O forwarding, and Figure 13(a) shows the results. The solid line in the figure represents the client's direct access latency to its bridge server, while the dashed line represents the I/O forwarding latency. I/O forwarding does cause an increase in latency. The larger the block size, the smaller the proportional increase in latency. For 8-KB and 1-MB block sizes, the latency increases by 34.4% and 17.7%, respectively.

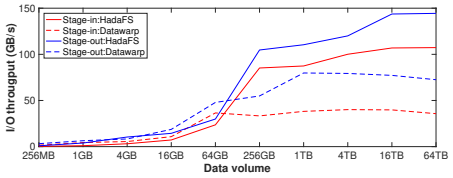


Figure 11: Data migration throughput comparison

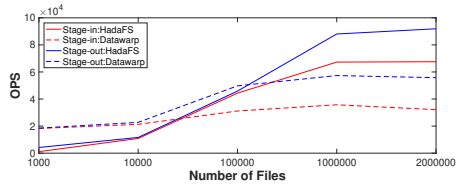


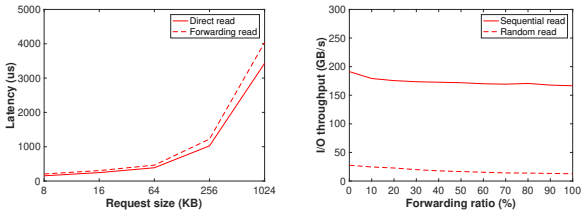
Figure 12: Number of files migrated per second comparison

We then evaluate the impact of the data forwarding ratio on the bandwidth of HadaFS. The higher the data forwarding ratio, the more data needs to be forwarded through the bridge server. In the experiment, HadaFS is configured with 16,384 clients and 64 servers, and the I/O request size is set to 1 MB for sequential read and 8 KB for random read. Figure 13(b) shows the results. As the I/O forwarding ratio increases, the throughput of HadaFS decreases, with a maximum loss of 18% for sequential read and 54% for random read. This is because the smaller the block size, the larger the forwarding overhead, and the larger the throughput loss.

However, note that HadaFS provides a runtime mount interface to control the mapping relationships flexibly, which can significantly reduce I/O forwarding. Let's take NEMO (a state-of-the-art modeling framework for research activities and forecasting services in the ocean and climate sci-

ences) [70] and its post-processing as an example to illustrate the advantages of the runtime mount interface. NEMO uses N-N I/O mode (also known as File Per Process) to read and write NetCDF files and is configured with 65,536 processes. And its post-processing is configured with 512, 1024, and 2048 processes. It is worth mentioning that in real application scenarios, the parallelism of the post-processing is significantly smaller than the parallelism of the module application. All 250,000 files (the total volume is more than 5 TB) output by NEMO are stored in 16 HadaFS servers.

Figure 14 shows the results. In the default configuration, the individual post-processing processes often need to access data that is not on the bridge server. So, the forwarding rate is high(up to 93%), and the performance is poor. After re-mapping the client-to-server connections with the mount interface, the forwarding rate can be greatly reduced, and performance can be significantly improved, up to 30% or more. This demonstrates that the flexible mount interface of HadaFS can significantly improve the performance of applications that need to share data.



(a) Forwarding latency (b) Forwarding throughput

Figure 13: Forwarding evaluation of HadaFS

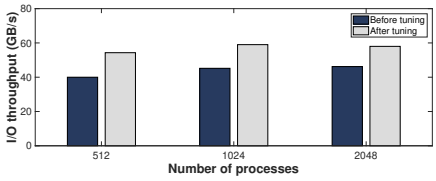


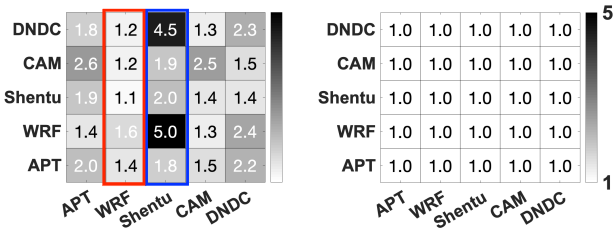
Figure 14: Performance improvement with mount interface

4.4.3 Performance evaluation on the interference

In this subsection, we evaluate HadaFS as a shared file system with 5 real-world applications, including APT (a particle dynamics simulation application) [62], WRF (a regional numerical weather prediction system) [1], Shentu (an extreme-scale graph engine) [34], CAM (a standalone global atmospheric model deriving from the CESM project for climate simulation/projection) [54], and DNDC (a biogeochemistry application for agroecosystems) [22].

First, we simulate the common I/O interference caused by sharing resources between jobs in HPC by co-running two applications on the same HadaFS server, and each application runs with 512 processes. Figure 15(a) shows the results. Each block's darkness reflects the application's slowdown factor at the row header by the application at the column header. As we can see, since different applications have different I/O behaviors, they share HadaFS with each other resulting in varying levels of performance slowdown. For example, WRF is a traditional serial I/O application that uses only one I/O process to access files through the NetCDF library, and its I/O load is very low (I/O bandwidth less than 200 MB/s). When WRF shares HadaFS server with other applications, it has less impact on them, as marked by the red box. On the contrary, Shentu is an I/O intensive application with N-N I/O mode, so its I/O bandwidth is very high (up to 2.5 GB/s). When Shentu shares HadaFS server with other applications, it has a high impact on them (up to 5x performance slowdown for other applications), as marked by the blue box.

HadaFS supports a runtime user-level mount interface and can assign the service resources according to the group name mentioned in Section 3.6.3. So in the production environment, HadaFS can flexibly change the mapping relationship from HadaFS clients to HadaFS servers through the mount interface to avoid I/O interference. Figure 15(b) shows the performance of avoiding sharing HadaFS server with other applications through the mount interface. This experiment demonstrates that the flexible mount approach provided by HadaFS can be beneficial for applications to avoid I/O interference.



(a) Before tuning (b) After tuning

Figure 15: Impact of HadaFS's interference avoidance on pairwise application co-run slowdown

4.4.4 Evaluation with large scale applications

This subsection shows the usage of HadaFS in five real-world large-scale applications, including NEMO [70], TK (Tokamak Plasma Simulation, 2021 Gorden Bell Prize finalist) [65], DiDA (an AI-enabled large-scale parallel atmospheric data-assimilation system) [23], Jstack (a debugging tool for the SNS) [47], and SWLBM (an efficient and scalable LBM) [37].

Figure 16 shows that applications can achieve significant I/O improvement (at least 7x) and reduce their runtimes and I/O ratios after using HadaFS, proving the effectiveness of HadaFS in improving the large-scale applications' performance. Details are as follows. TK runs with 32,768 I/O processes and 960 HadaFS servers, and the total runtime is more than 48 hours. With HadaFS, the I/O percentage of the total runtime dropped from 9.4% to 1.5%. NEMO runs with 480,000 I/O processes and 1200 HadaFS servers, and the total runtime of a model-year simulation is about 114 hours. With HadaFS, the I/O percentage of the total runtime dropped from

1.3% to 0.13%. DIDA runs with 65,536 I/O processes and 1200 HadaFS servers, and the total runtime is more than 2 hours. With HadaFS, the I/O percentage of the total runtime dropped from 4.1% to 0.5%. Jstack runs with 100,000 I/O processes and 256 HadaFS servers, and the total runtime is about 100 minutes every day. With HadaFS, the I/O percentage of the total runtime dropped from 5.0% to 0.46%. SunwayLBM runs with 18,000 I/O processes and 256 HadaFS servers, and the average runtime is about 7 days. With HadaFS, the I/O percentage of the total runtime dropped from 14.2% to 2.6%.

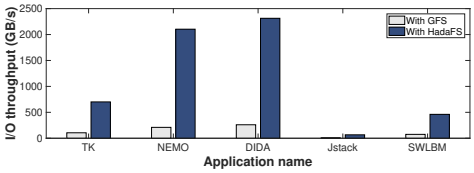


Figure 16: Performance improvement for large-scale real-world applications

5 Related work

Burst buffer on Top computers Many supercomputers deploy Burst Buffers to accelerate applications' I/O performance. Summit [26] adopts the local BBs technology and deploys SSDs on each computing node. To support data staging and migrate applications data to the GFS, Summit uses two technologies. One is Spectral, which provides the block-level data cache for applications. The other is SymphonyFS, which provides the file-level data cache for applications [43]. Fugaku [18] deploys the shared SSDs on the dedicated BB nodes (also used as the computing nodes but have two more cores), with each BB node serving a portion of the computing nodes, and provides users with three namespaces for different BB usage through LILO [21]. Applications can select the corresponding namespace according to the shared access requirements of the data (intra-node, intra-application, or inter-applications). As the world's first exascale supercomputer, Frontier [44] builds both the local BB and the shared BB. The local BB provides a burst data cache within the node, while the shared BB provides a shared data cache. In summary, the above situation shows that BB technology is moving towards integrating the local BB and the shared BB to support HPC applications' various requirements. For supercomputers with more than 100,000 nodes, local BB needs to deploy NVMe SSDs on each computing node, which will undoubtedly increase the cost. HadaFS combines the advantages of local BB and shared BB through the LTA architecture and the application-controllable mount interface, which can be deployed on ultra-scale supercomputers with more than 100000 nodes at a relatively low cost.

Researches for Burst Buffer Research on BBs has recently become a hot topic and can be divided into three technical routes. The first one is to improve the traditional distributed file systems and add new functions to support BBs, such as Lustre LPCC [48], which can cache data in client SSDs for transparent data caching. However, this mechanism inherits the scalability problems of traditional distributed file systems and is difficult to scale to ultra-scale. Similarly, there is BeeOND, which is based on BeeGFS [3]. The second one is to add data-sharing mechanisms based on the local BBs, such as Unifyfs [42], Burstfs [61], CHFS [57], Gfarm/BB [56], etc. These file systems run on the user layer and will be created when the job is submitted and destroyed when the job completes. In order to take advantage of BB's performance, these file systems also use a consistent relaxation protocol similar to HadaFS but does not consider data staging. The third one is to build a full-featured persistent BB storage system, e.g., DAOS [39]. DAOS [39] is an object storage system developed based on SPDK/PMDK. It is organized in an object-centric manner, supports transaction and multiple consistency management methods, and supports POSIX semantics based on object storage. Compared with the above works, HadaFS is built based on Shared BB, which has more advantages in scalability and has passed the verification of ultra-large-scale deployment of more than 100,000 nodes. In addition, HadaFS can provide applications with flexible and controllable POSIX consistency semantics.

6 Conclusion

We present a Burst Buffer file system named HadaFS, bridging the local BB and the shared BB based on the shared BB deployment. HadaFS can support ultra-scale deployments and balance the performance and the overhead with the novel architecture LTA and hierarchical metadata management mechanism. Besides, HadaFS integrates an internal data management tool named Hadash, which can provide a global data view and efficient data migration for users. HadaFS has been deployed on the SNS (over 100,000 computing nodes) and supports hundreds of applications. Especially, HadaFS supports several ultra-scale applications, providing stable and high-performance I/O services for these applications in preparation for the ACM Gordon Bell bid. Moreover, We demonstrate the high performance, high scalability, and low cost of HadaFS through a comprehensive experimental study.

Acknowledgement

We appreciate the thorough and constructive comments/suggestions from all reviewers. We thank our shepherd, Rob Ross, for his guidance during the revision process. This work is partially supported by the National Key R&D Program of China (Grant No. 2020YFB0204800), Marine S&T Fund of Shandong Province for Laoshan Laboratory (LSKJ202202100), National Natural Science Foundation of China (Grant No. U2242210), and the Major Key Project of PCL (No. PCL2022A05).

References

[1] A description of the advanced research WRF version 3. http://www2.mmm.ucar.edu/wrf/users/.

[2] Mdtest hpc benchmark, 2010. https://sourceforge.net/projects/mdtest/.

[3] David Abramson, Chao Jin, Justin Luong, and Jake Carroll. A beegfs-based caching file system for data-intensive parallel computing. In *Asian Conference on Supercomputing Frontiers*, pages 3–22. Springer, Cham, 2020.

[4] Mark F. Adams, Phillip Colella, Daniel T. Graves, Jeffrey N. Johnson, Hans Johansen, Noel Keen, Terry J. Ligocki, Daniel F. Martin, Peter McCorquodale, David Modiano, Peter O. Schwartz, T. D. Sternberg, and Brian van Straalen. Chombo software package for amr applications design document. 2014.

[5] Nawab Ali, Philip Carns, Kamil Iskra, Dries Kimpe, Samuel Lang, Robert Latham, Robert Ross, Lee Ward, and Ponnuswamy Sadayappan. Scalable i/o forwarding framework for high-performance computing systems. In *2009 IEEE International Conference on Cluster Computing and Workshops*, pages 1–10. IEEE, 2009.

[6] John Bent, Garth Gibson, Gary Grider, Ben McClelland, Paul Nowoczynski, James Nunez, Milo Polte, and Meghan Wingate. Plfs: a checkpoint filesystem for parallel applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–12. IEEE, 2009.

[7] Jean Luca Bez, Ahmad Karimi, Arnab Paul, Bing Xie, Suren Byna, Philip Carns, Sarp Oral, Feiyi Wang, and Jesse Hanley. Access patterns and performance behaviors of multi-layer supercomputer i/o subsystems under production load. pages 43–55, 06 2022.

[8] Peter Braam. The lustre storage architecture. *arXiv preprint arXiv:1903.01955*, 2019.

[9] Corey Brummel-Smith, Greg L. Bryan, Iryna S. Butsky, Lauren Corlies, et al. Enzo: An adaptive mesh refinement code for astrophysics. *The Astrophysical Journal Supplement Series*, 211, 2019.

[10] Lei Cao, Bradley W. Settlemyer, and John Bent. To share or not to share: comparing burst buffer architectures. In *SpringSim*, 2017.

[11] Philip Carns, Robert Latham, Robert Ross, Kamil Iskra, Samuel Lang, and Katherine Riley. 24/7 characterization of petascale i/o workloads. In *International Conference on Cluster Computing and Workshops*, pages 1–10, New Orleans, 2009. IEEE.

[12] Philip H Carns, Walter B Ligon III, Robert B Ross, and Rajeev Thakur. {PVFS}: A parallel file system for linux clusters. In *4th Annual Linux Showcase & Conference (ALS 2000)*, 2000.

[13] Qi Chen, Kang Chen, Zuo-Ning Chen, Wei Xue, Xu Ji, and Bin Yang. Lessons learned from optimizing the sunway storage system for higher application i/o performance. *Journal of Computer Science and Technology*, 35(1):47–60, 2020.

[14] Peter F Corbett and Dror G Feitelson. The vesta parallel file system. *ACM Transactions on Computer Systems (TOCS)*, 14(3):225–264, 1996.

[15] Cray. Lumi supercomputer, 2022. https://www.lumi-supercomputer.eu/.

[16] Bin Dong, Surendra Byna, Kesheng Wu, Prabhat, Hans Johansen, Jeffrey N. Johnson, and Noel Keen. Data elevator: Low-contention data movement in hierarchical storage system. *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*, pages 152–161, 2016.

[17] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. Rocksdb: Evolution of development priorities in a key-value store serving large-scale applications. *TOS*, 17(4):1–32, 2021.

[18] Jack Dongarra. Report on the fujitsu fugaku system. *University of Tennessee-Knoxville Innovative Computing Laboratory, Tech. Rep. ICLUT-20-06*, 2020.

[19] Matthieu Dorier, Gabriel Antoniu, Robert Ross, Dries Kimpe, and Shadi Ibrahim. CALCioM: Mitigating I/O interference in HPC systems through cross-application coordination. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2014.

[20] Brian C. Van Essen, Hyojin Kim, Roger A. Pearce, Kofi Boakye, and Barry Y. Chen. Lbann: livermore big artificial neural network hpc toolkit. *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments*, 2015.

[21] Fujitsu. File system and power management enhanced for supercomputer fugaku, 2021. https://www.fujitsu.com/.

[22] Donna L Giltrap, Changsheng Li, and Surinder Saggar. DNDC: A process-based model of greenhouse gas fluxes from agricultural soils. *Agriculture, Ecosystems & Environment*, 2010.

[23] Thomas M Hamill. Ensemble-based atmospheric data assimilation. *Predictability of weather and climate*, 124:156, 2006.

[24] Dave Henseler, Benjamin Landsteiner, Doug Petesch, Cornell Wright, and Nicholas J Wright. Architecture and design of cray datawarp. *Cray User Group CUG*, 2016.

[25] Frank Herold, Sven Breuner, and Jan Heichler. An introduction to beegfs, 2014.

[26] Jonathan Hines. Stepping up to summit. *Computing in science & engineering*, 20(2):78–82, 2018.

[27] Xu Ji, Bin Yang, Tianyu Zhang, Xiaosong Ma, Xiupeng Zhu, et al. Automatic, application-aware i/o forwarding resource allocation. In *17th USENIX Conference on File and Storage Technologies*, pages 265–279, 2019.

[28] chen Kang, Wu Yongwei, and zheng Weiming. Madfs: a high performance burst buffer file system. *Big Data*, 7(3):150, 2021.

[29] Jeongnim Kim, Andrew D. Baczewski, Todd D. Beaudet, Anouar Benali, et al. Qmcpack: an open source ab initio quantum monte carlo package for the electronic structure of atoms, molecules and solids. *Journal of Physics: Condensed Matter*, 30, 2018.

[30] Youngjae Kim, Scott Atchley, and Galen M. Shipman. LADS: Optimizing data transfers using layout-aware data scheduling. In *13th USENIX Conference on File and Storage Technologies (FAST)*, 2015.

[31] Steven Langer, Abhinav Bhatele, and Charles H. Still. pf3d simulations of laser-plasma interactions in national ignition facility experiments. *Computing in Science & Engineering*, 16:42–50, 2014.

[32] Paul Hermann Lensing, Toni Cortes, Jim Hughes, and André Brinkmann. File system scalability with highly decentralized metadata on independent storage devices. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 366–375. IEEE, 2016.

[33] Yan Li, Xiaoyuan Lu, Ethan L. Miller, and Darrell D. E. Long. ASCAR: Automating contention management for high-performance storage systems. In *IEEE International Conference on Massive Storage Systems and Technology (MSST)*, 2015.

[34] Heng Lin, Xiaowei Zhu, Bowen Yu, Xiongchao Tang, Wei Xue, Wenguang Chen, Lufei Zhang, Torsten Hoefler, Xiaosong Ma, Xin Liu, et al. Shentu: processing multi-trillion edge graphs on millions of cores in seconds. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 706–716. IEEE, 2018.

[35] Ning Liu, Jason Cope, Philip Carns, Christopher Carothers, Robert Ross, Gary Grider, Adam Crume, and Carlos Maltzahn. On the role of burst buffers in leadership-class storage systems. In *2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–11. IEEE, 2012.

[36] Yong Liu, Xin Liu, Fang Li, Haohuan Fu, Yuling Yang, Jiawei Song, Pengpeng Zhao, Zhen Wang, Dajia Peng, Huarong Chen, et al. Closing the" quantum supremacy" gap: achieving real-time simulation of a random quantum circuit using a new sunway supercomputer. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2021.

[37] Zhao Liu, XueSen Chu, Xiaojing Lv, Hongsong Meng, Shupeng Shi, Wenji Han, Jingheng Xu, Haohuan Fu, and Guangwen Yang. Sunwaylb: Enabling extreme-scale lattice boltzmann method based computing fluid dynamics simulations on sunway taihulight. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 557–566. IEEE, 2019.

[38] LLNL. Hacc i/o benchmark summary, 2017. https://asc.llnl.gov/sites/asc/files/2020-06/HACC_IO_Summary_v1.0.pdf.

[39] Jay Lofstead, Ivo Jimenez, Carlos Maltzahn, Quincey Koziol, John Bent, and Eric Barton. Daos and friends: a proposal for an exascale storage system. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 585–596. IEEE, 2016.

[40] Jay Lofstead, Fang Zheng, Scott Klasky, and Karsten Schwan. Input/output apis and data organization for high performance scientific computing. In *2008 3rd Petascale Data Storage Workshop*, pages 1–6. IEEE, 2008.

[41] Satoshi Matsuoka. Being "bytes-oriented" in hpc leads to an open big data/ai ecosystem and further advances into the post-moore era. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 5–5. IEEE Computer Society, 2017.

[42] Adam Moody, Danielle Sikich, Ned Bass, Michael J. Brim, and others. Unifyfs: A distributed burst buffer file system - 0.1.0, 10 2017.

[43] Sarp Oral, Sudharshan S Vazhkudai, Feiyi Wang, Christopher Zimmer, Christopher Brumgard, Jesse Hanley, George Markomanolis, Ross Miller, Dustin Leverman, Scott Atchley, et al. End-to-end i/o portfolio for the summit supercomputing ecosystem. In *Proceedings of the International Conference for High Performance*

Computing, Networking, Storage and Analysis, pages 1–14, 2019.

[44] ORNL. Frontier exascale system, 2022. https://www.olcf.ornl.gov/frontier/.

[45] Tirthak Patel, Suren Byna, Glenn K Lockwood, Nicholas J Wright, Philip Carns, Robert Ross, and Devesh Tiwari. Uncovering access, reuse, and sharing characteristics of {I/O-Intensive} files on {Large-Scale} production {HPC} systems. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 91–101, 2020.

[46] Md Mostofa Ali Patwary, Suren Byna, Nadathur Rajagopalan Satish, Narayanan Sundaram, Zarija Lukić, Vadim Roytershteyn, Michael J Anderson, Yushu Yao, Pradeep Dubey, et al. Bd-cats: big data clustering at trillion particle scale. In *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2015.

[47] Dajia Peng, Yunlong Feng, Yong Liu, Xin Liu, Wei Xue, Dexun Chen, Jiawei Song, and Zuoning Chen. Jdebug: A fast, non-intrusive and scalable fault locating tool for ten-million-scale parallel applications. *IEEE Transactions on Parallel and Distributed Systems*, 2022.

[48] Yingjin Qian, Xi Li, Shuichi Ihara, Andreas Dilger, Carlos Thomaz, Shilong Wang, Wen Cheng, Chunyan Li, Lingfang Zeng, Fang Wang, et al. Lpcc: hierarchical persistent client caching for lustre. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2019.

[49] WG Redis. Redis, 2016. http://redis.io/topics/faqAccessedNovember.

[50] RocksDB. A persistent key-value store for fast storage environments, 2022. http://rocksdb.org/.

[51] RSE-Cambridge. The data accelerator, 2022. https://www.hpc.cam.ac.uk/research/data-acc.

[52] SchedMD. Slurm workload manager, 2022. https://slurm.schedmd.com/overview.html.

[53] Hongzhang Shan and John Shalf. Using ior to analyze the i/o performance for hpc platforms. Technical report, Ernest Orlando Lawrence Berkeley NationalLaboratory, Berkeley, CA (US), 2007.

[54] RD Smith and PR Gent. Reference manual for the Parallel Ocean Program (POP), ocean component of the Community Climate System Model (CCSM2. 0 and 3.0). Technical report, Technical Report LA-UR-02-2484, Los Alamos National Laboratory, Los Alamos., (2002).

[55] Guangyu Sun, Jenő Kürti, Péter Rajczy, Miklós Kertész, Jürgen Hafner, and Georg Kresse. Performance of the vienna ab initio simulation package (vasp) in chemical applications. *Journal of Molecular Structure-theochem*, 624:37–45, 2003.

[56] Osamu Tatebe, Shukuko Moriwake, and Yoshihiro Oyama. Gfarm/bb — gfarm file system for node-local burst buffer. *Journal of Computer Science and Technology*, 35:61–71, 2020.

[57] Osamu Tatebe, Kazuki Obata, Kohei Hiraga, and Hiroki Ohtsuji. Chfs: Parallel consistent hashing file system for node-local persistent memory. *International Conference on High Performance Computing in Asia-Pacific Region*, 2022.

[58] Marat Valiev, Eric J. Bylaska, Niranjan Govind, Karol Kowalski, Tjerk P. Straatsma, Hubertus Van Dam, D. Wang, Jarek Nieplocha, Edoardo Aprá, Theresa L. Windus, and Wibe A. de Jong. Nwchem: A comprehensive and scalable open-source solution for large scale molecular simulations. *Comput. Phys. Commun.*, 181:1477–1489, 2010.

[59] Marc-André Vef, Nafiseh Moti, Tim Süß, Tommaso Tocci, Ramon Nou, Alberto Miranda, Toni Cortes, and André Brinkmann. Gekkofs-a temporary distributed file system for hpc applications. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 319–324. IEEE, 2018.

[60] Chen Wang, Kathryn Mohror, and Marc Snir. File system semantics requirements of hpc applications. In *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*, pages 19–30, 2021.

[61] Teng Wang, W Yu, K Sato, A Moody, and K Mohror. Burstfs: A distributed burst buffer file system for scientific applications. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2016.

[62] Yulei Wang, Jian Liu, Hong Qin, Zhi Yu, and Yicun Yao. The accurate particle tracer code. *Computer Physics Communications*, 2017.

[63] Junjie Wu, Yong Liu, Baida Zhang, Xianmin Jin, Yang Wang, Huiquan Wang, and Xuejun Yang. A benchmark test of boson sampling on tianhe-2 supercomputer. *National Science Review*, 5(5):715–720, 2018.

[64] Kesheng Wu, Surendra Byna, and Bin Dong. Vpic io utilities. Technical report, Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), 2018.

[65] Jianyuan Xiao, Junshi Chen, Jiangshan Zheng, Hong An, Shenghong Huang, et al. Symplectic structure-preserving particle-in-cell whole-volume simulation of tokamak plasmas to 111.3 trillion particles and 25.7 billion grids. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–13, 2021.

[66] Weixia Xu, Yutong Lu, Qiong Li, Enqiang Zhou, Zhenlong Song, Yong Dong, Wei Zhang, Dengping Wei, Xiaoming Zhang, Haitao Chen, et al. Hybrid hierarchy storage system in milkyway-2 supercomputer. *Frontiers of Computer Science*, 8(3):367–377, 2014.

[67] Bin Yang, Xu Ji, Xiaosong Ma, Xiyang Wang, Tianyu Zhang, Xiupeng Zhu, Nosayba El-Sayed, Haidong Lan, Yibo Yang, Jidong Zhai, et al. End-to-end {I/O} monitoring on a leading supercomputer. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 379–394, 2019.

[68] Bin Yang, Yanliang Zou, Weiguo Liu, and Wei Xue. An end-to-end and adaptive i/o optimization tool for modern hpc storage systems. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1294–1304. IEEE, 2022.

[69] Hisashi Yashiro, Koji Terasaki, Yuta Kawai, Shuhei Kudo, Takemasa Miyoshi, Toshiyuki Imamura, Kazuo Minami, Masuo Nakano, Chihiro Kodama, Masaki Satoh, et al. The nicam 3.5 km-1024 ensemble simulation: Performance optimization and scalability of nicam-letkf on supercomputer fugaku. In *EGU General Assembly Conference Abstracts*, pages EGU21–4771, 2021.

[70] Y. Ye, Z. Song, S. Zhou, Y. Liu, Q. Shu, B. Wang, W. Liu, F. Qiao, and L. Wang. swnemo_v4.0: an ocean model based on nemo4 for the new-generation sunway supercomputer. *Geoscientific Model Development*, 15(14):5739–5756, 2022.

[71] Orcun Yildiz, Matthieu Dorier, Shadi Ibrahim, Rob Ross, and Gabriel Antoniu. On the root causes of cross-application i/o interference in hpc storage systems. pages 750–759, 05 2016.

[72] Hao Yu, Ramendra K Sahoo, C Howson, George Almasi, José G Castanos, Manish Gupta, José E Moreira, Jeffrey J Parker, TE Engelsiepen, Robert B Ross, et al. High performance file i/o for the blue gene/l supercomputer. In *The Twelfth International Symposium on High-Performance Computer Architecture, 2006.*, pages 187–196. IEEE, 2006.