2018 IEEE/ACM 3rd International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems (PDSW-DISCS)

Integration of Burst Buffer in High-level Parallel I/O Library for Exa-scale Computing Era

Kaiyuan Hou
Northwestern University
Evanston, IL, USA
khl7265@eecs.northwestern.edu

Reda Al-Bahrani
Northwestern University
Evanston, IL, USA
rav650@eecs.northwestern.edu

Esteban Rangel
Northwestern University
Evanston, IL, USA
emr126@eecs.northwestern.edu

Ankit Agrawal
Northwestern University
Evanston, IL, USA
ankitag@eecs.northwestern.edu

Robert Latham
Argonne National Laboratory
Lemont, IL, USA
robl@mcs.anl.gov

Robert Ross
Argonne National Laboratory
Lemont, IL, USA
rross@mcs.anl.gov

Alok Choudhary
Northwestern University
Evanston, IL, USA
choudhar@eecs.northwestern.edu

Wei-keng Liao
Northwestern University
Evanston, IL, USA
wkliao@eecs.northwestern.edu

***Abstract*— While the computing power of supercomputers continues to improve at an astonishing rate, companion I/O systems are struggling to keep up in performance. To mitigate the performance gap, several supercomputing systems have been configured to incorporate burst buffers into their I/O stack; the exact role of which, however, still remains unclear. In this paper, we examine the features of burst buffers and study their impact on application I/O performance. Our goal is to demonstrate that burst buffers can be utilized by parallel I/O libraries to significantly improve performance. To this end, we developed an I/O driver in PnetCDF that uses a log-based format to store individual I/O requests on the burst buffer – later to be flushed to the parallel file system as one request. We evaluated our implementation by running standard I/O benchmarks on Cori, a Cray XC40 supercomputer at NERSC with a centralized burst buffer system, and Theta, a Cray XC40 supercomputer at ALCF with locally available SSDs. Our results show that IO aggregation is a promising role for burst buffers in high-level I/O libraries.**

Keywords— burst buffer; Log; Exa-scale

I. INTRODUCTION

Historically, storage technology has struggled to keep pace with processing power. As CPUs have become faster each year, hard disks -- although increasing largely in capacity -- have not improved much in access speed [1]. This disparity has created an increasingly challenging environment for modern High-Performance Computing (HPC) systems, where fast computation is paired with slow I/O. The I/O bottleneck heavily affects the performance of many scientific simulations, as they typically generate large amounts of data. The next generation (exascale) systems will likely have in excess of 10 petabytes of system memory with applications generating datasets of similar size -- as large as an entire Parallel File System (PFS)'s capacity in 2009. Without improvements to I/O systems, slow disk speeds will make accessing file systems infeasible. To counter this trend, strategies to incorporate faster devices into the I/O stack have been proposed [2,3,4,5]. The most common strategy is the use of Solid State Drives (SSDs) a new type of storage device that is faster than the hard disk. Modern HPC systems are now starting to incorporate SSDs as a burst buffer that is accessible to users.

While burst buffers are now becoming more common on supercomputers, the use of this new technology has not been fully explored. In the literature, burst buffers have been used by file systems [6] and lower level I/O middleware such as MPI-IO [7]. To our best knowledge, burst buffers have not yet been utilized in high-level I/O libraries, i.e., I/O libraries that are built on top of other libraries to expose a high-level abstract interface, rather than merely byte offset and count as in POSIX I/O. Many high-level I/O libraries [8,9,10] were designed at a time when hard disk-based PFS was the primary storage target. Since SSDs have different characteristics from hard disks, some design choices made at that time may no longer be ideal when burst buffers are available. As a result, for high-level I/O libraries to utilize the new hardware effectively, new approaches are needed for the modern IO hierarchy.

In this paper, we explore the idea of using a burst buffer to do I/O aggregation in a high-level I/O library. Due to the physical limitations of hard disks, and the way modern PFS handle parallel I/O operations, writing large and contiguous chunks of data is more efficient than writing many small and more fragmented ones [11]. We conclude that the burst buffer is an ideal device to aggregate I/O requests; by storing the data of write operations on the burst buffer to later flush, we have the opportunity to turn many small write requests into larger ones without consuming limited system memory space. This concept is already proven effective in lower-level I/O middleware [7].

We developed a lightweight module in the PnetCDF (Parallel-NetCDF) [8] library to make use of burst buffers for I/O aggregation. The module intercepts write requests from the user application and saves them in the burst buffer using a log-

978-1-7281-0192-7/18/$31.00 ©2018 IEEE
DOI 10.1109/PDSW-DISCS.2018.00004

based structure. A log entry is created to record every write request in a high-level abstract representation, which is later used to generate aggregated I/O requests to the PFS.

Performing I/O aggregation in the higher levels of the I/O hierarchy has many potential advantages. The I/O requests generated by many scientific simulations are often subarrays of multi-dimensional arrays. Aggregating at a high level allow us to retain the structure of the original data. It can then be used by either the in-situ components of the application or the I/O library itself to improve performance or to support additional features. In section 3 for example, we demonstrate how to utilize this structured information to resolve a major performance issue we encountered in our implementation. Applications performing in-situ analysis can also benefit by accessing the original data structure directly from the log files stored in burst buffers. I/O request aggregation at a high level reduce the memory footprint by preventing one subarray I/O request from being translated into many noncontiguous requests when it is flattened out to a list of offsets and length representations. Also, being closer to the application reduces the depth of calling the stack, and hence the overhead.

Despite the benefits, I/O aggregation at the high level faces some challenges. One of them comes from the limitation of MPI-IO which PnetCDF is built on top of. A single MPI write call only allows the flattened file offsets in a monotonically non-decreasing order, if the request contains multiple noncontiguous file regions. As a result, aggregation by simply appending one write request after another may violate the MPI-IO file offset order requirement. However, flattening all requests into offsets and lengths in order to reorganize the request offsets to abide the MPI-IO requirement can be expensive. To avoid the costly flattening operation, we calculate the first and last offsets of the access regions which are sufficient to tell whether two requests overlap. This strategy allows us to perform flattening only for the overlapping requests. In fact, enabling aggregation while maintaining the I/O semantics can make the burst buffering layer infeasibly complex. We make use of the semantics of user's I/O intention to design a solution that balances transparency and efficiency.

We ran experiments on two supercomputers: Cori at the National Energy Research Scientific Computing Center (NERSC) and Theta at the Argonne Leadership Computing Facility (ALCF). We used IOR [12], FLASH I/O [13] and BTIO [14] benchmarks for evaluation. We compared the performance of our approach with the case where a burst buffer is not involved, aggregation at lower level, as well as using the burst buffer directly as a high-speed file system and copying the data to the PFS later. The experiments show that our burst buffer driver can increase the performance up to 4 times. It suggests that I/O aggregation is a very useful feature in a high-level I/O library, and burst buffers are the ideal tool for the job.

II. RELATED WORK

Many supercomputer vendors have introduced their burst buffer solutions. One of the widely adopted solutions is Cray DataWarp. DataWarp is a centralized architecture that uses SSDs to provide a high-speed storage device between the application and the PFS in hope to decouple computation and PFS I/O operation [15]. The burst buffer is mounted as a file system on computing nodes. By default, DataWarp operates in striped mode. The data is striped across the burst buffer servers. All processes running a job share the same space. Files created on the burst buffer are accessible by all processes. We refer to it as shared mode in this paper. Under this mode, there is only 1 metadata server serving all the compute nodes. If data sharing is not needed, DataWarp can be set to operate in private mode in which it works like a node-local burst buffer. Files created by a process will not be visible to other processes, however, the capacity of the burst buffer is still shared. The metadata workload is distributed across all metadata servers, preventing the metadata server from becoming the bottleneck.

Bhimji et al. studied the characteristics of DataWarp on well-known applications and concluded that it delivers significantly higher performance compared to the PFS [2,16]. Hicks introduced a client-side caching mechanism for DataWarp as DataWarp currently has no client-side caching [17]. Dong et al. proposed a way to transfer data on the burst buffer to the PFS using compute nodes [18]. They found that moving the file to the PFS using compute nodes can be faster than having DataWarp staging out the file because of the limited number of burst buffer servers that must serve all the compute nodes. Moving data with compute nodes increases the overall performance and reduces resource contention on burst buffer servers. Kougkas et al. introduced a distributed buffering system that automatically manages data movement between different memory layers and storage hierarchy. Their system covered a wide range of storage devices including non-volatile memory, burst buffer and PFS [19].

The role of the burst buffer is also explored by other researchers. The most popular is using the burst buffer as a cache to the PFS. Most applications do not maintain a constant I/O demand throughout the execution, instead, I/O requests tend to arrive in a burst followed by a period of silence [20]. Placing a buffer before the PFS helps smooth out the peak of the demand, allowing more efficient utilization of PFS bandwidth. Wang et al. introduced a log-styled data structure to record write requests from the user application and a mechanism that flushes the data to the PFS in the background [6]. A similar concept was proposed by Sato et al. [21] where they utilized the burst buffer to develop a file system that aims to improve check-pointing performance. Kimpe et al. introduced a log-based buffering mechanism for MPI-IO called LogFS [7]. It records low-level I/O requests in a log-based data structure on the burst buffer. Requests are indexed in an R-tree structure of the start and end offset they access. To generate non-decreasing aggregated request, they sort every branch of the R-tree into increasing order. LogFS is implemented either as a standalone library or as an MPI-IO module. The common design characteristic of these approaches is that they put the burst buffer at a lower level in the I/O stack. At such level, the information provided is low-level offset and count instead of abstracted high-level descriptions such as variable and range.

Log-based data structures have long been used to organize data on a storage media. Bent et al. developed a virtual parallel log structured file system that remaps the preferred data layout of user applications into one which is optimized for the underlying file system [22]. Rosenblum and Ousterhout

designed a file system that stores data using a log-based structure [23]. Dai et al. designed a log-structured FLASH I/O file system that is tailored for buffering sensor data in microsensor nodes [24].

III. Background

A. What is PnetCDF

PnetCDF is a high-level parallel I/O library providing high-performance parallel access to NetCDF [9,25] files. It has been publicly available since 2003. It uses MPI-I/O [26] to access data on the PFS. The library is made up of a unified API layer that sits on top of many extensible I/O modules called drivers (an I/O module in the library, not OS driver). Drivers are used to interact with a specific lower-level library, to apply a specific I/O strategy, or to support a specific file format. This modular design makes it easy to develop extensions, such as burst buffer support in this work, for the library. PnetCDF provides functions to aggregate I/O requests using the memory to increase performance [27]. It is shown that this approach can significantly improve I/O performance under most scenarios [28].

A NetCDF file can contain variables that are logically structured in a multi-dimensional array of a data type. The size of the variable is described by dimension entities defined in the file. Attribute entities can be added as annotations to individual variables as well as the entire file. Unlike accessing the file using lower-level interface where the user specifies offset and length, NetCDF files are accessed in terms of variables, array indices, shape, and other high-level description of the I/O request.

B. Burst Buffer Architectures

There are two common types of burst buffer architectures in modern supercomputers. One is node-local burst buffer where computing nodes are equipped with an SSD as local storage. The SSD is usually managed by the local file system that provides the traditional POSIX I/O interface. The other architecture is a shared burst buffer where a group of burst buffer servers use SSDs or other fast storage devices to provide a high-speed storage service that is shared by all computing nodes in a way similar to the PFS. In this way, the burst buffer resource can be concentrated on jobs in need. The data will remain available after a node fails. Some implementations can also be configured to mimic node-local burst buffer.

IV. Design of Burst Buffer Layer

We implemented the burst buffer I/O driver in PnetCDF for I/O aggregation. It intercepts write requests from the user application and records them on the burst buffer. Other requests such as file initialization and attribute write are carried out on the PFS as usual. The only difference in the file produced by the burst buffer driver is the I/O pattern. Such transparency eliminates concern on compatibility. The burst buffer driver can be enabled by an environment variable without the need to modify the application. In this way, it benefits legacy programs that were not optimized for modern storage hierarchy. The burst buffer driver allows application developers to write data in whichever manner fits their needs with less concern on performance.

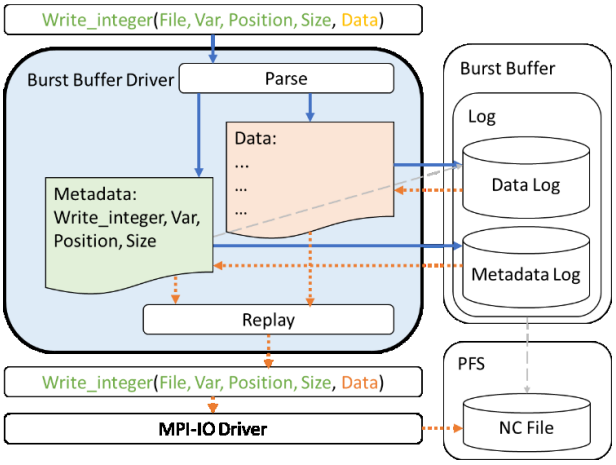

Fig. 1. Handling variable write request. The burst buffer driver is shaded in light blue. Data are shaded in light green while metadata re shaded in pink. Blue arrows show the data flow when writing to the burst buffer. Orange arrows show the data flow when flushing to the PFS.

Figure 1 gives a summary of the design. Our burst buffer driver is marked blue with a bolded outline. Solid green arrows show the traditional data flow of PnetCDF. Dashed blue arrows and red arrows show the data flow of buffering on burst buffer and flushing to PFS respectively.

A. File Creation

When a NetCDF file is opened, the burst buffer driver initializes a log consisting of two log files on the burst buffer: a metadata log, and a data log. The data log stores the request data of I/O requests while the metadata log stores other information describing the I/O request. Log entries are recorded in pairs; a metadata log entry always binds to a data log entry and vice-versa. Each pair of log entries represents an I/O operation made by the application. Successive entries are appended to the log file in a linear fashion. Since request data is only needed when flushing the log entries, separation of request metadata and request data allows efficient traversal of metadata entries without the need to do file seeks. It also eliminates the need to reorganize the data before sending it to the PFS when flushing the log.

In order to work efficiently on different burst buffer architectures and configurations, we support two log file to process mappings - log-per-process, and log-per-node. In the log-per-process mapping, each process will create its own set of log files. Without the need to coordinate with other processes, data can be accessed in an efficient way. One concern of log-per-process mapping is that when the number of processes is large, the workload of log file creation can overwhelm the metadata server, causing performance issues. This problem affects centralized burst buffers the most as the number of burst buffers does not increase when the application scales. For example, the DataWarp on Cori uses only one metadata server to serve one job. The metadata server becomes a bottleneck when every process tries to create log files. Because of such

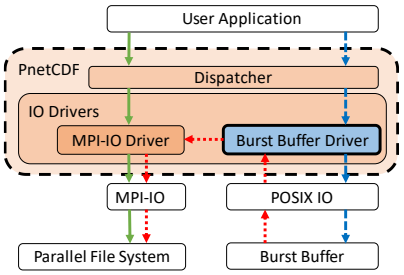

Fig. 2. An overview of burst buffer driver. Blue arrows show the data flow when writing to the burst buffer using the burst buffer driver. Red arrows show the dataflow when flushing the data to the PFS. Green arrows show the original data flow when the burst buffer is not in use.

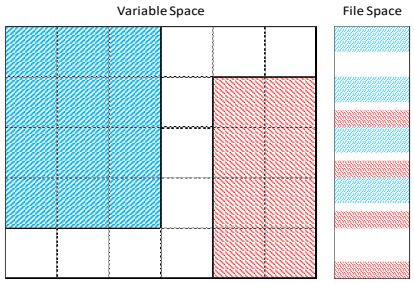

Fig. 3. Variable Space vs. File Space. Two submatrices marked blue and red are completely separated in a 2D variable. However, their file space interleves with each other.

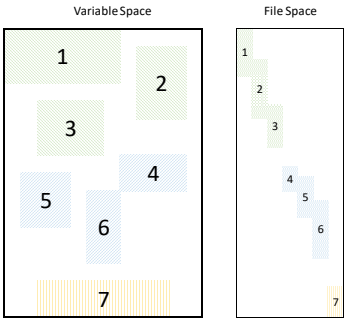

Fig. 4. Two-level reordering strategy. Submatrices are colored according to their group. For submatrices in different groups, they are guaranteed to not interleve each other in the file space.

issue, log-per-process mapping only works well on node-local burst buffer or in architectures where the workload is distributed. We will demonstrate this problem in the experiment section.

Since we are likely to have a large number of processes in an exascale environment, we need to reduce the metadata workload in case we are using a shared burst buffer. Log-per-node mapping is designed to alleviate the issue faced by log-per-process mapping on initialization by reducing the number of files created. Processes in the same node share the same pair of log files. The file is divided into blocks and those are assigned to processes in a round robin fashion so that every process has its own space as if having its own log file. Modern supercomputers can hold tens of processes in a single node. The number of log files created can be reduced significantly, mitigating metadata workload issues. However, file sharing may affect the performance of data access. For applications that do not suffer from file creation overhead, log-per-node mapping may not be beneficial.

B. *I/O Operations*

PnetCDF only involves the burst buffer for write requests. It intercepts variable write requests and records them in the log on the burst buffer. Other request types including file initialization, adding attributes, defining dimensions, and read operations pass through to the default I/O driver. They are carried out directly on the PFS in the usual way as the burst buffer driver is not involved. If the I/O request tries to read the data that is still buffered in the log, the log is flushed to bring the NetCDF file on the PFS up to date before carrying out the operation. Since NetCDF files are mostly used to store checkpointing data or simulation results, we do not expect read after write scenario to happen frequently to the extent that causes performance problems for doing a flush before any variable read. We keep track of several file properties that can be affected by buffered variable data such as the size of the number of records in a record variable in order to prevent flushing the log on simple file property querying operation. The log is also flushed when the file is closed, when the user performs a file sync, and when the user explicitly requests a flush.

Figure 2 summarizes the process to record a variable write call and its reconstruction when flushing. Blue arrows indicate the data flow when the application makes a write API call. Red dashed arrows show the data flow when flushing the log. Thin dashed gray arrows indicate a pointer that links the metadata to the data as well as log file to NetCDF file.

C. *Log File Format*

The flexibility of PnetCDF interface brings another challenge to log format design. The user can write to a variable using a variety of operations. A log format that records all types of operations will be too complicated to implement as a light-weighted module. Due to this concern, our log only records one category of operations – writing to a subarray of a variable. We assume every write operation writes to a rectangular sub-region within a variable. An optional stride can be specified along each dimension to skip cells in between. Operations that do not fit this format are being decomposed into many operations that fit. We also translate all user-defined datatypes to native types before recording the operation in the log to eliminate the need to deal with derived types.

The metadata log consists of a header followed by metadata entries. The header records information about the NetCDF file as well as the number of log entries. Each log entry describes one I/O operation. It includes the type of the operation, the variable involved, the location within the variable, the shape of the subarray to write, and the location of corresponding data log entry. Since variables have a different number of dimensions, the size of the metadata log entry varies. For performance consideration, a copy of metadata log is cached in the memory during the entire file open session. The log file is purely a backup in case of interruption.

The header of the data log does not contain any information except for a serial for identification purpose. A data log entry only records the data passed by the application in the request related to its corresponding metadata entry. The data in the data log is in the native representation of the underlying hardware, enabling us the possibility to read from the data log directly without the need of conversions. For example, an analysis program running in parallel can retrieve the data directly from the data log without flushing the log to the PFS. Although it is not implemented in the current version of the burst buffer driver,

we are considering supporting such functionality in future releases.

D. Flush Data to PFS

Although the data on the burst buffer can be flushed by simply repeating every I/O operation recorded in the log, doing so will not provide any performance advantage but only the overhead of a round trip to the burst buffer. Instead, we want to combine all the recorded requests into one. This aggregation step is the key to performance improvements in our burst buffer driver.

Unfortunately, MPI-IO, which PnetCDF is built on, requires that the displacement of each file section accessed within a single I/O request needs to be monotonically non-decreasing [29]. This restriction prevents us from taking the straight-forward solution of appending all aggregated requests one after another. Instead, we can only aggregate consecutive requests where file offsets accessed by a request are all before that accessed by later requests. Since variables in a NetCDF file are stored one after another, aggregation is only possible when the application accesses them in the order they are stored. Furthermore, a subarray of a high-dimensional variable, when being flattened into offset and length, maps to many non-contiguous regions that span across a great range. Even two non-overlapping subarrays can produce decreasing access offset when stacked together.

Figure 3 demonstrates this scenario. The blue and red subarrays do not logically intersect each other, but the range they access in the file space do intersect. Due to the reason above, the chance of applying successful aggregation becomes very slim even under usual I/O patterns. In the worst I/O pattern, we are set back to do one request at a time, making buffering meaningless.

To overcome the problem above, we want to rearrange the data so that every offset and length in the aggregated request are in increasing order. However, the computational cost can be prohibitive. Because the way variables are stored in the file, it is possible to have 2 requests in which neither of them accesses only regions before all accessed regions of the other. In such case, these two requests cannot be put together no matter the order. We call these 2 requests interleaved with each other. To deal with interleaved requests, we need to break the requests down into offset and length for reordering. To do so, we need to flatten all requests out into a long list of offsets and lengths and sort them into non-decreasing order. Sorting such huge amount of offset and length imposes a significant drag on the performance. Sorting also takes additional memory that can otherwise use to achieve larger aggregation size.

To mitigate the impact of sorting, we take advantage of the original data structure to eliminate unnecessary reordering tasks. Since each request is writing to a rectangular array, every offset it accesses is already in non-decreasing order. Reordering regions within a request is totally unnecessary. Also, if two requests are not interleaving each other, we can simply reorder them as a whole; there is no need to break them down for reordering. Using these properties, we came up with a two-level reordering strategy, a high-level reordering followed by a low-level reordering. We start by sorting requests by the first byte accessed. It ensures that any 2 adjacent requests that do not interleave each other can be safely stacked together as an aggregated request. Later, we scan through the sorted request for interleaved requests. This can be done by comparing the offset of the last byte accessed by one request to the offset of the first byte accessed by the other. Whenever 2 consecutive requests interleave each other, we combine them as a large request which can also interleave with later requests. A low-level reordering is then performed within each of the combined requests to make sure those offsets are in non-decreasing order. Finally, we can safely stack all requests together as a single aggregated request.

The method described above is demonstrated in Figure 4. Each submatrix corresponds to one request. The number indicates its order after high-level reordering. Requests of the same color are in the same group for low-level reordering if there is more than one request within the group. Requests in different color do not intersect each other in file accessing range. After reordering requests within each group, it is safe to stack all request together to obtain aggregated request.

In practice, we may still not be able to aggregate all requests together due to buffer size limitation. The driver continues to include requests in the order they are recorded until the buffer space runs out. It then reads from the log for data and descriptions of the requests to construct an aggregated request. Since the amount of data, as well as buffer size limit, can be different on each process, some processes may need to flush the data in more rounds than others, processes must agree on the number of rounds required to flush in collective I/O mode.

V. Experiments

We ran the experiments on two supercomputers, Cori and Theta. Cori is a Cray XC40 supercomputer at NERSC [30,31]. It has 2,388 nodes connected by Cray Aries with Dragonfly topology providing 5.625 TiB/s global bandwidth. Each node has 2 Intel® Xeon™ E5-2698 v3 processors providing 32 cores/64 threads that are matched with 128 GB DDR4 2133 MHz memory. The parallel file system used on Cori is Cray Sonexion 2000, a Lustre with 248 OSTs on 248 servers. Together, they can deliver 744 GiB/s of peak bandwidth. The DataWarp implementation on Cori a centralized burst buffer consisting of 288 servers shared by all compute nodes and providing 1.7 TiB/s of peak I/O performance. At the time of our experiments, the burst buffer was in its first stage where the only supported type of data management operations were copying the entire file form and to the PFS. In the experiments, we used the shared mode configuration and then sml_pool which has 80 servers. We set stripe count to 64 on both the burst buffer and Lustre. The stripe size on Lustre is set to 8 MiB to match the 8MiB stripe size of the burst buffer. We ran 32 MPI processes per node on Cori.

Theta is another Cray XC40 supercomputer at ALCF. It consists of 4,392 nodes connected by a 3-layer Aries Dragonfly network. Each node is equipped with a 64 core Intel KNL 7230 processor, 16 GiB of MCDRAM, 192 GiB of DDR4 memory. Unlike Cori, Theta does not have a centralized burst buffer file system. Instead, there is an SSD in each cabinet that can be accessed locally by nodes in the cabinet. The model of the SSDs is either Samsung SM951 or SM961 SSD. The Lustre on Theta

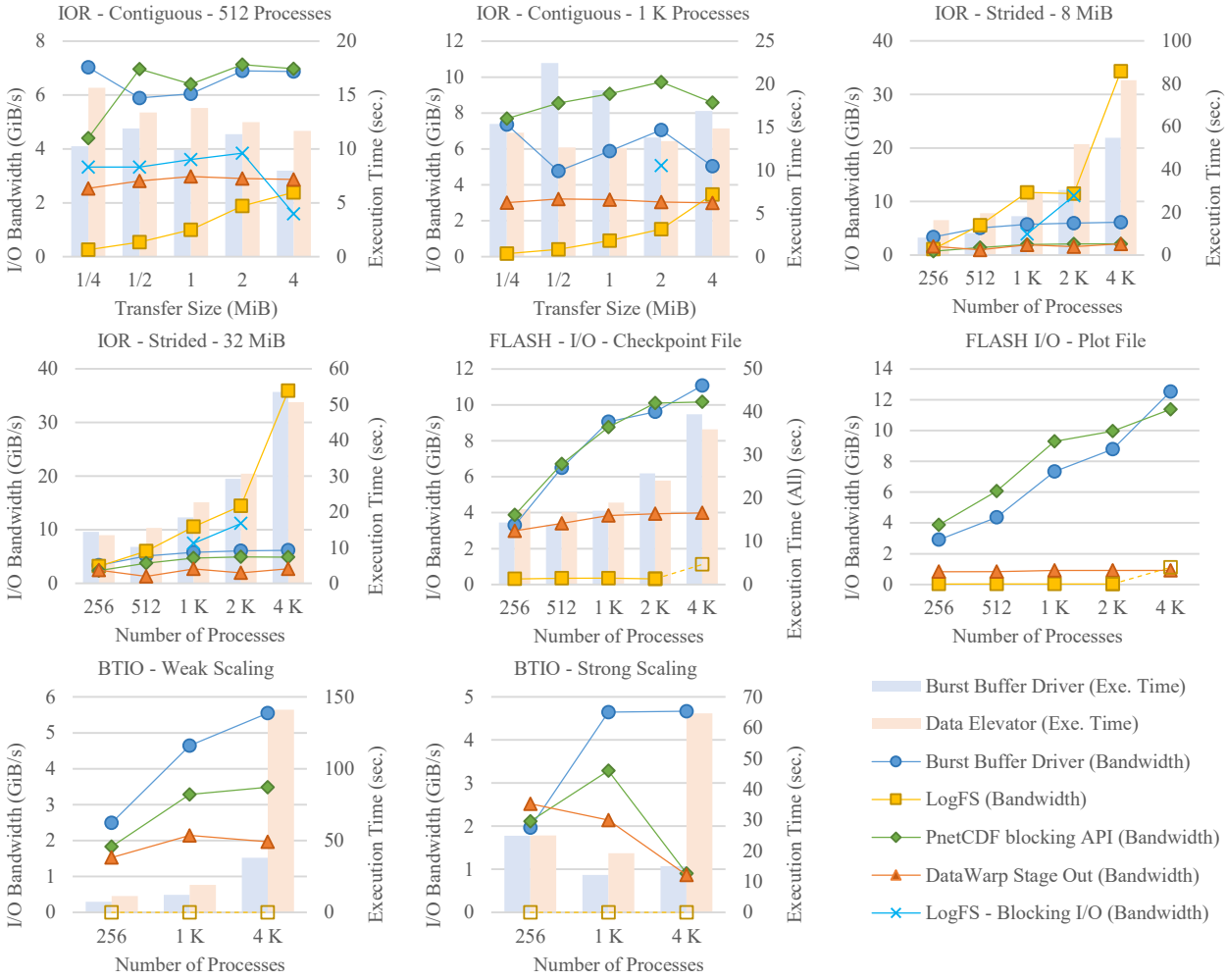

Fig. 5. Bandwidth and Execution Time on Cori. The lines show the andwidth measured by the benchmark porograms. The bars show the end to end execution time of each benchmark including time spent on computation. In case the benchmark program did not finish within a reasonable time, we present the bandwidth approximate by actuall size written in dashed lines and hollow markers. Cases discussing LogFS characteristic is marked in ×

has 56 OSTs. We used 8 MiB stripe size and stripe count of 56 in the experiment. We ran 64 processes per node on Theta. We ran the experiment on both machines using up to 4,096 processes. To mitigate interference from system loading, every single experiment is repeated at least twice. The best run is selected to represent the results.

A. I/O Benchmark

Our performance evaluation uses three well-known I/O benchmarks: IOR [12], FLASH I/O [13] and BTIO [14]. IOR is a well-known synthetic benchmark used within the I/O community to test the performance of PFS with configurable access patterns through a wide variety of interfaces. The file is made up of segments and each segment is made up of blocks. Each process is responsible to write 1 block within a segment. In each iteration, processes write a fixed amount called transfer size to its corresponding block until it is filled. The processes then move on to the next segment [32]. We use IOR to simulate 2 different patterns: contiguous and strided. While running the contiguous pattern the block size and the transfer size are set to be equal, making the aggregated access domain contiguous. The workload resembles the pattern of appending records to the file, a common I/O pattern in parallel-computing jobs. For the strided pattern, we write only 1 segment. Transfer size is set to a divisor of block size so that each process writes a portion of its block in each iteration. This setting creates a strided access pattern in the file space which is expected to cause performance problems when doing small I/O without aggregation as the access domain maps only to a subset of file servers while leaving other servers idle. In both cases, we write a total of 64 MiB per process. We used various transfer sizes to study the behavior of our model under different I/O request sizes.

The FLASH I/O benchmark [13] is extracted from the I/O kernel of a block-structured adaptive mesh hydrodynamics simulation program base on FLASH I/O [33, 34], a modular and extensible set of physics solvers, maintained by The FLASH I/O Center for Computational Science's [35]. The computational domain is a 3-dimensional array divided into equal sized blocks. Each process holds 80 to 82 blocks to simulate load imbalance. The block size, which is configurable, along with the number of processes determines the total I/O size. The simulation space is represented by 24 variables. It generates a checkpoint file

containing all variables and 2 plot files containing only 4 variables. We set the block size to 16 x 16 x 16, resulting in roughly 75 MiB of write amount per process. This amount written by each process consists of the size of checkpoint file and the 2 plot files.

BT-I/O [14] is derived from the Block Tri-diagonal (BT) solver as part of the NAS Parallel Benchmarks (NPB) [36]. The BT solver involves a complex diagonal multi-partitioning in which each process is responsible for multiple Cartesian subsets of the entire data set. Such a division often results in a high degree of fragmentation at the output stage where all processes need to combine their fragmented regions into one shared file [14]. The I/O size of BTIO is controlled by the size of the global mesh point array as well as the number of iterations. Due to the unique way BTIO distributes data across processes, it exhibits a complex file layout and access patterns. We studied both weak and strong scaling cases in BTIO. In the strong scaling experiment, the array size was set to 512 x 512 x 512 and number of iterations to 8. This setting produced 5 GiB per iteration and a total size of 40 GiB. In the weak scaling experiment, the first 2 dimensions of the array were adjusted so that each process writes 40 MiB. Other settings were kept the same as the strong scaling case.

B. Compare with PnetCDF Blocking I/O

To evaluate performance gains in I/O aggregation using the burst buffer, we compare the bandwidth of each benchmark against PnetCDF blocking collective APIs. The results from Cori and Theta are shown in Figure 5 and Figure 6 respectively.

As we expected, the performance on IOR strided pattern without aggregation is poor unless the size of individual fragment is very large. The burst buffer driver improves the situation significantly by aggregating small writes into a whole block. Once the combined region across all processes covers the entire file, the strided I/O pattern is turned into the more efficient contiguous pattern. The burst buffer driver improves the performance by up to 4.2 times on Cori and 3.8 times on Theta over the blocking collective I/O using small transfer size. Even when the transfer size becomes large, the burst buffering still outperforms the collective I/O by a large margin. However, for contiguous patterns, the degree of improvement by burst buffering declines. Since the contiguous pattern is already near ideal, the benefit of aggregation becomes less significant except when the transfer size is very small.

For FLASH I/O, the burst buffer driver does not show noticeable performance improvements. The reason is similar to the IOR contiguous pattern. Since FLASH I/O divides its file space in an interleaved fashion, the combined region in a single PnetCDF collective I/O operation forms a single large and contiguous block – an ideal I/O pattern. As a result, the remaining benefit of aggregation is only the potential reduction in the number of I/O requests to the PFS. When the number of processes becomes large enough to generate a combined size that saturates the aggregator buffer, the marginal advantage is further reduced to merely a saving of collective I/O initialization. The burst buffer driver performs slightly slower than the blocking collective I/O, except for the case of 4,096 processes where the cost of collective I/O initialization starts to overtake the overhead of burst buffer aggregation.

The advantage of the burst buffer driver is most obvious in BTIO. The complex file layout of BTIO results in a non-contiguous I/O pattern similar to that in IOR strided mode. This pattern usually results in poor parallel I/O performance for PFS even when collective buffering is used. The burst buffer driver improves the performance by combining and reordering complex I/O patterns into one that is favorable to the file system. In the weak scaling experiment, the burst buffer driver increases the I/O bandwidth by up to 37% on Cori and 16% on Theta.

In the strong scaling experiment, the performance of blocking collective I/O decreases when running on a large number of processes. The reason behind that is the way BTIO divides the global array into small, non-contiguous requests. Moreover, the BTIO takes more steps to write a single global variable when it is accessed in parallel by more processes. The burst buffer driver remedies this by combining those fragmented I/O requests into a contiguous one, largely improving the scalability. In this case, the burst buffer driver increases the write bandwidths of up to 2.6 times on Cori and 4.3 times on Theta.

In nearly all cases where raw PnetCDF I/O performance is not ideal, the burst buffer driver can improve I/O performance by a significant margin. It suggests that I/O aggregation is a simple yet very effective technique to increase I/O performance in an I/O library. Regarding this, we recommend that other high-level I/O libraries should also provide the feature or APIs to enable request aggregation. Doing so may significantly improve the overall I/O bandwidth of the entire HPC system, allowing hardware resource to be used more efficiently. In addition, the log-based data structure on the burst buffer is a good solution to the task.

C. Compare with DataWarp Stage Out

Aside from I/O aggregation, the burst buffer can also be used in other ways. In a centralized burst buffer architecture, a straight-forward method to utilize the burst buffer is to use it in place of the PFS. Instead of doing I/O on the PFS, we do it on the burst buffer. The files on the burst buffer are then copied to the PFS when the job finishes. It is interesting to find out how the performance of this approach compares to using the burst buffer driver. Since this approach requires a centralized burst buffer, we only evaluate it on Cori. We compare the bandwidth running each of the benchmarks using our burst buffer driver using traditional PnetCDF API on top of the burst buffer. The time copying the files from the burst buffer to the PFS is counted toward I/O time. We also want to compare the time taken to write the data on the burst buffer as well as the time taken to move the data from the burst buffer to the PFS using these two methods.

Contrary to our intuition, using the burst buffer as a PFS does not always improves the performance. In many cases, it instead decreases the overall performance of all the benchmarks regardless of configurations. The reason is that most of the time is not spent on writing the file to the burst buffer but on copying it to the PFS using the function provided by the burst buffer server. The timing breakdown is shown in Figure 7. Since there are only a fixed number of burst buffer servers that must be

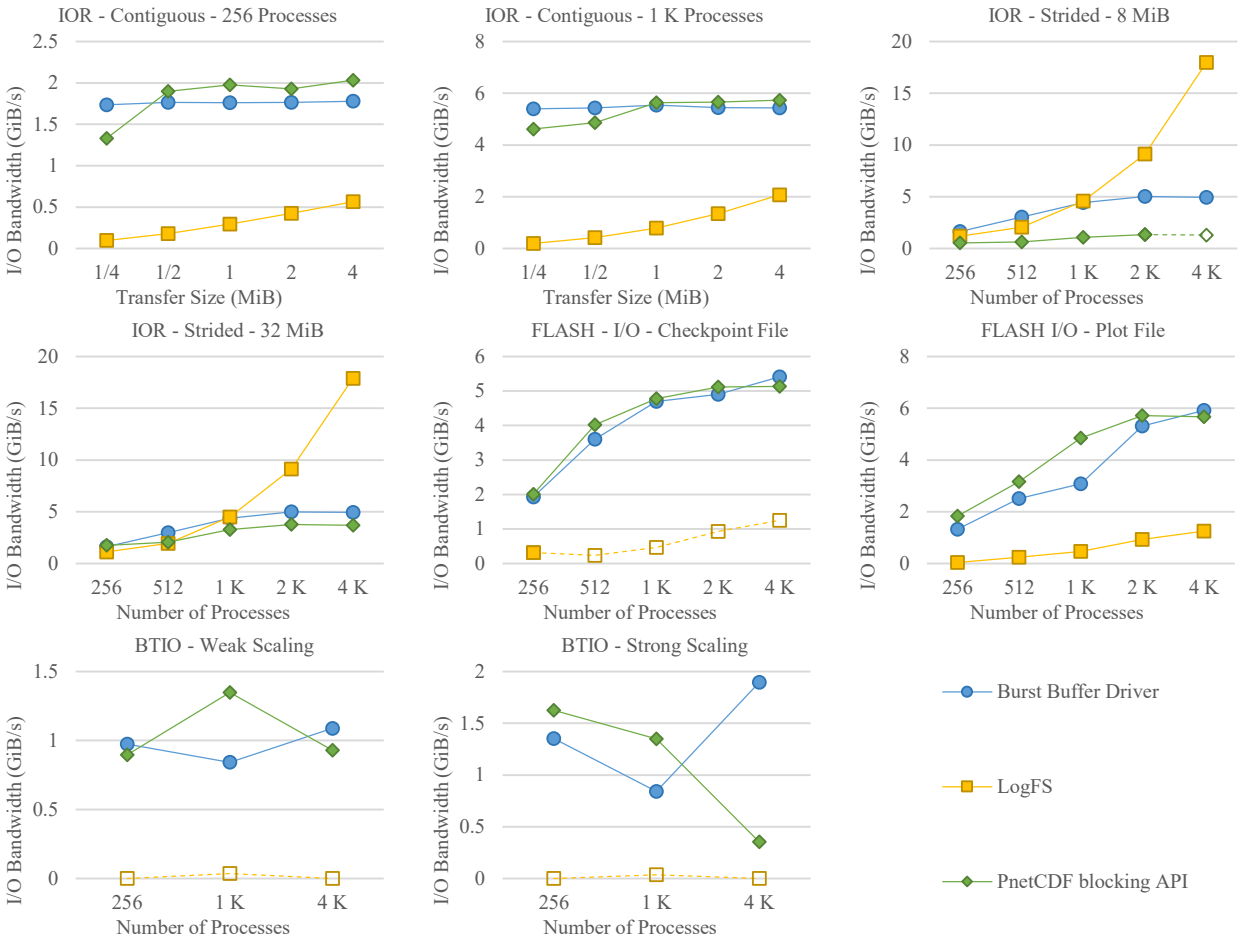

Fig. 6. Bandwidth and Execution Time on Theta. The lines show the andwidth measured by the benchmark porograms. In case the benchmark program did not finish within a reasonable time, we present the bandwidth approximate by actuall size written in dashed lines and hollow markers.

shared by all users, the speed of moving files to the PFS does not scale with the number of processes.

On all of the benchmarks, the burst buffer driver significantly outperforms the case using the burst buffer in place of the PFS. The overall performance of the burst buffer driver is up to 70% faster for IOR contiguous pattern and up to 4 times faster for IOR strided pattern. For FLASH I/O, the burst buffer driver is about 2 times faster than DataWarp staging out. For BTIO, compared to DataWarp staging out, using the burst buffer driver can be up to 3.9 times faster in the weak scaling case and 6.5 times faster in the case of strong scaling.

A more interesting point appears when we compare timing breakdown. In terms of flushing the data to the PFS, the burst buffer only takes 32 computing nodes to achieve a higher speed than the built-in file copying function provided by the burst buffer server running on 64 servers. At 4096 processes, the burst buffer driver is more than 2 times faster than the burst buffer servers when moving data to the PFS. In terms of I/O time on the burst buffer, the burst buffer driver takes significantly less amount of time than writing entire file on the burst buffer. Since the amount of data written to the burst buffer is roughly the same, we may conclude that the I/O pattern can still affect performance on the burst buffer similar to that on the PFS.

The results above suggest an important lesson - burst buffers should not be used to replace the role of the PFS directly, a proper file structure is required to achieve high efficiency when caching data on the burst buffer. Although I/O operations on the burst buffer are much faster than that on the PFS, the speed of copying files from the burst buffer back to the PFS may become a bottleneck. Regarding this, we believe that the role of the burst buffer is a temporary storage for the application to store disposable data that will be discarded after the execution unless the underlying burst buffer architecture supports efficient data movement between the burst buffer and the PFS. A good way to use the burst buffer is to divert the data from the memory, freeing up space for computation.

D. Compare with Data Elevator

A major source of low performance of using DataWarp stage out is the time it takes to copy the file from the burst buffer to the PFS. The issue is studied and discussed in [18]. The work showed that using compute nodes to move data to the PFS can be faster than having the DataWarp server to move the files especially when there are more compute nodes than burst buffer servers. Data elevator introduced a user-level service process that runs in the background alongside the application to actively

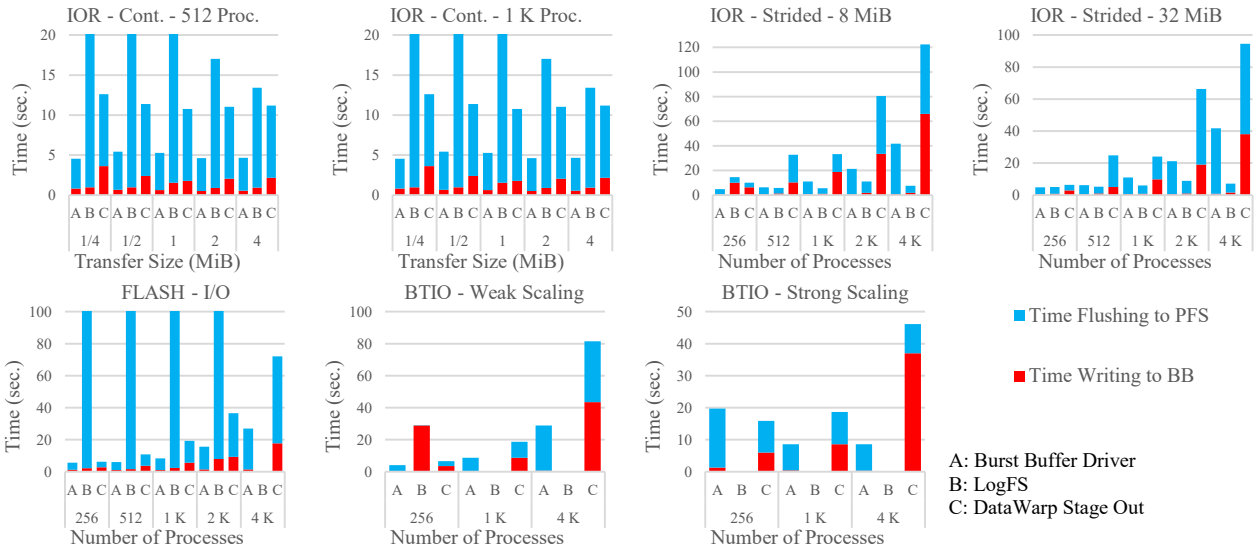

Fig. 7. Writing to Burst Buffer VS Flusing to PFS Time on Cori.

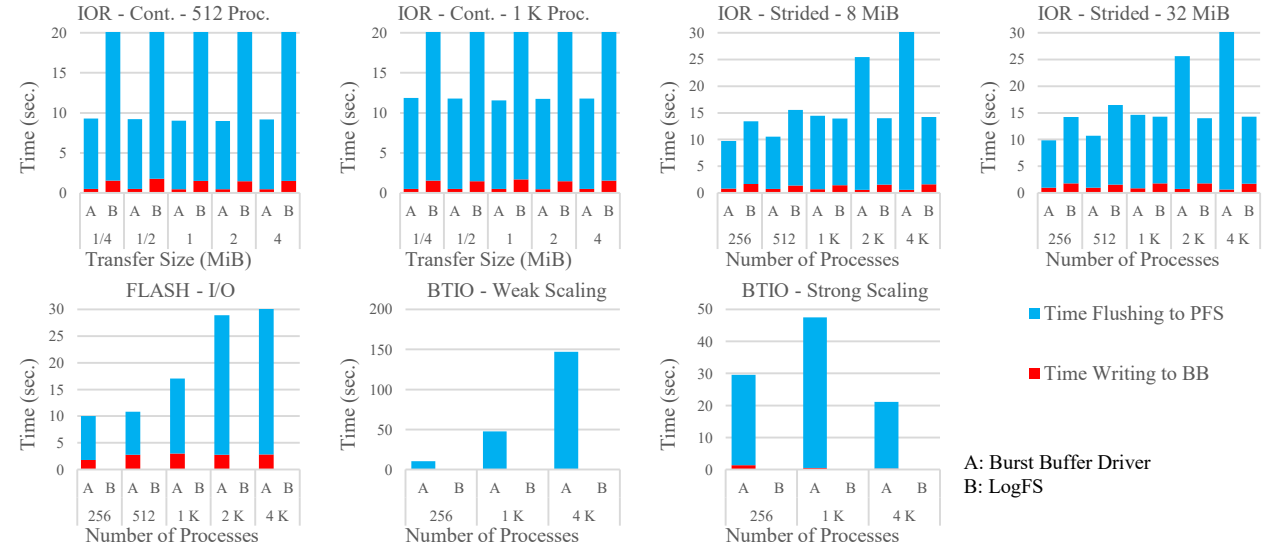

Fig. 8. Writing to Burst Buffer VS Flushing to PFS Time on Theta. To prevent large values from distorting the chart, we set an upperbound on the vertical axis. Bars reaching the top of the chart indicate larger than the maximum labeled value on vertical axis.

read files from the burst buffer and write them to the PFS. With increased file copying performance, it may now be feasible to use the burst buffer as a file system. Since the majority space of a NetCDF file is occupied by variables. The amount of data moved by the Data Elevator should be similar to the amount of data movement involved using our burst buffer drive. To find out which solution performs better, we repeated the experiment in subsection C using Data Elevator in place of DataWarp API. We used the latest version of Data Elevator (as of Jun. 2018) with its default setting.

Because Data Elevator relies on background processes to move data from the burst buffer to the PFS, the application need not stop when the file is being flushed to the PFS until the end of the job. Due to this characteristic, it is difficult to measure the exact time spent on I/O. Instead, we present the end to end time from starting the benchmark program and the background processes until both finishes. We do not present the result on Theta because Data Elevator only works under global (shared) burst buffer as it does not keep any metadata that is required to combine data scattered on nodes. The result is shown in Figure 5. We are interested in comparing the burst buffer driver to Data Elevator. On large and sequential I/O pattern, Data Elevator performs better than the burst buffer driver. Data Elevator achieved 70% higher bandwidth for IOR contiguous pattern. However, the burst buffer driver catches up when I/O size becomes small and non-sequential. For IOR strided pattern, the burst buffer driver achieves up to 50% higher I/O bandwidth. For BTIO, in both weak and strong scaling cases, the burst buffer driver outperforms Data Elevator by around 70%. The two solutions are on par when the I/O pattern falls in between the two extremes such as for FLASH I/O and IOR strided pattern with large transfer size. In short, they complement each other.

The cause of such coincidence on performance characteristic of the two solutions is due to the way the burst buffer is used. The burst buffer driver uses the burst buffer to store organized log files. Regardless of the I/O pattern from the application, the I/O pattern on the burst buffer is always contiguous. The Data Elevator, on the other hand, does not regulate the files on the burst buffer. It redirects I/O operations from the user application to the burst buffer as is. The I/O pattern remains the same as the original pattern given by the application. Although SSDs are more resilient to noncontiguous I/O patterns than hard disks, the performance penalty of non-sequential access is still significant. On simple I/O patterns where the access is near sequential, both solutions write to the burst buffer in an efficient way while the burst buffer driver bears the overhead of organizing the log files. On complex patterns, the performance gain form sequential access outweighs the cost of additional computation. Although we cannot accurately measure the timing breakdown of the Data Elevator due to its flush-in-background approach, we can still confirm the reasoning above by comparing the time it takes to write to the burst buffer in the DataWarp stage out approach in Figure 7. We can see that, on difficult benchmarks such as BTIO, it takes the application significantly more time to write the native file to the burst buffer in stage out solution than it takes the burst buffer driver to write log files to the burst buffer.

E. Compare with LogFS

One of our motivation to design the burst buffer driver is the opportunity to utilize the high-level information about the original data structure to increase the I/O performance. To verify such advantage, we compare our burst buffer driver with LogFS [7]. The main differences between the two are that LogFS records I/O operations in the broken-down offsets and lengths representation while our burst buffer driver records it in its original form, and that LogFS employs a tree structure to generate non-decreasing offsets on aggregated request while our two-level reordering strategy relies on high-level information to speed up the sorting process.

The experiment shows that LogFS performs poor on nearly all experiments except in IOR strided pattern. For IOR contiguous pattern, the burst buffer driver achieved 50% and 1.6 times higher bandwidth than LogFS on Cori and Theta respectively. For FLASH I/O, the burst buffer driver achieved 3 times the bandwidth on Cori and 2 times the bandwidth on Theta. For BTIO, LogFS could not finish within a reasonable time on larger runs so they are not shown in the charts. For IOR strided pattern, LogFS outperformed all other solutions significantly. It achieved more than 4 time the bandwidth compares to our burst buffer driver which is in the second place.

To understand such unusual behavior of LogFS, we did a deep profiling of LogFS flushing procedure. We found that a majority of time is spent on posting MPI asynchronous I/O. In ROMIO, MPI_File_iwrite_at is implemented using the AIO interface (POSIX asynchronous I/O) which starts immediately writing data in the background once the requests are posted. For the I/O patterns exhibited in the benchmarks used in our experiments, its performance is expected to be worse than the collective I/O, because the two-phase I/O is not used. For Lustre, it is known that the two-phase I/O arranges the requests to avoid file lock conflicts and is critical to achieve high performance on parallel computers. We show a few cases in Figure 5 to demonstrate that replacing the MPI asynchronous writes with blocking collective writes does improve the bandwidths significantly. From this experiment, we believe LogFS can perform equivalently well to our burst buffering approach, if collective writes were used. We show a few cases in Figure 5 to demonstrate that replacing the MPI asynchronous writes with blocking collective writes does improve the bandwidths significantly. From this experiment, we believe LogFS can perform equivalently well to our burst buffering approach, if collective writes were used.

There are several factors allowing LogFS to top the list on IOR strided pattern. First, since each process only writes a single block in IOR strided pattern, the aggregated I/O pattern is actually contiguous. On the other hand, processes in IOR contiguous pattern write to multiple strided blocks, the aggregated I/O pattern is strided. It is understandable that contiguous I/O access performs better. In addition, processes within the same node are given consecutive ranks. This arrangement caused the combined access region within a single node to form a single large and contiguous chunk. As a result, the I/O client on each computing nodes only needs to acquire lock from the PFS once, eliminating the possibility of extended lock contention. In short, it achieved similar effect of collective I/O but without the overhead of communication. In such case, it is possible to LogFS to outperform other solutions doing collective I/O. Finally, variables in IOR benchmark are all one-dimensional. In this case, LogFS does not surfer any disadvantage compared to the high-level approach.

F. Performance Analysis on Log File to Process Mappings

Other than the performance comparison with other solutions, we also want to find out how different log file to process mappings affect the performance on different burst buffer architectures. We expect the log-per-process mapping to work best on node-local burst buffers while the log-per-node mapping to work best on centralized burst buffers. To verify such assumption, we measure the time used by our burst buffer driver in each step when aggregating I/O requests on the burst buffer. We found that the overhead is dominated by only two steps – the time it takes to initialize logfiles as well as the time it uses to exchange data with the burst buffer. The results on Cori and Theta are shown in Figure 9 and Figure 10. Times taken by other steps are eliminated because they are too short to visualize in the figure.

On Cori with centralized burst buffer, the cost to initialize the log increases significantly when the number of processes increases. This confirms our concern of metadata server contention. In an exascale environment with thousands or even millions of processes. The initialization cost will be infeasibly high, rendering the idea useless. For this reason, we suggest configuring the burst buffer to private mode whenever possible. Doing so can largely reduce the initialization cost. Should there be a need to use striped mode, Log-per-node mapping can be used to mitigate the contention because the number of files it needs to initialize is reduced by a factor equal to number of the processes per node, which can be high on larger systems. In our experiment, Log-per-node mapping reduces the initialization cost by about 50%. However, since many processes are sharing

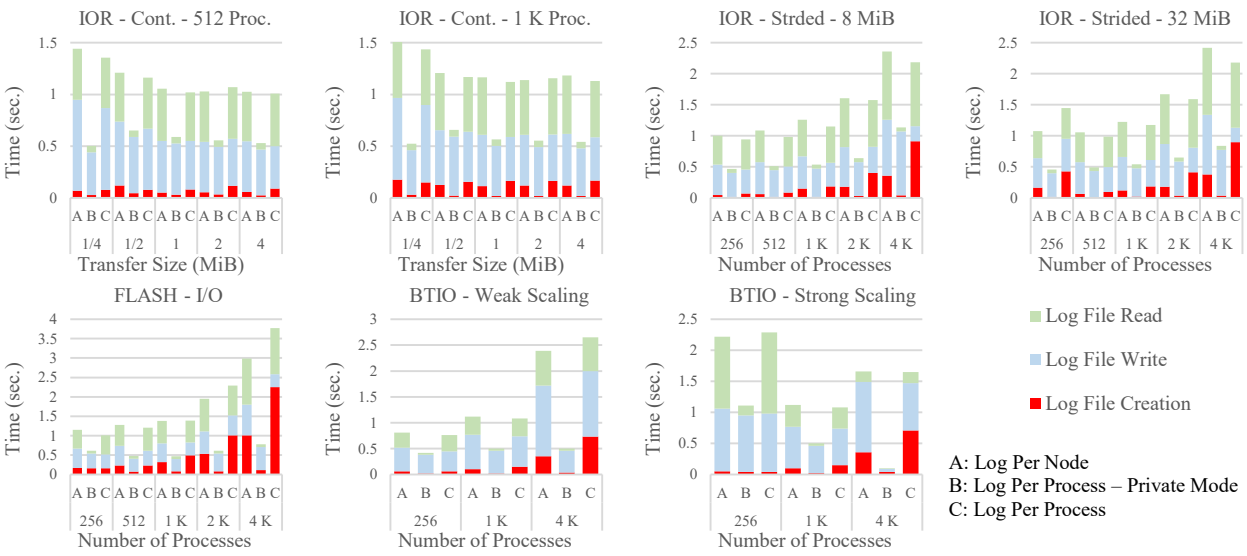

Fig. 9. Burst Buffer Driver Overhead with Different Log to Process Mapping on Cori.

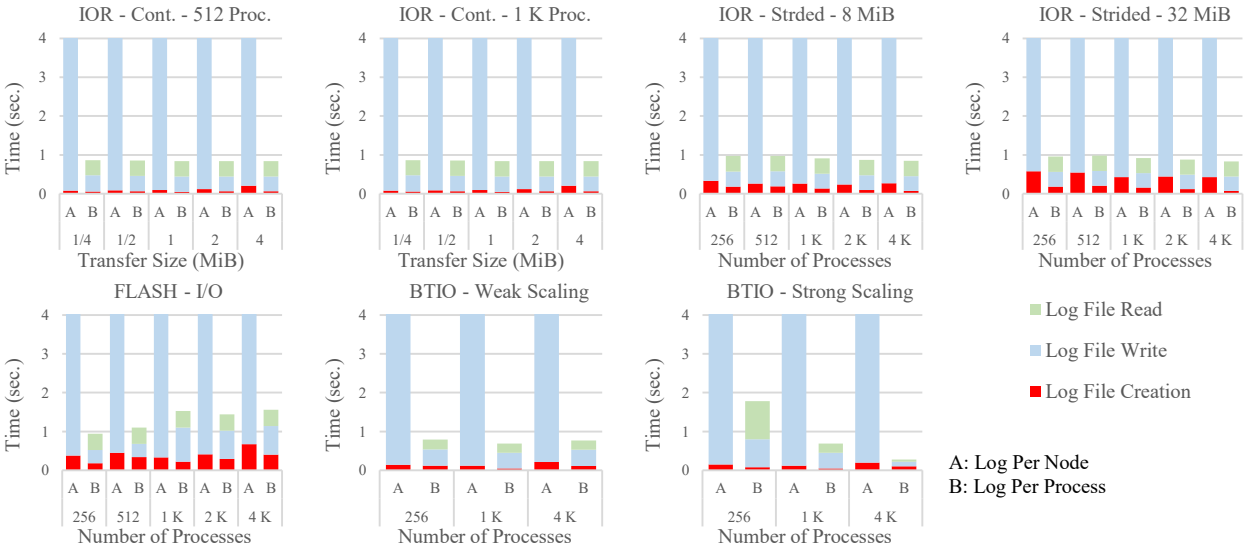

Fig. 10. Burst Buffer Driver Overhead with Different Log to Process Mapping on Theta. To prevent large values from distorting the chart, we do not show times beyond 4 sec. Bars reaching the top of the chart indicates a time larger than 4 sec in both Figure 9 and Figure 10.

one log file, there will be a performance penalty on data access. In its implementation, the file space is allocated to processes in a round-robin fashion, a necessary approach to deal with unknown size of data. As a result, each process is assigned a non-contiguous file space, forcing it to do non-contiguous I/O operations at the boundary of each block. For these reasons, we expect slower data access on log-per-node mapping.

On Theta with node-local burst buffer, log-per-node mapping is not preferred. Since data access time dominates the overhead of burst buffer driver, reducing file creation does not provide any benefit on performance. Instead, log-per-node mapping significantly slows down data access because local filesystem must maintain data consistency on shared files. Overall, the overhead of the burst buffer measured on Theta is significantly lower than that measured on Cori. The performance of the burst buffer is also more stable on Theta. It is largely due to interference from other jobs in a centralized burst buffer architecture and the fact that the number of burst buffers, and hence, the combined bandwidth in node-local setup automatically scales up on larger jobs when more nodes join the computation. It suggests that node-local burst buffer is a better choice for I/O aggregation.

VI. Conclusion

In this work, we discover a new role of burst buffers in the high-level library. We show the benefit of I/O aggregation in high-level library despite being more complicated to implement. We designed a log-based data structure that balance between performance and transparency. We take advantage of high-level information in a high-level I/O library to speed up our aggregation operations. We implemented this idea as an I/O module in PnetCDF that can be used by other applications. We

found that simply redirecting files to the burst buffer usually results in poor performance, suggesting that a new hardware cannot achieve its full potential without the support of properly tuned software. We demonstrate that I/O aggregation is a worth-considering feature to provide in a high-level I/O library. We hope our study can benefit future I/O library developers.

ACKNOWLEDGMENT

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.

REFERENCES

[1] B. Dong, X. Li, L. Xiao and L. Ruan, "A New File-Specific Stripe Size Selection Method for Highly Concurrent Data Access," Journal of Grid Computing, pp. 22-30, 2012.

[2] W. Bhimji, et al., "Accelerating science with the NERSC burst buffer early user program," in CUG2016 Proceedings, 2016.

[3] N. Liu, et al., "On the role of burst buffers in leadership-class storage systems," in Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on, 2012.

[4] T. Wang, et al., "Trio: burst buffer based i/o orchestration," in Cluster Computing (CLUSTER), 2015 IEEE International Conference on, 2015.

[5] T. Wang, et al., "An ephemeral burst-buffer file system for scientific applications," in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis., 2016.

[6] T. Wang, H. S. Oral, Y. Wang, B. W. Settlemyer, S. Atchley and W. Yu, "BurstMem: A high-performance burst buffer system for scientific applications," in Big Data (Big Data), 2014 IEEE International Conference, 2014.

[7] D. Kimpe, R. Ross, S. Vandewalle and S. Poedts, "Transparent log-based data storage in MPI-IO applications," in Proceedings of the 14th European conference on Recent Advances in Parallel Virtual Machine and Message Passing Interface , Paris, 2007.

[8] J. Li, et al., "Parallel NetCDF: A Scientific High-Performance I/O Interface," arXiv: Distributed, Parallel, and Cluster Computing, 2003.

[9] R. . Rew and G. . Davis, "NetCDF: an interface for scientific data access," IEEE Computer Graphics and Applications, vol. 10, no. 4, pp. 76-82, 1990.

[10] M. Folk, A. Cheng and K. Yates, "HDF5: A file format and I/O library for high performance computing applications," Proceedings of supercomputing, vol. 99, pp. 5-33, 1999.

[11] R. Thakur, W. Gropp and E. Lusk, "Data sieving and collective I/O in ROMIO," in Frontiers of Massively Parallel Computation, 1999. Frontiers' 99. The Seventh Symposium on the. IEEE, 1999.

[12] "HPC IO Benchmark Repository," [Online]. Available: https://github.com/LLNL/ior.

[13] "FLASH I/O Benchmark Routine -- Parallel HDF 5," [Online]. Available: http://www.ucolick.org/~zingale/FLASH I/O_benchmark_io/.

[14] P. Wong, R. F. VanderWijngaart and B. Biegel, "NAS Parallel Benchmarks I/O Version 2.4. 2.4," 2002.

[15] Henseler, Dave, B. Landsteiner, D. Petesch, C. Wright and N. J. Wright, "Architecture and design of cray DataWarp," in Cray User Group CUG, 2016.

[16] W. Bhimji, et al., "Extreme I/O on HPC for HEP using the Burst Buffer at NERSC," Journal of Physics: Conference Series, vol. 898, no. 8, p. 082015, 2017.

[17] B. Hicks, "Improving I/O bandwidth with Cray DVS Client‐side Caching," Concurrency and Computation: Practice and Experience, vol. 30, no. 1, 2018.

[18] D. Bin, S. Byna, K. Wu, H. Johansen, J. N. Johnson and N. Keen, "Data Elevator: Low-Contention Data Movement in Hierarchical Storage System," in High Performance Computing (HiPC), 2016 IEEE 23rd International Conference, 2016.

[19] A. Kougkas, H. Devaraja and X.-H. Sun , "Hermes: AHeterogeneous-AwareMulti-TieredDistributedI/O BufferingSystem," in Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing, Tempe, 2018.

[20] N. Liu, et al., "Modeling a leadership-scale storage system," in International Conference on Parallel Processing and Applied Mathematics, 2011.

[21] K. Sato, et al., "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, 2014.

[22] J. Bent, et al., "PLFS: a checkpoint filesystem for parallel applications," in Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, 2009.

[23] M. Rosenblum and J. Ousterhout, "The Design and Implementation of a Log-Structured File System," ACM Transactions on Computer Systems (TOCS), vol. 10, no. 1, pp. 26-52, 1992.

[24] H. Dai, M. Neufeld and R. Han , "ELF: an efficient log-structured FLASH I/O file system for micro sensor nodes," in Proceedings of the 2nd international conference on Embedded networked sensor systems, 2004.

[25] R. Rew and G. Davis, "The unidata NetCDF: Software for scientific data access," Sixth International Conference on Interactive Information and Processing Systems for Meteorology, Oceanography and Hydrology, pp. 33-40, 1990.

[26] Thakur, Rajeev, W. Gropp and E. Lusk, "On implementing MPI-IO portably and with high performance," in Proceedings of the sixth workshop on I/O in parallel and distributed systems, 1999.

[27] K. Gao, W. Liao, A. Choudhary, R. Ross and R. Latham, "Combining I/O operations for multiple array variables in parallel NetCDF," Cluster Computing, vol. , no. , pp. 1-10, 2009.

[28] R. Latham, et al., "A case study for scientific I/O: improving the FLASH I/O astrophysics code," Computational Science & Discovery, vol. 5, no. 1, p. 015001, 2012.

[29] "MPI: A Message-Passing Interface Standard V3.1," [Online]. Available: http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf.

[30] K. Antypas, N. Wright, N. P. Cardo, A. Andrews and M. Cordery, "Cori: A Cray XC Pre-Exascale System for NERSC," in Cray User Group Proceedings, 2014.

[31] T. Declerck, et al., "Cori - A System to Support Data-Intensive Computing," in Cray User Group, 2016.

[32] H. Shan and J. Shalf, "Using IOR to analyze the I/O Performance for HPC Platforms," Lawrence Berkeley National Laboratory, 2007.

[33] B. Fryxell, et al., "FLASH I/O: An Adaptive Mesh Hydrodynamics Code for Modeling Astrophysical Thermonuclear FLASH I/Oes," Astrophysical Journal Supplement Series, vol. 131, no. 1, pp. 273-334, 2000.

[34] "The FLASH I/O Code," [Online]. Available: http://FLASH I/O.uchicago.edu/site/FLASH I/Ocode/.

[35] R. Latham, et al., "A case study for scientific I/O: improving the FLASH I/O astrophysics code," Computational Science & Discovery, vol. 5, no. 1, p. 015001, 2012.

[36] "NASA Parallel Benchmarks," [Online]. Available: https://www.nas.nasa.gov/publications/npb.html.