From Islands to Archipelago: Towards Collaborative and Adaptive Burst Buffer for HPC Systems

Mingtian Shao
College of Computer Science and Technology, National University of Defense Technology
Changsha, Hunan, China
shaomt@nudt.edu.cn

Ruibo Wang
College of Computer Science and Technology, National University of Defense Technology
Changsha, Hunan, China
ruibo@nudt.edu.cn

Wenzhe Zhang
College of Computer Science and Technology, National University of Defense Technology
Changsha, Hunan, China
zhangwenzhe@nudt.edu.cn

Kai Lu
College of Computer Science and Technology, National University of Defense Technology
Changsha, Hunan, China
kailu@nudt.edu.cn

Yiqin Dai
College of Computer Science and Technology, National University of Defense Technology
Changsha, Hunan, China
daiyq@nudt.edu.cn

Huijun Wu
College of Computer Science and Technology, National University of Defense Technology
Changsha, Hunan, China
wuhuijun@nudt.edu.cn

Abstract

Modern supercomputers increasingly use node-local storage as burst buffers (BB) to address I/O bottlenecks. However, current BBs do not naturally support workflows, a common workload in HPC consisting of many interconnected subtasks. Workflow I/O can be divided into three types: intra-task I/O, inter-task I/O, and stage-in/out I/O. While BBs can accelerate intra-task I/O, they often overlook the other two. Inter-task I/O relies on migrating data through the Parallel File System (PFS), which can slow down overall performance. Although allocating more resources to create larger BBs could help, it increases costs. Additionally, temporary BBs lack permanent storage, requiring data migration between the PFS and BB for stage-in and stage-out I/O. This process often involves multiple data copies and reduces I/O efficiency. Even for intra-task I/O, unbalanced data distribution can cause bottlenecks on heavily loaded nodes.

To improve workflow acceleration in BB systems, it is important to address the needs of all the above-mentioned three types of I/O. This paper presents CodepFS, a new collaborative and adaptive burst buffer. CodepFS uses a file pattern-aware data distribution strategy to improve data balance and boost intra-task I/O efficiency. It integrates smoothly with PFS, allowing fast stage-in/out I/O without extra data copies. For inter-task I/O, CodepFS introduces a technique called Inheritance-Fusion Namespace (IFN), which merges separate burst buffer instances into a unified system, thereby avoiding large-scale data migration during task coordination. Extensive evaluations show that CodepFS outperforms existing shared burst buffer technologies like UnifyFS and GekkoFS.

CCS Concepts

• **Information systems** → **Distributed storage**.

Keywords

Parallel File Systems, Burst Buffer (BB), High-Performance Computing (HPC), Workflow

ACM Reference Format:
Mingtian Shao, Ruibo Wang, Wenzhe Zhang, Kai Lu, Yiqin Dai, and Huijun Wu. 2025. From Islands to Archipelago: Towards Collaborative and Adaptive Burst Buffer for HPC Systems. In *2025 International Conference on Supercomputing (ICS '25), June 08–11, 2025, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3721145.3729515

[†]Corresponding author: Huijun Wu (wuhuijun@nudt.edu.cn).

ICS '25, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-1537-2/25/06
https://doi.org/10.1145/3721145.3729515

1 Introduction

Despite the slowdown of Moore's Law, the performance improvement of high-performance computing (HPC) systems has far outpaced that of storage systems, driven by advances in heterogeneous architectures and multi-chip packaging

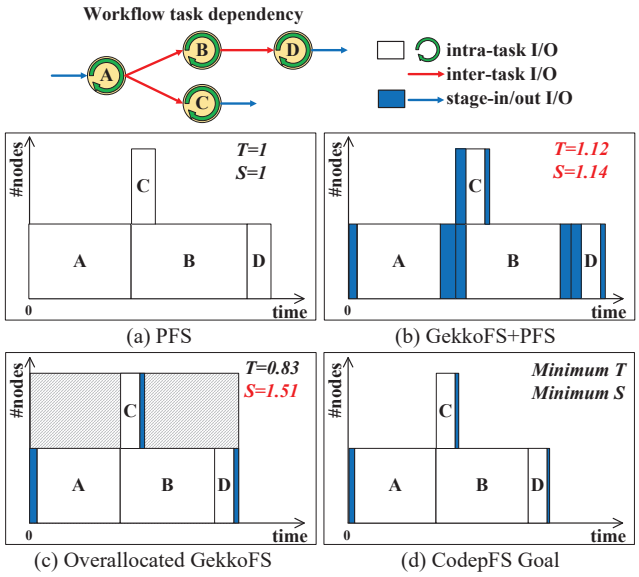


Figure 1: workflow with the existing BBs.

technologies [23]. In the face of today's exascale supercomputers with node counts reaching tens of thousands, traditional Parallel File Systems (PFS) struggle to scale, exacerbating performance bottlenecks. Burst buffer technology[17, 22] utilizes SSD or NVRAM resources located on computing nodes or at I/O forwarding nodes[5, 34] to handle bursty I/O and enhance I/O performance.

However, as one of the key workloads in HPC, workflow applications currently lack adequate support from BBs. Workflows consist of numerous subtasks that have data dependencies, which can be represented as directed acyclic graphs (DAG). These subtasks follow a specific execution order dictated by their dependencies, necessitating data sharing not just within the nodes of a single task, but also across different tasks within the workflow. Typically, the tasks in workflows are I/O-sensitive. To illustrate the limitations of the current BBs, we have classified the workflow I/O into three distinct stages (as shown in Figure 1): intra-task I/O, inter-task I/O, and stage-in/out I/O. Figure 1(a) shows how the PFS-based workflow works, which is the most common way. We used two metrics for workflow, execution time (represented by T) and billing (represented by area S, i.e. resources multiplied by time). We use this running pattern in Figure 1(a) as a baseline to show how current BB performs in workflow tasks.

Global BBs like DataWarp [18, 19], which is built on dedicated storage servers, typically exhibit strong data sharing capabilities but suffer from poor performance, particularly under significant competition effects. Existing BB systems deployed on computing nodes can be classified into node-local BB and shared BB, depending on the establishment of a unified namespace. Local BBs restrict node storage usage to the node itself, posing challenges in supporting file sharing among computing nodes. Conversely, shared BBs aggregate storage resources from multiple nodes to offer a unified namespace. That being said, existing shared BB research focuses on utilizing techniques like local writes and client-side caching to accelerate I/O performance. Recent studies have extensively explored various shared BBs, such as GekkoFS[29, 30], UnifyFS[8], DAOS[15, 21], and HadaFS[12]. These systems aim to provide a scalable and unified approach to BB management. However, despite the potential benefits and extensive explorations for shared BB systems, they just focus on accelerating intra-task I/O, but ignore the inter-task I/O and stage-in/out I/O. Drawing from our extensive experience in operating Tianhe supercomputers, we find that for different I/O types of workflow, three fundamental concerns are so far overlooked by current BBs.

First, **existing BBs cannot provide inter-task I/O directly.** Due to the lack of data sharing capabilities between different BB instances, users are compelled to manually migrate data among these instances through PFS to achieve inter-task I/O (Figure 1(b)). This is an undesirable burden for users. The substantial overhead of data migration overshadows the benefits of high-speed I/O offered by BBs. To avoid data migration between tasks, users tend to over-allocate resources, running the entire workflow within a single large BB instance (Figure 1(c)). However, this approach leads to excessive resource costs and over-billing. In Figure 1, we visually compare the expenses of users (represented by area S) of existing BB technologies in workflow scenarios, while also demonstrating how CodepFS effectively reduces the total charge and execution time (Figure 1(d)).

Second, **BBs often ignore efficient stage-in/out I/O in collaboration with PFS.** BBs, as a temporary file system, do not have persistent storage capacity, necessitating data stage-in I/O from PFS before tasks and data stage-out I/O to PFS after tasks. Existing shared BB systems, like GekkoFS, often neglect parallel data migration. Moreover, using the BB client's I/O path for data migration (e.g., UnifyFS) introduces overhead, such as metadata maintenance and multiple memory copies. Existing shared BBs are lack of coordinated design with the underlying PFS. Therefore, data stage-in/out I/O often introduces non-trivial overheads, which may obscure the I/O acceleration benefits of BBs.

Third, **BBs' inflexible and unbalanced data layout limits intra-task I/O performance.** Hash-based data distribution is the most widely used data distribution schema for existing shared BBs, such as GekkoFS and MadFS. Unfortunately, the hashing shows an imbalance, which causes I/O performance to suffer from a bottleneck on the most heavily loaded nodes.

To address the challenges mentioned above, this paper proposes CodepFS, a shared burst buffer system that utilizes the node-local storage resources of computing nodes. CodepFS seamlessly integrates with workflow applications, enabling file view merging across multiple BB instances employed in different workflow stages, thereby achieving inter-task I/O without extensive data movement. Meanwhile, CodepFS collaborates with PFS, achieving highly parallel data stage-in/out I/O, circumventing data transfer bottlenecks across storage hierarchies. Additionally, CodepFS introduces an adaptive file-pattern-aware data distribution mechanism to eliminate load unbalance caused by non-uniform hashing. This mechanism not only accelerates intra-task I/O performance, but also enhances BB's scalability.

The core innovation of CodepFS lies in its enhancement and expansion of shared BB's support for three I/O types of HPC workflows, effectively addressing the limitations and shortcomings of existing BBs. To the best of our knowledge, CodepFS is the first work on shared BB that natively supports workflow in the production system. CodepFS has been deployed on the Tianhe Exascale Prototype Upgrade System[31]. Contributions in this paper are as follows:

- This paper, to the best of our knowledge, is the first to point out the different I/O types according to the stages in the HPC workflow. We then highlight the shortcomings of existing shared BB designs for these I/O types.
- This paper proposes CodepFS, a shared burst buffer that achieves enhanced support for HPC workflows and minimizes both running time and user costs. This paper delves into the design and implementation of CodepFS.
- Experimental results demonstrate that CodepFS outperforms the state-of-the-art shared BBs under different workloads, in terms of intra-task I/O, inter-task I/O and stage-in/out I/O.

The rest of the paper is organized as follows. Section 2 presents the motivation. We introduce the design and implementation of CodepFS in Section 3. Section 4 presents the evaluation methodology and results. Section 5 discusses the related work. Conclusions are provided in Section 6.

2 Motivations

This paper focuses on using shared BBs that utilize node-local storage resources to support workflows, and delves into three key factors that hinder the practical implementation of shared BBs. These include: (1) Lack of data sharing mechanism between different tasks within prevalent workflow application models. (2) Inefficient data migration between shared BBs and global PFS. (3) The load unbalance caused by uneven data distribution.

2.1 Data sharing in workflow models

Current temporary BBs lack native support for data sharing between workflow tasks, forcing manual data migration to achieve inter-task I/O, which introduces a significant bottleneck. HPC workflows, vital in fields like materials science and biomedicine, often involve a lot of interdependent tasks generating and transferring large datasets. These workflows typically take the form of directed acyclic graphs (DAG), where the nodes represent tasks, and the edges represent dependencies [9].

While different workflow tasks might run on separate groups of nodes, they can all access data stored in the PFS. However, existing BBs struggle to support workflows due to the challenge of sharing data across them. Since each BB serves a single task on a specific set of nodes, directly accessing data between dependent tasks becomes impossible. Consequently, workflows rely on costly intermediate data transfers through the PFS. This inefficiency becomes particularly problematic for large intermediate data commonly found in workflows. To address this, shared BBs should support direct data sharing among different tasks within the workflow to massively accelerate overall I/O efficiency.

2.2 Data stage-in/out

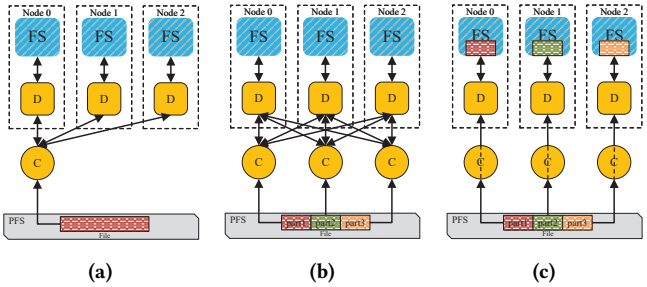


Figure 2: Data staging between current temporary BBs and PFS.

Temporary BBs achieve persistent storage via PFS like Lustre [25, 33]. Efficient data movement is crucial, involving pre-loading task data (stage-in I/O) and returning results (stage-out I/O). However, achieving efficient data staging is non-trivial. Effective data staging goes beyond simply copying data in parallel to the BB system. Figure 2 illustrates three methods for staging-in/out data between the temporary BBs and the PFS.

Figure 2(a) depicts a data migration approach where a single client maintains full connectivity to all daemons. While this method is straightforward, it suffers from a performance bottleneck due to its reliance on a single client, and it necessitates data redistribution between the client and the daemons.

The data redistribution process involves transferring a substantial number of remote procedure calls (RPCs) and discrete data blocks, which significantly hampers the efficiency of the migration. Notable examples of this approach include GekkoFS and typical DataWarp.

In Figure 2(b), the number of clients is increased to improve performance; however, it still introduces the challenge of data redistribution between clients and daemons. Typical examples include our GekkoFS parallel copy tool (Section 4.2) and DataWarp which employs methods such as mpiFileUtils[20].

Figure 2(c) presents a stage-in approach that can bypass the client and eliminate the need for data redistribution. Nevertheless, this approach often overlooks a critical aspect. The ultimate goal of data stage-in is to serve subsequent data read requests. Consider UnifyFS and HadaFS as an example: importing data directly without prior knowledge of future access patterns would trigger cross-node read requests, which are already detrimental to performance.

Efficient data staging solutions should be closely integrated with the underlying PFS to reduce unnecessary data copying and mitigate the adverse impact of inconsistent data distribution on subsequent read performance.

2.3 Data layout

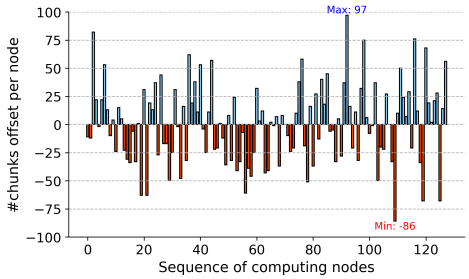


Figure 3: Unbalanced data distribution of GekkoFS with 128 nodes. The average number of chunks per node is 1024.

BBs can be categorized as stateful or stateless based on data placement strategy. Stateful BBs like BeeOND[13] offer configuration options but introduce scalability bottlenecks and user complexity due to reliance on a central service. Stateless approaches used in systems like GekkoFS and HadaFS leverage hash mapping, but this can lead to unbalanced data distribution and long-tail latency for frequently accessed data. To illustrate this data imbalance, we evaluated the distribution of GekkoFS chunks across nodes. As shown in Figure 3, this distribution exhibits variations of up to 17.9% between nodes. Additionally, all BBs require Remote Procedure Call (RPC) requests and Remote Direct Memory Access (RDMA) operations to access data [16], which can introduce significant overhead for small files, negating the benefits of a distributed system. Conversely, BurstFS[32] and UnifyFS prioritize write performance by enabling direct local writes, but this approach can exacerbate capacity and load imbalances.

In summary, current BB data layout lacks flexibility to accelerate intra-task I/O. Although stateless approaches offer better scalability, they can lead to an unbalanced data distribution. This paper proposes a new approach that recognizes the differing needs of large and small files. Large files should be strategically placed to ensure balanced access across nodes. In contrast, smaller file access should be optimized for speed, minimizing request overhead.

3 Design and Implementation

3.1 Overview of CodepFS

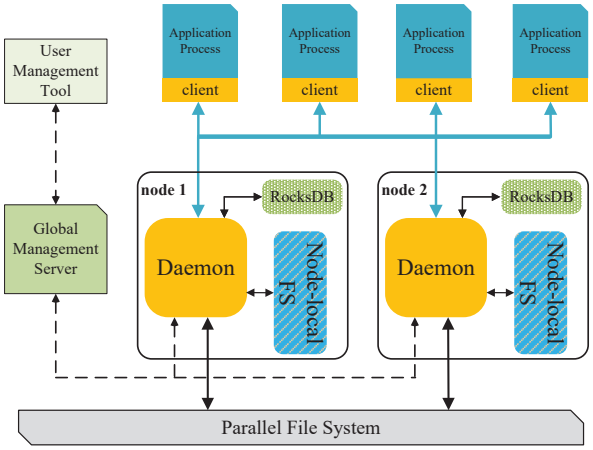


Figure 4: Architecture of CodepFS. Consider an example of temporarily aggregating two computing nodes.

CodepFS employs a client-server architecture that features a service process on each computing node. These service processes act as servers, while tasks running on the nodes become access the BB service through a dynamic link library (DLL). The service process aggregates the available storage capacity of the local file system on its node, presenting a unified view to the client. Metadata, which describe files, are distributed between nodes using RocksDB, a proven LSM-tree-based key-value store[10]. The CodepFS client intercepts system calls related to file operations initiated by tasks. Based on the mount point, the client dynamically decides whether to redirect the call to the CodepFS server. Data storage locations within CodepFS are determined by applying a hash function to the file's absolute path.

To accelerate intra-task I/O, CodepFS introduces a dynamic adaptive data distribution strategy. This strategy leverages a hybrid approach for large files, combining hashing together with contiguous placement to optimize access. For small files, a straightforward hashing approach suffices. This method transparently achieves efficient storage for all file sizes.

CodepFS streamlines data movement between itself and the underlying PFS via a dedicated data migration interface to optimize stage-in/out I/O. Leveraging the predefined data distribution strategy, each CodepFS daemon process independently retrieves its designated data blocks, eliminating the need for BB client intervention. Furthermore, CodepFS implements transparent directory mapping. This feature allows users to interact with data directly through existing PFS paths, concealing the underlying CodepFS infrastructure.

A concept called Inheritance-Fusion Namespace (IFN) is introduced in CodepFS to enhance data collaboration within workflows. These intelligent namespaces analyze the data dependencies between tasks and allow CodepFS to merge namespaces seamlessly, granting tasks direct access to previously generated data of its dependent tasks. By eliminating data transfer and duplication, IFN ensures effortless data sharing (inter-task I/O) across tasks.

3.2 Adaptive data layout

The commonly used data placement strategy in BB systems based on node-local storage involves partitioning files into multiple chunks. This approach is widely employed in systems such as GekkoFS, MadFS, and HadaFS. However, such systems tend to uniformly divide files of different sizes into chunks. Due to the inherent randomness in data distribution, hashing algorithms exhibit an inevitable imbalance. Therefore, data partition approaches of existing BBs struggle to achieve a truly balanced distribution of data. This not only leads to uneven storage capacity usage among nodes but also results in increased latency for I/O requests, thereby impacting the availability and performance of the system.

While small files (fitting within a single block) avoid the imbalance issues associated with hashing, they introduce higher transmission costs. Reading and writing files in shared BBs generally involves a client request to the target node's daemon followed by RDMA data transfer. However, for small file access, the dominant time factor becomes the round-trip time (RTT) per Remote Procedure Call (RPC) rather than the actual data transmission. This is because the data size is minimal compared to the overhead associated with the RPC call itself.

To address the above challenges, CodepFS employs a differentiated data placement strategy. Files are categorized as "small" (fitting one block) or "fat" (exceeding one block). File metadata includes a boolean `fat_file` flag to determine access methods based on file size efficiently.

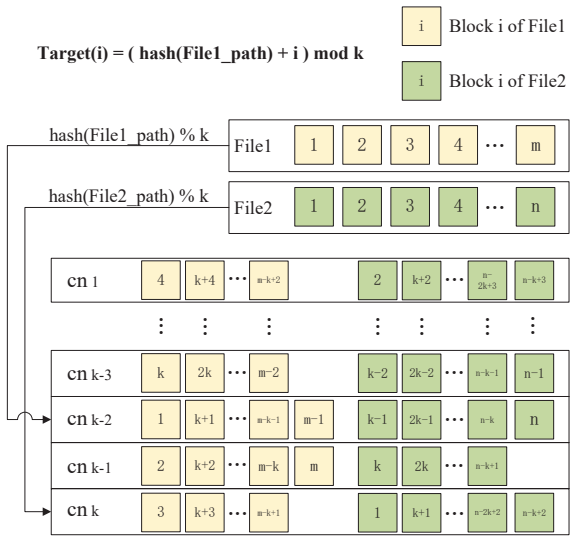


Figure 5: Data layout of fat files.

3.2.1 fat file. For "fat files" (larger than one block), CodepFS employs a data placement strategy that prioritizes balanced storage across computing nodes. As shown in Figure 5, the absolute file path is hashed to determine a starting node for data block placement. Subsequent blocks are then distributed sequentially across participating nodes following a hash ring scheme. It means that CodepFS only computes the hash value of the absolute file path as the location of the first data block, rather than hashing each data block independently. This approach guarantees a uniform distribution of the fat file across all nodes, maintaining a maximum load difference of only one data block among them. If the number of data blocks is exactly an integer multiple of the number of nodes, then perfect load balancing will be achieved. In other cases, there is only a load difference of one data block among all nodes within the CodepFS instance. This data distribution effectively avoids the huge load imbalance in Figure 3, thereby optimizing file-level I/O concurrency and enhancing overall application I/O performance.

While the first block placement might vary for individual files, the sequential nature balances storage across nodes for multiple files. This minimizes load imbalance in storage utilization and mitigates potential long-tail delays that could impact parallel computing efficiency.

Reading and writing fat files in CodepFS leverages a two-step approach to optimize efficiency. First, the client calculates target nodes and data blocks based on file path, offset, and data size. In a CodepFS instance, all nodes are organized

into a ring structure of data blocks. The starting point of each file's data blocks on this ring is determined by the hash value of the file path. As illustrated in Figure 5, a CodepFS instance consists of a total of k computing nodes (denoted as *cn* in Figure 5). For instance, File 1 is composed of m data blocks. The location of each data block is computed using the hash value of the file path combined with the block number, followed by taking the remainder when divided by k. The block number for a data block can be calculated based on the offset and the block size.

The design of the fat file layout not only achieves a well-balanced data distribution but also reduces computational overhead within the I/O stack. In the existing BB, The block number and file path are used together as input to the hash function to compute the storage target (such as GekkoFS), and the number of hash operations is equal to the number of accessed data blocks. In contrast, CodepFS only needs one hash operation to determine the location of the data block, that is, only the hash value of the absolute path of the file is calculated. In other words, CodepFS reduces a large number of hash operations to one, which will effectively reduce the amount of computation on the I/O stack, especially for large file accesses.

Crucially, CodepFS combines multiple block requests destined for the same node into a single Remote Procedure Call (RPC), minimizing Round-Trip Time (RTT) overhead. Once the target node receives the combined RPC, it utilizes RDMA (Remote Direct Memory Access) for efficient data transfer, bypassing the software stack. This combined approach significantly reduces communication overhead compared to per-block requests, improving overall performance for large file access.

3.2.2 small file. CodepFS optimizes storage and access for small files by co-locating their object data and metadata within RocksDB. Each file's metadata resides as a single record, containing a `fat_file` flag set to false to indicate a small file. The actual object data is stored in the `obj_data` field. This co-location strategy simplifies access by requiring only one RPC for both reading and writing, minimizing communication overhead compared to separate data and metadata retrieval.

CodepFS optimizes small file access through co-located storage of object data and metadata in RocksDB. This approach minimizes communication overhead. For writes, the client combines updated metadata and object data into a single RPC request sent directly to the target node. The receiving daemon then updates the corresponding record in RocksDB. Reads are even faster: upon receiving an open request, the target node returns the entire record, including both metadata and object data, to the client. The client associates the file descriptor with a memory pointer storing the received data. Subsequent read system calls can directly access this local memory, eliminating the need for further RPCs, RDMA, or network transfers. By utilizing co-location and the inherent need to access metadata before data, CodepFS significantly accelerates small file access.

3.2.3 file volume boundaries crossing. CodepFS caters to various file sizes by employing distinct storage strategies for small and large files. To ensure efficient handling of file size boundaries, CodepFS utilizes the same hashing strategy on the absolute file path for both the metadata and the first data block of "fat files" (larger than one block). This ensures the first block resides on the same node as the metadata. Similarly, for small files co-located with their metadata in RocksDB, their data can be considered the "first block," again simplifying boundary handling. By leveraging these differentiated storage strategies, CodepFS effectively manages file size spans, enhancing its overall usability across diverse file size scenarios.

Small file → fat file: CodepFS efficiently handles small file growth into "fat files" by utilizing co-located storage of metadata and data. When a small file requires more space, it simply transfers the existing object data from the metadata's `obj_data` field to the designated first data block on the same node and flip the `fat_file` flag. Since all these operations occur within a single node, no data transfer across the network is needed, simplifying the process and minimizing overhead.

Fat file → small file: CodepFS efficiently handles both file growth and shrinkage. Shrinking a fat file involves storing the reduced data directly in the metadata and flipping the `fat_file` flag , eliminating the need for separate data blocks. Likewise, truncating large files only requires localized data movement within the first block. These optimizations minimize network overhead for file size changes.

3.3 Global Management Server and directory mapping

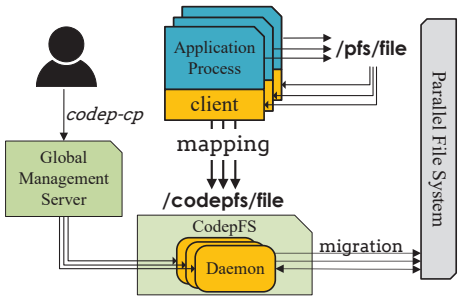


Figure 6: Global Management Server for parallel stage-in/out and directory mapping.

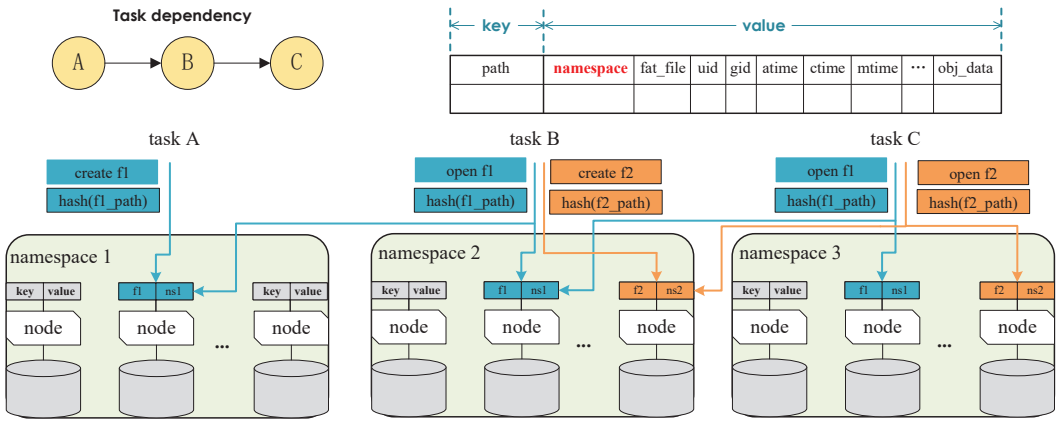


Figure 7: Inheritance-Fusion Namespace for workflow tasks.

BBs primarily serve as temporary file storage, necessitating data staging in/out from/to PFS before/after computational tasks. CodepFS simplifies data movement between PFS and itself with a user-friendly *codep-cp* command, mirroring the familiar *cp*. This tool enables efficient data staging from PFS to CodepFS before jobs and result transfer back to PFS after completion. Behind the scenes, a global data management service on the login node orchestrates these migrations. Users specify source and destination locations, and the service directs CodepFS daemons to migrate data block-by-block based on file paths. This eliminates complex data movement for users and optimizes performance.

While existing systems like HadaFS and UnifyFS offer data migration between PFS and other file systems, they often require manual user intervention for each task. What's worse, using the BB client's I/O path for data migration (e.g., UnifyFS) introduce overhead for metadata maintenance and memory copies. Through the *codep-cp* command, we achieve ad-hoc data movement and transparent directory mapping. By configuring directory mappings (`PFS_dir`:`CodepFS_dir`) during startup, users can access PFS data migrated to `CodepFS_dir` seamlessly, eliminating the need for manual data movement commands altogether. This focus on user experience reduces the burden of data management compared to existing solutions.

CodepFS's directory mapping simplifies data movement between PFS and itself. When data resides in a pre-configured PFS directory (`PFS_dir`), the system automatically migrates it to the corresponding CodepFS location upon job launch, using the global management service. This ensures efficient data access through CodepFS. Similarly, writes destined for `PFS_dir` are first staged in `CodepFS_dir`. A "stage bit" in the metadata tracks files slated for migration. Upon CodepFS shutdown, the system checks this bit and migrates any flagged data back to `PFS_dir`. This automation eliminates the need for manual data movement or path modifications in application code or job parameters. Users simply define directory mapping to seamlessly leverage CodepFS's high-speed data access while maintaining data consistency with PFS.

Note that the CodepFS daemon leverages the hybrid hash and sequential layout algorithm to efficiently determine and fetch required data blocks, consolidating them into a single RDMA transfer. This approach aligns data staging with how clients will subsequently access the data, avoiding the need for additional data redistribution while simply adopting parallel data copying. CodepFS circumvents the BB client by enabling direct data retrieval from PFS by the file system daemons. This approach facilitates fully parallel data staging, maximizing the utilization of PFS bandwidth and minimizing BB-related data migration overhead. Overall, the CodepFS daemon simplifies data management by automating the process, reducing errors, and optimizing data transfer for efficient data access.

3.4 Inheritance-Fusion Namespace

In modern High Performance Computing (HPC) systems, workflow jobs are a crucial component. These jobs consist of multiple interconnected tasks, often relying on large datasets. However, current shared BB file systems, particularly those aggregating node-local storage, are not optimized for direct workflow job support. This necessitates manual data movement between BBs or between BBs and the PFS, leading to a significant burden on users and hindering overall efficiency.

CodepFS supports workflow applications with Inheritance-Fusion Namespace (IFN) that grant tasks access to other BB instance namespaces. This capability effectively merges file views across multiple BB instances.

CodepFS manages data access through namespaces, one per BB instance. Clients within a namespace can access all data managed by that instance's daemon. The "namespace"

element in the metadata store tracks the relevant namespace. By default, it points to the local namespace. Workflow tasks efficiently access data using IFN. Tasks primarily scan their own and directly dependent tasks' namespaces (Figure 7). When a task needs data not in its local namespace, it queries the dependent task's namespace. If found, the metadata is cached locally for future access, streamlining data retrieval for dependent tasks.

For tasks in the presence of indirect dependencies, such as task C and task A. When task C accesses file `f1`, the metadata cannot be found in the local namespace, and then it looks up in the namespace of the directly related preorder task B. Since the metadata of `f1` is stored in namespace of task B, and the `namespace` entry in it points to task A. Task C stores this metadata into the local namespace and accesses `f1`'s object data in the namespace of the task A.

CodepFS supports data sharing among tasks within workflows, with temporary file systems being removed after workflow completion. IFN ensures optimal storage efficiency by storing object data of files once, reducing space costs. Metadata is stored in separate namespaces to enhance access performance without significant space overhead. For efficient metadata querying, IFN only need to query a maximum of `N` target nodes based on the number of dependencies, with concurrent querying. Additionally, CodepFS employs Bloom filters [27] on daemons to swiftly identify file locations, bypassing irrelevant namespaces and accelerating the process.

CodepFS collaborates with the resource manager to optimize resource allocation. Even while serving I/O requests for subsequent tasks, the computing nodes from completed tasks can be repurposed for non-BB-accessing tasks, such as compute-intensive tasks and PFS-only tasks. The resource manager divides nodes into compute-idle nodes (still providing CodepFS I/O with only 1 daemon occupying 1 core) and entire-idle nodes. This maximizes node utilization without compromising data security. CodepFS and the resource manager meticulously manage file system permissions to ensure only authorized tasks have access to data.

4 Evaluation

We carried out the evaluation on the Tianhe Exascale Prototype Upgrade System, which was ranked first in the Graph500 SSSP [1] and Green Graph500[2] in June 2021. In our evaluation, a total of 512 computing nodes were used. The high-performance interconnection network of the Tianhe HPC system is TH-Express [24]. All computing nodes have access to the same shared storage using the Lustre file system.

We compare the performance of CodepFS with GekkoFS, BeeOND, UnifyFS, HadaFS-style and PFS (Lustre). For the fairness of the evaluation, We simply modified BeeOND and UnifyFS to make them better adapted to TH-express. Since HadaFS is closed source, to evaluate CodepFS performance relative to HadaFS, we implemented modifications inspired by HadaFS logic within CodepFS. This allowed us to simulate an HadaFS-like environment for our benchmarks. GekkoFS, BeeOND, UnifyFS and HadaFS-style use the same number of daemons and clients as CodepFS in each test. Experiments are designed to verify the effectiveness of CodepFS's design. We used IOR[26] benchmark to compare the I/O bandwidth. The ability of data migration is also tested. Finally, experiments are designed to highlight the powerful support of CodepFS for workflow. To reduce the chance error of the experiment, all results reported in this section represent the average of 10 experiment repetitions.

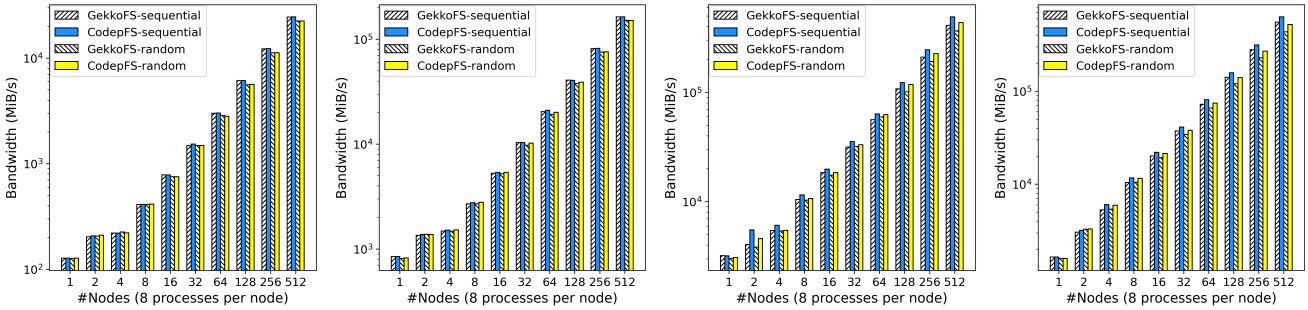
4.1 Data performance evaluation

The adaptive data layout strategy of CodepFS is treated differently for large and small files. When we tested the data access performance of CodepFS, we also tested the performance of regular files (fat files) and small files separately.

We employed the IOR benchmark to evaluate the bandwidth of CodepFS for large files (fat files). We compared its performance against GekkoFS using various transfer sizes: 8KB, 64KB, 1MB, and 32MB. Additionally, we tested both random and sequential ordering within the files.To eliminate the influence of client-side caching, we utilized the *-C* parameter of IOR to control task ordering during read-back. Furthermore, the *-F* parameter ensured each process accessed a single file. Each process read a total of 2GB of data.The evaluation spanned various cluster sizes, ranging from 1 to 512 nodes. Each node executed eight IOR processes concurrently.

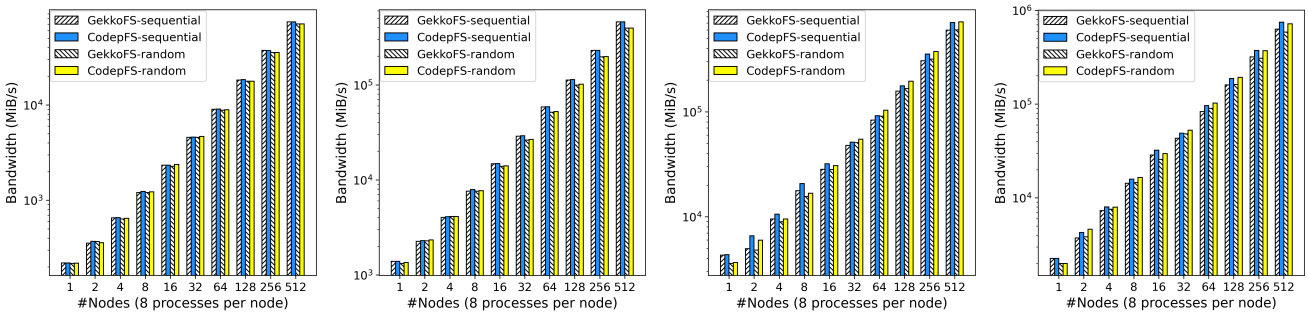
Figure 8 presents the test results for IOR write bandwidth. CodepFS and GekkoFS perform similarly when transferSize is 8KB and 64KB. This is because a small transfer size introduces a large number of RPCs, the actual data transfer time takes a small proportion, and the advantage of load balancing is weakened. As the transfer size increases, as shown in Figure 8(c) and Figure 8(d), the advantage of CodepFS highlights. At this time, the data transmission time has become dominant, and the balanced data distribution of CodepFS effectively avoids the long-tail delay problem in parallel programs.

Our experiments reveal that CodepFS exhibits superior performance as the number of computing nodes increases. And, in this trend, the increase of CodepFS's performance is almost linear. This demonstrates the excellent performance and scalability of CodepFS. This advantage likely stems from potential storage imbalances within GekkoFS at larger scales. Nodes holding more data experience heavier access pressure, creating a bottleneck for the entire computation. CodepFS,



(a) transferSize: 8KB (b) transferSize: 64KB (c) transferSize: 1MB (d) transferSize: 32MB

Figure 8: IOR write bandwidth for each process operating on its own file.



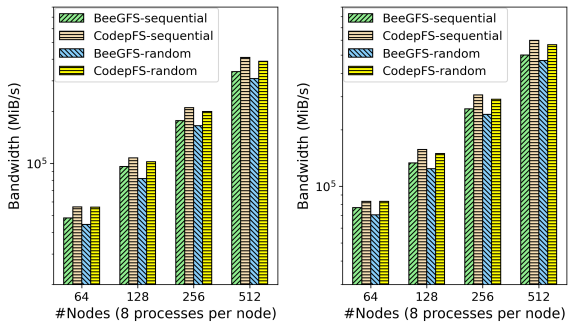
(a) transferSize: 8KB (b) transferSize: 64KB (c) transferSize: 1MB (d) transferSize: 32MB

Figure 9: IOR read bandwidth for each process operating on its own file. Tasks are reordered for readback to avoid client cache.

on the other hand, achieves a balanced distribution of storage pressure across all nodes, effectively mitigating long-tail latency in data access. This is evident in the 512-node test with a 32MB transfer size, where CodepFS delivers bandwidth 1.2× faster than GekkoFS. Figure 9 presents the read bandwidth test results of IOR. The overall trend is consistent with the write bandwidth.

We also compared the performance of CodepFS with BeeOND. The transfer size was set to 1MB, and random and sequential reads and writes were tested using different node scales. CodepFS and BeeOND used the same number of daemons and clients at each scale.

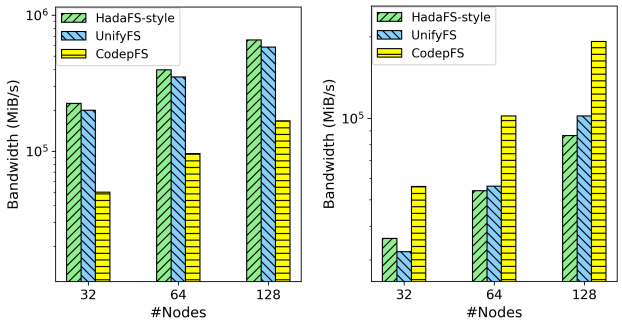
Figure 10 shows that CodepFS performances better under all settings. The speedup of CodepFS is more pronounced as the node size increases. The acceleration ratio of random reads and writes is higher than that of sequential reads and writes. Especially in the test of 512 nodes, CodepFS's sequential write performance is 1.21×, and sequential read is 1.19× of BeeOND. CodepFS's random write performance is 1.26×, and random read is 1.21× of BeeOND. Due to the stateless design, CodepFS is significantly more scalable than BeeOND.



(a) write bandwidth (b) read bandwidth

Figure 10: IOR bandwidth for each process operating on its own file compared with BeeGFS's BeeOND.

UnifyFS and HadaFS significantly reduce I/O overhead and latency for improved write performance by direct local writes. However, the read performance of the process on the

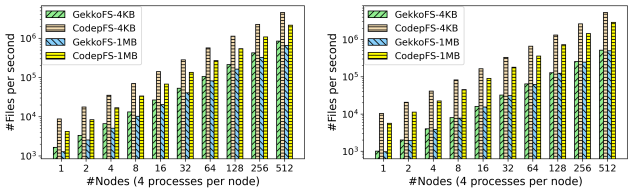


(a) write bandwidth (b) read bandwidth

Figure 11: Bandwidth of reordered IOR tasks for single shared file compared with UnifyFS and HadaFS-style.

other nodes is hurt inevitably. In this case, we compare the performance of CodepFS with UnifyFS and HadaFS-style.

Figure 11 shows the test results. UnifyFS and HadaFS-like configurations demonstrate a clear advantage in write performance as local writes bypass network overhead. However, CodepFS outperforms them significantly in reordered reads, which are common in many parallel applications, such as sparse matrix computations and graph algorithms. In the 128-node test, CodepFS delivers read performance 1.9 times faster than UnifyFS and 2.2 times faster than HadaFS-like. The huge read-write performance difference makes the application scenarios of UnifyFS and HadaFS very limited. In contrast, the balanced data distribution mechanism of CodepFS ensures scalability for both read and write operations, thereby facilitating the effective handling of a wide range of applications.



(a) write throughput (b) read throughput

Figure 12: Small files throughput.

For the read and write performance test of small files, we designed different small file sizes of 4KB and 1MB. We use parallel programs to read and write small files. Each node runs eight processes. Each process writes or reads 10,000 small files. During the test, all files are written, and then all files are read using offset reading. The offset is 8 processes. This avoids the performance benefits of caching.

Figure 12(a) gives the throughput of small file writes. The ordinate is the number of files written per second. Experimental results show that CodepFS is significantly better than GekkoFS in small file write performance. For small 4KB files, the throughput of CodepFS is 5.3× of GekkoFS. For small files of 1MB, CodepFS has 3.3× the throughput of GekkoFS. The smaller the file size, the more obvious the speedup effect of CodepFS. Figure 12(b) presents the throughput test results for small file reads. For small 4KB files, the throughput of CodepFS is 10.3× of GekkoFS. The throughput of CodepFS is 5.8× that of GekkoFS for small files of 1MB. Obviously, the performance benefit of CodepFS is more significant for small file reads, because there is only one RPC during the read, which is the open file. The write process requires two RPC, one for the open file and one for the write. For both reading and writing small files, CodepFS shows near-linear scalability.

These results of the data performance evaluation indicate that CodepFS's adaptive data layout is capable of accommodating a wide variety of file access patterns: sequential or random read and write, large or small files, a single shared file or a separate file accessed by each process, which almost covers the mainstream file access patterns in HPC.

4.2 Data migration evaluation

In this section, we tested the performance of migrating data into the BB. UnifyFS provides specialized tools to import data. CodepFS imports data using a special command called *codep-cp*. GekkoFS does not provide a tool for data import. We designed a parallel copy tool and loaded the intercept library of GekkoFS to realize parallel data stage-in. For a fair comparison, the number of concurrent processes of the tool is the same as the number of daemons in CodepFS.

Typically, the imported data will be read by tasks shortly after. In the process of data migration to BB, it is not possible to predict the I/O behavior of subsequent tasks. In contrast to UnifyFS, where read performance is sensitive to data placement, CodepFS can improve performance with adaptive data layout. So we tested the read performance after the data was imported. A total of 128 nodes were used for the test, and the amount of data migrated ranged from 128MB to 4TB.

Figure 13 shows the results of the test. The experimental results show that CodepFS has a massive advantage over GekkoFS and UnifyFS regarding data migration ability. When the data volume is 4TB, CodepFS is 6.2× faster than GekkoFS while 24.7× faster than UnifyFS.

For the read performance test after importing the data, CodepFS showed the best performance. UnifyFS has the lowest read bandwidth. Since UnifyFS imports data blindly and cannot predict the I/O mode of tasks, it has a natural disadvantage for read performance. When the data volume is

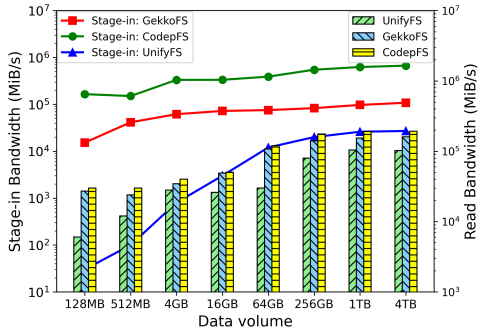


Figure 13: Stage-in and Read (afterwards) bandwidth.

4TB, the read-afterward bandwidth of CodepFS is 1.2× that of GekkoFS and 1.9× that of UnifyFS. In summary, CodepFS achieves the best performance in terms of both data migration and read-afterward performance.

4.3 Workflow evaluation

Table 1: Data migration in the PE workflow

FS	TaskA	TaskB, TaskC		TaskD	
	stage-out	stage-in	stage-out	stage-in	stage-out
CodepFS + dir_map	-	-	-	-	-
CodepFS + IFN	-	-	C_data	-	D_data
GekkoFS + local-FS	A_data	A_data	B_data C_data	B_data	D_data
GekkoFS + PFS	A_data	A_data	B_data C_data	B_data	D_data
PFS	-	-	-	-	-

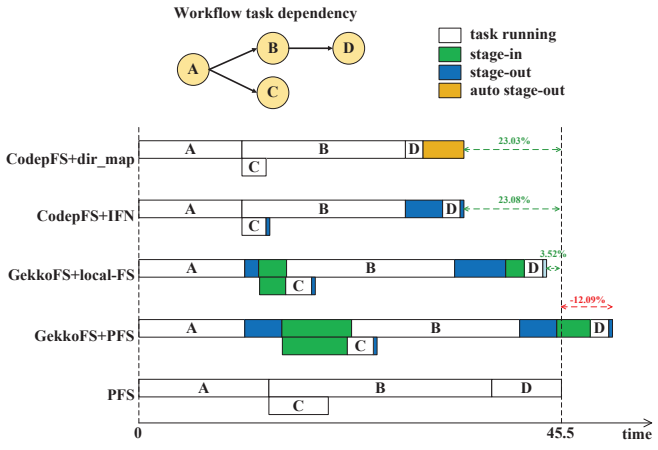


Figure 14: PE workfolw test with different BB/FS.

CodepFS supports data sharing among workflow tasks through IFN, which effectively avoids the overhead of data movement. Directory mapping allows users to use CodepFS to run workflows as if they were using PFS. To test the efficient support of CodepFS for workflows, we tested four different workflows separately.

First, we conducted tests using a typical petroleum exploitation (PE) simulation program running on the Tianhe HPC system. The workflow of the PE is shown at the top in Figure 14. PE only has four tasks A, B, C, and D. Task A generates data but does not read it. Tasks B, C, and D both read and generate data. Task A is responsible for generating geological information, which will generate more data and save it in file `A_data`. These data will be continued to be used by subsequent tasks. Task B analyzes the data generated by task A, extracts and calculates according to the data characteristics, and generates data such as oil distribution and probability `B_data`. Task C reads the data generated by task A, initially draws the information map `C_data`, and restores the geological scene in the field to achieve the purpose of visualization. Task D visualizes the data from task B, annotates the information such as oil distribution and probability, and saves it to the file `D_data`. The final data `C_data` and `D_data` need to be persisted. Each task uses 128 nodes with eight processes on each node.

Table 1 shows the various approaches to supporting workflows. The PFS-based approach is the most straightforward choice for users today. All tasks read and write data directly from/to PFS. It is difficult for GekkoFS to share data between tasks directly, and it must be relayed through an intermediate medium, so we choose PFS and node-local storage as the relay medium for testing.

CodepFS is designed to interact efficiently with PFS while providing full transparency through directory mapping to reduce the users' burden. Users do not need to perform explicit data migration between PFS and CodepFS to gain the I/O acceleration capabilities of CodepFS. In our workflow tests, we tested the benefits of CodepFS directory mapping. The CodepFS+IFN approach represents that the directory mapping is unused. In this case, the user needs to manually migrate the data that needs to be persisted to PFS via *codep-cp*.

The timeline in Figure 14 records the test results for the different file systems in this test workflow. The white rectangles represent when the task was running, the blue rectangles represent when the data was migrated to PFS, and the green rectangles represent when the data was migrated to BB. The blue and green rectangles need to be migrated manually by the user. The experimental results show that users do not need to perform any additional operations when running the workflow in PFS. However, the running time is longer. We use this time as the baseline. The total running time of

Table 2: workflow test of different I/O density

workflow	I/O density	#tasks	#edges	total nodes	total process	#shared_file	shared size(GB)	total time(s) and charge			
W1	**low**	60	191	960	7680	191	382.12	PFS	371.13	speedup	charge
								CodepFS	366.36	**1.29%**	**-1.42%**
W2	**medium**	80	1082	2560	20480	1509	17,602.42	PFS	2012.23	speedup	charge
								CodepFS	1691.7	**15.93%**	**-18.48%**
W3	**high**	80	1082	2560	20480	1509	105,614.71	PFS	4117.54	speedup	charge
								CodepFS	2681.48	**34.88%**	**-41.16%**

the whole workflow for GekkoFS using PFS as the staging medium increases by 12.09% due to the large amount of data migration required. GekkoFS, which uses node-local storage, is even better, with a 3.52% reduction in runtime for the entire workflow. However, both approaches require users to manually migrate data before and after the task runs.

CodepFS with IFN effectively eliminates the need to migrate data between workflow tasks. Users only need to migrate the output files to PFS, reducing the time spent on the entire workflow by 23.08%. With directory mapping, the user does not need to do any migration work, and the data is automatically migrated to PFS after the task is completed. The time required to complete the workflow in this way is almost the same as that of CodepFS+IFN. However, it is worth noting that this way greatly reduces the burden on the user.

We designed three additional workflows (shown in Table 2) for more comprehensive testing. Unlike the PE workflow, these three workflows are composed of many tasks, and the dependencies (count of edges in the DAG) are complex, making it difficult for users to manually migrate data between different BBs (such as GekkoFS). So, we only tested these three workflows running on PFS and CodepFS.

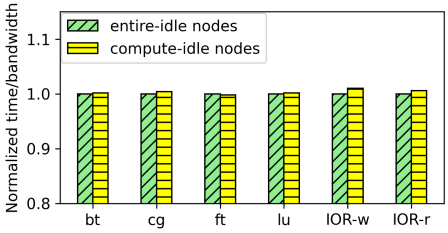


Figure 15: IFN overhead test.

The three workflows are low, medium, and high in I/O intensity, and the amount of shared data is also vastly different. Results show that CodepFS can support workflow efficiently. Directory mapping dramatically reduces the user's burden. The speedup effect of CodepFS increases as the I/O intensity rises. For the most I/O intensive workflow W3, CodepFS can achieve a 34.88% speedup of the total running time and reduce user cost by 41.16%.

We also tested the overhead of IFN. Using nodes that are free of compute resources but still provide CodepFS I/O, we run the compute-intensive benchmark NPB to test the compute overhead and IOR to test the overhead of PFS bandwidth. As shown in the Figure 15, IFN introduces a negligible overhead.

4.4 Evaluation with real-world applications

We compare CodepFS with Lustre using three real-world applications. HACC is an IO-performace benchmark. It uses a n-body simmulation to simulate collisionless fluids in space. HPIO[7] is a benchmark that tests reads, and writes to and from noncontiguous data layouts. S3aSim is a sequence similarity search algorithm framework for testing and evaluating various I/O strategies using MPI-IO. We run HACC on 128 computing nodes, each of which runs four processes. HPIO and S3aSim each occupy 64 computing nodes, with four processes on each node.

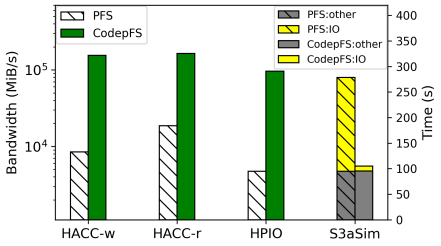


Figure 16: Evaluation with three applications.

The experimental results are shown in Fig 16. Where the HACC and HPIO metrics are bandwidth, corresponding to the left vertical axis in the figure. The metric of S3asim is the running time, which corresponds to the vertical axis on the right. The experimental results show that the write bandwidth of HACC using CodepFS is 18.2× that of PFS, and the read bandwidth is 8.8× that of PFS. The bandwidth of HPIO is increased by 20.3× by CodepFS.

CodepFS significantly optimizes S3aSim's running time. By reducing I/O time by 94.9%, CodepFS brings down the I/O time ratio from 66.0% to 8.9% of the total runtime. This

reduction in I/O overhead translates to a substantial 62.3% improvement in S3aSim's overall running time.

5 Related work and Discussion

Burst Buffer file system is a high-speed acceleration layer between the global PFS and applications. According to the shared ability of data, it can be divided into shared BB and node-local BB. Node-local BBs use the node-local storage as a cache, and its data cannot be accessed by other nodes. Shared BBs typically use dedicated storage or forwarding nodes for caching. In recent years, some studies have aggregated node-local storage to serve as shared BBs, such as BeeOND, GekkoFS, UnifyFS.

Both GekkoFS and MadFS adopt a path-based hash strategy for data storage. Although MadFS uses LSM-tree to optimize the creation of small files, they are not friendly to small files. CodepFS can improve the access performance of small files by centralizing the object data and metadata of small files, prefetching object data, and reducing the number of RPC. For large files, the central service of BeeOND presents a scalability bottleneck. The hash mapping of file blocks employed by GekkoFS and HadaFS may lead to I/O intensity imbalances among nodes within the BB instance, adversely affecting its scalability. In contrast, CodepFS sustains a balanced data distribution among nodes by utilizing an adaptive file data layout. At file granularity, the maximum difference in load among all nodes is only one data block. As the node scale increases, the benefits of this data distribution become even more pronounced, which leads to better scalability.

HadaFS bridges the two BB and enhances filesystem scalability with Localized Triage Architecture (LTA). However, LTA introduces additional RTT overhead. CHFS [28] is a temporary file system for node-local persistent memory devices such as Intel Optane, which is similar to DAOS, but CHFS requires the user to manually modify the code, which imposes a burden on the user. Common problems with these BB file systems also include no support for workflows, as they are difficult to share data between two BBs. CodepFS introduces a data view fusion technique for workflow applications. Directory mappings allow users to use CodepFS transparently. However, despite the potential benefits and extensive explorations for shared BBs, their use remains minimal among leading TOP500 systems [3, 4, 6, 11, 14].

Data migration to and from PFS is an essential requirement for BB. UnifyFS provides a migration tool, but it does not match the application's I/O pattern, resulting in limited read performance. GekkoFS does not provide a dedicated tool, and it needs to redirect data between client and daemon through parallel copy, which seriously affects migration efficiency. CodepFS seamlessly integrates with PFS, enabling rapid data stage in/out without additional copies.

6 Conclusion

This paper proposes CodepFS, a novel collaborative and adaptive burst buffer to address the limitations of current shared BBs for HPC workflows. CodepFS makes a significant effort for shared BB on data layout strategies, data interaction with PFS, and data sharing among workflow tasks. CodepFS is user-friendly and includes efficient data migration tools and transparent PFS directory mapping, allowing users to reap the benefits of CodepFS's I/O acceleration without additional burdens. Experimental results on the Tianhe Exascale Prototype Upgrade system demonstrate that CodepFS outperforms the state-of-the-art shared burst buffer systems.

Acknowledgments

This work is supported by the National Natural Science Foundation of China under grant No. 62306328, No. U24A2033, No. 62421002, No. 62302514, the science and technology innovation Program of Hunan Province under grant No. 2023RC3021, No. 2024RC1047 and the Natural Science Foundation of Hunan Province of China under Grant No. 2024JJ6471.

We also sincerely thank our shepherd and the ICS reviewers for their insightful feedback, which helped improve the quality of this paper.

References

[1] 2021. Graph500. https://graph500.org/?pageid=944
[2] 2021. GreenGraph500. https://graph500.org/?pageid=946
[3] 2023. Introduction to advanced High performance computing systems. https://top500.org
[4] 2024. Frontier exascale system. https://www.olcf.ornl.gov/frontier/
[5] Hideyuki Akimoto, Takuya Okamoto, Takahiro Kagami, Ken Seki, Kenichiro Sakai, Hiroaki Imade, Makoto Shinohara, and Shinji Sumimoto. 2020. File System and Power Management Enhanced for Supercomputer Fugaku. https://api.semanticscholar.org/CorpusID:245667237
[6] Scott Atchley, Christopher Zimmer, John Lange, David E. Bernholdt, Verónica G. Melesse Vergara, Thomas Beck, Michael J. Brim, Reuben D. Budiardja, Sunita Chandrasekaran, Markus Eisenbach, Thomas M. Evans, Matthew Ezell, Nicholas Frontiere, Antigoni Georgiadou, Joe Glenski, Philipp Grete, Steven P. Hamilton, John K. Holmen, Axel Huebl, Daniel A. Jacobson, Wayne Joubert, Kim H. McMahon, Elia Merzari, Stan G. Moore, Andrew Myers, Stephen Nichols, Sarp Oral, Thomas Papatheodore, Danny Perez, David M. Rogers, Evan Schneider, Jean-Luc Vay, and P. K. Yeung. 2023. Frontier: Exploring Exascale. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2023, Denver, CO, USA, November 12-17, 2023*, Dorian Arnold, Rosa M. Badia, and Kathryn M. Mohror (Eds.). ACM, 52:1–52:16. doi:10.1145/3581784.3607089
[7] Jean Luca Bez, Suren Byna, and Shadi Ibrahim. 2023. I/O access patterns in HPC applications: A 360-degree survey. *Comput. Surveys* 56, 2 (2023), 1–41.
[8] Michael J. Brim, Adam T. Moody, Seung-Hwan Lim, Ross G. Miller, Swen Boehm, Cameron Stanavige, Kathryn M. Mohror, and Sarp Oral. 2023. UnifyFS: A User-level Shared File System for Unified Access to Distributed Local Storage. In *IEEE International Parallel and Distributed Processing Symposium, IPDPS 2023, St. Petersburg, FL, USA, May 15-19,*

2023. IEEE, 290–300. doi:10.1109/IPDPS54959.2023.00037

[9] Haijun Cao, Hai Jin, Xiaoxin Wu, Song Wu, and Xuanhua Shi. 2010. DAGMap: efficient and dependable scheduling of DAG workflow job in Grid. *The Journal of supercomputing* 51 (2010), 201–223.

[10] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. 2021. RocksDB: Evolution of Development Priorities in a Key-value Store Serving Large-scale Applications. *ACM Trans. Storage* 17, 4, Article 26 (oct 2021), 32 pages. doi:10.1145/3483840

[11] Jack Dongarra. 2020. Report on the Fujitsu Fugaku system. *University of Tennessee-Knoxville Innovative Computing Laboratory, Tech. Rep. ICLUT-20-06* (2020).

[12] Xiaobin He, Bin Yang, Jie Gao, Wei Xiao, Qi Chen, Shupeng Shi, Dexun Chen, Weiguo Liu, Wei Xue, and Zuoning Chen. 2023. HadaFS: A File System Bridging the Local and Shared Burst Buffer for Exascale Supercomputers. In *21st USENIX Conference on File and Storage Technologies, FAST 2023, Santa Clara, CA, USA, February 21-23, 2023*, Ashvin Goel and Dalit Naor (Eds.). USENIX Association, 215–230. https://www.usenix.org/conference/fast23/presentation/he

[13] Frank Herold, Sven Breuner, and Jan Heichler. 2014. An introduction to BeeGFS. *ThinkParQ, Kaiserslautern, Germany, Tech. Rep* (2014).

[14] Jonathan Hines. 2018. Stepping up to Summit. *Comput. Sci. Eng.* 20, 2 (2018), 78–82. doi:10.1109/MCSE.2018.021651341

[15] Adrian Jackson and Nicolau Manubens. 2023. DAOS as HPC Storage: Exploring Interfaces. *CoRR* abs/2311.18714 (2023). doi:10.48550/ARXIV.2311.18714 arXiv:2311.18714

[16] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. Design Guidelines for High Performance RDMA Systems. In *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016*, Ajay Gulati and Hakim Weatherspoon (Eds.). USENIX Association, 437–450. https://www.usenix.org/conference/atc16/technical-sessions/presentation/kalia

[17] Harsh Khetawat, Christopher Zimmer, Frank Mueller, Scott Atchley, Sudharshan S. Vazhkudai, and Misbah Mubarak. 2019. Evaluating Burst Buffer Placement in HPC Systems. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. 1–11. doi:10.1109/CLUSTER.2019.8891051

[18] Benjamin R. Landsteiner, Dave Henseler, Douglas Petesch, and Nicholas J. Wright. 2016. Architecture and Design of Cray DataWarp. https://api.semanticscholar.org/CorpusID:96443652

[19] Benjamin R. Landsteiner and David Paul. 2018. DataWarp Transparent Cache : Implementation , Challenges , and Early Experience. https://api.semanticscholar.org/CorpusID:85520996

[20] LLC. Lawrence Livermore National Security. 2015. mpiFileUtils. Retrieved Dec 7, 2024 from http://github.com/hpc/mpifileutils

[21] Zhen Liang, Johann Lombardi, Mohamad Chaarawi, and Michael Hennecke. 2020. DAOS: A Scale-Out High Performance Storage Stack for Storage Class Memory. In *Supercomputing Frontiers: 6th Asian Conference, SCFA 2020, Singapore, February 24–27, 2020, Proceedings* (Singapore, Singapore). Springer-Verlag, Berlin, Heidelberg, 40–54. doi:10.1007/978-3-030-48842-0_3

[22] Ning Liu, Jason Cope, Philip H. Carns, Christopher D. Carothers, Robert B. Ross, Gary Grider, Adam Crume, and Carlos Maltzahn. 2012. On the role of burst buffers in leadership-class storage systems. *012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)* (2012), 1–11. https://api.semanticscholar.org/CorpusID:9676920

[23] Yinqiu Liu, Kai Qian, Kun Wang, and Lei He. 2022. Effective Scaling of Blockchain Beyond Consensus Innovations and Moore's Law: Challenges and Opportunities. *IEEE Syst. J.* 16, 1 (2022), 1424–1435. doi:10.1109/JSYST.2021.3087798

[24] Zhengbin Pang, Min Xie, Jun Zhang, Yi Zheng, Guibin Wang, Dezun Dong, and Guang Suo. 2014. The TH Express high performance interconnect networks. *Frontiers Comput. Sci.* 8, 3 (2014), 357–366. doi:10.1007/s11704-014-3500-9

[25] Juan Piernas, Jarek Nieplocha, and Evan J. Felix. 2007. Evaluation of active storage strategies for the lustre parallel file system. In *Proceedings of the ACM/IEEE Conference on High Performance Networking and Computing, SC 2007, November 10-16, 2007, Reno, Nevada, USA*, Becky Verastegui (Ed.). ACM Press, 28. doi:10.1145/1362622.1362660

[26] Hongzhang Shan and John Shalf. 2007. Using IOR to analyze the I/O Performance for HPC Platforms. *Lawrence Berkeley National Laboratory* (2007). https://api.semanticscholar.org/CorpusID:1260396

[27] Sasu Tarkoma, Christian Esteve Rothenberg, and Eemil Lagerspetz. 2011. Theory and practice of bloom filters for distributed systems. *IEEE Communications Surveys & Tutorials* 14, 1 (2011), 131–155.

[28] Osamu Tatebe, Kazuki Obata, Kohei Hiraga, and Hiroki Ohtsuji. 2022. CHFS: Parallel Consistent Hashing File System for Node-local Persistent Memory. In *International Conference on High Performance Computing in Asia-Pacific Region* (<conf-loc>, <city>Virtual Event</city>, <country>Japan</country>, </conf-loc>) *(HPCAsia '22)*. Association for Computing Machinery, New York, NY, USA, 115–124. doi:10.1145/3492805.3492807

[29] Marc-Andre Vef, Nafiseh Moti, Tim Süß, Markus Tacke, Tommaso Tocci, Ramon Nou, Alberto Miranda, Toni Cortes, and André Brinkmann. 2020. GekkoFS - A Temporary Burst Buffer File System for HPC Applications. *J. Comput. Sci. Technol.* 35, 1 (2020), 72–91. doi:10.1007/S11390-020-9797-6

[30] Marc-Andre Vef, Nafiseh Moti, Tim Süß, Tommaso Tocci, Ramon Nou, Alberto Miranda, Toni Cortes, and André Brinkmann. 2018. GekkoFS - A Temporary Distributed File System for HPC Applications. In *IEEE International Conference on Cluster Computing, CLUSTER 2018, Belfast, UK, September 10-13, 2018*. IEEE Computer Society, 319–324. doi:10.1109/CLUSTER.2018.00049

[31] Ruibo Wang, Kai Lu, Chen Juan, Wen zhe Zhang, Jin wen Li, Yuan Yuan, Ping jing Lu, Libo Huang, Shengguo Li, and Xiaokang Fan. 2021. Brief introduction of TianHe exascale prototype system. *Tsinghua Science & Technology* 26 (2021), 361–369. https://api.semanticscholar.org/CorpusID:228997868

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

[33] Md. Wasi-ur-Rahman, Xiaoyi Lu, Nusrat Sharmin Islam, Raghunath Rajachandrasekar, and Dhabaleswar K. Panda. 2014. MapReduce over Lustre: Can RDMA-Based Approach Benefit?. In *Euro-Par 2014 Parallel Processing - 20th International Conference, Porto, Portugal, August 25-29, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8632)*, Fernando M. A. Silva, Inês de Castro Dutra, and Vítor Santos Costa (Eds.). Springer, 644–655. doi:10.1007/978-3-319-09873-9_54

[34] Weixia Xu, Yutong Lu, Qiong Li, Enqiang Zhou, Zhenlong Song, Yong Dong, Wei Zhang, Dengping Wei, Xiaoming Zhang, Haitao Chen, Jianying Xing, and Yuan Yuan. 2014. Hybrid hierarchy storage system in MilkyWay-2 supercomputer. *Front. Comput. Sci.* 8, 3 (jun 2014), 367–377. doi:10.1007/s11704-014-3499-6