LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-681480

BurstFS: A Distributed Burst Buffer File System for Scientific Applications

T. Wang, W. Yu, K. Sato, A. Moody, K. Mohror

January 28, 2016

International Conference on Supercomputing
Istanbul, Turkey
May 30, 2016 through January 1, 2016

Disclaimer

An Ephemeral Burst Buffer File System for Scientific Applications

Teng Wang[†] Kathryn Mohror[‡] Adam Moody[‡] Kento Sato[‡] Weikuan Yu[†]
[†]Florida State University [‡]Lawrence Livermore National Lab
{twang, yuw}@cs.fsu.edu {kathryn, moody20, sato5}@llnl.gov

***Abstract*—Burst buffers are becoming an indispensable hardware resource on large-scale supercomputers to buffer the bursty I/O from scientific applications. However, there is a lack of software support for burst buffers to be efficiently shared by applications within a batch-submitted job and recycled across different batch jobs. In addition, burst buffers need to cope with a variety of challenging I/O patterns from data-intensive scientific applications. In this study, we have designed an ephemeral Burst Buffer File System (BurstFS) that supports scalable and efficient aggregation of I/O bandwidth from burst buffers while having the same life cycle as a batch-submitted job. BurstFS features several techniques including scalable metadata indexing, co-located I/O delegation, and server-side read clustering and pipelining. Through extensive tuning and analysis, we have validated that BurstFS has accomplished our design objectives, with linear scalability in terms of aggregated I/O bandwidth for parallel writes and reads.**

I. Introduction

With the explosive growth of scientific and analytic datasets, burst buffers have been stipulated as an indispensable component on large-scale high performance computing (HPC) systems [2, 3, 6, 9, 10, 11, 13, 24]. There are two main strategies for deploying burst buffers. One strategy is to attach fast storage locally to each compute node, referred to as node-local burst buffers. The other is to provision an additional layer of fast storage that can be remotely shared by compute nodes, referred to as remote or shared burst buffers. While both strategies are being employed on current and next-generation systems [9, 10, 12], we focus on the node-local burst buffer strategy in this work.

Burst buffers are a powerful hardware resource for scientific applications to buffer their bursty I/O traffic. However, the usage of burst buffers is not yet well-studied, nor are burst buffer software interfaces standardized across systems. Currently, users are left with the freedom to explore the use of burst buffers in an ad-hoc manner. However, domain scientists would rather focus on their scientific problems instead of fiddling with the complexity of how to best use burst buffers.

Several efforts have explored the use of locality-aware distributed file systems (e.g., HDFS [37]) to manage node-local burst buffers [4, 45, 48]. In such file systems, each process stores its primary data to the local burst buffer. Because compute processes can be co-located with their data, it is feasible to achieve linearly scalable aggregated bandwidth [48, 32]. However, burst buffers are only available to user jobs temporarily. A user job can utilize local burst buffers within the duration of its allocation, but the job loses access to the burst buffer storage when the allocation terminates. Conventional file systems such as HDFS [37] or Lustre [17, 31] are typically designed to persist data indefinitely, on the order of an HPC system's lifetime. They utilize long-running daemons for I/O services, which are not necessary for temporary burst buffer usage. In addition, the construction and cleanup of I/O services for these file systems can lead to a waste of resources in terms of compute cores, storage and memory. Therefore, for effective use of burst buffers by scientific users, it is critical to develop software for standardizing the use of burst buffers, so that they can be seamlessly integrated into the repertoire of HPC tools on leadership supercomputers.

HPC applications typically exhibit two main I/O patterns: shared file (N-1) and file-per-process (N-N) [15] (see details in Section II-A). For node-local burst buffers, with the N-N pattern, applications can achieve scalable bandwidth by having each process write/read its files locally. The difficulty for node-local burst buffers lies with the N-1 I/O pattern, in which all processes write a portion of a shared file. In particular, a shared file requires the metadata for all data segments to be properly constructed, indexed, and collected at the time of writes, then later formulated with a global layout before any process can locate its targeted data for read access. While this issue has been investigated on persistent parallel file systems [15, 47], the problem of efficiently formulating and serving the global layout of a shared file remains a critical issue for a temporary file system across burst buffers.

In addition, datasets from scientific applications are typically multi-dimensional. Such datasets are usually stored in one particular order of multiple dimensions, but frequently read from different dimensions based on the nature of scientific simulation or analysis. Often, there is an incompatibility between the order of writes and the order of reads for data elements in a multi-dimensional dataset, which typically leads to many small non-contiguous read operations for one process to retrieve its desired data elements [39] (see Section II-B for more details). An effective node-local burst buffer file system also needs to provide a mechanism for scientific applications to efficiently read multi-dimensional datasets without many costly small read operations.

In this research, we have designed an ephemeral Burst

978-1-4673-8815-3/16/$31.00

Buffer File System (BurstFS) that has the same temporary life cycle as a batch-submitted job. BurstFS organizes the metadata for the data written in local burst buffers into a distributed key-value store. To cope with the challenges from the aforementioned I/O patterns, we designed several techniques in BurstFS including scalable metadata indexing, co-located I/O delegation, and server-side read clustering and pipelining. We used a number of I/O kernels and benchmarks to evaluate the performance of BurstFS, and validate our design choices through tuning and analysis.

In summary, our research makes the following contributions.

- We present the design and implementation of a burst-buffer file system to meet the need of effective utilization of burst buffers on leadership supercomputers.
- We introduce several mechanisms inside BurstFS, including scalable metadata indexing for quickly locating data segments of a shared file, co-located I/O delegation for scalable and recyclable I/O management, and server-side clustering and pipelining to support fast access of multi-dimensional datasets.
- We evaluate the performance of BurstFS with a broad set of I/O kernels and benchmarks. Our results demonstrate that BurstFS achieves linear scalability in terms of aggregated I/O bandwidth for parallel writes and reads.
- To the best of our knowledge, BurstFS is the first file system designed to have a co-existent and ephemeral life cycle with one or a batch of scientific applications in the same job.

II. Background on I/O Patterns for Burst Buffers

In this section, we provide an overview of typical I/O patterns that need to be supported on burst buffers, in order to facilitate understanding of our design objectives for BurstFS.

A. Checkpoint/Restart I/O Using Shared or Per-Process Files

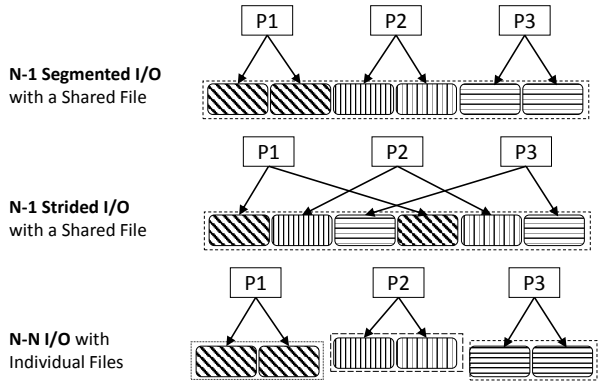


Fig. 1: Checkpoint/restart I/O patterns (adapted from [15]).

Checkpoint/restart is a common fault tolerance mechanism used by HPC applications. During a run, application processes periodically save their in-memory state in files called checkpoints, typically written to a parallel file system (PFS). Then, in the event of a failure, the most recent checkpoint can be read to restart the job. For simplicity, checkpointing operations are usually concurrent across all processes in an application, and occur at program synchronization points when no messages are in flight. On current HPC systems, checkpointing can account for 75%-80% of the total I/O traffic [32]. While there is ongoing debate on how checkpointing operations will change on exascale systems compared to today's systems, there is general consensus that the data size per checkpoint will increase due to larger job scales and the interval between checkpoints will decrease due to increased overall failure rates [18, 29]. The larger file sizes and shorter intervals for checkpointing will require orders of magnitude faster storage bandwidth [25].

There are two dominant I/O patterns for checkpoint/restart, N-1 and N-N patterns as shown in Fig. 1. In N-N I/O, each process writes/reads data to/from a unique file. In N-1 I/O, all processes write to, or read from, a single shared file. N-1 I/O can be further classified into two patterns: N-1 segmented and N-1 strided. In N-1 segmented I/O, each process accesses a non-overlapping, contiguous file region. In N-1 strided I/O, processes interleave their I/O amongst each other.

In conventional parallel file systems [31, 34, 36, 43] large files are striped over one or more storage servers. While striping allows each process to interact with multiple storage servers in parallel, it also leads to striping overhead [46] and I/O contention [15, 41] when multiple processes concurrently access the same storage server. A distributed burst buffer file system that localizes writes/reads is highly beneficial for such checkpoint/restart workload because of the reduced contention and the performance advantage of burst buffer device (e.g. NVRAM and SSD).

B. Multi-dimensional I/O Access in Scientific Applications

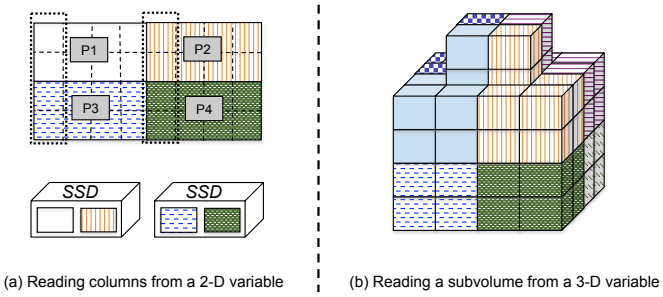


Fig. 2: I/O access patterns with multi-dimensional variables.

Another common I/O pattern on HPC systems is data access to multi-dimensional data variables in scientific applications. While multi-dimensional variables are written in one particular order, they are often read for analysis or visualization in a different order than the write order [27, 39].

Fig. 2(a) shows a sample read pattern on a two-dimensional variable. This variable is initially decomposed into four blocks, which are written to two SSDs as four data blocks. When this variable is read back for analysis, one process may require only one or more columns from this variable. However, these two columns are stored non-contiguously across the data blocks.

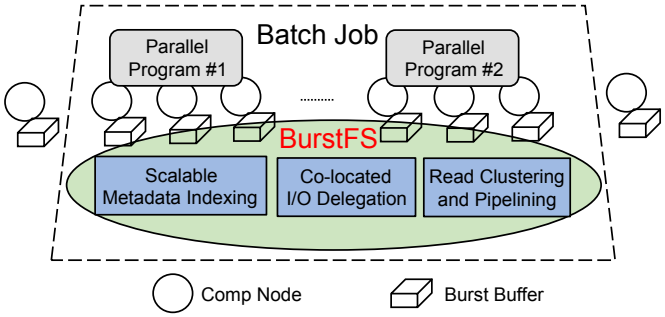


Fig. 3: BurstFS system architecture.

Therefore, this process needs to issue small read requests to four different data blocks in order to retrieve its data for analysis. Fig. 2(b) illustrates a similar but more complex scenario with a three-dimensional variable. The 3-D variable is initially stored as eight different blocks across burst buffers. A process may only need a subvolume in the middle of the variable for analysis. This subvolume has to be gathered from eight different blocks to complete its data access, resulting in many small read operations. Taken together, a high-performance file system for burst buffers must provide a mechanism to efficiently read multi-dimensional datasets without many small read operations.

III. Ephemeral Burst Buffer File System

We designed the Burst Buffer File System (BurstFS) as an ephemeral file system, with the same lifetime as an HPC job. Our overarching goal for BurstFS is to support scalable aggregation of I/O operations across distributed, node-local storage for data-intensive simulations, analytics, visualization, and checkpoint/restart. BurstFS instances are launched at the beginning of a batch job, provide data services for all applications in the job, and terminate at the end of the job allocation. Fig. 3 shows the system architecture of BurstFS.

When a batch job is allocated a set of compute nodes on an HPC system, an instance of BurstFS will be constructed on-the-fly across these nodes, using the locally-attached burst buffers, which may consist of memory, SSD, or other fast storage devices. These burst buffers enable very fast log-structured local writes; i.e., all processes can store their writes to the local logs. Next, one or more parallel programs launched on a portion of these nodes can leverage BurstFS to write data to, or read data from, the burst buffers. In addition, a BurstFS instance exists only during the lifetime of the batch job. All allocated resources and nodes will be cleaned up for reuse at the end of the scheduled execution. This avoids any post-mortem interference with other jobs or potential unforeseeable complications to the operation of file and storage systems. Furthermore, parallel programs within the same job allocation (e.g., programs launched within the same batch script) can share data and storage on the same BurstFS instance, which can greatly reduce the need of back-end persistent file systems for data sharing across these programs.

BurstFS is mounted with a configurable prefix and transparently intercepts all POSIX functions under that prefix [40].

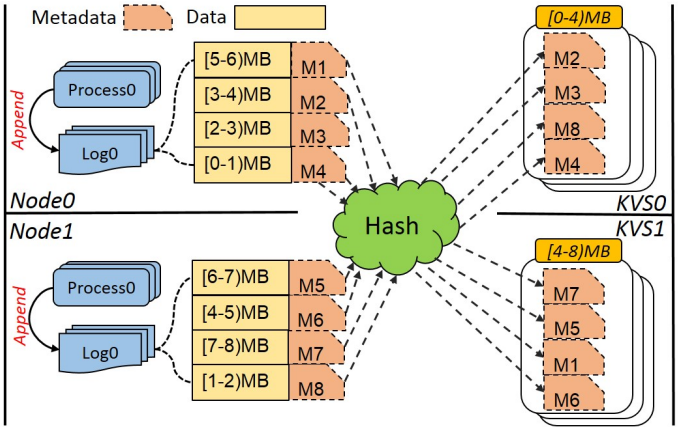


Fig. 4: Diagram of the distributed key-value store for BurstFS.

Data sharing between different programs can be accomplished by mounting BurstFS using the same prefix. Upon the unmount operation from the last program, all BurstFS instances sequentially flush their data for data persistence (if requested), clean up their resources and exit.

To support the challenging I/O patterns discussed in Section II, we designed several techniques in BurstFS including scalable metadata indexing, co-located I/O delegation, and server-side read clustering and pipelining as shown in Fig. 3. BurstFS organizes the metadata for the local logged data into a distributed key-value store. It enables scalable metadata indexing such that a global view of the data can be generated quickly to facilitate fast read operations. It also provides a lazy synchronization scheme to mitigate the cost and frequency of metadata updates. In addition, BurstFS supports co-located I/O delegation for scalable and recyclable I/O management. Furthermore, we introduce a mechanism called server-side read clustering and pipelining for improving the read performance. We elaborate on these techniques in the rest of this section.

A. Scalable Metadata Indexing

As discussed in Section I, one of the challenges for the N-1 I/O pattern is accessing the metadata of segments scattered across all nodes. This leads to a huge scalability problem when all processes are reading their data and each one needs to gather the metadata from all nodes.

1) Distributed Key-Value Store for Metadata: BurstFS solves this issue using a distributed key-value store for metadata, along with log-structured writes for data segments. It leverages MDHIM [23] for the construction of distributed key-value stores and provides additional features for efficient handling of bursty read and write operations.

Fig. 4 shows the organization of data and metadata for BurstFS. Each process stores its data to the local burst buffer as data logs, which are organized as data segments. New data are always appended to the data logs, i.e., stored via log-structured writes. With such log-structured writes, all segments from one process are stored together regardless of their global logical position with respect to data from other processes.

When the processes in a parallel program create a global shared file, a key-value pair (e.g., M1 or M2, etc) is generated for each segment. A key consists of the file ID (8-byte hash value) and the logical offset of the segment in the shared file. The value describes the actual location of the segment, including the hosting burst buffer, the log containing the segment (there can be more than one log from multiple processes on the same node), the physical offset in the log, and the length. The key-value pairs (KVP) for all the segments can then provide the global layout for the shared file. All the KVPs are consistently hashed and distributed among the key-value servers (e.g., KVS0, KVS1 and so on). With such an organization, the metadata storage and services are spread across multiple key-value servers. Many processes from a parallel application can quickly retrieve the metadata and form a global view of the layout of a shared file.

2) Lazy Synchronization: In BurstFS, we also develop *lazy synchronization* to provide efficient support for bursty writes. Each process provides a small memory pool for holding the metadata KVPs from write operations, and, at the end of a configurable interval, KVPs are periodically stored to the distributed key-value stores. An *fsync* operation can force an explicit synchronization. BurstFS leverages the batch put operation from MDHIM to transfer these KVPs together in a few round-trips, minimizing the latency incurred by single put operations. During the synchronization interval, BurstFS searches for contiguous KVPs in the memory pool to potentially combine. A combined KVP can span a bigger range. As shown in Fig. 4, segments [2-3) MB and [3-4) MB are contiguous and map to the same server (KVS0), so their KVPs are combined into one KVP. Lazy synchronization can significantly reduce the number of KVPs required when many data segments issued by each process are logically contiguous (e.g. N-1 segmented and N-N write in Fig. 1).

3) Parallel Range Queries: To begin a read operation, BurstFS has to first look up the metadata for the distributed data segments. Thus, it searches for all KVPs whose offsets fall in the requested range, e.g., [`offset`, `offset+count`] is the requested range in `pread`. With batched read requests, BurstFS needs to search for all KVPs that are targeted by the read requests in the batch. To retrieve the requested metadata entries for different read operations, we need support for a variety of range queries to the key-value store. However, range queries are not directly supported by MDHIM; the clients can indirectly perform range queries by iterating over consecutive KVPs within a range with repeated cursor-type operations. Clients must sequentially invoke one or more cursor operations for one range server, and must search multiple range servers until all KVPs have been located. The additive round-trip latencies by all cursor operations to multiple range servers can severely delay read operations.

To mitigate this, we introduce parallel extensions for both MDHIM clients and servers. On the client side, we transform an incoming range request and break it into multiple small range queries to be sent to each server based on consistent hashing. Compared with sequential cursor operations, this extension allows a range query to be broken into many small range queries, one for each range server. These small queries are then sent in parallel to all range servers to retrieve all KVPs. On the server side, for the small range query within its scope, all KVPs inside that range are retrieved through a single sequential scan in the key-value store. With this parallel optimization, any combination of queries can be accomplished through only parallel range queries to all servers and a single local scan operation at each key-value server.

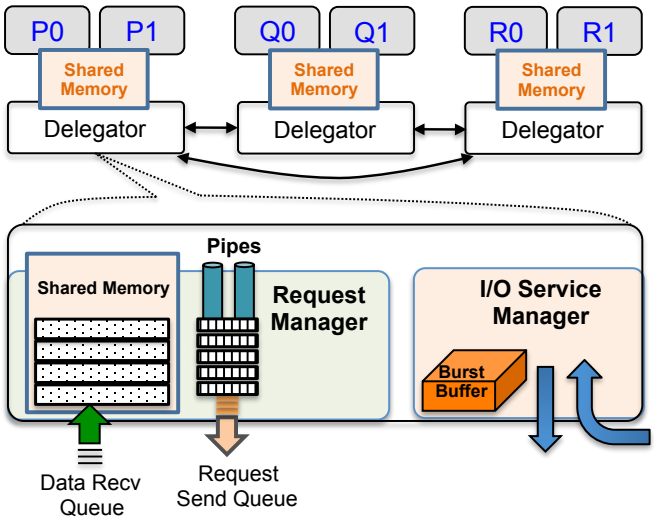


Fig. 5: Diagram of Co-located I/O Delegation on three compute nodes P, Q and R, each with 2 processes.

B. Co-located I/O Delegation

In contrast to BurstFS write operations that store data locally, read operations in BurstFS may need to transfer data from remote burst buffers to a process initiating a read. To ensure the efficiency of reads, we need to support fast and scalable data transfer for read operations. A common approach adopted by many parallel programming models such as MPI [28] and PGAS [20] is to have each process make read function calls to persistent file and storage service daemons. Because BurstFS has a limited lifetime to that of a single job, BurstFS has special requirements for I/O services. One implementation option might be to have persistent I/O daemons to support BurstFS; however, that would lead to a waste of computation and memory resources. Another implementation choice could be to utilize a simple I/O service thread spawned from the parent process in a parallel program. However, with this approach, the service thread can only serve the I/O needs for processes in the same program, and cannot serve subsequent or concurrent programs in the batch job.

In BurstFS, we introduce scalable read services through a mechanism called *co-located I/O delegation*. We launch an I/O proxy process on each node, a *delegator*. Delegators are decoupled from the applications in a batch job, and are launched across all compute nodes. The delegators collectively provide data services for all applications in the job.

As shown in Fig. 5, processes on three compute nodes will have all their I/O activities delegated to the delegator on the same node. Each delegator consists of two main components: a *request manager* and an *I/O service manager*. In this way, a conventional client-server model for I/O services is transformed into a peer-peer model among all delegators. With this arrangement, individual processes no longer communicate with I/O servers directly, but go through their I/O delegators. This leads to a great reduction on the total number of network communication channels and the associated resources across the compute nodes. The I/O service manager in each delegator is dedicated to serve the incoming read requests from peer delegators. The I/O service managers exploit opportunities to consolidate requests, pipeline data retrieval from local storage, and transfer data back to requesting delegators (See Section III-C for details).

The request manager of a delegator is composed of two main data structures: a *request send queue* and a *data receive queue*, as shown in Fig. 5. The request send queue is a circular list with a configurable number of entries. When not full, it receives the read requests from all client processes through named pipes. Requests are queued based on the destination delegator. Requests to the same delegator are chained together, which consolidates multiple requests into a single network message. The data receive queue resides in a shared memory pool constructed across delegator and client processes on the same node. For each I/O request, an outstanding request entry is created in the receive queue. Data returned from remote delegators is directly deposited in the shared memory pool, and the receive queue is searched for a matching outstanding request entry. When a match is found, the outstanding request is marked as complete. An additional acknowledgment is sent via the pipe to notify the client process to consume the data.

The request manager monitors the utilization level of the shared memory pool. When it is higher than a configurable threshold (default 75%), the delegator (1) informs processes of the urgent need to consume their data and (2) throttles request injection to remote delegators. The request manager also monitors the ingress bandwidth based on the received data for read requests in the send queue. When the ingress bandwidth is saturated, the request manager creates additional network communication channels to send requests and receive data.

C. Server-Side Read Clustering and Pipelining

As discussed in Section II-B, with multi-dimensional variables, a process can issue many small, noncontiguous read requests for scattered data segments in each data log. Various I/O libraries and tools have provided special support for such noncontiguous read access. For instance, POSIX `lio_listio` allows read requests to be transferred in batches; and OrangeFS supports batched read requests. While being able to combine small requests into a list or a large request, these techniques mainly work from the client side and rely on the underlying storage system such as the disk scheduler to prefetch or merge requests for fast data retrieval. However, there is still a lack of distributed file systems that can globally optimize these batch read requests from all processes.

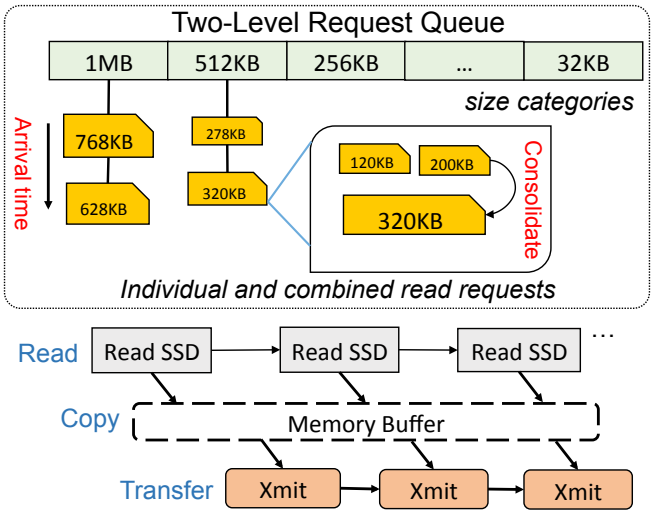


Fig. 6: Server-Side Read Clustering and Pipelining.

As an ephemeral file system in a batch job, BurstFS directly manages accesses to the datasets from scientific applications via delegators. Therefore, besides leveraging the existing techniques of batched reads from the client side, BurstFS can exploit its visibility of read requests at the server side (via the I/O service manager) for further performance improvements. To this end, we introduce a mechanism called *server-side read clustering and pipelining (SSCP)* in the I/O service manager to improve the read performance of BurstFS.

SSCP addresses several concurrent, sometimes conflicting objectives: (1) the need of detecting spatial locality among read requests and combining them for large contiguous reads. (2) and the need of serving on-demand read requests as soon as possible for execution progress. As shown in Fig. 6, SSCP provides two key components to achieve these objectives, a *two-level request queue* for read clustering and a three-stage pipeline for fast data movement.

In the two-level request queue, SSCP first creates several categories of request sizes, ranging from 32KB to 1MB (see Fig. 6). Incoming requests will be inserted to the appropriate size category either individually, or if contiguous with other requests, combined with the existing contiguous requests and then inserted into the suitable size category. As shown in the figure, two contiguous requests of 120KB and 200KB are combined by the service manager. Within each size category, all requests are queued based on their arrival time. A combined request will use the arrival time from its oldest member. For best scheduling efficiency, the category with largest request size is prioritized for service. Within the same category, the oldest request will be served first. BurstFS enforces a threshold on the wait time of each category (default 5*ms*). If there is any category having not been serviced longer than this threshold, BurstFS selects the oldest read request from this category for service and resets the category's wait time.

The I/O service manager creates a memory pool to tem-

porarily buffer outgoing data. This facilitates the rearrangement of data segments for network transfer and allows the formulation of a pipeline. Fig. 6 shows the three-stage data movement pipeline: reading, copying, and transferring. In the reading stage, the I/O service manager picks up a request from the request list based on the aforementioned scheduling policy, reads the requested data from the local burst buffer to a slot in the memory buffer. In the copying stage, the data in the memory buffer is prepared as an outgoing reply for the remote delegator, and then copied from the memory buffer to the network packet. Data inside the memory buffer may need to be divided into multiple replies for different remote delegators. The I/O service manager then creates multiple network replies, one for each delegator. In the transferring stage, the I/O service manager can pack one or more network replies for the same remote delegator into one network message (1MB maximum), and transmit (*Xmit* in Fig. 6) it to the delegator.

IV. Experimental Evaluation

A. Testbed

Our experiments are conducted on the Catalyst cluster [3] at Lawrence Livermore National Laboratory (LLNL), consisting of 384 nodes. Each node is equipped with two 12-core Intel Xeon E5-2695v2 processors, 128 GB DRAM and an 800-GB burst buffer comprised of PCIe SSDs.

Configuration: We focus on comparing BurstFS with two contemporary file systems: OrangeFS 2.8.8, and the Parallel Log-Structured File System 2.5 (PLFS [15]). As a representative parallel file system (PFS), OrangeFS stripes each file over multiple storage servers to enable parallel I/O with high aggregate bandwidth. In our experiments, we establish OrangeFS server instances across all the compute nodes allocated to a job to manage all the node-local SSDs. PLFS is designed to accelerate N-1 writes by transforming random, dispersed, N-1 writes into sequential N-N writes in a log-structured manner. Data written by each process are stored on the backend PFS as a log file. In our experiments, we use OrangeFS (over node-local SSDs) as the backend PFS for PLFS. We use PLFS's MPI interface for read and write.

Since version 2.0, PLFS has had burst buffer support. In PLFS with burst buffer support (referred to as "PLFS burst buffer" in the rest of this paper), instead of writing the log file on the backend PFS, processes store their *metalinks* on the backend PFS, which point to the real location of their log files in the burst buffers. This allows each process to write its log file to the burst buffer instead of the backend PFS. In our experiments, we have each process write to its node-local SSD, and the location is recorded in the metalink stored on the center-wide Lustre parallel file system. This configuration can deliver scalable write bandwidth. In order to read data from PLFS burst buffer, each node-local SSD has to be mounted on all other compute nodes as a global file system (e.g., NFS), which requires system administrator support. A primary goal for BurstFS is that it be completely controllable from user space, including mounting the file system. Thus, due to the requirement of administrator intervetion to establish the cross-mount environment for read with PLFS burst buffer, we only evaluated the write scalability of PLFS burst buffer and include this result in Section IV-B.

Benchmarks: We have employed microbenchmarks that exhibit three checkpoint/restart I/O patterns (see Fig. II-A). Note that N-1 strided pattern is a case of 2-D scientific I/O as described in Section II-B.

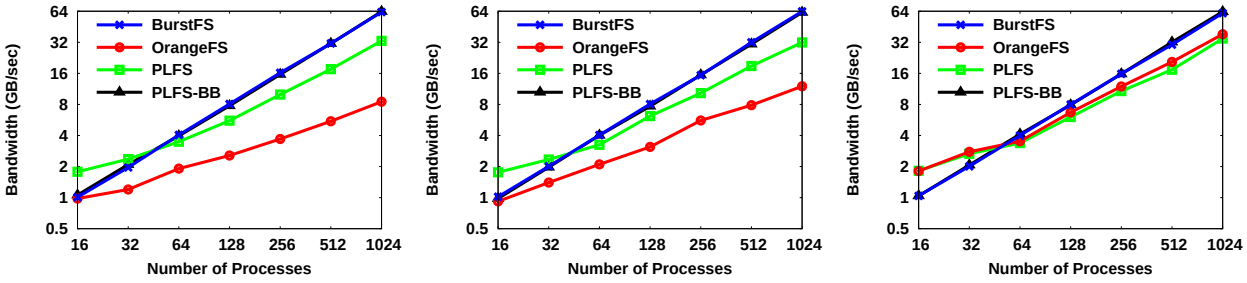
To assess BurstFS's potential to support scientific applications, we evaluate BurstFS using I/O workloads extracted from MPI-Tile-IO [33] and BTIO [44]. MPI-Tile-IO is a widely adopted benchmark used for simulating the workloads that exist in visualization and numerical applications. The two-dimensional dataset is partitioned into multiple tiles, each process rendering pixels inside one tile. Developed by NASA Advanced Supercomputing Division, BTIO partitions a three-dimensional array across a square number of processes, each process processing multiple Cartesian subsets. In both workloads, all processes first write their data into a shared file, then read back into their memory. To evaluate the support for a batch job of multiple applications, we employ the Interleaved Or Random (IOR) benchmark [26] to read data provided by Tile-IO and BTIO programs in the same job.

B. Overall Write/Read Performance

We first evaluate the overall write/read performance of BurstFS. In this experiment, 16 processes are placed on each node, each writing 64MB data following an N-1 strided, N-1 segmented, or N-N pattern. After each process writes all of its data, we use `fsync` to force all writes to be synchronized to the node-local SSD. We set the stripe size on OrangeFS as 1MB and fix the transfer size at 1MB to align with the stripe size, and each file is striped across all nodes in OrangeFS. This configuration gives OrangeFS the best read/write bandwidth over other tuning choices (e.g. 64KB default stripe size).

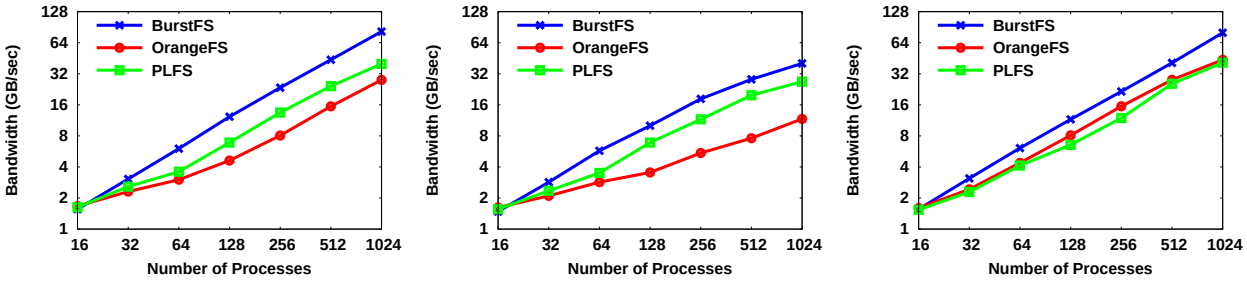
Fig. 7 compares the write bandwidth with PLFS burst buffer (PLFS-BB), PLFS, and OrangeFS. In all three write patterns, both BurstFS and PLFS burst buffer scale linearly with process count. This is because processes in both systems write locally and the write bandwidth of each node-local SSD is saturated. While we also observe linear scalability in OrangeFS and PLFS, their bandwidths increase at a much slower rate. This is because both PLFS and OrangeFS stripe their file(s) across multiple nodes, which can cause degraded bandwidth due to contention when different processes write to the same remote node. On average, BurstFS delivers 3.5×, 2.7×, and 1.3× the performance of OrangeFS for N-1 segmented, N-1 strided, and N-N patterns, respectively. Its performance is 1.6×, 1.6×, and 1.5× the performance of PLFS, respectively, for the three patterns.

We observe that PLFS initially delivers higher bandwidth than BurstFS at small process counts (16 and 32 processes), for all three patterns. After further investigation, we find this is because, internally, PLFS transforms the N-1 writes into N-N writes. However, when `fsync` is called to force these N-N files to be written to PLFS's back end file system, (i.e.,



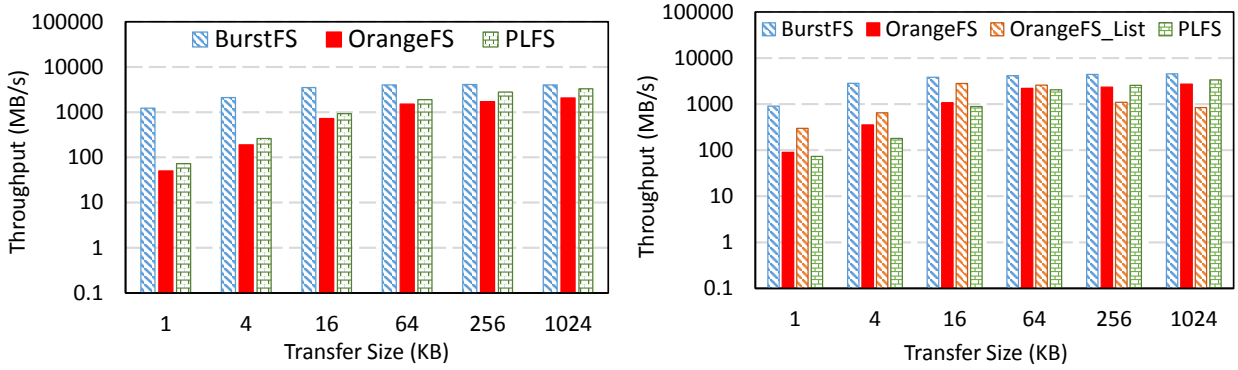
(a) N-1 Segmented Write (b) N-1 Strided Write (c) N-N Write

Fig. 7: Comparison of BurstFS with PLFS and OrangeFS under different write patterns.



(a) N-1 Segmented Read (b) N-1 Strided Read (c) N-N Read

Fig. 8: Comparison of BurstFS with PLFS and OrangeFS under different read patterns.



(a) Write (b) Read

Fig. 9: Comparison of BurstFS with PLFS and OrangeFS under different transfer sizes.

OrangeFS), OrangeFS does not completely flush the files to the SSDs before `fsync` returns. The measured bandwidth is even higher than the aggregate SSD bandwidth on the local file systems.

Fig. 8 compares the read bandwidth of BurstFS with OrangeFS and PLFS. Each process reads 64MB data under N-1 strided, N-1 segmented and N-N patterns. For the N-1 strided reads, we first create a shared file using N-1 segmented writes, then read all data using the N-1 strided reads. In this way, each process needs to read data from multiple logs as discussed in Section II-B. In order to cluster the non-contiguous read requests under this pattern, we use POSIX `lio_listio` to transfer read requests to BurstFS in batches. In the case of OrangeFS, when we enable its list I/O operations, we observe the bandwidth is two times lower than the configuration without list I/O operations. This is because OrangeFS list I/O does not benefit large read operations. Thus, for this experiment, we report only the performance of N-1 strided pattern in OrangeFS without its list I/O support.

As we can see from Fig. 8(a), the bandwidth of N-1 segmented read scales linearly with process count for BurstFS, since each process reads all data directly from its local node. In contrast, both PLFS and OrangeFS need to read data from remote nodes, losing the benefit from locality. On the other hand, the bandwidth of N-1 strided read in Fig. 8(b) increases at a much slower rate in BurstFS compared with segmented read. This is because the strided read pattern results in higher contention due to all-to-all reads from remote burst

buffers. BurstFS with N-1 strided read still scales better and outperforms both OrangeFS and PLFS. This is because instead of servicing each request individually, BurstFS delegators cluster read requests from numerous processes and serve them through a three-stage read pipeline. On average, BurstFS delivers 2.2×, 2.5× and 1.4× the performance of OrangeFS, respectively, for N-1 segmented, N-1 strided and N-N patterns. It delivers 1.6×, 1.4× and 1.6× the performance of PLFS, respectively, for the three patterns.

C. Performance Impact of Different Transfer Sizes

Fig. 9 shows the impact of transfers sizes on the bandwidth of BurstFS. We focus on N-1 strided I/O, because it is a challenging I/O pattern. Similar to the experiment in Section IV-B, for BurstFS strided read operations, we first create a shared file using N-1 segmented writes and then read the data back using an N-1 strided pattern. In this way, BurstFS will not benefit from local reads.

The results in Fig. 9(a) demonstrate the impact of transfer sizes on write bandwidth when 64 processes write to a shared file. BurstFS outperforms OrangeFS and PLFS by having each process write data locally, and it delivers outstanding performance improvement at small transfer sizes, for example, 24.4× and 16.7× at 1KB compared to OrangeFS and PLFS, respectively. This is because both PLFS and OrangeFS suffer from the cost of random writes and repeated data transfers to the remote burst buffers.

Fig. 9(b) shows the impact of transfer size on read bandwidth. For small read requests, OrangeFS provides list I/O support so that a list of read requests can be combined into one function call. The result of this type of read operations is shown in Fig. 9(b) as OrangeFS_List. As we can see from this figure, although OrangeFS_List enhances the performance of small reads, it is still lower than BurstFS. This is because the additional benefits of server-side clustering and pipelining in BurstFS. Overall, BurstFS yields up to 10.2×, 3× and 12.3× performance improvement compared to OrangeFS, OrangeFS_List and PLFS, respectively.

D. Analysis of Metadata Performance

As discussed in Section III-A1, BurstFS distributes the global metadata indices over distributed key-value store. During file open, each process in PLFS needs to construct a global view of a shared file by reading and combining metadata from other processes. After this step, all look-up operations are conducted locally. To evaluate the benefit of our design, we compare the metadata look-up time of BurstFS with that of PLFS (i.e. PLFS's total time on index construction during file open and local look-up during read), as well as the original look-up time from the MDHIM functions. We examine the look-up performance using both cursor and batch get functions from MDHIM. Each cursor operation triggers a round-trip transfer for each key-value pair, and a look-up for a range can invoke multiple cursor operations as described in Section III-A3. The total look-up time is significantly higher than other cases. For instance, it takes 81 seconds for the 4KB case in Fig. 10(a). So we omit the look-up time with cursor operations in our figures.

Fig. 10(a) compares the look-up time of BurstFS, PLFS, and MDHIM batch get (denoted as MDHIM). In all three cases, we launch 32 processes, each to look up the locations of 64MB data written under the N-1 strided pattern. The total data volume is the product of the transfer size and the number of segments. Thus a smaller transfer size will lead to more segments, therefore more indices. As we can see from the figure, the look-up time of all cases drops along with the increasing transfer size. This is because of fewer metadata look-ups. The look-up time of BurstFS is significantly faster than PLFS. This validates that the scalable metadata indexing technique in BurstFS can quickly establish a global view of the metadata for a shared file. In contrast, every process in PLFS has to load all the indices generated during write and construct the global indices for read, this all-to-all load dominates the look-up time. BurstFS also outperforms MDHIM by minimizing the number of read operations with only one sequential scan at the range server because of its support for parallel range queries (see Section III-A3). On average, BurstFS reduces the look-up time by 77% and 58% compared with PLFS and MDHIM, respectively.

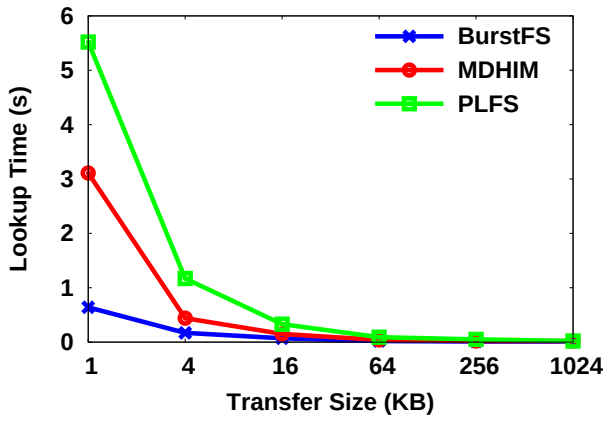
Fig. 10(b) shows the metadata performance with an increasing process count. In this test, each process looks up 64 MB data written with a transfer size of 64 KB. More processes lead to more look-up operations. As shown in the figure, the look-up time of PLFS increases sharply with the process count. In contrast, the look-up time for BurstFS and MDHIM increases slowly with more processes, because of the use of a distributed key-value store for metadata.

E. Tile-IO Test

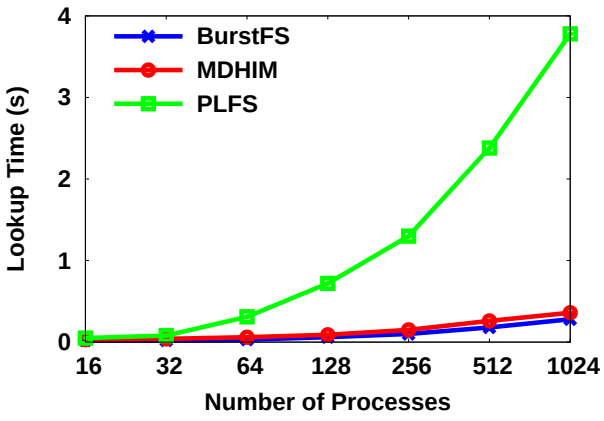
Fig. 11 shows the performance of BurstFS with Tile-IO. In this experiment, a 32GB global array is partitioned over 256 processes. Each process first writes its tile to several noncontiguous regions of the shared file, then reads it back to its local memory. For write operations, BurstFS outperforms OrangeFS and PLFS by directly writing data to local SSDs. For reads, although all three file systems benefit from the buffer cache, BurstFS still performs best since each process reads data locally. Overall, BurstFS delivers 6.9× and 2.5× improvement over OrangeFS for reads and writes, respectively, and 7.3× and 1.4× improvement over PLFS for reads and writes, respectively.

F. BTIO Test

Fig. 12 shows the performance of BurstFS under the BTIO workload with problem size D. In this experiment, the 408 × 408 × 408 global array is decomposed over 64 processes. Similar to Tile-IO, each process first writes its own cells to several noncontiguous regions of the shared file, then reads them back to its local memory. Due to the 3-D partitioning, the transfer size (2040B) of each process is much smaller than Tile-IO (32KB), so the I/O bandwidth of both PLFS and OrangeFS with BTIO decreases rapidly, compared with



(a) Metadata performance with varying transfer sizes



(b) Metadata Performance with varying process counts

Fig. 10: Analysis of metadata performance as a result of transfer size and process count.

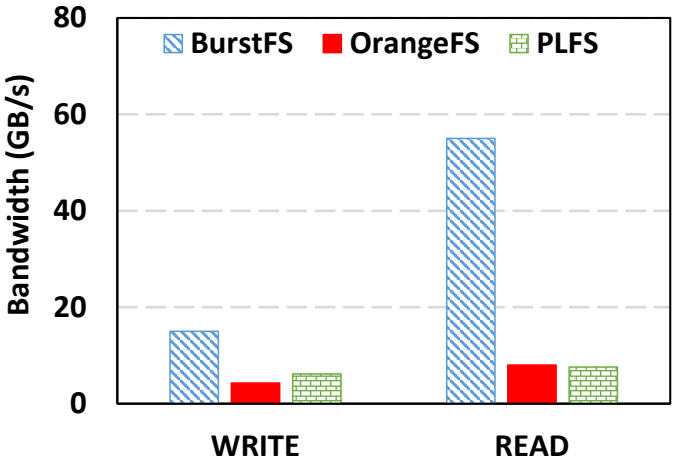


Fig. 11: Performance of Tile-IO.

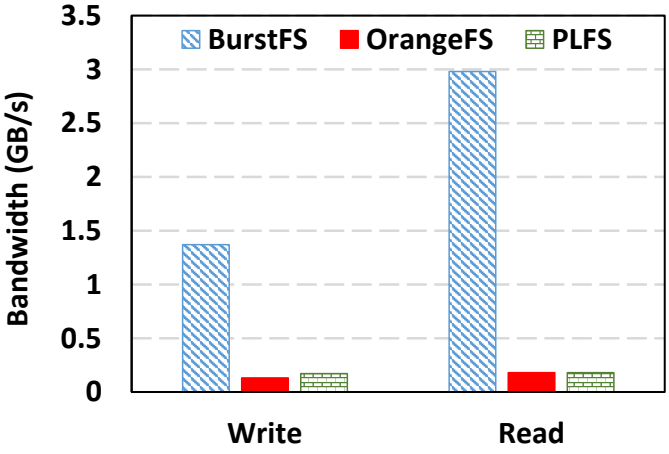


Fig. 12: Performance of BTIO.

Tile-IO. BurstFS sustains this small-message workload with the benefits of local reads and server-side read clustering. Overall, it delivers 15.6× and 9.5× performance improvement over OrangeFS for reads and writes, respectively. It also outperforms PLFS by 16.2× and 7×, respectively, for reads and writes.

G. IOR Test

In order to evaluate the support for data sharing among different programs in a batch job, we conduct a test with IOR. We run IOR with a varying number of processes reading a shared file written by another set of processes from a Tile-IO program. Processes in both MPI programs are launched in the same job. Each node hosts 16 Tile-IO processes and 16 IOR processes. Once Tile-IO processes complete writing on the shared file, this file is read back by IOR processes using the N-1 segmented read pattern. We keep the same transfer size of IOR as Tile-IO. Since the read pattern does not match the initial write pattern of Tile-IO, each process needs to read from multiple logs on remote nodes. We fix the size of each tile as 128MB and the number of tiles along Y axis as 4, and then increase the number of tiles along X axis. Thus the number of tiles on the X axis will increase along with the number of reading processes. Fig. 13 compares the read bandwidth of BurstFS with PLFS and OrangeFS. Both PLFS and OrangeFS are vulnerable to small transfer size (32KB). BurstFS maintains high bandwidth because of locally combining small requests and server-side read clustering and pipelining. On average, when reading data produced by Tile-IO, BurstFS delivers 2.3× and 2.5× the performance of OrangeFS and PLFS, respectively.

We also evaluate the read bandwidth of IOR over the dataset generated by BTIO, using two BTIO classes D and E. For Class D, we use 64 processes to write an array of 408 × 408 × 408 to a shared file. For Class E, 225 processes write an array of 1020 × 1020 × 1020 to a shared file. In both cases, the shared file is then read back by the IOR processes using the N-1 segmented read pattern. Fig. 14 shows the read bandwidth. Due to the much smaller transfer size (2040B for Class D and 2550B for Class E), the bandwidths of OrangeFS and PLFS with BTIO are much lower than with Tile-IO. While the performance of BurstFS is also impacted by the small

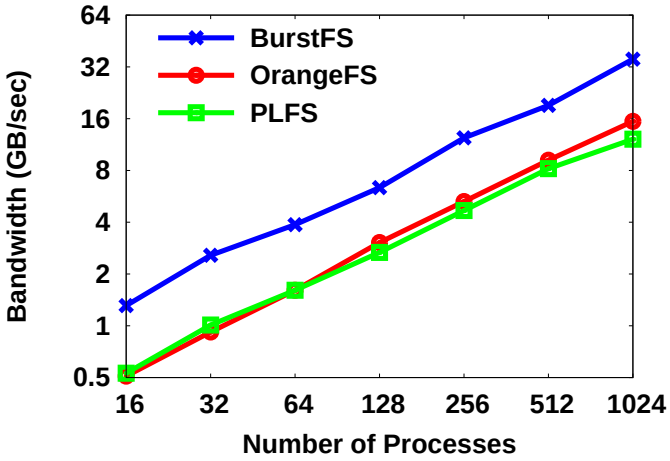


Fig. 13: Read bandwidth of IOR on the shared file written by Tile-IO.

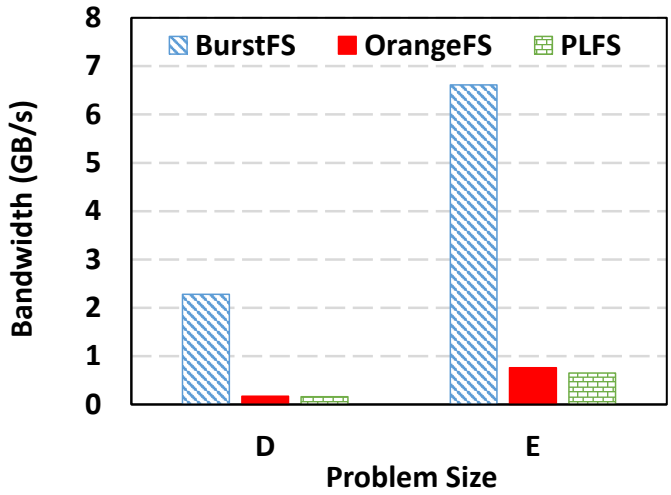


Fig. 14: Read bandwidth of IOR on the shared file written by BTIO.

transfer size, it delivers much better bandwidth to these small requests. On average, when reading data produced by BTIO, BurstFS delivers 10× and 12.2× performance improvement compared to OrangeFS and PLFS, respectively.

V. Related Work

The importance of burst buffers is shown by their inclusion in the blueprint of many next-generation supercomputers [2, 8, 9, 10, 11, 12] with a broad investment in supporting software. DataWarp [5], IME [7] and aBBa [1] are three ongoing projects in Cray, DDN and EMC. Their potential benefits have been explored from various research angles [25, 35, 42]. All these works target remote, shared burst buffers. In contrast, our work centers on node-local burst buffers, an equally important architecture that currently lacks standardized support software. Compared with work on remote burst buffers, our work delivers linear scalability for checkpointing/restart since most I/O requests are serviced locally. PLFS burst buffer [14] supports node-local burst buffers (see Section IV-A) and can deliver fast, scalable write performance. It relies on a global file system (e.g., Lustre, NFS) to manage metalinks, which can be an overhead if the number of metalinks is large. In addition, reading data from PLFS burst buffer requires each of the node-local burst buffers to be mounted across all compute nodes. BurstFS differs from PLFS burst buffer in that it is structured as a standalone file system. BurstFS achieves scalable read performance using the collective services of its delegators. Moreover, BurstFS is specialized for managing node-local burst buffers, while PLFS burst buffer supports both the node-local burst buffers and remote shared burst buffers.

The I/O bandwidth demand from checkpoint/restart has been increasing on par with the computing power. SCR [29], CRUISE [32] and FusionFS [48] are notable efforts designed to address this increasing I/O challenge and achieve linear write bandwidth by having each process write individual files to node-local storage (N-N). Different from these works, BurstFS supports both N-1 and N-N I/O patterns and delivers scalable read/write bandwidth for both patterns. Multidimensional I/O has long been a challenging workload for parallel file systems. The small, non-contiguous read/write requests issued from individual processes can dramatically constrain parallel file system bandwidth. Several state-of-the-art approaches have been developed to address this issue. PLFS accelerates small, non-contiguous N-1 writes by transforming them into sequential, contiguous N-N writes [15]. However, PLFS (without burst buffer support) still relies on a back end parallel file system to store the individual files from the N-N writes. Contention can occur when two files are striped on the same storage server. In contrast, BurstFS provides an independent file system service. It addresses write contention via local writes, and is optimized for read-intensive workloads. Two-phase I/O [38] is another widely adopted approach to optimize small, non-contiguous I/O workloads. All processes send their I/O requests to aggregators, which consolidate them into large, contiguous requests. The read service of BurstFS has some similarity to two-phase I/O: its delegators are akin to making the I/O aggregators used in two-phase I/O into a service. However, there are two key distinctions. First, the consolidations of BurstFS are directly conducted at the file system instead of the aggregators. This avoids the extra transfer from aggregators to client processes. Second, the consolidation is done by each delegator individually without extra synchronization overhead.

Cross-application data sharing is a daunting topic since many contemporary programming models (e.g. MPI, PGAS) define separate name spaces for each program. A widely adopted approach is leveraging existing distributed systems, such as distributed file systems (e.g. Lustre [17], OrangeFS [16], HDFS [37]) and distributed key-value stores (e.g. Memcached [30], Dynamo [21], BigTable [19]). However, these services are usually distant from computing processes, yielding limited bandwidth. In addition, the heavy overhead from start up, tear down, and management makes them unsuitable to be co-located with applications in batch jobs. On

the other hand, a couple of service programs are developed to be run with applications in batch jobs. Docan *et al.* [22] develop DART, a communication framework that enables data sharing via separate service processes located on a different set of nodes from the simulation applications (in the same job). Their later work DataSpaces [22] extends the original design. In both studies, application processes write to and read from the service process in an ad hoc manner. Each operation requires a separate network transfer. In contrast, the delegator in BurstFS is designed as an I/O proxy process co-located with application processes on the same node. All writes are local. Reads are deferred to the I/O delegator, which provides many opportunities to optimize the read operations.

VI. Conclusion

In this paper, we examined the requirements of data management for node-local burst buffers, a critical topic since node-local burst buffers are in the designs of next-generation, large-scale supercomputers. Our approach to managing node-local burst buffers is BurstFS, an ephemeral burst buffer file system with the same lifetime as batch jobs and designed for high performance with HPC I/O workloads. BurstFS can be used by multiple applications within the same job, sequentially as with checkpoint/restart, or concurrently as with ensemble applications. We implemented several techniques in BurstFS that greatly benefit challenging HPC I/O patterns: scalable metadata indexing, co-located I/O delegation, and server-side read clustering and pipelining. These techniques ensure scalable metadata handling and fast data transfers. Our performance results demonstrate that BurstFS can efficiently support a variety of challenging I/O patterns. Particularly, it can support shared file workloads across distributed, node-local burst buffers with performance very close to that for non-shared file workloads. BurstFS also scales linearly for parallel write and read bandwidth and outperforms the state-of-the-art by a significant margin.

Acknowledgment

We greatly appreciate Dr. John Bent from EMC for his guidance on configuring and running PLFS with burst buffer support. We are also thankful for the insightful comments from the anonymous reviewers.

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. LLNL-CONF-681480-DRAFT. This work was also supported in part by the National Science Foundation award 1561041.

References

[1] Active Burst Buffer Appliance. http://www.theregister.co.uk/2012/09/21/emc_abba//.

[2] Aurora. http://aurora.alcf.anl.gov/.

[3] Catalyst. http://computation.llnl.gov/computers/catalyst.

[4] Characterization and optimization of memory-resident mapreduce on hpc systems.

[5] Datawarp. http://www.cray.com/products/storage/datawarp.

[6] Hyperion. https://hyperionproject.llnl.gov/index.php.

[7] Infinite Memory Engine. http://www.ddn.com/products/infinite-memory-engine-ime/.

[8] NERSC-8. https://www.nersc.gov/users/computational-systems/cori/.

[9] Sierra. https://www.llnl.gov/news/next-generation-supercomputer-coming-lab.

[10] Summit. https://www.olcf.ornl.gov/summit/.

[11] Theta. https://www.alcf.anl.gov/articles/alcf-selects-projects-theta-early-science-program.

[12] Trinity. http://www.lanl.gov/projects/trinity.

[13] TSUBAME2. http://tsubame.gsic.titech.ac.jp/en/hardware-architecture.

[14] John Bent, Sorin Faibish, Jim Ahrens, Gary Grider, John Patchett, Percy Tzelnic, and Jon Woodring. Jitter-Free Co-Processing on a Prototype Exascale Storage Stack. In *IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–5. IEEE, 2012.

[15] 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*. ACM, 2009.

[16] Michael Moore David Bonnie, Becky Ligon, Mike Marshall, Walt Ligon, Nicholas Mills, Elaine Quarles Sam Sampson, Shuangyang Yang, and Boyd Wilson. OrangeFS: Advancing PVFS.

[17] Peter J Braam and R Zahir. Lustre: A Scalable, High Performance File System. *Cluster File Systems, Inc*, 2002.

[18] Michael J Brim, David A Dillow, Sarp Oral, Bradley W Settlemyer, and Feiyi Wang. Asynchronous Object Storage with QoS for Scientific and Commercial Big Data. In *Proceedings of the 8th Parallel Data Storage Workshop*, pages 7–13. ACM, 2013.

[19] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Berkeley, CA, USA, 2006. USENIX Association.

[20] Cristian Coarfa, Yuri Dotsenko, John Mellor-Crummey, François Cantonnet, Tarek El-Ghazawi, Ashrujit Mohanti, Yiyi Yao, and Daniel Chavarría-Miranda. An Evaluation of Global Address Space Languages: Co-array Fortran and Unified Parallel C. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 36–47. ACM, 2005.

[21] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 205–220. ACM, 2007.

[22] Ciprian Docan, Manish Parashar, and Scott Klasky. DataSpaces: An Interaction and Coordination Framework for Coupled Simulation Workflows. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 25–36, New York, NY, USA, 2010. ACM.

[23] Hugh Greenberg, John Bent, and Gary Grider. MDHIM: A Parallel Key/Value Framework for HPC. In *7th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 15)*, 2015.

[24] Jiahua He, Arun Jagatheesan, Sandeep Gupta, Jeffrey Bennett, and Allan Snavely. DASH: a Recipe for a Flash-Based Data Intensive Supercomputer. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE Computer Society, 2010.

[25] 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 *IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–11. IEEE, 2012.

[26] LLNL. IOR Benchmark. https://github.com/LLNL/ior.

[27] Jay Lofstead, Milo Polte, Garth Gibson, Scott Klasky, Karsten Schwan, Ron Oldfield, Matthew Wolf, and Qing Liu. Six Degrees of Scientific Data: Reading Patterns for Extreme Scale Science IO. In *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, pages 49–60. ACM, 2011.

[28] Ewing Lusk, S Huss, B Saphir, and M Snir. MPI: A Message-Passing Interface Standard, 2009.

[29] Adam Moody, Greg Bronevetsky, Kathryn Mohror, and Bronis R De Supinski. Design, Modeling, and Evaluation of a Scalable Multi-Level Checkpointing System. In *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, 2010.

[30] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. Scaling Memcache at Facebook. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 385–398, 2013.

[31] Sarp Oral, David A Dillow, Douglas Fuller, Jason Hill, Dustin Leverman, Sudharshan S Vazhkudai, Feiyi Wang, Youngjae Kim, James Rogers, James Simmons, et al. OLCFs 1 TB/s, Next-Generation Lustre File System.

[32] Raghunath Rajachandrasekar, Adam Moody, Kathryn Mohror, and Dhabaleswar K Panda. A 1 PB/s File System to Checkpoint Three Million MPI Tasks. In *Proceedings of the 22nd International Symposium on High-Performance Parallel and Distributed Computing*, pages 143–154. ACM, 2013.

[33] R. B. Ross. Parallel I/O Benchmark Consortium.

[34] Robert B Ross, Rajeev Thakur, et al. PVFS: A Parallel File System for Linux Clusters. In *Proceedings of the 4th annual Linux Showcase and Conference*, pages 391–430, 2000.

[35] Kiminori Sato, Kathryn Mohror, Adam Moody, Todd Gamblin, Bronis R De Supinski, Naoya Maruyama, and Shingo Matsuoka. A User-Level Infiniband-Based File System and Checkpoint Strategy for Burst Buffers. In *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, pages 21–30. IEEE, 2014.

[36] Frank B Schmuck and Roger L Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *FAST*, 2002.

[37] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10. IEEE, 2010.

[38] Rajeev Thakur, William Gropp, and Ewing Lusk. Data Sieving and Collective I/O in ROMIO. In *Frontiers of Massively Parallel Computation, 1999. Frontiers' 99. The Seventh Symposium on the*, pages 182–189. IEEE, 1999.

[39] Yuan Tian, Scott Klasky, Hasan Abbasi, Jay Lofstead, Ray Grout, Norbert Podhorszki, Qing Liu, Yandong Wang, and Weikuan Yu. EDO: Improving Read Performance for Scientific Applications through Elastic Data Organization. In *2011 IEEE International Conference on Cluster Computing*, pages 93–102. IEEE, 2011.

[40] Teng Wang, Kathryn Mohror, Adam Moody, Weikuan Yu, and Kento Sato. BurstFS: A Distributed Burst Buffer File System for Scientific Applications. In *Poster Presented at the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2015.

[41] Teng Wang, Sarp Oral, Michael Pritchard, Bin Wang, and Weikuan Yu. Trio: Burst buffer based i/o orchestration. In *2015 IEEE International Conference on Cluster Computing*, pages 194–203. IEEE, 2015.

[42] Teng Wang, Sarp Oral, Yandong Wang, Brad Settlemyer, Scott Atchley, and Weikuan Yu. BurstMem: A High-Performance Burst Buffer System for Scientific Applications. In *Big Data (Big Data), 2014 IEEE International Conference on*, pages 71–79. IEEE, 2014.

[43] Brent Welch, Marc Unangst, Zainul Abbasi, Garth A Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. Scalable Performance of the Panasas Parallel File System. In *FAST*, pages 1–17, 2008.

[44] Parkson Wong and R der Wijngaart. NAS Parallel Benchmarks I/O Version 2.4. *NASA Ames Research Center, Moffet Field, CA, Tech. Rep. NAS-03-002*, 2003.

[45] Jiangling Yin, Jun Wang, Jian Zhou, Tyler Lukasiewicz, Dan Huang, and Junyao Zhang. Opass: Analysis and Optimization of Parallel Data Access on Distributed File Systems. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 623–632. IEEE, 2015.

[46] W. Yu, J.S. Vetter, R.S. Canon, and S. Jiang. Exploiting Lustre File Joining for Effective Collective I/O. In *7th Int'l Conference on Cluster Computing and Grid (CCGrid'07)*, Rio de Janeiro, Brazil, May 2007.

[47] W. Yu, J.S. Vetter, and H.S. Oral. Performance Characterization and Optimization of Parallel I/O on the Cray XT. In *22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS'08)*, Miami, FL, April 2008.

[48] Dongfang Zhao, Zhao Zhang, Xiaobing Zhou, Tonglin Li, Ke Wang, Dries Kimpe, Philip Carns, Robert Ross, and Ioan Raicu. FusionFS: Toward Supporting Data-Intensive Scientific Applications on Extreme-Scale High-Performance Computing Systems. In *Big Data (Big Data), 2014 IEEE International Conference on*, pages 61–70. IEEE, 2014.