NORTHWESTERN UNIVERSITY

Scalable Parallel I/O in the Exa-scale Era

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY

Field of Computer Engineering

By

Kaiyuan Hou

EVANSTON, ILLINOIS

March 2023

ABSTRACT

The speed of the storage device has long lagged behind the computation speed of processors. As a result, the I/O performance of storage systems in a supercomputer fails to keep up with its computational power. This gap continues to widen in modern supercomputers. On future exascale supercomputers, this issue can worsen to the extent that I/O operations become a bottleneck to many applications.

In this dissertation, we experiment with three ideas to improve I/O performance and scalability on supercomputers. One of them is to try to utilize new types of storage hardware being introduced in modern supercomputers. They bridge the performance gap between the primary storage system and compute nodes, but also complicate the I/O stack. Another is to compress the data to ease the demand on I/O bandwidth. Compression reduces the bandwidth required to store the same amount of information but makes the data more difficult to access and manage. The other is to improve the I/O pattern to utilize I/O resources more efficiently. Large and contiguous I/O requests can generally be handled more efficiently; however, it takes computation and memory resources to reorganize the data into the desired pattern.

We present four projects that study the practicality of the three ideas for future exascale HPC environments on various applications. The first one tries to utilize the

burst buffer to perform I/O aggregation that combines and reorders I/O requests so it can be handled more efficiently by the file system. The second one is an experiment on the idea of compressing checkpoint data to reduce I/O time, and file system workload. The third one introduces a framework we designed to enable efficient parallel I/O on compressed variables in classic NetCDF files. The final one presents an HDF5 VOL we developed that stores datasets in a log-based storage layout that results in efficient I/O patterns. We show that the ideas mentioned above are effective and provide a decent solution to scalable I/O in the exascale era.

Table of Contents

List of Tables

List of Figures

CHAPTER 1

INTRODUCTION

1.1. Parallel I/O in the Exa-scale Era

Throughout the past, the computing power of computers has increased significantly. High-Performance Computing (HPC) systems are expected to pass the exascale milestone in less than a year. Increasing computation power increases the demands on the underlying storage solution in both capacity and performance. Although the capacity of storage devices improved significantly during the past years, the improvement in performance is inadequate. As a result, the speed of the file systems fails on supercomputers to keep pace with the growth of computational power. As the performance gap between computation and I/O continues to widen, I/O operation has become a potential performance bottleneck of modern HPC applications.

An exascale HPC system can perform at least a quadrillion floating-point computation per second. It is also expected to have tens of petabytes of memory. Such computation power, combined with colossal system memory, enables scientific applications to run at an unprecedented pace, scale, or resolution. Those applications are expected to produce data that is several magnitudes larger than current applications do and to produce them at a much faster rate. To handle data that size in a reasonable time, the I/O solution must be efficient and scalable.

1.2. Possible Approaches to Scalable I/O in Exa-scale Systems

We briefly discuss some potential solutions to improve I/O performance and scalability and challenges associated with them.

1.2.1. Utilizing new storage hardware

The various storage device has been proposed in the past decade to replace rotating hard disk to improve performance. Those devices are not only faster but also more flexible to a different type of I/O patterns. They provide enough performance to handle the data generated by modern HPC applications. However, due to their high cost, it is prohibitively expensive to replace traditional hard disk-based storage systems entirely with these new devices. Usually, they are deployed in limited quantity in modern HPC systems to serve as a buffer to the traditional hard disk-based parallel file system. As a result, the I/O stack in modern supercomputers becomes deeper and deeper. The existing software must adapt to increasing depth of the I/O stack to achieve I/O high performance. In consequence, managing data efficiently across layers in the I/O stack becomes a challenge to applications and I/O libraries.

1.2.2. Data reduction

Instead of increasing I/O performance to meet the demand, an alternate approach is to reduce the size of the data. An obvious solution is to compress the data before writing to the file system. While it may not reduce I/O time for applications due to time spent on compression and decompression, the I/O bandwidth is being used more efficiently. It allows the same I/O resource to serve more running jobs. An undesirable side effect of

compression is that the data generally must be compressed and decompressed as a whole. This limitation makes parallel access to compressed data difficult. Due to its complexity, applications usually rely on I/O libraries or middlewares to handle compressed data. Supporting parallel I/O on compressed data efficiently then becomes a challenge faced by all I/O library developers.

1.2.3. Improve I/O pattern

The amount of data is not the only determining factor of I/O workload. Due to the characteristic of the underlying storage device and the design of the particular PFS, performance is usually very sensitive to I/O pattern. Generally, I/O operations involving fewer I/O requests to large chunks of contiguous data are easier to the PFS than fragment I/O requests of discontiguous chunks. I/O pattern that aligns with the file system block or stripe boundary can also be handled more efficiently. Common approaches to improve the I/O pattern includes I/O aggregation, data buffering, subfiling, and log-based storage layout. I/O aggregation and buffering require additional memory space that can compete for limited memory resources with the application. Other solutions also require accommodation from the application.

1.3. Exploring Scalable Parallel I/O Solutions on Exascale Systems

In this dissertation, we study the ideas mentioned in subsection section 1.2. The article is divided into four chapters. Each chapter presents a project in which we apply one or more of the ideas to a particular type of application. They are summarized below.

1.3.1. I/O aggregation using the burst buffer

In this project, we examine the features of burst buffers and study their impact on application I/O performance. We demonstrate that burst buffers can be utilized by parallel I/O libraries to significantly improve performance. By offsetting the data onto the burst buffer, we can enable I/O aggregation without regard to available memory. We extend PnetCDF to store individual I/O requests on the burst buffer in a log-based format and later flush them 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 in high-level I/O libraries is a good way to use burst buffers on modern supercomputers. We will discuss details of this work in subsection chapter 3.

1.3.2. Checkpoint compression

In this project, we explore the idea of speeding up the I/O operation by compressing the data before writing to the file system. We augmented OpenAD in the MITgcm framework to write checkpoints in a compressed form. We study the effect of different compression algorithms on different applications. We show that it is possible to reduce end-to-end

I/O time by compressing the data. We will discuss details of this work in subsection chapter 4.

1.3.3. Paralle I/O framework for compressed NetCDF variables

In this project, we design and implement the variable compression feature in the Parallel NetCDF library. Our design employs the same concept of chunking used by the HDF5 library, but we focus on enabling I/O aggregation across multiple requests to address the challenges of performance and scalability. We demonstrate the idea of using I/O aggregation to amortize the communication cost to synchronize chunk access. We evaluate our solution using the I/O kernel of real-world scientific applications and analyze the impacts of data compression on parallel I/O performance. Experiment results suggest that handling multiple requests at once can significantly improve the parallel I/O performance on chunked and compressed data. We will discuss details of this work in subsection chapter 5.

1.3.4. Log-based storage layout for HDF5 datasets

In this work, we introduce a log-based storage layout for HDF5 datasets. HDF5 is a popular file format and I/O library among scientific applications. Despite its strong functionality, the performance of HDF5 is subpar to libraries explicitly designed for parallel I/O, such as PnetCDF and ADIOS, especially when the I/O pattern is complex. One of the contributing factors is the overhead to rearrange the data in canonical order which is required by the HDF5 format. Log-based storage layout is known to produce an efficient

write pattern as it delays the rearranging overhead to the reading stage. However, a log-based layout generates additional metadata to describe the logical position of the data. We explored several techniques to mitigate the metadata overhead, including encoding, deduplication, and compression. Experiment results show the effectiveness of our metadata reduction methods and that our solution can match or exceed the performance of other I/O libraries. We will discuss details of this work in subsection chapter 6.

CHAPTER 2

BACKGROUNDS

In this subsection, we provide some background related to this dissertation.

2.1. Storage System on HPC Systems

2.1.1. Parallel file system

Usually, the primary persistent storage on a supercomputer is a Parallel File System (PFS). PFS provides global shared storage for all compute nodes. They are designed to support high-performance parallel access to shared files. Most PFS store data on a set of dedicated storage servers separate from the compute nodes. The storage servers are connected to the compute nodes either directly on the network or via dedicated I/O nodes. Files are usually divided into stripes that are distributed across multiple servers to improve performance. Notable PFS implementation includes GPFS [**1**] and Lustre [**2**].

2.1.2. Burst buffer

Burst Buffers are faster storage devices usually placed in front of the PFS to absorb burst in I/O demand. While various devices can be used to implement burst buffers, most of the implementations use the Solid State Drive (SSD). They are abundant in the market and ubiquitous in computing devices. SSDs can be implemented using various storage media and technologies. As a result, they have diverse characteristics in terms of cost

and performance that fit a wide range of positions in the I/O stack. Many supercomputers nowadays incorporate SSDs to improve performance. Cori at NERSC employs Cray Datawrap [**3**], a burst buffer solution made of SSDs that can be requested in by jobs. Theta at ALCF has 2 SSD per cabinet shared by local nodes as the burst buffer.

2.1.3. Non-volatile random-access memory

Non-volatile random-access memory (NVRAM) is a type of Non-volatile memory that provides a memory-like interface instead of a disk-like interface of SSDs and memory sticks. One of their advantage, compared to other non-volatile memory, such as SSDs, is that they support random-access without little to no performance penalty. It allows them to serve a wide range of roles in an HPC system. They can be configured to serve as an extension to the main memory, to serve as local storage for the application, or to serve as the burst buffer to the PFS. The performance and the cost of NVRAMs sit in between the system memory and disk-like non-volatile memory, such as SSDs. As a result, they are often placed right below the main memory in the I/O stack. NVRAMs are relatively new to HPC systems. At the time of this writing, the only supercomputer incorporating NVRAMs is Summit [**4**] at ORNL.

2.2. Parallel I/O Libraries and Middlewares

2.2.1. MPI-IO

MPI-IO [**5, 6**] is part of the MPI standard that provides an interface for multiple processes to access the same file. It is a low-level I/O middleware with a POSIX-like API. MPI-IO provides many convenient features for file sharing. One is a derived datatype that can be used to define non-contiguous regions in the file and the memory space. Another one is the file view that restricts the I/O operation of a process to part of the file, so the user does not need to seek past space that belongs to other processes explicitly. MPI-IO includes an abstraction layer called Abstract Device I/O (ADIO) that apply different optimization strategy on different file systems to ensure efficient I/O operation under different environment. In the past, MPI-IO has been chosen by many high-level I/O library developers as the I/O middleware between their library and the operating system.

2.2.2. NetCDF and PnetCDF

NetCDF (Network Common Data Form) [**7, 8**] is a self-describing, machine-independent (portable) data format for array-oriented scientific data. In a NetCDF file, user data are stored in multi-dimensional arrays called variables. The size of variables is defined using global shared dimensions. Attributes can be added to describe variables and the file. In the newer NetCDF-4 standard, users can create groups to organize variables and attributes.

The NetCDF library is a high-level I/O library for accessing NetCDF files developed by unidata. It provides an abstract view of the NetCDF file so applications can access the NetCDF file without the need to deal with the NetCDF file format. PnetCDF [**9**] is

a parallel alternative to the NetCDF library. It supports parallel I/O on classic NetCDF files using MPI-IO. For more details regarding NetCDF and PnetCDF, please refer to section 5.1.3.

2.2.3. HDF5

Hierarchical Data Format version 5 (HDF5) is a file format designed to store a large amount of data. Similar to NetCDF, the HDF5 format is also self-describing and machine-independent. In an HDF5 file, user data are stored in multi-dimensional arrays called datasets. Datasets are organized into groups. Attributes can be added to describe groups and datasets. Internally, data objects are indexed using a B-tree. Compared to NetCDF, the HDF5 data model offers more functionality such as chunked storage layout, data filters (see section 5.1.1), and user-defined datatype.

HDF5 library is a high-level I/O library for accessing HDF5 files. Applications can access HDF5 data objects by providing a path similar to accessing files in the file system. HDF5 library supports various I/O modes and middlewares, including POSIX I/O for sequential applications and MPI-IO for parallel applications. Due to the rich set of library features and the flexibility of the data model, it has gained a huge user base in the HPC community.

2.2.4. ADIOS

The Adaptable IO System (ADIOS) provides an easy and flexible way for applications to write their runtime variables to the file. The I/O pattern is predefined by the application using an XML file. ADIOS differs from other I/O libraries in the way it stores the data.

It uses a log-based storage layout in which the data from processes always append one after another regardless of the I/O pattern from the application. They place an index at the end of the file to track scattered pieces of variables to provide the applications with a global view when reading variables. Since the contiguous I/O pattern is known to be efficient on most PFS, ADIOS usually achieves higher write bandwidth compared to other I/O libraries on applications that write complex I/O patterns.

CHAPTER 3

I/O AGGREGATION ON BURST BUFFER

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 [**10**]. This disparity has created an increasingly challenging environment for modern HighPerformance 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 [**11, 12, 13, 14**]. 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 [**15**] and lower level I/O middleware such as MPI-IO [**16**]. 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 [**9, 7, 17**] 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 chapter, 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 [**18**]. 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 [**16**].

We developed a lightweight module in the PnetCDF (Parallel-NetCDF) [**9**] 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 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 insitu 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 nondecreasing 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 [**19**], FLASH I/O [**20**] and BTIO [**21**] 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.

3.1. 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 [**3**]. 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 [**11, 15**]. Hicks introduced a client-side caching mechanism for DataWarp as DataWarp currently has no client-side caching [**22**]. Dong et al. proposed a way to transfer data on the burst buffer to the PFS using compute nodes [**23**]. 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 [**24**]. 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 [**25**]. 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 [**15**]. A similar concept was proposed by Sato et al. [**26**] 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 [**16**]. 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 [**27**]. Rosenblum and Ousterhout designed a file system that stores data using a log-based structure [**28**]. Dai et al. designed a log-structured FLASH I/O file system that is tailored for buffering sensor data in microsensor nodes [**29**].

3.2. Design of the 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.

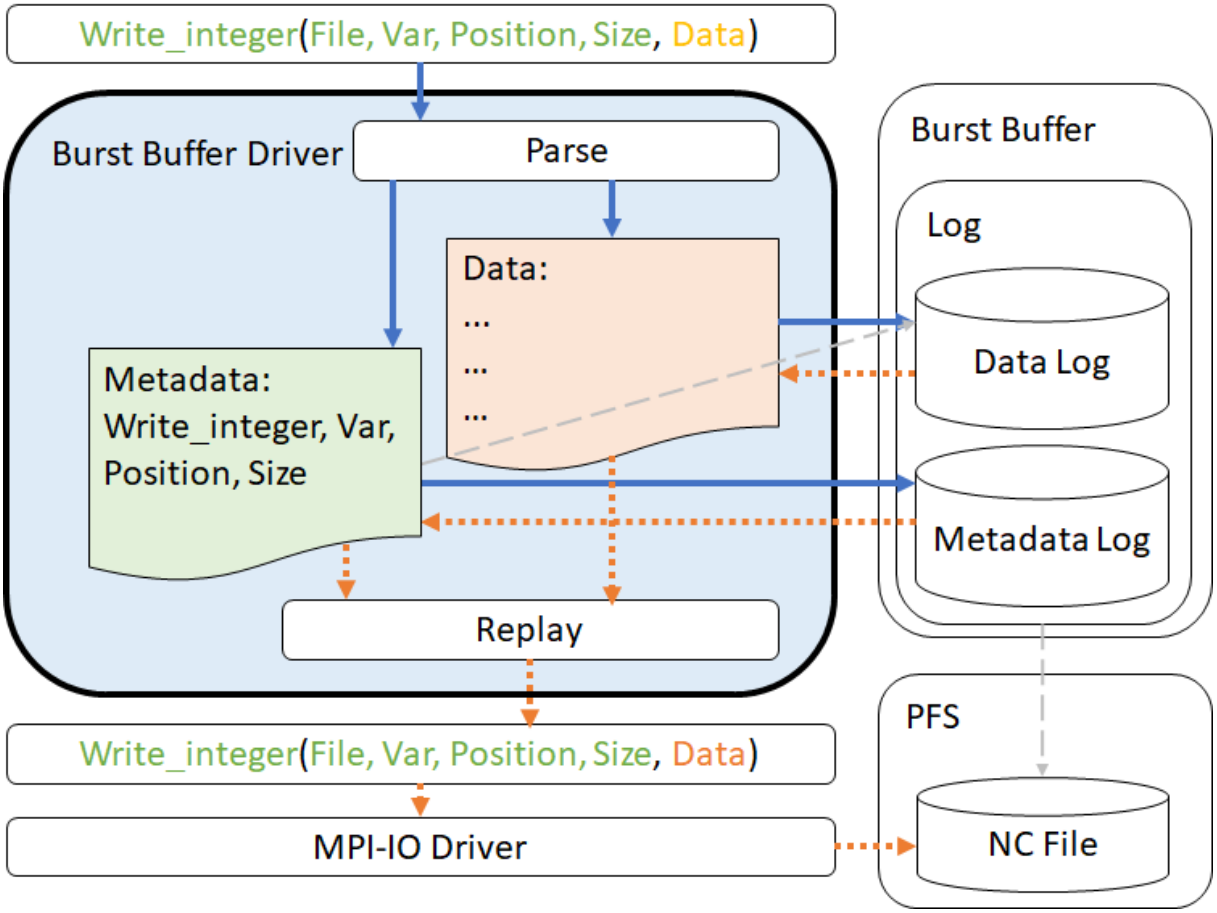


Figure 3.1. Handling variable write request. The burst buffer driver is shaded in light blue. Data are shaded in light green while metadata re shaded in pink. Blue arrows show the data flow when writing to the burst buffer. Orange arrows show the data flow when flushing to the PFS.

Figure 3.1 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.

3.2.1. 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 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.

3.2.2. 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.

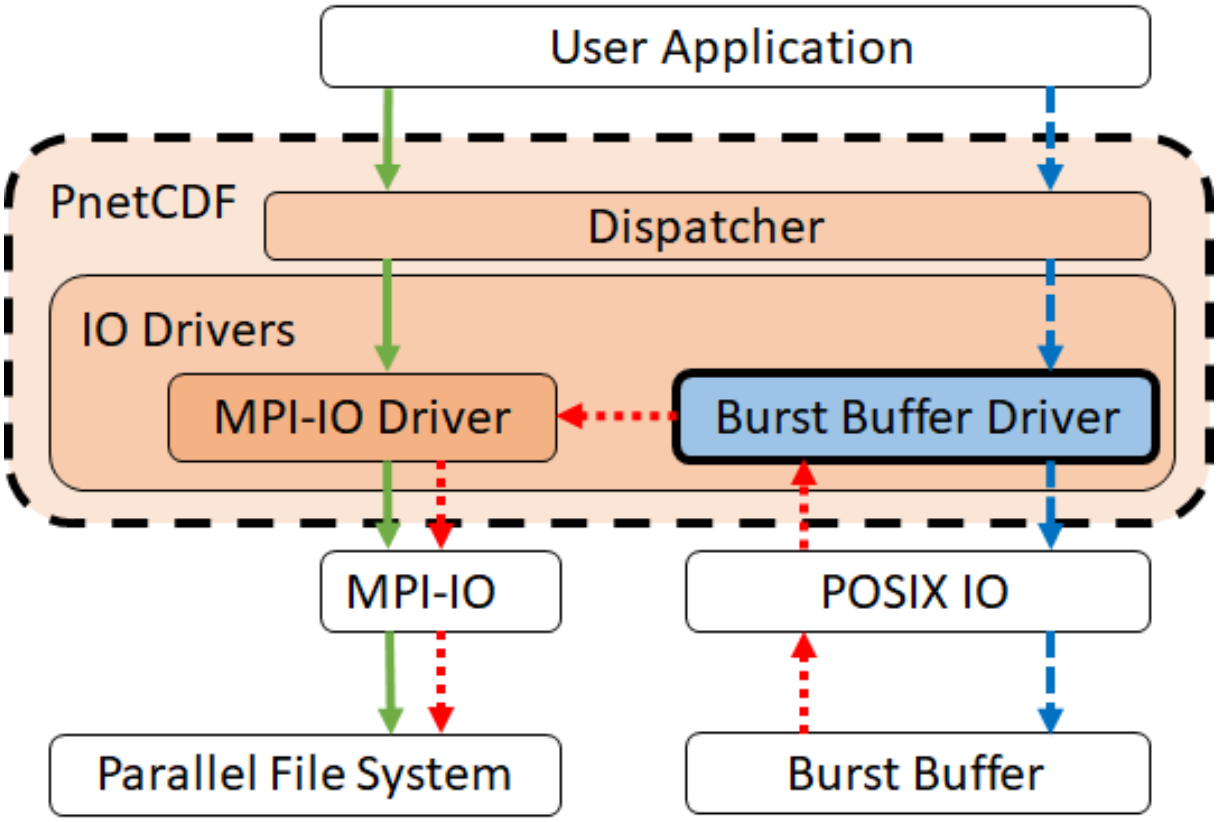


Figure 3.2. An overview of burst buffer driver. Blue arrows show the data flow when writing to the burst buffer using the burst buffer driver. Red arrows show the dataflow when flushing the data to the PFS. Green arrows show the original data flow when the burst buffer is not in use.

Figure 3.2 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.

3.2.3. 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.

3.2.4. 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.

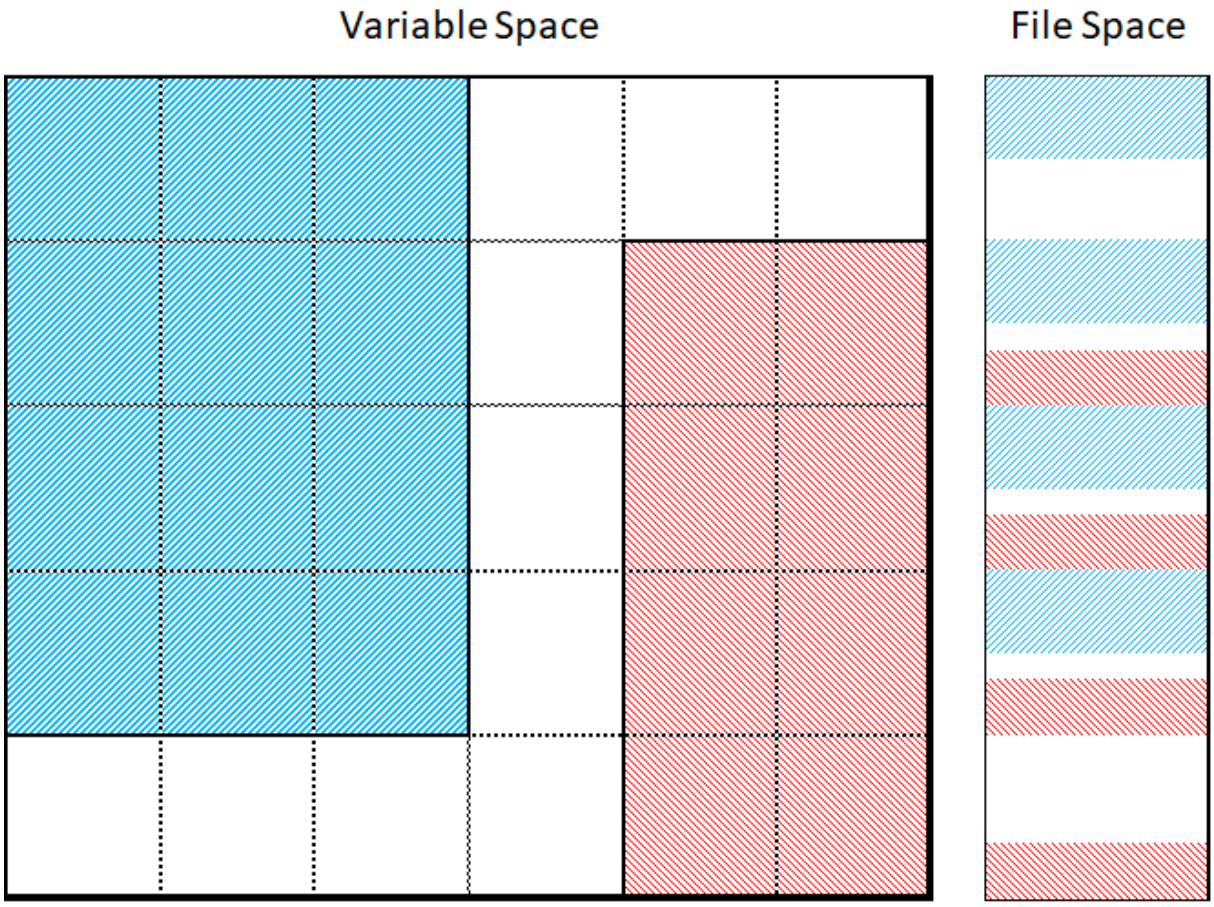


Figure 3.3. Variable Space vs. File Space. Two submatrices marked blue and red are completely separated in a 2D variable. However, their file space interleves with each other.

Figure 3.3 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.

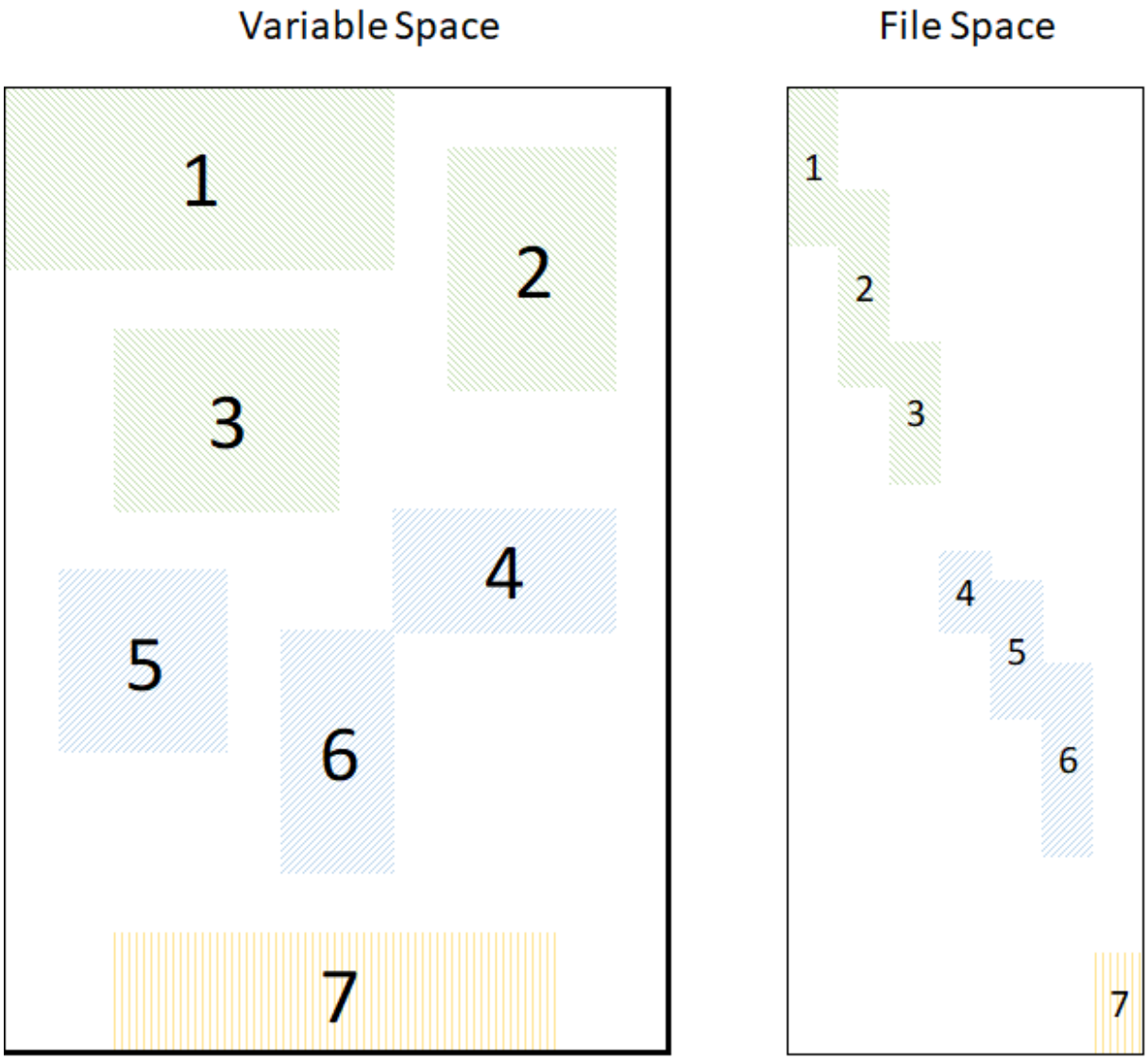


Figure 3.4. Two-level reordering strategy. Submatrices are colored according to their group. For submatrices in different groups, they are guaranteed to not interleve each other in the file space.

The method described above is demonstrated in Figure 3.4. 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.

3.3. Experiment

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 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.

3.3.1. I/O benchmarks

Our performance evaluation uses three well-known I/O benchmarks: IOR [**19**], FLASH I/O [**20**], and BTIO [**21**]. 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 [**20**] 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 [**21**] 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 [**21**]. 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.

3.3.2. 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 3.5 and Figure 3.6 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

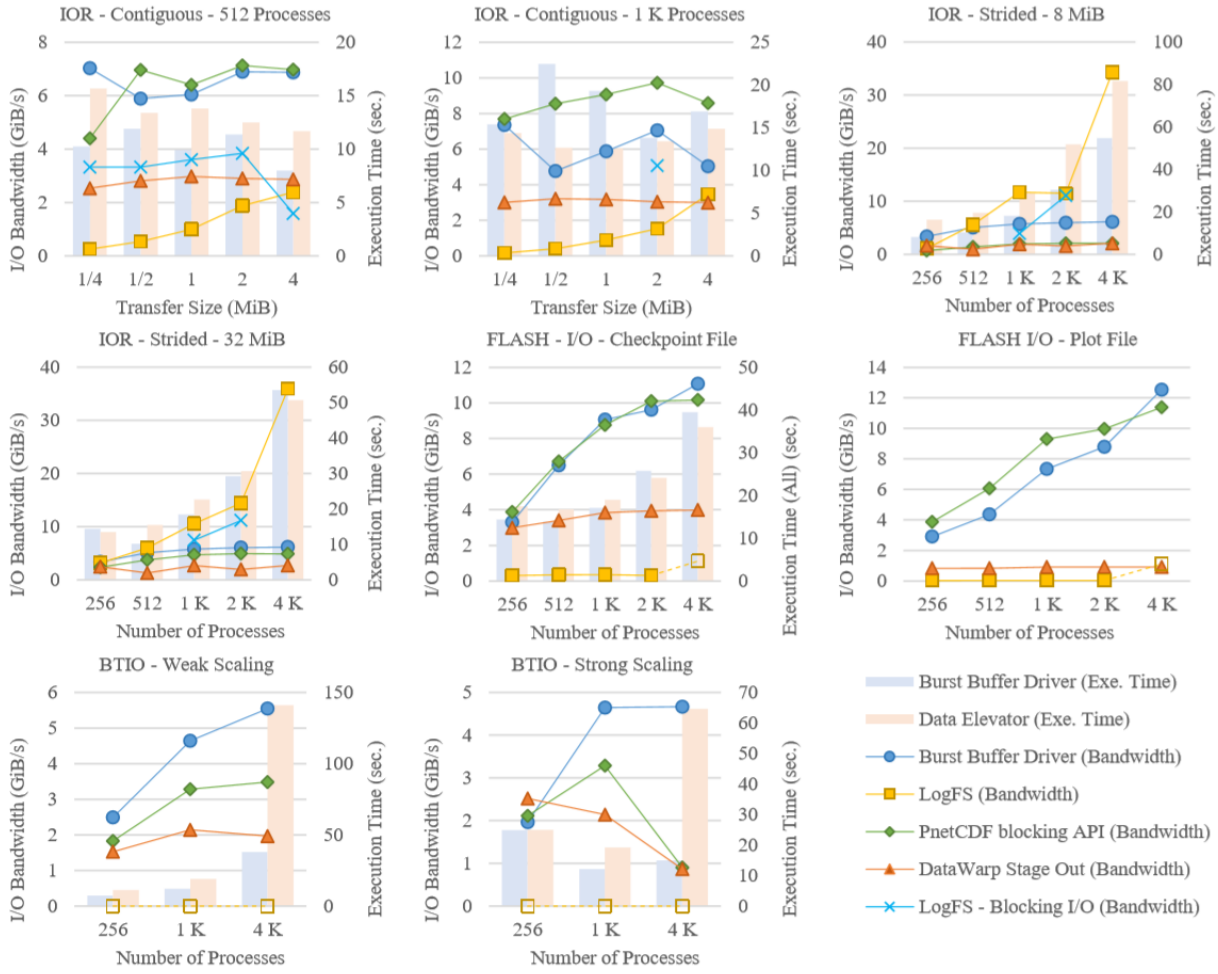


Figure 3.5. Bandwidth and Execution Time on Cori. The lines show the andwidth measured by the benchmark porograms. The bars show the end to end execution time of each benchmark including time spent on computation. In case the benchmark program did not finish within a reasonable time, we present the bandwidth approximate by actuall size written in dashed lines and hollow markers. Cases discussing LogFS characteristic is marked in "x"

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

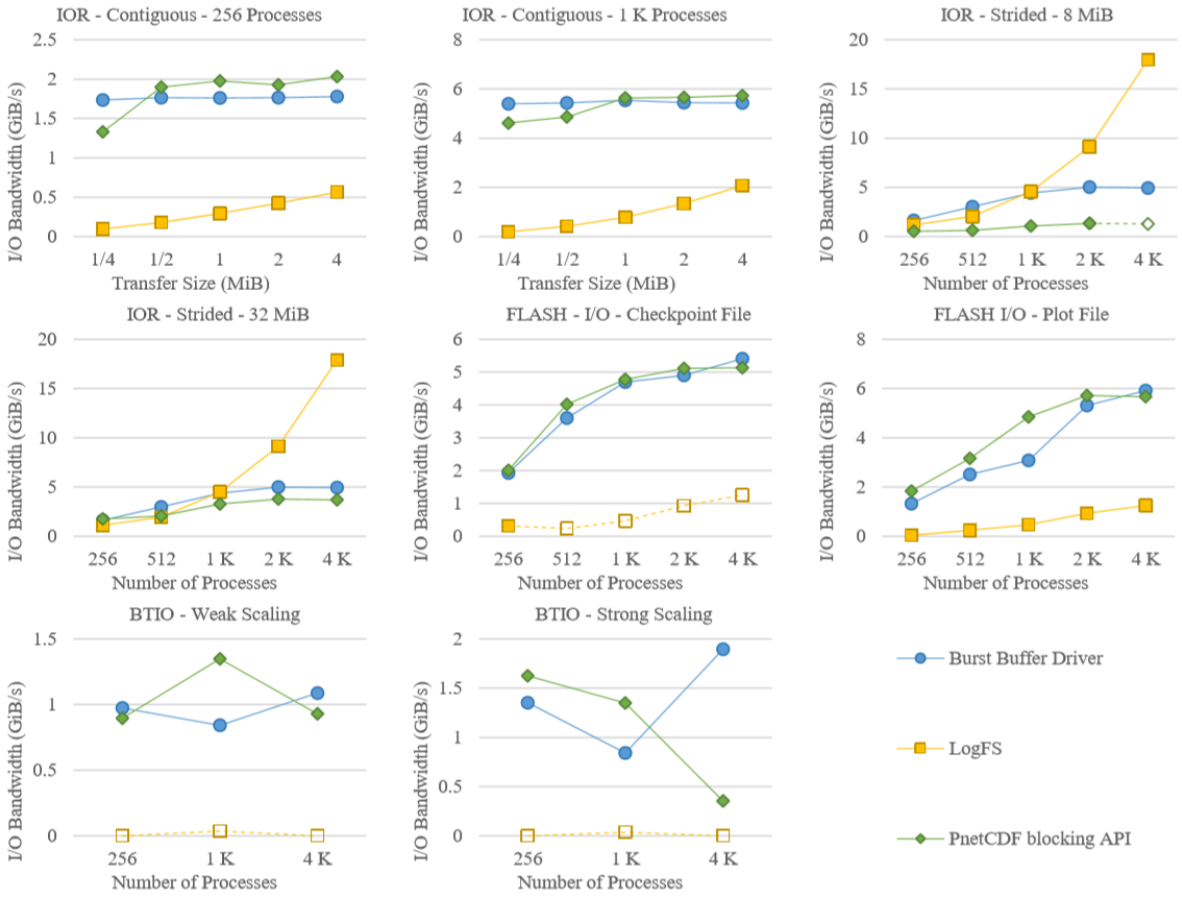


Figure 3.6. Bandwidth and Execution Time on Theta. The lines show the andwidth measured by the benchmark porograms. In case the benchmark program did not finish within a reasonable time, we present the bandwidth approximate by actuall size written in dashed lines and hollow markers.

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.

3.3.3. 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

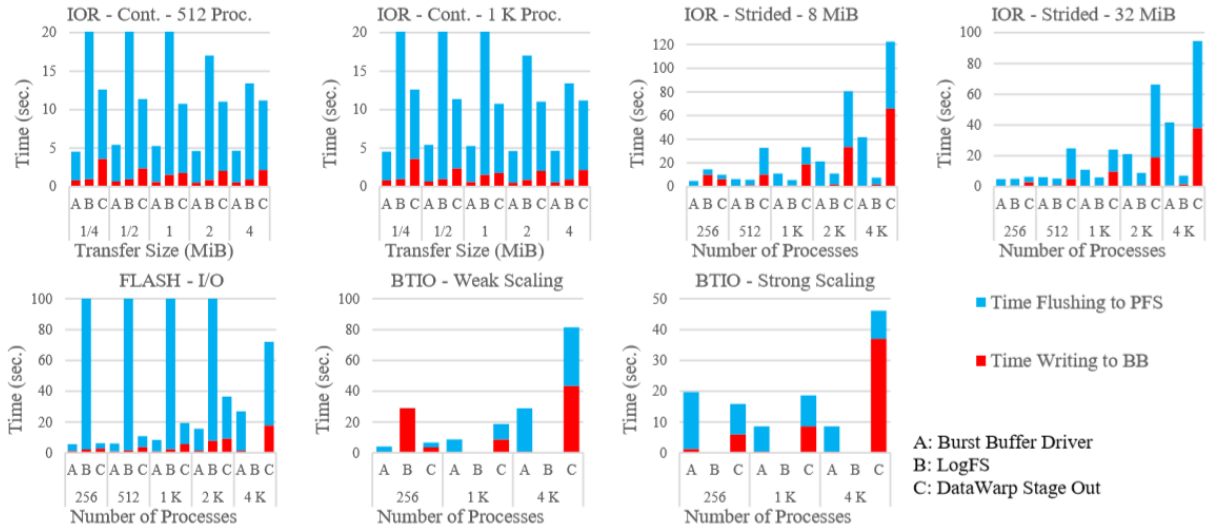


Figure 3.7. Writing to Burst Buffer VS Flusing to PFS Time on Cori.

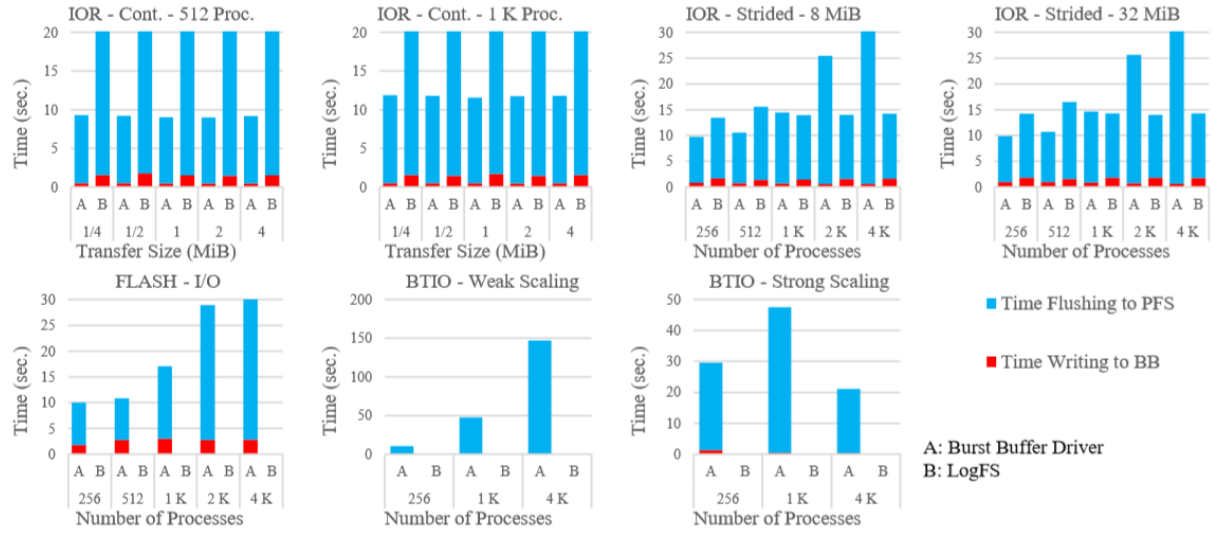


Figure 3.8. Writing to Burst Buffer VS Flushing to PFS Time on Theta. To prevent large values from distorting the chart, we set an upperbound on the vertical axis. Bars reaching the top of the chart indicate larger than the maximum labeled value on vertical axis.

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 3.7. Since there are only a fixed number of burst buffer servers that must be 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 string 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.

3.3.4. 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 [**23**]. 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 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 3.5. 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 3.7. 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.

3.3.5. 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 [**16**]. 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 3.5 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 3.5 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.

3.3.6. Performance analysis on log file to process pappings

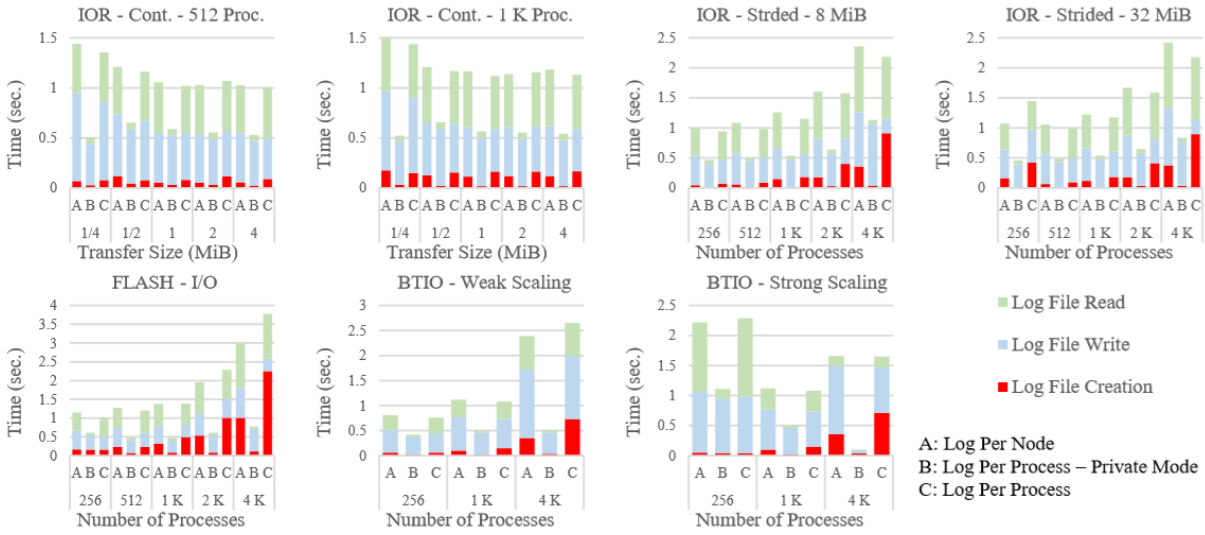


Figure 3.9. Burst Buffer Driver Overhead with Different Log to Process Mapping on Cori. To prevent large values from distorting the chart, we do not show times beyond 4 sec. Bars reaching the top of the chart indicates a time larger than 4 sec.

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

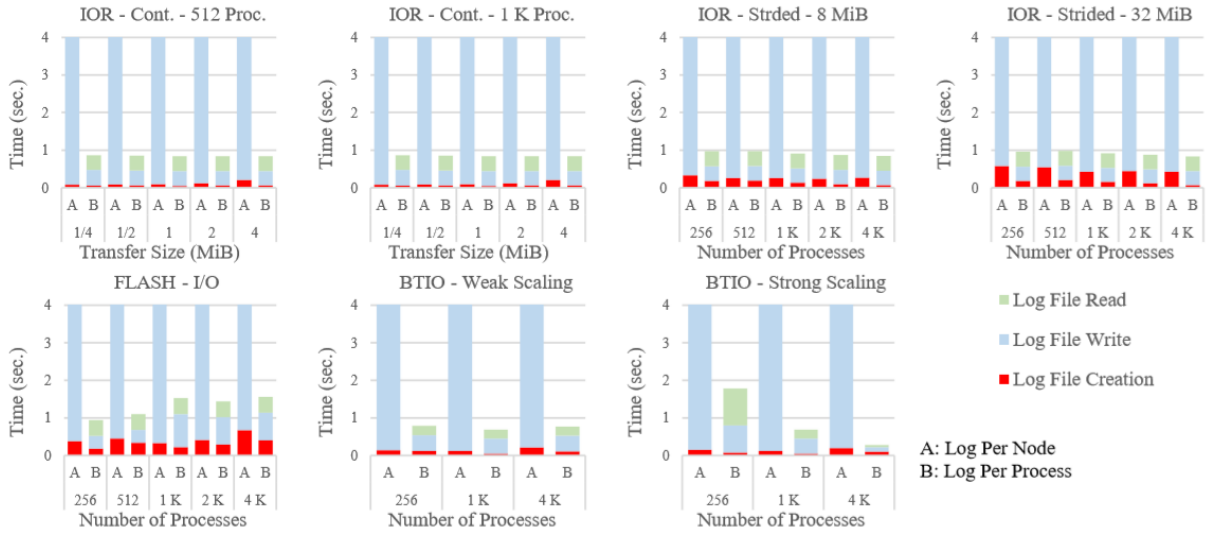


Figure 3.10. Burst Buffer Driver Overhead with Different Log to Process Mapping on Theta. To prevent large values from distorting the chart, we do not show times beyond 4 sec. Bars reaching the top of the chart indicates a time larger than 4 sec.

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 3.9 and Figure 3.10. 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 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.

3.4. Summary

In this project, 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.

CHAPTER 4

A STUDY ON CHECKPOINT COMPRESSION FOR ADJOINT COMPUTATION

Many scientific applications involve computing the gradient of cost functions or simulation models. Gradients are usually used to estimate the sensitivity of model output with respect to input [**37**] or to optimize an objective function. The adjoint state method is an efficient numerical method to compute the gradient that is widely employed by automatic differentiation (AD) tools [**38, 39, 40, 41**]. Every function can be decomposed into a sequence of basic operations. The gradient of a function can be computed automatically by using the chain rule without manually deriving the derivative. The adjoint state method computes the gradient by applying the chain rule iteratively on the sequence of operations in reverse order. In many cases, it involves much less computation than do methods that follow the original order of the operations, such as the tangent linear model. By recording every intermediate result when computing a function, the adjoint state method can then be used to compute the gradient of that function in a more efficient way. It is a powerful tool in many applications such as machine learning, data analysis, and scientific simulations. In this chapter, we focus on adjoint computation in scientific simulation applications where each operation can be viewed as an iteration of time steps.

Although the adjoint state method provides a computation speed advantage over methods that apply the chain rule in the forward direction, it does come with a major limitation

– memory usage. In order to apply the adjoint state method, every step during the computation of the function value along with its intermediate values must be retained. When the method is applied to large-scale scientific simulations, the state of each iteration must fit in the memory. When running simulation across a long time period, there can be thousands or even millions of iterations. Even if the size of each individual checkpoints is not large, the aggregated size across iterations can still be in terabytes or even petabytes. Although memory capacity has increased significantly in recent years, it is still far from enough to record the state of the entire simulation. In some cases, even the main file system does not have enough space to store the entire history of simulation.

To perform the adjoint state method with limited memory capacity, we need to trade time for space. Instead of saving the results of all iterations, we keep only a subset of them, called checkpoints. Whenever the value of a non-checkpoint iteration is needed, we recover it by restarting the simulation model from the nearest checkpoint [**42**]. Such an operation is called recomputation. In addition to memory, checkpoints can also be stored on other devices such as local hard disks, global parallel file system, or even tapes. These storage media can be employed together to create multilevel checkpointing in which higher-level (faster) checkpointing is used to recompute data between two lower-level checkpoints [**43, 44**]. In many cases, the hard disk is used alongside the memory to create a two-level checkpointing strategy. Different factors must be considered when doing checkpointing on the hard disk. Unlike doing checkpointing in memory where storing and retrieving checkpoints are virtually instantaneous, hard disk I/O involves a significant latency and a limited bandwidth. Usually, disk-based checkpointing is bandwidth bound rather than capacity bound; in other words, the number of checkpoints we can store (called the

budget) is limited not by disk or file system capacity but by the amount of data we can transmit from and to the storage device. The higher the bandwidth, the more budget we have on checkpoints. Improving I/O speed gives us a larger budget for checkpoints on the second level, which in turn allow the first-level budget to be applied to a shorter period, reducing recomputation and increasing overall performance. When disk-based checkpointing is applied alone as a single-layer strategy, reducing the I/O time leads to direct improvement of overall execution time.

An intuitive way to speed the checkpoint I/O time is to reduce the size of the checkpoint. A straightforward approach is to compress the checkpoints. However, it also introduces the overhead of compression and decompression. The compression algorithm must be efficient so that the saving from I/O time can cover its overhead, yet it also must be effective in reducing the size of the checkpoints. When a lossy compression algorithm is used, the error introduced may also become an issue.

Since I/O speed is known to be far slower than computation, the idea of applying compression on checkpoints used for resilience purposes has been thoroughly explored [**45, 46, 47, 48**]. In such application, most of the checkpoints will just be discarded after execution unless there are anomalies that interrupts the execution. Unlike the checkpoints created for resilience purpose where decompression time does not matter, every checkpoint created by adjoint state method will be read back later, making decompression performance as important as compression performance. Form the other aspect, the checkpoints in adjoint state method are not used to compute the simulation output but only used to compute the gradient. As a result, the presence of error is more acceptable

than checkpoints used for resilience purpose, giving us more opportunity to employ lossy compression methods.

In this chapter, we study the concept of compressing the checkpoint data for a disk-based checkpointing strategy used by adjoint computation. We run experiment using the MITgcm [**49, 50**] simulation framework with OpenAD [**38**] for gradient computation. We used Zlib [**51**] for lossless compression. For lossy compression, we tried ZFP [**52, 53**] and SZ [**54, 55**]. We also tried to do checkpointing with reduced precision. We ran experiments on a small cluster and stored checkpoints to the local hard disk. In the experiment, we were able to reduce the checkpoint size by up to 90% and reduce the checkpointing time by 87%. When using lossy compression, the error did not exceed 1% even after tens of iterations. Overall, the experiment results suggest that compression is a promising approach to improve disk-based checkpointing performance.

4.1. MITgcm and Automatic Differentation

4.1.1. Automatic differentation using the adjoint state method

An iterative function $f : R^p \rightarrow R^q$ can be viewed as a composite function of its iterations.

$$f(x) = f_{n-1} \circ f_{n-2} \circ \ldots f_0(x)$$

The function can be evaluated by iterate through the iterations.

$$x_0 = x, x_i = f_{i-1}(x_{i-1}), f(x) = x_n$$

Applying the chain rule, one can compute the gradient of f as follows.

$$\frac{\partial x_i}{\partial x} = f'_{i-1}(x_{i-1})\frac{\partial x_{i-1}}{\partial x} = \bar{x}_{i-1}\frac{\partial x_{i-1}}{\partial x}$$

where

$$\bar{x}_i = \frac{\partial x_i}{\partial x}$$

$$f'(x) = \frac{\partial x_n}{\partial x} = f'_{n-1}(x_{n-1})f'_{n-2}(x_{n-2})\dots f'_0(x_0) = \bar{x}_{n-1}\bar{x}_{n-2}...\bar{x}_0$$

The multiplication (accumulation) of $\bar{x}_i$s can be done in either the forward (starting from $\bar{x}_0 0$) or backward (starting from $\bar{x}_{n-1}$) direction. Usually, the number of output variables is significantly smaller than the number of input variables (i.e., $p \gg q$) in scientific simulation applications since the input variables are the initial state of the entire simulation grid whereas the output variables are just a few features we are interested in. In such cases, backward accumulation used by the adjoint state method involves significantly less computation. However, every $\bar{x}_i$s must be available in order to compute $\bar{x}_i$s, leading to the problem we are studying in this chapter.

The adjoint state method consists of a forward sweep and an adjoint (backward) sweep. During the forward sweep, the intermediate steps to compute a function's value along with all intermediate values are recorded. The backward sweep follows recorded steps in reverse order, applying the chain rule recursively along the way to calculate the gradient of the function at the same point.

4.1.2. MITgcm

MITgcm is an atmospheric and oceanic simulation framework written mainly in Fortran. It supports gradient computation using either the tangent linear or adjoint state method

for sensitivity study or parameter optimization in some of the applications. OpenAD is used for adjoint computation. MITgcm employs a single-level disk-based checkpointing that uses a revolving scheduling strategy where the checkpoint budget is configurable. It uses native Fortran I/O to read and write checkpoints. A file is created per process per checkpoint. During each checkpoint, every variable in the simulation model is written to the file one after another in its native representation. In order to retrieve the checkpoints, data is read to the variables following the same order.

4.2. Supporting Checkpoints Comrpession in OpenAD

To perform data reduction, we reimplemented the checkpointing routine of MITgcm in C. We performed compression on each variable before writing it to the file. We decompressed the data read from the file before returning the results to the simulation routines. We studied three types of data reduction method: reduced precision, lossless compression, and lossy compression.

In the first study, we reduced the precision of floating-point values among the checkpoint data. This approach can be viewed as a simple lossy compression method. MITgcm by default stores all floating-point variables in double precision. We modified it to store single-precision checkpoints by performing type casting between the variable and the I/O buffer. This method applies only to floating-point data. We leave variables of other data types unchanged.

To study the effect of lossless compression, we used the compression library Zlib [**51**]. We deflate variables before writing them to the file. When we needed to restore the checkpoints, we inflated the data before putting them into variables. The size of compressed

data was appended to the front of the compressed stream. Since small variables, such as scalars, are likely to be expanded instead of compressed by Zlib, we set a threshold of 1 KiB on variable size to prevent such an issue.

For lossy compression, we used the ZFP library [**52**] developed at Lawrence Livermore as well as the SZ library developed at Argonne [**55**]. ZFP takes a structure called the field from the caller that contains the dimensionality and shape of the array to decide the best compression strategy. It accepts fields only up to three dimensions. For higher-dimensional variables, we merged its higher dimensions into the third (third fastest changing) dimension. We used ZFP in fixed-accuracy mode in which the error is bounded to within 10^{-3}. Similar to the lossless case, we compressed only those floating-point variables with a size larger than 1 KiB. SZ also use the dimensionality and shape of the array to help improve the compression results. It supports arrays with up to four dimensions. We performed a transformation similar to that used in the ZFP case for arrays with higher dimensionality. We set the error bound to 10^{-3} in absolute error to match the setting used in ZFP. SZ supports two modes: best compression mode for better compression ratio (original size / compressed size) and best speed mode for faster compression/decompression time.

4.3. Experiment

Table 4.1. End to End Run Time

Program	**Origin**	**Float**	**Zlib**	**ZFP**	**SZ-S**	**SZ-C**
hs94	104.2	78.8	59.4	54.7	71.9	72.0
halfpipe	807.2	831.4	804.7	832.5	897.2	894.0

SZ-C and SZ-S refers to SZ library in best compression and best speed mode respectively. Time is shown in seconds.

We ran the experiment on a small cluster connected by Intel Omni-Path. Each node is equipped with two Intel Xeon Platinum 8180M processors (56 cores in total) paired with 384 GiB of memory. Each node has access to a GPFS, and two local SSDs. We use GPFS in the experiment.

We evaluated the concept of apply compression on the hs94.1x64x5 (hs94) and halfpipe stream-ice (halfpipe) example programs. Since checkpointing in MITgcm is done in a file-per-process basis, evaluating on a single node should be sufficient to indicate the compression ratio and its impact on performance. However, to measure I/O time more accurately, we need to scale up the simulation to generate larger checkpoint files. More nodes are then added in order to speed the computation. We used direct and synchronized I/O to eliminate the interference of caches. We aggregated checkpointing data in the memory so that there is only one I/O request per checkpoint. We used the latest release of OpenAD (2014-03-15).

For the hs94 experiment, we used 256 processes on 4 nodes. We set tile size per process to 256 x 1 and the number of processes along each dimension to 256 x 1. The resulting grid is 65,536 cells evenly distributed across 180 degrees in the spherical polar grid. The corresponding checkpoint size per iteration is about 437 MiB.

In the halfpipe experiment, we adapted the same scale as in [**56**]. We fixed the global grid to 160 x 80 with 500 m spacing. Because of the smaller problem size, we used only 64 processes on a single node in this experiment. There are 8 processes on both dimensions, each of them holding a 20 x 10 tile. The resulting data size per checkpoint is around 31 MiB. We used the same input files as in [**56**].

4.3.1. Performance study

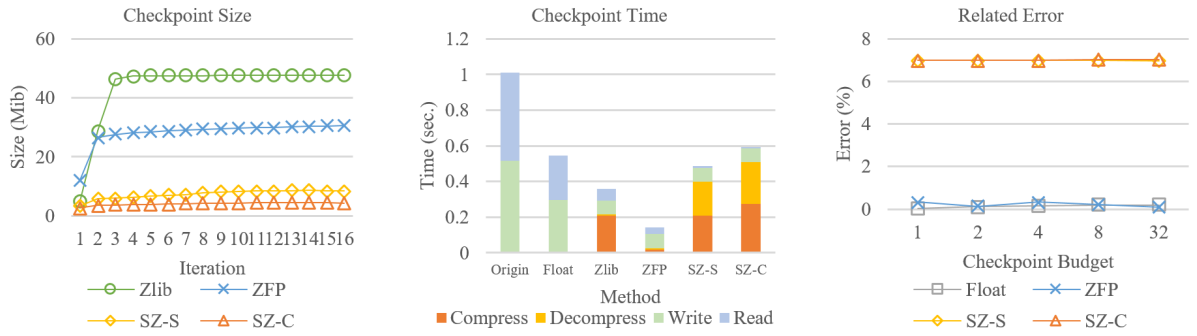


Figure 4.1. Results of hs94 experiment. SZ-C and SZ-S refers to SZ library in best compression and best speed mode, respectively. Original checkpoint size is 436 MiB. Checkpotining time is averaged across 32 iterations. Errors are the maximum among the four sampled points.

To study the effectiveness of compression on increasing checkpoint performance, we measured the time spent on writing checkpointing files as well as time spent on doing compression. We also measured the checkpoint size after each iteration.

In the hs94 experiment, lossless compression achieves a 90% compression ratio and 65% reduction in checkpointing time. Performing checkpointing in reduced precision also results in 46% reduction in checkpointing time with negligible computational overhead. ZFP achieves a 94% compression ratio and 86% reduction in checkpointing time. SZ achieves a 98% and 99% compression ratio in best speed and best compression mode, respectively. Because of its longer computational time, however, it reduces the checkpointing time only by 51% and 41%. The best compression mode does not improve performance because the reduction in size is not large enough to compensate additional time on computation. We believe it can outperform other reduction methods if the checkpoint size is several magnitudes larger. Overall, ZFP performs the best among all methods tested.

Although the time spent on performing compression may exceed the time writing our compressed data, it is still a fraction of the time we saved by reducing data size. We can see in Table 4.3 that applying compression significantly reduces the overall execution time of the simulation. Since even lossless compression can achieve a decent compression ratio, we believe that checkpointing data in the hs94 example program is loosely packed or contains many redundancies. Also, we can see that the compression ratio during the first few iterations is significantly higher than in later iterations. In the first iteration, it is nearly 99% regardless of the method used. This result suggests that the initial state of the model may have low variance.

In the halfpipe experiment, lossless compression achieves an 84% compression ratio and 43% reduction in checkpointing time. Performing checkpointing in reduced precision results in a 34% reduction in checkpointing time. ZFP achieves a 65% compression ratio and 59% reduction in checkpointing time. SZ achieves a 71% and 84% compression ratio in best speed and best compression mode, respectively. SZ fails to improve overall performance because it has a Huffman coding step which has a constant cost that slow it down on small sized data. Contrary to intuition, lossless compression achieves the best compression ratio. We think the reason is that the tile size is too small (only 20 x 10) for lossy compression to utilize information from adjacent cells or to do meaningful block transformation. In terms of overall performance, ZFP again performs the best because of its faster computational time. Unlike in hs94, the end to end execution time does not reduce. We believe it is caused by the computation-intensive characteristic of the halfpipe example. Only a fraction time is spent on checkpointing compared to doing computation. In such case, the extremely small share of checkpointing time makes

compression non-profitable. The small I/O time improvement may even not be measurable due to measuring noise. In addition, running compression algorithm can also compete cache resource with the computation, leading to slower overall execution time.

In all of our experiments, compression significantly reduces the checkpoint size. Even when the improvement on performance is too small to be seen, it still reduces the load on shared I/O resource for other applications in need. With neglectable computation overhead, we can significantly speed up the checkpointing procedure while saving a large amount of space and bandwidth. Our results suggest that compression is an intuitive but effective way to improve I/O-based checkpointing performance or resource utilization for adjoint computation.

4.3.2. Lossy compression error

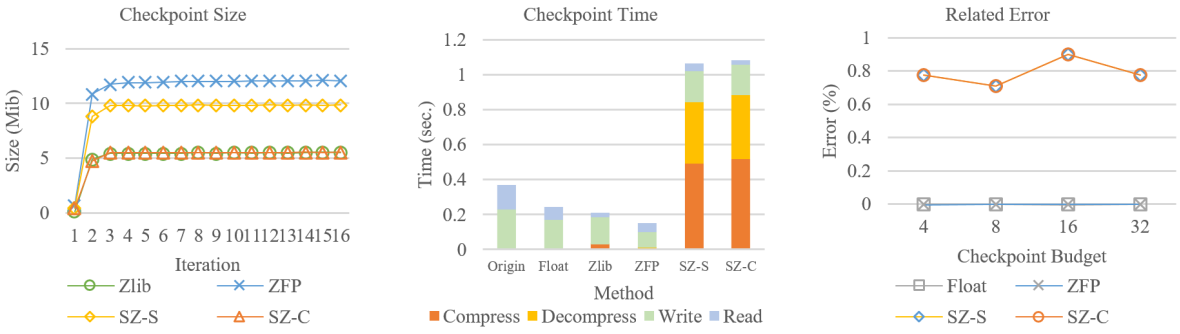


Figure 4.2. Results of halfpipe experiment. SZ-C and SZ-S refers to SZ library in best compression and best speed mode, respectively. Original checkpoint size is 31 MiB. Checkpotining time is averaged across 32 iterations. Errors are the maximum among the four sampled points.

A major concern about lossy compression is the error introduced. Even if we can bound the error on the checkpoint itself, the error can be magnified when propagating across

the model via recomputation. To utilize lossy compression for better compression ratio, we want to make sure that the error is within a reasonable bound. MITgcm comes with a gradient check module that reports the gradient of four points sampled from the simulation grid. We compared the sampled gradient computed with compressed checkpoints to that computed with the uncompressed one. We measured the relative error after 32 iterations. By controlling the number of checkpoint budgets, we can study how fast the error propagate across the iterations via recomputation.

In the hs94 experiment, checkpointing with reduced precision introduces only negligible error. ZFP also manages to keep the error lower than 1% even when given only one checkpoint budget. SZ, on the other hand, introduces more error. Both best speed mode and best compression mode result in around 7% deviation from the uncompressed version on the gradient. Considering its extremely high compression ratio, such error is understandable. In this experiment, the checkpointing budget, or the number of recomputations, does not affect the error significantly. This result suggests that errors are not magnified when they propagate across iterations.

Since the halfpipe experiment is computationally intensive, performing many recomputations will be too slow. As a result, we started from a larger amount of checkpoint budgets in this experiment. In the halfpipe_stream-ice experiment, all the methods tested are able to keep the error lower than 1%. Using ZFP or checkpointing in reduced precision introduces almost no error in the result. SZ again brings the most error because of its high compression ratio.

Our experiment results suggest that the error will usually not do much harm when applying lossy compression on checkpoints. For most applications, such as sensitivity

study and parameter tuning, we believe that such an error rate is acceptable. Unless precision is absolutely needed, lossy compression can be used to further increase the checkpointing performance.

4.4. Summary

In this project, we studied the concept of applying data compression to reduce checkpoint size. We implemented compressed checkpointing in the MITgcm framework and evaluated the performance using different compression methods. We also investigated the error propagation caused by lossy compression. Our results show that compression can effectively reduce checkpoint size and increase overall performance when doing checkpointing to a slow device. We believe it is a good answer to the bandwidth limitation of I/O-based checkpointing architectures.

4.4.1. Future work

We briefly describe two possible research direction related to applying compression in checkpointing.

4.4.1.1. Extension to memory-based CP. Although we focus on I/O-based checkpointing in this chapter, the concept can also be applied to memory-based checkpointing. Since memory access can be considered as instantaneous, compression cannot reduce checkpointing time; instead, it reduces space usage. Less space usage means a larger checkpoint budget and less re-computation. Unlike the case of I/O-based checkpointing where I/O time is linearly related to checkpoint size, applying compression on memory-based checkpointing is more complicated. The effect of checkpoint compression on overall

performance depends on the checkpoint scheduling strategy and the complexity of the model. As a result, the effectiveness needs to be studied in a case-by-case fashion.

4.4.1.2. Scheduling algorithm. Many checkpoint scheduling strategies are being proposed that optimize performance on many scenarios. Many of these strategies assume that checkpoint size remains the same across all iterations; however, such an assumption may not hold true when compression is applied. It is not the case after checkpoints are compressed. In addition, many strategies consider that memory-based checkpointing does not take time to complete. This assumption no longer holds when we need time to do compression and decompression. In order to achieve the best performance using compression, a tailored scheduling strategy is needed.

CHAPTER 5

EFFICIENT PARALLEL I/O FOR CHUNKED AND COMPRESSED CLASSIC NETCDF VARIABLES

As the scale of modern HPC systems grows at a rapid pace, the volume of data produced by applications follows. To allow efficient storing, managing, and sharing, many scientific data are now stored in compressed format [**57**]. Due to the diverse nature of scientific data, compression is usually performed inside high-level I/O libraries where the characteristics of the data are known instead of the parallel file system.

A major challenge regarding parallel I/O on compressed data is that most compressed data cannot be partially decompressed. Due to this limitation, accessing part of the compressed data, except a few specially designed algorithms [**58, 59**] for a limited type of operations, requires decompression of the entire dataset. Modifying part of a compressed dataset requires decompressing, re-compressing, and overwriting the entire dataset. It implies that a compressed dataset can only be modified by one process at a time.

One solution adopted by the HDF5 library [**60**] is using a chunked storage layout. A dataset is divided into equal-sized chunks that are compressed independently. To ensure data consistency of a chunk, only one process, called the owner, is allowed to modify it directly. Other processes write to the chunk by forwarding the request to the owner.

The strategy above comes with several tradeoffs. There are communication overheads to manage chunk access and to forward the write requests. Also, the number of chunks

limits the scalability. Finally, since writing to datasets requires communication, all write operations must be collective.

In this project, we introduce a compression feature for variables in the PnetCDF [**9**] library. Scientific applications in various domains, including much of the climate applications, store their data in classic NetCDF format [**61, 62, 63**]. The classic NetCDF format [**7**] is more efficient than other complex file formats, but it does not support compressing data objects. We saw increasing demand from the community for data compression features in NetCDF variables.

We enable variable compression using a chunked storage layout similar to HDF5[**60**]. Our design uses an array-based reference table to organize the chunks. We store chunking and compression-related metadata as special NetCDF attributes under the variable. The file containing chunked variables remains a valid NetCDF file, though chunked variables can only be interpreted by PnetCDF. We took the same approach as HDF5 to ensure data consistency, but we adopt a different policy for picking chunk owners that not only tries to minimize the communication cost but also tries to balance the compression workload among the processes.

Although the concept of chunking is not new, our design emphasizes enabling I/O aggregation and utilizing it to mitigate the limitation of the existing solution. By handling multiple I/O requests together, we can reduce the number of interprocess communications and hence the contention of the interconnect. We can also avoid decompressing and recompressing chunks when a chunk is written more than once. Most importantly, by aggregating requests across all variables, the total number of chunks we handle is more than that if we handle each request individually. Having more chunks to handle allows

us to achieve better load-balancing, parallelism, and hence scalability. We support I/O aggregation through PnetCDF's non-blocking API [**64**].

We evaluated our implementation on Cori [**30**] at the National Energy Research Scientific Computing Center (NERSC), using up to 4096 processes. We tested various I/O patterns, including rank-based appending I/O patterns commonly used by AMR applications, a checkerboard I/O pattern, and I/O kernels from the E3SM [**65**] simulation framework, as well as the Pandana module [**66**] in the NuMI Off-axis ν_e Appearance (NOvA) experiment [**67**].

The experiment results suggest that I/O requests aggregation can significantly improve the efficiency of parallel I/O on chunked and compressed data. Our solution generally achieves 2 to 3 times the parallel write performance compared to HDF5. When multiple variables are involved, our solution with I/O aggregation can be up to 14 times faster than HDF5. Our solution can improve the end-to-end I/O performance up to 2.7 times compared to writing variables without compression on highly compressible datasets.

5.1. Related Work

Many works apply compression to improve parallel I/O performance. Welton et al. employ compression to increase the network throughput between compute nodes and I/O nodes of the file system [**68**]. Filgueira et al. use compression to reduce the communication time between compute nodes and I/O aggregators in MPI-IO [**69**]. Islam et al. utilize compression to reduce checkpointing overhead [**70**]. Bui et al. proposed several techniques, including compression, to improve the I/O performance on IBM Blue Gene/Q supercomputers [**71**].

Chunked storage layout is widely used to enable parallel I/O on compressed data. Hadjidoukas and Wermelinger introduce a compressed data format for the Cubism framework [**72**] that utilizes the block nature of the application to divide the data into chunks for compression. They incorporate efficient wavelet-based techniques and state-of-the-art floating-point compressors [**73**]. Bicer et al. proposed a solution based on a chunked layout to access compressed NetCDF variables in parallel in which padding is being added to compressed chunk to accommodate the future growth [**74**]. Other than a workaround to access the compressed dataset in parallel, chunked storage layout is also used to store data growing along multiple dimensions [**75, 76**]. Zarr is a python package that implements chunked multi-dimensional arrays that can be accessed concurrently by all threads or processes [**77**]. Zarr allows chunks to be transformed using the Numcodecs [**78**] package. N5 is a library providing primitive operations to store chunked n-dimensional arrays [**79**]. It stores every chunk as a single file under a directory representing the array. ADIOS [**80**] supports data compression by compressing individual data blocks in a log-based storage layout.

5.1.1. Data compression in parallel HDF5

HDF5 [**60**] is an I/O library widely used for handling scientific data. HDF5 datasets can be stored in a contiguous layout that flattens the data into a single block or a chunked layout. In a chunked dataset, the data is stored as non-intersecting, fixed-size, and rectangular chunks. When the chunks are stored in the file, each chunk is flattened either in a row-major or a column-major order. The chunks can be stored in any order and at any location. HDF5 uses a B-tree-based data structure to track the location of chunks. HDF5 supports

data compression through filters on chunked datasets. Filters are data transformations applied to chunk data before writing the chunk to the file. Applications can apply different filters to the dataset; some of them provide compression functionality.

The parallel write operation in HDF5 consists of 2 phases. The first phase is to exchange the data among processes.To ensure data consistency, each chunk can only be written directly by a process called the chunk owner. The chunk owner is the process with the most data to write to that chunk. Processes exchange the data such that the entire data of each chunk is aggregated to the owner process. The second phase is to compress the chunks and to write the compressed data to the file. Each process independently compresses the chunks it owns. Then, processes exchange the compressed data size to calculate the write offset in the file and collectively write the compressed chunks to the file. Finally, the file offset of the chunks is written in the b-tree index.

The parallel read operation in HDF5 for compressed datasets is implemented in 4 steps. First, each process calculates the intersection of the requested data space and all the chunks in the dataset. Second, all the processes look up the location of the requested chunks in the b-tree index. Third, the processes collectively read the chunks they need. Finally, each process independently decompresses the chunks and gets the requested part of the chunks.

5.1.2. NetCDF

NetCDF (Network Common Data Form) [**7, 8**] is a self-describing, machine-independent (portable) data format for array-oriented data. A classic NetCDF file contains three types of objects: attribute, dimension, and variable. *Attributes* contain metadata for the file

and variables. *Dimensions* are named scalar which describes a dimension of variables in the problem domain. One dimension, called the 'record dimension', in a NetCDF file can have unlimited size. *Variables* are multi-dimensional array of data (counterpart of a dataset in HDF5). NetCDF variables are always stored in a contiguous layout in which data is flattened into the file in canonical order. The shape of a variable is defined by referring to dimension objects.

A variable that has the record dimension is called a 'record variable'. The record dimension can only be the most significant dimension of the variable. A unit slice of a record variable along the record dimension is called a 'record'. The coordinate of the slice along the record dimension is referred to as the record number. Record variables can be resized arbitrarily along the record (first) dimension. The size of the record dimension increases automatically to fit the variable with most records. There is only one unlimited dimension in a NetCDF file shared by all record variables. As a result, all record variables contain the same number of records. Whenever a record variable gets a new record, the size of all other record variables also increases.

In addition to the classic format, there is also the NetCDF-4 [**81**] format. NetCDF-4 store NetCDF data objects as HDF5 data objects. A NetCDF-4 file is a particular type of HDF5 file that follows NetCDF-4 specification. Compared to the classic format, NetCDF-4 introduces the concept of group as well as features from HDF5, such as chunked storage layout and filters. In this project, we focus on supporting variable compression in the classic NetCDF format.

5.1.3. PnetCDF

PnetCDF [**9**] is a high-level parallel I/O library for managing dimensions, variables, and attributes in classic NetCDF files. Applications can access part of the variable by specifying a subarray of the variable to read or write. In addition to conventional read and write APIs, PnetCDF also provides a set of non-blocking APIs. Non-blocking APIs allow applications to post multiple I/O operations, and let PnetCDF aggregate them into a large request for better performance.

5.2. Design and Implementation

Our approach adopts a chunked storage layout similar to HDF5 to enable compression for NetCDF variables. Applications can enable chunked storage layout and/or compression on individual variables. We designed a data structure to store chunked variables using classic NetCDF data objects. We implement our solution in the PnetCDF library.

5.2.1. Chunking and compression metadata

As the classic NetCDF format does not include metadata entries for describing data chunking and compression, we use the following NetCDF attributes to store such metadata for each chunk-enabled variable. These attributes all have names with the first character '_', which are reserved for special names with meaning to implementations, as restricted in the NetCDF convention [**82**].

- **_chunk_dims** is a 1D integer array attribute of size equal to the number of dimensions of the chunked variable. It contains the dimension IDs previously defined through calls

to `ncmpi_def_dim()`. This attribute also serves as an indicator of a chunked variable. If this attribute is missing, the variable is not chunked (a traditional variable).

- **__chunk__refs** is an attribute storing the starting file offsets of the chunk reference table. For fixed-size variables, this attribute is a 64-bit integer. The chunk reference table is a 1D 64-bit integer array of size equal to the number of chunks of the variable. The table stores the file starting offsets of individual chunks. For record variables, this attribute is a 1D 64-bit integer array of size equal to the number of records. Each of its array elements points to the file starting offset of a record's chunk reference table. There is one chunk reference table for each record, and reference tables can be stored in non-contiguous locations in the file.
- **__chunk__ext__ndims** is an attribute of a 64-bit integer, storing the number of effective records for the variable. Effective records are referred to as the number of records that have been written in the file. This value is less than or equal to the value of the unlimited dimension stored in the file.
- **__filters** is an attribute of an integer array, storing the IDs of predefined filters that are applied on the data chunks. The array indices also indicate the order of filters applied to the data chunks in a pipelined fashion.
- Other filter-specific attributes, such as compression level of lossless compression method, error tolerance of lossy compression method, etc. These attributes will be added when the corresponding filter is incorporated into PnetCDF.

5.2.2. Data chunks and chunk reference table

Two new data objects introduced in our design for describing the chunking and compression settings are the chunk reference table and data chunks. A fixed-size variable's chunk reference table is comprised of two 1D arrays of 64-bit integer type, both of size equal to the number of chunks of the variable. One array is the offset array that stores file offsets pointing to the starting locations of individual chunks. The other is the size array that stores the sizes of each compressed chunk in the file. For record variables, each record of a variable has its own chunk reference table, and the tables of consecutive records are not necessarily stored contiguously in the file.

Data chunks contain the parts of the variable in compressed form. They can be compressed and decompressed independently. Each data chunk occupies a contiguous space in the file, but chunks of a variable are not required to be stored contiguously. This design allows flexibility in file space management but can adversely affect I/O performance if chunks are dispersed all over the file.

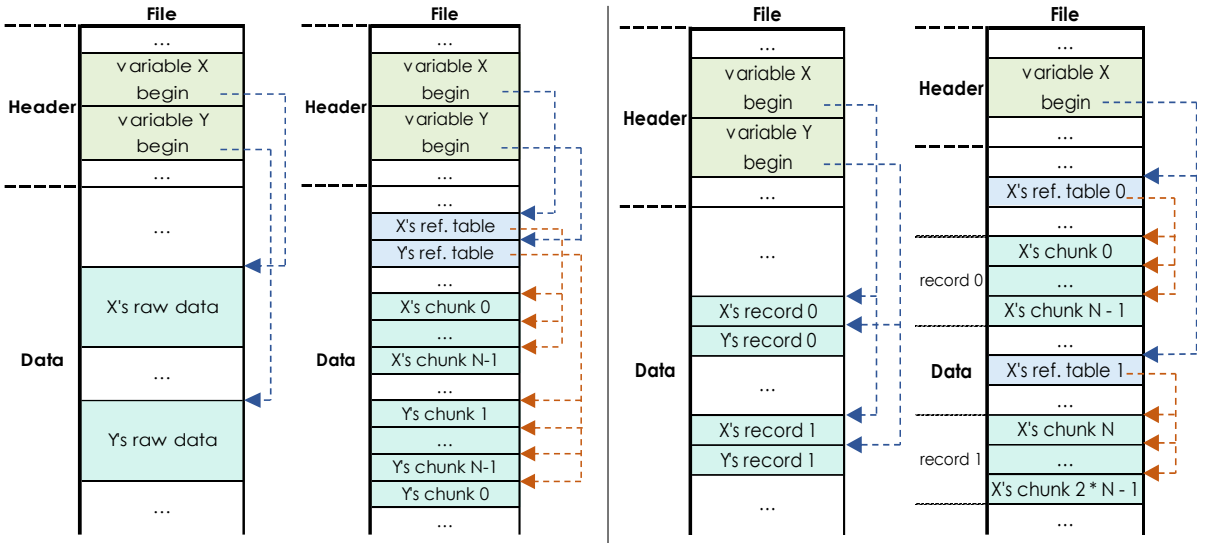
Without violating the NetCDF file format specification, we store the new data objects in the "free space" between variables. The free spaces are paddings between the end of a variable and the beginning of the next variable. Paddings are required to comply with the NetCDF format specification in which variables must be aligned to 4-byte boundaries. The use of such alignment has also been extended in both PnetCDF and NetCDF libraries to align the variable's file space to file system striping boundaries in order to achieve better I/O performance [**83, 84**]. Our design utilizes this feature to make space for chunk reference tables and data chunks.

A new API named `ncmpi_var_set_chunk` is used to set the size of the chunks on a chunked variable. Once chunk sizes have been set, the number of chunks for a fixed-size variable or a record variable's record is known. For fixed-size variables, their reference tables can be allocated when calling API `ncmpi_enddef`. During this time, PnetCDF will check all defined variables and adjust the "begin" fields of all variables to make room for chunk reference tables. For record variables, their reference tables are allocated when new records are created. Because the compressed size of a chunk is not known at the time of `ncmpi_enddef`, the file space for data chunks is not allocated until the application writes to the chunk. The chunk offsets in the chunk reference table are set to -1 to indicate that the chunks haven't been allocated yet.

A new API named `ncmpi_var_set_filter` is used to enable compression for variables and choose the compression method. Our pluggable interface allows incorporation of any compression algorithm, such as deflate [**85, 51**], zstd [**86**], SZ [**54**], ZFP [**53**] ... etc.

5.2.3. Chunk data layout for fixed-size variables

fig. 5.1 illustrates the data layouts of chunked and un-chunked variables. A classic NetCDF file is divided into header and data sections. The file header stores the metadata of dimensions, variables, and attributes, while the data section stores variables' raw data. fig. 5.1(a) shows two traditional un-chunked fixed-size variables, X and Y. When variables X and Y are defined one after another, their metadata is stored consecutively in the file header. The metadata field "begin" of each variable points to the starting file location of its raw data, shown as blue arrows. In this example, a gap appears between the space occupied by two variables' raw data, which is legit to the NetCDF file format specification.



(a) original fixed-size variables (b) chunked fixed-size variable (c) original record variable (d) chunked record variable

Figure 5.1. Data layout of compressed variables versus uncompressed variables. The anchor variable is painted light green. The reference table is painted cyan. The chunk data is painted light green. Note that chunkes does not need to be stored in order.

Our design makes use of such gaps to store the chunked variables, including their reference tables and data chunks.

fig. 5.1(b) shows the file layouts of two chunked fixed-size variables X and Y. Although chunk reference tables are metadata, we store them in the data section of a NetCDF file rather than in the file header. For fixed-size variables, their chunk reference tables are placed at the beginning of the data section. The chunk reference table contains the location of individual chunks, as shown by the red arrows. Entries in the chunk reference table are ordered according to the row-major canonical order of the chunk in the variable. The index of a chunk in the chunk reference table is referred to as the chunk ID and is used to identify the chunk within the library. As depicted in this example, both variables X and Y have N chunks. Chunks are not required to be stored adjacent to each other;

however, our implementation tries to place them in contiguous space if possible for better I/O performance.

5.2.4. Chunk data layout for record variables

fig. 5.1(c) shows two traditional un-chunked record variables, X and Y, with two records each. Record variables have the same header as fixed-size variables except that the first dimension is always the record dimension. The metadata field "begin" of each variable points to the starting file location of its first record, shown as blue arrows. Records with the same record number from all record variables are stored together in a block referred to as a record of the NetCDF file. File records are ordered according to their record number. Paddings are allowed between records of variables and between file records as long as each file record has the same size.

NetCDF format specification requires that file records be stored after all fixed-sized variables [**87**]. To comply with it, PnetCDF needs to move all records downward every time it inserts padding for new chunks. Constantly relocating record variables can degrade the performance significantly. To avoid the issue, we keep the file free of traditional record variables. Traditional record variables are emulated by chunked variables with a chunk size equal to a record's size.

fig. 5.1(d) shows the file layouts of a chunked record variable X with two records. We fix the chunk size along the record dimension to one, so no chunk spans across multiple records. A record of record variables is structurally similar to a fixed-size variable, except they share the metadata with other records. For record variables, their chunk reference tables are placed randomly within the data section, similar to data chunks. We only

allocate the chunk reference table of a record when the application writes the record. For records that are not written, the corresponding field in the **__chunk__refs** attribute is set to NULL to indicate that the chunk does not exist. As depicted in this example, both records 0 and 1 have N chunks. Similar to fixed-size variables, we try to store chunks of a record in a contiguous space for performance consideration. However, chunks from different records are stored separately to avoid the need to relocate existing records.

NetCDF data model allows at most one shared record dimension and requires it to be the first dimension of all record variables. Under this assumption, we can view each record as a fixed-size variable with its own chunks and reference table. A record variable can then be viewed as an array of records that only need to support appending. These properties motivated us to use the lightweight, array-based chunk reference table in our design. In contrast, HDF5 has a more flexible data model in which the number and location of unlimited dimensions are unrestricted. The array-based data structure used in our design may not work in this scenario since it will require frequent insertion and reordering that can degrade performance.

5.2.5. Parallel access policy for data chunks

For parallel access to data chunks, we employ a similar policy as HDF5. Chunks are assigned to a single process called the **owner**. Only the owner can directly modify the chunk in the file. While a process can own multiple chunks, a chunk can only be owned by one process. The owner is responsible for performing compression, decompression, and I/O operations on the chunk. If a process needs to access a chunk owned by another process, it sends a request to the owner of that chunk to have the owner access the chunk

on its behalf. As it requires the participation of other processes, writing to compressed variables must be a collective operation.

Since accessing local chunks does not require communication, chunk ownership assignments can affect the communication overhead. We define the **access size** of a process to a chunk as the size of all intersections between the process's local I/O requests and the chunk. To minimize communication costs, we should assign a chunk to the process with the largest access size. The total size of data being exchanged under this assignment is minimal since the selected chunk owner will have more data to send than any other processes that can replace it. However, compression and I/O workload can also affect the overall I/O performance. Chunk owners are responsible for compression, decompression, performing raw data I/O of the chunk, and handling requests from other processes writing to the chunk. An imbalanced assignment may concentrate the compression and I/O workload onto a few processes, resulting in poor parallelism. In the worst case, a process owning too many chunks may run out of memory for chunk caching. As a result, chunks need to be assigned as evenly as possible, not only for better performance but also to fit within resource limitations.

While HDF5 prioritizes minimizing the communication cost and only consider load balancing when there is a tie, our design adopts a more flexible approach. We introduce a per-process **workload penalty** that is proportional to the total size of chunks a process already owns. For each chunk, we calculate the **preference score** of a process by deducting the workload penalty from the access size. The process with the highest preference score becomes the owner of the chunk. After assigning a chunk to a process,

the **workload penalty** of the process increases to make it more difficult for that process to own another chunk.

5.2.6. Parallel writing to compressed variables

A process writes to a chunk owned by another process by sending its write request to the chunk owner. If a process accesses multiple chunks owned by the same owner, we use MPI datatypes to combine the request, so there is only one message sent to the owner. This strategy differs from HDF5's approach in that HDF5 sends one message per chunk. Each request consists of the chunk ID and the position (a subarray) within the chunk to write, followed by the data to write. The data is converted to match the data type of the variable before packing into requests. If an I/O request spans multiple chunks, the sender breaks it into multiple requests.

A collective write starts by having processes perform a collective communication, so chunk owners know the number of incoming messages they need to receive. We use MPI datatype to encapsulate the metadata and the data of requests without explicitly packing the data into a contiguous memory buffer. If the intersection covers a non-contiguous region, we use an MPI subarray type; otherwise, we use a contiguous datatype with a lighter overhead that benefits small I/O requests.

Chunk owners allocate a memory buffer, called the chunk buffer, for every chunk they own. The chunk buffer store the data of the chunk before compression. When the owner does not completely write a chunk, the owner fills the chunk with the variable's fill value. If the chunk already exists in the file, it is read back and decompressed into the buffer to be merged with the new data. Once chunk buffers are initialized, the chunk

owner process incoming request messages, copying the data to the corresponding place in the chunk buffer. After processing all requests, including the owners' own, the owners compress the chunk buffer.

New chunks are stored together in the "free space" between NetCDF variables. They are arranged in the order of the chunk ID. The file offset of a chunk can be calculated from the size of the chunks. For existing chunks that are being modified, the existing space is reused if the compressed size fits into the existing location; otherwise, it is treated as a new chunk and relocated to a new location. For now, we do not recycle the space as we do not expect frequent modification variables.

We use MPI-IO to write compressed chunks directly into the space reserved for data chunks. If a process owns more than one chunk, we define an MPI file view to write all chunks collectively. For each variable, one process will update the reference table with the new offset and size of the compressed chunks. We overwrite the existing reference table if it exists. Otherwise, we create the reference and update the **__chunk__refs** attribute to point to the new reference table.

5.2.7. Parallel reading from chunked variables

Reading works similar to writing, except the data flows in reverse, from the chunk owner to the process that reads the chunk. A process reading from a chunk sends a read request to the chunk owner and waits for the response. Read requests share the same metadata with write requests to describe the chunk and the data's location within the chunk to read. The chunk owner constructs an MPI datatype to select the data from the chunk buffer and send it to the requesting process. The requesting process also uses MPI datatype to

distribute the data received directly into the application buffer. If the application requests a data type inconsistent with the native type of the variable, the data is converted by the requesting process locally before returning to the user.

Our design relies on chunk owners to read and decompress chunks for other processes. HDF5, on the other hand, supports reading by having each process independently read and decompress the chunks they need. It allows chunk owners to reuse the chunk buffer during the entire session without the concern of data consistency. Also, we avoid performing repeated decompression work when multiple processes are reading the same chunk. From a different aspect, the HDF5s approach enjoys the advantage of low communication overhead since the only collective operation is reading the raw data. Their strategy can be very efficient when a chunk is read by only one process. We will further discuss the pros and cons of the two approaches in the experiment section.

5.3. I/O Request Aggregation

Most applications store their data across multiple variables. Each variable may represent a variable in the simulation or a feature in the gathered data. When applications access a NetCDF file, they often make multiple I/O requests to access multiple variables or to access different parts of a variable. If we can aggregate and handle these I/O requests together, we will have more opportunities to perform optimization. PnetCDF's non-blocking API [**64**] provides native support to I/O aggregation. It allows the application to stage I/O operations in PnetCDF and process them at once.

Aside from reducing the communication overhead, the most important benefit of I/O aggregation is improved scalability. Recall that only the owner can access a chunk directly,

the degree of parallelism is then capped at the number of chunks. Reducing the chunk size to get more chunks may not be feasible because chunks need to have a certain size to achieve a decent compression ratio. Handling multiple requests across different variables at a time allows us to parallelize across chunks from all variables. With more chunks to distribute among processes, we can further extend the scalability.

We provide an example to help illustrate the idea. Consider an application running on N processes that write to M ($<$ N) small variables. Each variable has only one chunk due to its small size. PnetCDF API only allows the application to access one variable per call. Since there is only one chunk, only one process can perform compression and the I/O, serializing the entire operation. If we can handle all requests together, the penalty incurred from owning a chunk prevents a process from winning chunks in another variable. It allows M processes to perform the compression and the I/O in parallel. Since requests to different variables are combined into one message, the number of point-to-point communication does not increase.

Supporting non-blocking I/O on compressed variables requires very little modification to the procedure described in section 5.2.6 and section 5.2.7. We append the variable ID as metadata in each chunk access request so that the chunk owner can tell which variable is accessed when the request message contains requests of different variables. It allows a process to access chunks of different variables owned by the same owner in a single request message. Since the way a process accesses a chunk is irrelevant to the variable containing the chunk, no change is required on the other part of the procedure. The owner of the first chunk of a variable is responsible for updating the variable's reference table.

When the application uses non-blocking I/O, the chunk owner assignment is delayed until the time the aggregated request is flushed. Having the information of all I/O requests allows PnetCDF to make better chunk owner assignments compared to the case where only the first request is visible. When assigning chunk owners for multiple variables, PnetCDF will overlap the communication in assigning a variable with the computation of the access size of another variable to hide the communication overhead.

5.4. Usage

We plan to integrate our implementation into future releases of PnetCDF. The integration will be transparent to applications. Existing applications that do not need chunked storage and compression feature will not be affected by the change.

In the future, we will introduce new APIs for applications to enable chunked storage on selected variables, to specify chunk size, and to set compression filters, similar to NetCDF4. The default chunk size of fix-sized variables is the entire variable, for record variables, it is the size of a record. Currently, we support ZLIB and SZ as compression filters. We plan to introduce other compression algorithms in the future.

The performance of our design relies on I/O aggregation. As a result, we only support non-blocking APIs for reading and writing on chunked variables. Blocking APIs are simulated by calling wait immediately after posting a non-blocking operation. For performance reasons, we highly encourage the use of non-blocking APIs.

5.5. Experiment

We ran experiments on Cori, a Cray XC40 supercomputer at National Energy Research Scientific Computing Center (NERSC) [**30, 31**] boosting both Haswell and KNL nodes.

We used Haswell nodes for our experiment. There are 2,388 Haswell 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. We ran the experiments on Cori's Cray Sonexion 2000 file system, a Lustre with 248 OSTs on 248 servers. We configured our test folder to use 64 stripes and 1 MiB stripe size. The theoretical peak parallel I/O bandwidth under this setup is around 57.54 GiB/s for the lustre file system [**31**] and about 1 GiB/s per compute node [**88**].

We evaluated our solution on two commonly seen I/O patterns in HPC applications – checkerboard data partition pattern and FLASH I/O pattern. We also tested it on I/O kernels extracted from two real-world applications. One of them is the E3SM application [**65**] with a fragment and near-random I/O pattern. Another is the Pandana I/O [**67**] module with a block-appending I/O pattern similar to FLASH I/O. We used a mix of real and artificial datasets. Artificial datasets give us precise control of the compression ratio, while real datasets represent the compression ratio of real-world applications.

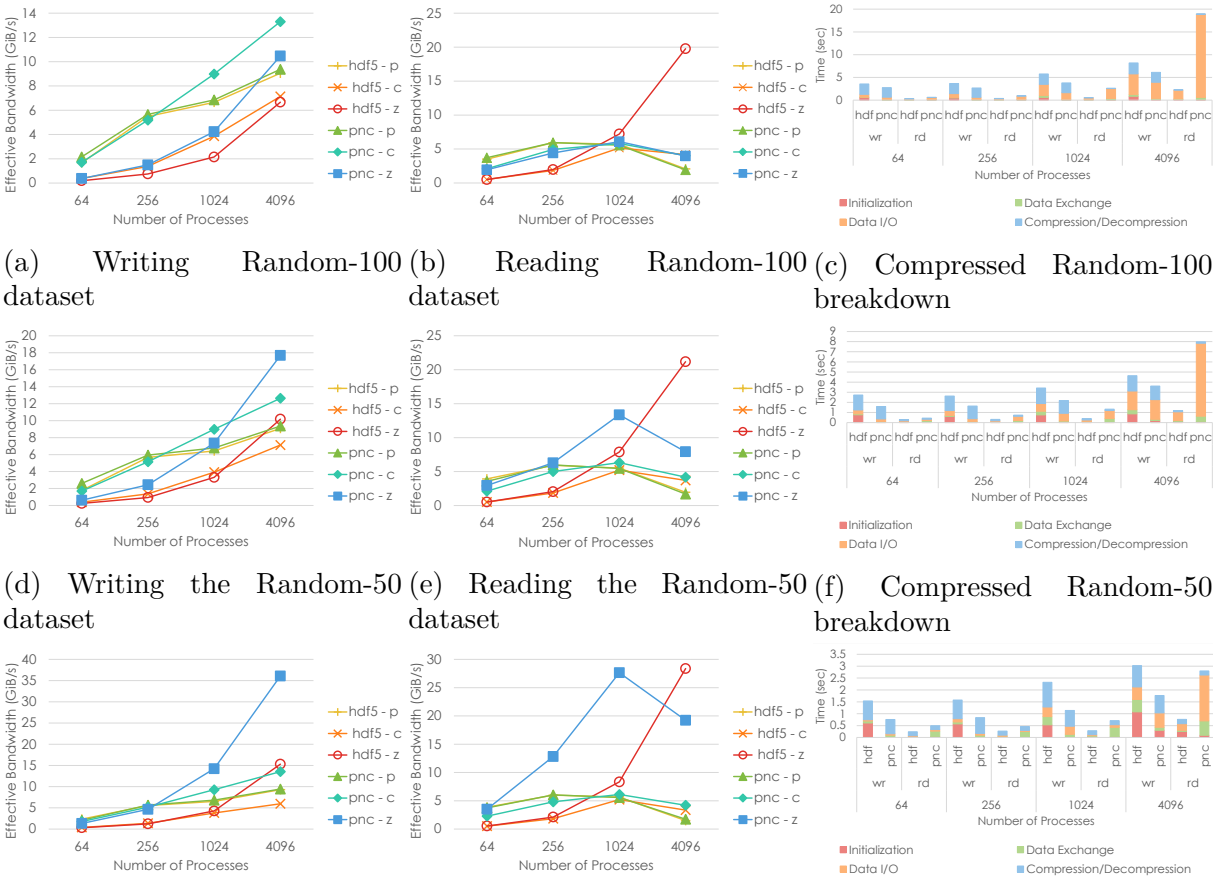
To compare the I/O performance between compressed and non-compressed data, we introduced a measurement call effective I/O bandwidth. The effective I/O bandwidth is defined as the size of the data before compression divided by the total time of I/O operation. The effective bandwidth accounts for the size reduction effect of doing compression.

For each dataset, we compared the effective bandwidth between PnetCDF (our solution) and HDF5 using a contiguous storage layout, chunked storage layout, and chunked storage layout with the level 6 deflate (zlib) filter. We repeated each experiment at least

3 times and take the best result to mitigate the interference from other applications. Experiments of different configurations are interleaved in which reading tests are set as far apart as possible from the corresponding writing test to reduce the effect of file system caching.

We used the latest version of PnetCDF (1.11.2) and HDF5 (1.12.0) at the time of our experiment. Both libraries were built with the default toolchain on Cori and ran with their default configurations. The chunk size and compression related parameters are set to the same for both libraries.

We augmented both PnetCDF and HDF5 to measure time spent in internal functions so we can make a more detailed comparison and identify potential areas for improvement. Since the architecture and implementation of PnetCDF and HDF5 are quite different, it is not possible to make a one-to-one comparison of steps between the two libraries. Instead, we organize them into four types of operations - initialization, data exchange, compression/decompression, and I/O. Initialization includes operations to initialize the data structure to represent compressed variables (datasets), such as building the reference table and reading static variable metadata. The data exchange time is the time spent on exchanging chunk access requests between processes, including packing and unpacking the request messages and communications to synchronize the message size. The compression time is the time spent compressing the data chunks. The I/O step includes writing compressed chunks to the file and updating the chunk reference table.



(a) Writing Random-100 dataset (b) Reading Random-100 dataset (c) Compressed Random-100 breakdown

(d) Writing the Random-50 dataset (e) Reading the Random-50 dataset (f) Compressed Random-50 breakdown

(g) Writing the Random-10 dataset (h) Reading the Random-10 dataset (i) Compressed Random-10 breakdown

Figure 5.2. Checkerboard I/O end to end time (bars) and bandwidth (lines). In the legend, 'pnc' means PnetCDF and our solution, '-p' means contiguous layout, '-c' means chunked layout, and '-z' means chunked and compressed layout. (c), (f), and (i) shows the time spent in each steps on compressed Random-100, Random-50, and Random-10 datasets respectively.

5.5.1. Checkerboard I/O

In the checkerboard I/O pattern, a multi-dimensional variable is divided into fixed-size rectangular subarrays. Each process accesses one subarray. In MPI-IO, it is equivalent to setting the file view with a two-dimensional subarray datatype (MPI_Type_create_subarray).

Checkerboard patterns are commonly seen in simulation frameworks using fixed-sized grids. We used this pattern to study the relation of I/O performance to the compression ratio. To do so, we generated a chunk with completely random numbers so that it is almost not compressible. We mixed it with a different portion of 0s (fully compressible) to create chunks of different compression ratios. We repeated this chunk to form a 2-D variable so that all chunks' size and compression costs are consistent. Using this method, we generated three different datasets. The first one is an uncompressible dataset (random-100) containing all random bits. The second one is a reasonably compressible dataset (random-50) that contains half random bits and half 0s. The third one is a highly compressible dataset (random-10), which contains only 10% random bits. Since the goal was to evaluate our chunked storage solution's performance instead of the underlying compression algorithm, the data's content is irrelevant. Using an artificial dataset allows us precise control of the compression ratio to achieve our goal.

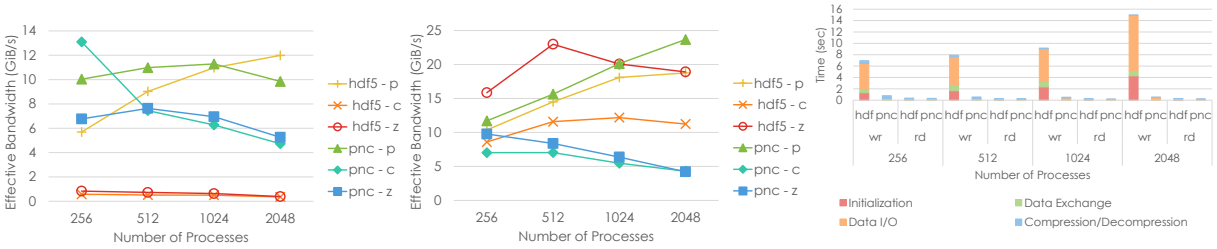
In this experiment, each process writes a 4k by 4k subarray in a squared variable. We set chunk size along each dimension to be twice the per-process subarray size so that each process writes to a quarter of a chunk. This setting simulates the situation mentioned above while creating the need to exchange data for evaluation purposes. The dataset contains only one variable, so aggregation provides no advantage. We ran it on up to 128 nodes with 32 processes per node.

Figure 5.2 shows the result of the checkerboard I/O under different compression ratios. In terms of write performance, PnetCDF out-perform HDF5 in most cases. On 4096 processes, PnetCDF is up to three times faster than HDF5. Timing breakdown suggests that HDF5 spent significantly more time on initialization. In addition to HDF5's

inherently heavier metadata operation, we found that HDF5 always fills new chunked and filtered datasets (variables) with background value regardless of settings (property list), effectively writing the dataset another time. We have no clue about the reason behind their design.

When it comes to reading performance, HDF5 scales better than PnetCCDF. It is a result of HDF5's approach to handling collective read on compressed datasets. As discussed in section 3, HDF5 has each process independently read and decompress the chunks they need. In this experiment, a chunk is only shared by four processes, making independent MPI read faster than the collective one. Since each process only reads from a single chunk, performing repeated decompression across processes does not increase the overall decompression time. Without the communication overhead to exchange data, HDF5 overtook PnetCDF when scaling to 4096 processes.

Chunked and compressed layout out-performs the contiguous storage layout even on the non-compressible dataset due to more efficient I/O pattern [**89, 90**]. While compression cannot improve write performance compared to chunked storage layout except on the highly compressible dataset, it can boost read performance on the 50% compressible dataset in both PnetCDF and HDF5. The main reason is that inflate (decompression) runs significantly faster than deflate (compression) [**91**]. We need more size reduction in I/O to compensate for compression cost than that to compensate for decompression cost.



(a) Writing the GCM dataset (b) Reading the GCM dataset (c) Compressed GCM dataset breakdown

Figure 5.3. FLASH I/O pattern end to end time (bars) and bandwidth (lines) on the Galaxy Cluster Merger (GCM) dataset. In the legend, 'pnc' means PnetCDF and our solution, '-p' means contiguous layout, '-c' means chunked layout, and '-z' menas chunked and compressed layout. (c) shows the time spent in each steps on compressed GCM datasets.

5.5.2. FLASH I/O

In the FLASH simulation framework, the problem space consists of equal-sized blocks. Blocks do not always combine into a single rectangular array as in checkerboard I/O pattern; instead, they can form irregular shapes or multiple discontinued rectangular spaces. Blocks are assigned to processes. A common way to store the blocks on a disk is to stack them one after another that resembles a rectangular variable in which the number of blocks is another dimension. Blocks handled by the same process are usually stored together. The resulting I/O pattern is a block-appending pattern where each process writes blocks in a contiguous space after blocks from processes with a smaller rank. This type of I/O pattern is commonly seen in applications that use adaptive mesh refinement (AMR) [**92**], such as the FLASH code [**93**] and AMReX [**94**].

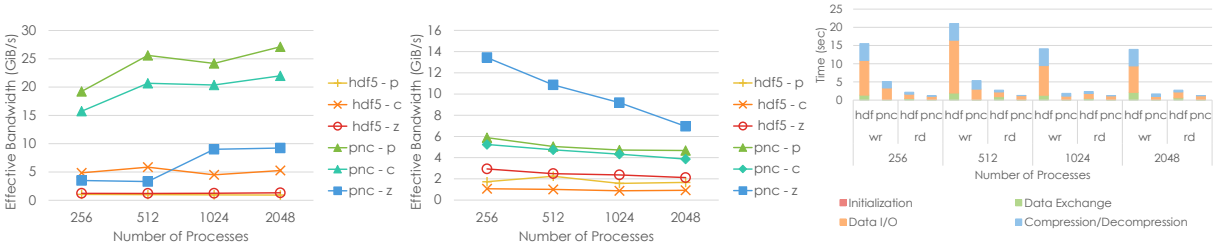
We used the data generated by the Galaxy Cluster Merger simulation, a FLASH [**93**] application, from the sample datasets of the yt project [**95**]. The variables in the dataset are made up by stacking 3-dimensional blocks into a 4-dimensional variable. There are

nine variables of size 43065 x 16 x 16 x 16 in the dataset, totaling 5.91 GiB. We set the chunk size along the stacking dimension to roughly the number of blocks per process to reduce communication overhead.

Figure 5.3 shows the result of Galaxy Cluster Merger dataset. Our solution significantly outperforms HDF5 when writing the compressed variables. The timing breakdown shows that our data exchange and I/O time are noticeably lower than HDF5, suggesting a significant advantage of aggregating nine variables' requests. HDF5 has an edge when it comes to reading. The reason is that we set the chunk size to match the per-process block size. It makes chunk boundaries mostly align with the processes' access boundary, resulting in a chunk-per-process I/O pattern. This kind of pattern favors the independent decompression approach used by HDF5 as there is no repeated read and decompression work among processes. Due to an efficient I/O pattern and a mild compression ratio of 2.65, both HDF5 and our solution cannot achieve higher I/O bandwidth than writing to uncompressed variables.

5.5.3. Pandana I/O

We implemented an I/O benchmark to simulate the I/O patterns in the Pandana framework [**66**] used in the NuMI Off-axis ν_e Appearance (NOvA) experiment designed to study neutrino oscillations [**67**]. In the NoVA experiment, sensors are set up to monitor particle collision events and other events of interest. The events gathered in a round of the NoVA experiment are collected in an HDF5 file. For each type of event, there is an HDF5 group to store the event of that type. Events are stored as a set of 1-D variables. Each variable



(a) Writing the NoVA dataset (b) Reading the NoVA dataset (c) Compressed NoVA dataset breakdown

Figure 5.4. Pandana I/O pattern end to end time (bars) and bandwidth (lines) on the NoVA dataset. In the legend, 'pnc' means PnetCDF and our solution, '-p' means contiguous layout, '-c' means chunked layout, and '-z' menas chunked and compressed layout. (c) shows the time spent in each steps on compressed NoVA datasets.

represents an attribute of the events. Elements at the same index across all variables represent an event.

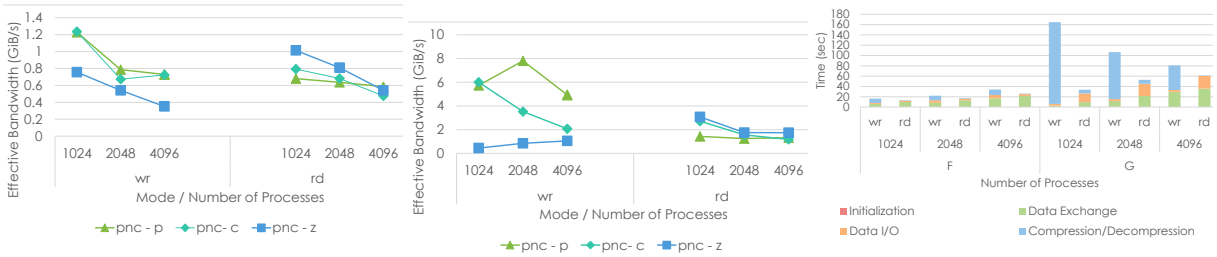
The files generated by multiple rounds of the NoVA experiment are combined into a single HDF5 file and fed to the analysis program. The concatenation is performed group by group. Each process reads the events in the files they are assigned to and appends the events to a contiguous space in the combined file. The I/O pattern of the concatenation resembles the block-appending pattern of the FLASH I/O benchmark.

The analysis program assigns each process a contiguous block of events to analyze and then output the combined result. The read pattern of Pandana I/O also resembles the concatenation process's block-appending pattern except that the I/O size per process can differ. Events are assigned to process based on some predefined rule unrelated to the location of the event before concatenation. Depending on the need of the analysis task, only required types of events are read.

We gathered the data from 1951 rounds of NoVA experiments. We conducted the experiment using a subset of events (groups) used in one of the NOvA experiment's analysis tasks. The subset contains 108 variables organized in 15 groups totaling 7.09 GiB. We set the chunk size to 1 MiB. In this experiment, we assume the read pattern is the same as the write pattern. We ran the experiment on up to 64 nodes with 32 processes per node.

Figure 5.4 shows the result on the NoVA dataset. Despite a decent compression ratio of 4.92, the write performance of the compressed storage layout is poor due to the imbalanced compression workload. The dataset contains 15 groups of varying sizes, some are large while others are small. Small groups do not contain enough chunks to enable good parallelism. Consequently, only a few processes are performing the compression while other processes wait.

With the advantage of I/O aggregation, PnetCDF out-perform HDF5 by a huge margin on all storage layouts for both reading and writing. PnetCDF performs 5 and 14 times faster than HDF5 on writing and reading, respectively, when using a compressed storage layout. Unlike that in FLASH I/O, HDF5 does not perform well on reading. We can see in the breakdown chart that HDF5 spent most of the time reading the chunks. The reason, aside from the lack of I/O aggregation, is the way the dataset is structured. Since data is divided into 109 variables, each variable is relatively small and has only a few chunks. When reading a variable, multiple processes will read and decompress the same chunk for part of the data they need. Those repeated independent read requests from different processes effectively increase the total amount read and put additional workloads on the file system.



(a) E3SM atmorspheric component (b) E3SM oceanic component (c) Compressed E3SM datasets breakdown

Figure 5.5. E3SM I/O end to end time (bars) and bandwidth (lines). (c) shows the time PnetCDF spent in each steps on compressed E3SM datasets. We use 'F' for atmorspheric component, 'G' for oceanic component.

5.5.4. E3SM I/O pattern

Energy Exascale Earth System Model (E3SM) is a coupled model used for modeling, simulation, and prediction of the Earth's climate [**65**]. We evaluated our implementation with a benchmark program that reconstructs E3SM's I/O kernel using the I/O pattern captured by the PIO library [**96**]. The problem domain is represented by cubed sphere grids, which produce long lists of small and non-contiguous I/O requests across MPI processes. On top of that, it involves a large number of variables, resulting in a high metadata handling workload. The E3SM I/O pattern presents one of the most challenging I/O patterns to the underlying I/O library.

We used the data and the I/O pattern collected from a high-resolution simulation of E3SM [**97**]. E3SM contains specific models for each component in the earth system. Each model writes its own output file with a different file structure and I/O pattern. We take the output from the atmospheric component (F case) and the oceanic component (G case). The F case contains 414 variables totaling 15 GiB in size. The largest variables have a shape of 72 x 777602, followed by variables of shape 1 x 777602, and various small-sized

variables. The G case contains 52 variables totaling 80 GiB in size, including variables with a shape of 3693225 x 80, 7441216 x 80, 11135652 x 80, and various small variables.

We adjusted the chunk length along the longest dimension of a variable to control the size of the chunks. The chunk length along other dimensions is set to the dimension of the variable. We set the chunk size to roughly 1 MiB for the F case and 10 MiB for the G case. Due to its smaller size, the F case needs a smaller chunk size to ensure there are enough chunks to divide among the processes. We used a larger chunk size in the G case to reduce the overhead on managing the chunks.

The E3SM benchmark poses many challenges. One of them is the number of variables in the output file. Building the reference table and assigning the chunks of those variables can take be time-consuming. Another challenge is a large number of fragment I/O requests. Packing, sending, and unpacking those requests results in significant overhead to the data exchange phase. They also slow down the chunk owner assignment calculation as there are a large number of requests to consider. The other factor is the highly irregular I/O pattern in which any process may write to regions scattered throughout the entire variable. No matter how the chunk owners are assigned, a large amount of remote chunk access is unavoidable.

Due to the disadvantages mentioned above, we do not expect the chunked and compressed layout to outperform the contiguous layout in terms of overall I/O time. We only hope aggregation can help to manage the communication overhead to an acceptable level. We tried to run the E3SM I/O benchmark using HDF5 API for comparison. Unfortunately, HDF5 could not finish in a reasonable time despite our optimization efforts. The main reason is that HDF5 API only accepts one request to one dataset at a time.

With hundreds of datasets and millions of small I/O requests per process, it is infeasible to process them one by one. For this reason, we focus on comparing our solution to the contiguous data layout in PnetCDF and studying the timing breakdown.

Results are shown in figure 5.5. The overall compression ratio is 2.27 for the F case and 1.99 for the G case. Compared to contiguous storage layout, parallel write to compressed variables only provide 20%∼50% of effective bandwidth in both F and G cases. In terms of parallel read performance, the compressed layout can out-performing the contiguous layout on a smaller number of processes. A possible reason is that the decompression workload is generally lighter than compression.

As we scale up the experiment, the performance of the compressed storage layout drops significantly. The timing breakdown shows that the time spent in data exchange increases with the number of processes. It is caused by the overhead to manage MPI asynchronous communications. The E3SM I/O pattern results in a near-all-to-all communication pattern in the data exchange step. A process has to send chunk access requests to many chunk owners since the data it accesses spans across a large number of chunks. Thus, the number of MPI asynchronous communications increases with the number of processes writing the variables. A large number of simultaneous MPI asynchronous communications can significantly impact the performance, as suggested in [**98**].

5.6. Summary

In this chapter, we introduced the compression feature for classic NetCDF variables. We referenced HDF5's approach and designed a chunked storage layout for the classic NetCDF data model. We compared discussed the pros and cons of our design versus

HDF5's under different I/O patterns. We proposed the concept of using I/O aggregation to alleviate the limitations on parallel I/O performance on compressed data.

We evaluated our solution on a supercomputer using I/O kernels of real-world applications. The result shows that I/O aggregation is very effective at improving parallel I/O performance on compressed data when there is more than one variable involved. Based on this finding, we strongly encourage other developers to support I/O aggregation in their I/O library or application to enable the opportunity for optimization across multiple I/O requests.

We plan to incorporate our work in the PnetCDF library. We hope our compression feature in PnetCDF can accelerate the adaptation of data compression of NetCDF applications.

CHAPTER 6

IMPROVING PARALLEL HDF5 PERFORMANCE WITH LOG-BASED STORAGE LAYOUT

As the scale of HPC (High-performance Computing) systems continue to grow, the bottleneck on parallel I/O performance has become more and more obvious. Most HPC applications perform their I/O tasks through high-level I/O libraries. ADIOS, HDF5, and NetCDF are the most widely used I/O libraries among HPC applications. ADIOS is known to outperform the other two libraries in many HPC applications, especially those with a complex I/O pattern.

A major contributing factor to the performance discrepancy is the way they organize the data in the file. HDF5 and NetCDF store the data in a contiguous layout in which data are flattened into the file space according to canonical order. It requires MPI-IO, the I/O middleware for parallel file access, to perform inter-process communication to reorder the data into canonical order. The overhead on communication can often lead to subpar I/O performance, especially on complex I/O patterns.

HDF5 offers an alternative chunked storage layout in which a dataset is divided into chunks and the canonical order is only enforced within individual chunks instead of the entire dataset. This feature has shown improvement in I/O performance in many applications. However, the effect becomes limited when facing I/O patterns consisting of a high volume of noncontiguous and irregular I/O requests.

ADIOS, on the other hand, adopts a log-based storage layout in which a process always writes a contiguous block in the file regardless of the I/O pattern. Log-based storage layout stores the data as-is along with metadata to describe their logical location. In this way, the expensive overhead of rearranging the data is deferred to the reading stage. Lack of expensive rearranging overhead makes log-based storage layouts less susceptible to complex I/O patterns [**16**].

Despite many optimization efforts, the inherent disadvantage makes contiguous storage layouts difficult to compete with log-based storage layouts on parallel I/O. As applications scale to more compute nodes the performance gap will grow. We expect log-based storage layouts to receive more attention in future HPC applications.

In this chapter, we studied different I/O challenges when storing data in a log-based layout using PnetCDF, ADIOS, and HDF5. We designed two log-based storage layout architectures, one for HDF5 files and another one for NetCDF files. For each file format, We explore a different approach to tackle a major limitation of log-based storage layout - metadata overhead. Because the data is unorganized, log-based layouts need to store additional metadata to keep track of the structure of the data so they can be read. While the size of the metadata is usually negligible, it can exceed the size of the data in some cases. The metadata overhead increases the total I/O amount and negates the performance gain from the improved I/O pattern.

For HDF5 files, we developed an HDF5 Virtual Object Layer (VOL) plug-in that enables a log-based storage layout for HDF5 datasets. The plug-in intercepts HDF5 dataset operations and records write requests into a log-based data structure. We designed a framework to store the data and metadata using HDF5 data objects. We explored several

methods to reduce the metadata overhead inspired by our observation of the I/O pattern in some real-world applications.

For NetCDF files, we extend the PnetCDF library to support a log-based storage layout. We store the log structure in NetCDF data objects. We introduce new APIs that allow applications to define an I/O pattern and associate NetCDF variables with predefined I/O patterns. In this way, the application can define its I/O pattern in an efficient way that minimizes the metadata size.

We evaluated the three designs on Summit at Oak Ridge Leadership Computing Facility (OLCF) [**99, 100, 4, 101**] and Cori at National Energy Research Scientific Computing Center (NERSC) [**30, 31**]. We studied them on various I/O patterns, including rank-based appending I/O patterns, subarray I/O patterns, and the extracted I/O kernels of the E3SM [**65**] simulation framework.

The experiment demonstrates the advantage of using a log-based storage layout. log-layout based VOLachieves up to 9 times the parallel write performance of PnetCDF which is currently the fastest I/O library that stores data in a canonical layout. log-layout based VOLalso outperform ADIOS most of the time, approaching the performance of tailored log-based I/O module in many cases. It suggests that the metadata reduction techniques and other optimizations we incorporated in our design are very effective.

6.1. Related Work

6.1.1. Log-based Storage Layout

Log-based storage layout achieves its parallel write efficiency by deferring the expensive overhead of data rearranging to the reading stage. fig. 6.1 demonstrates the difference

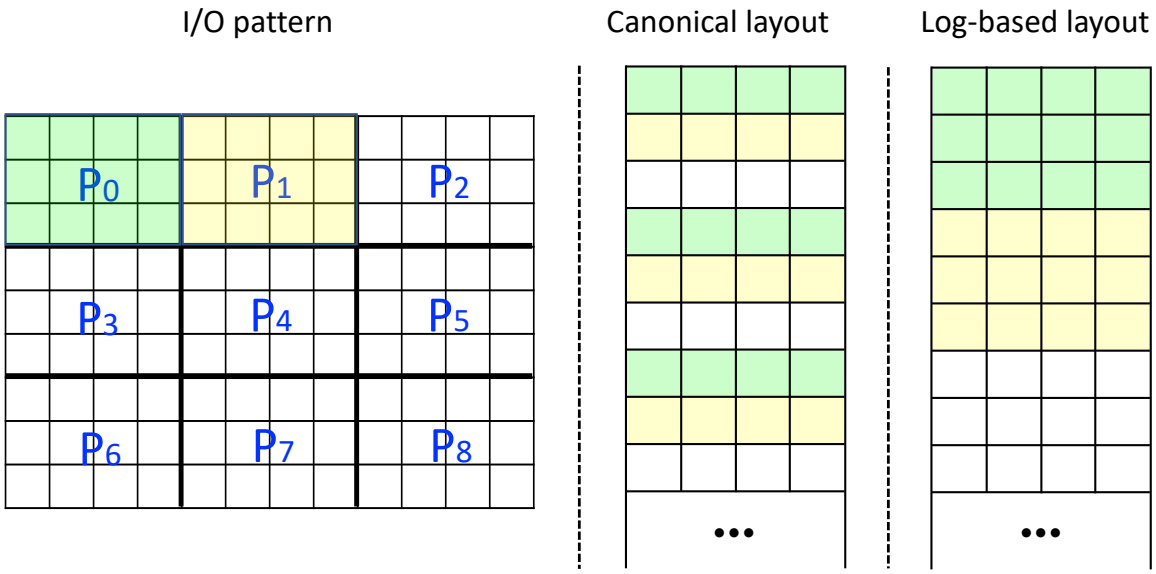


Figure 6.1. Canonical storage layout vs log-based storage layout.

between canonical storage layout and log-based storage layout. Instead of organizing the data in canonical order, a log-based storage layout appends the data as-is and records its logical location, so the data can be reordered later on. It allows applications that are less sensitive to read performance to trade it for write performance.

Log-based storage layout is widely used to organize data across a wide range of applications, including high-level I/O libraries [**80, 102, 103**], low-level I/O middlewares [**104**], and file systems [**105, 106**]. Kimpe et al. introduced a log-based buffering mechanism for MPI-IO that records I/O requests in a log-based data structure and replays it into canonical order when the file is closing [**16**]. Bent et al. developed a virtual parallel file system that remaps the preferred data layout of user applications into a log-based storage layout optimized for the underlying file system [**27**]. Rosenblum and Ousterhout designed

a file system that stores data in a log-based storage structure [**28**]. Dai et al. designed a log-structured file system for buffering sensor data in microsensor nodes [**107**].

6.1.2. HDF5 Data Model

Hierarchical Data Format 5 (HDF5) [**60**] is a self-describing file format popular among scientific applications. The HDF5 Abstract Data Model [**90, 108**] provides a variety of data objects for organizing data.

The data objects highly related to this work are **attributes**, **datasets**, and **groups**. Attributes are metadata that can be attached to describe other data objects. Groups are containers for organizing data objects that can be nested. Every HDF5 file comes with a **root group** that contains all data objects.

A dataset is a high-dimensional array for storing applications' data. The shape of a dataset is specified by a **dataspace**. Applications can access any part of a dataset by setting a **selection** on the corresponding dataspace in the form of **hyper-slabs** (sub-arrays).

6.1.3. HDF5 Virtual Object Layer

While the HDF5 data model is popular in a wide range of scientific applications, it is difficult to cater to the need of all types of applications in a single file format [**109**]. To address this limitation, HDF5 introduces a **virtual object layer (VOL)** [**110**]. HDF5's VOL is an abstraction layer that intercepts HDF5 API calls related to data objects and forwards them to a pluggable module that handles the operation. The module handling the operations is called an **object driver**, or, in short, driver. The VOL allows the

implementation of the HDF5 data model using different file formats and I/O strategies. The VOL is transparent to applications and can be enabled through environment variables. Users can select the driver that fits the specific characteristics of their data and the underlying file system without the need to modify their source code.

A VOL object driver does not have to implement all HDF5 operations from scratch. It can utilize the functions of other VOL drivers. The developer can implement a part of the HDF5 operations and pass other operations to another driver. This stackable design provides flexibility to developers while maintaining compatibility.

The HDF5 library includes 2 object drivers. The **native** driver implements the native HDF5 file format. The **pass-through** driver pass all operation to another driver in a way that is transparent to applications. It serves as an example for driver developers. Other notable object drivers includes the async VOL[**111**], the PDC VOL[**109**], the data elevator VOL [**23**], the cache VOL [**112**], and the DAOS VOL [**113**].

6.1.4. ADIOS and BP File Format

The Adaptable Input/Output System (ADIOS) [**80**] is a high-level unified framework for handling extreme-scale parallel I/O workloads. It has demonstrated superior I/O performance on modern parallel file systems over other parallel I/O libraries on leadership supercomputers credited to the efficient log-based binary pack (BP) format it uses [**114**].

A BP file is made up of a series of process groups stored one after another starting at the beginning of the file and followed by the footer. A process group (PG) is a self-contained log data structure for recording I/O operations from a process in a timestep (a session of file access by the application). Each PG occupies a contiguous space in

the file and is written exclusively by a single process. A process can write multiple PGs. I/O operations in different timesteps are stored in separate PGs. A PG starts with a header containing information about the timestep, the transport methods used, and other metadata of the PG. Immediately after the header is the list of variable write records followed by the list of attribute write records.

For each variable write operation, ADIOS records the name of the variable written, the data to write, the logical location of the data within the variable, the datatype, and other characteristics of the data. Attributes are recorded in a way similar to variables. A special form of variables is **local variables**. Local variables are only a collection of data blocks without any additional metadata. It is left to the application to interpret the data blocks.

Unlike most self-describing file formats that use a header to store the file metadata, BP files place the metadata in a footer at the end of the file. The footer contains three indexes, one for PGs, one for variables, and one for attributes. The PG index is a list in which each entry records the file offset of a PG and the rank of the process that wrote it. The variable index contains references to all variables in the PGs along with a copy of all metadata. Similarly, the attribute index contains references to all attributes in the PGs. Having a copy of the metadata in both the footer and individual PGs allows efficient construction of both local and global views of the variables. A the end of the footer are four 64-bit integers. The first three are the file offset of the indexes, the last one is the version number.

ADIOS 2 [**115**] is the successor of ADIOS. ADIOS 2 introduces two new versions of the BP format - BP4 and BP5. BP4 records data and metadata in separate files to reduce

the number of file-seeking operations. BP5 introduces optimizations to improve efficiency when there are a large number of variables and timesteps.

6.2. Design of Log-layout based VOL Driver

We designed and developed the log-layout based VOL, a lightweight VOL plug-in to enable a log-based storage layout for HDF5 datasets. Instead of recording the content of datasets in the file, the log-layout based VOL records a history of the write operations performed on datasets. The records can be used to reconstruct the data when it is needed. The log-layout based VOL only intercepts and handles HDF5 operations related to the use of a log-based storage layout. Other operations are passed directly to the underlying VOL in a way similar to the pass-through to maximize the compatibility for existing HDF5 applications.

While the concept of log-based storage layout is not new, we introduced several optimizations in our design of log-layout based VOL. They were inspired by our observation of the I/O pattern in some real-world applications and the bottlenecks we encountered when applying log-layout based VOL on them. Some optimizations address limitations of log-based storage layout while others target bottlenecks in the HDF5 framework.

6.2.1. Metadata and Log Layout

The log-layout based VOL creates additional datasets, attributes, and groups in an HDF5 file to store additional data and metadata required for its operation. Data objects used internally by the log-layout based VOL are identified by a special prefix in their name and are hidden from applications. The file created through the log-layout based VOL is

a valid HDF5 file except that the meaning of the content can only be understood by the log-layout based VOL.

We refer to the data structure used to record dataset operations as the log. The log is made up of two parts: a metadata log, and a data log. The data log stores the data of dataset write operations while the metadata log stores other information describing the operation. Separating data and metadata allows the log-layout based VOL to search through metadata entries without the need to do file seeks when handling a read request. A record in the log is a pair of an entry in the data log and an entry in the metadata log. It represents a single write operation to a dataset.

The log is stored in a hidden group under the root group created when the file is created. We store the log entries in one-dimensional contiguous datasets. The log entries are encoded and stored in the datasets one after another. We refer to this kind of encoded (not understandable by other applications) data block that occupy a contiguous space in the file as a **blob**. The metadata log and the data log are stored in separate datasets. There can be multiple datasets for the metadata log and for the data log. The Log-layout based VOL records the number of datasets for the metadata log and the data log as internal attributes.

The log-layout based VOL creates all HDF5 datasets as scalar datasets regardless of the specified dimensions. Since the log-layout based VOL stores the data of datasets separately in log-based data structures, the space allocated for datasets in an HDF5 file is left unused. Making datasets scalar reduces the file size while complying with the HDF5 specification. The original dimensions and maximum dimensions of the dataset are stored

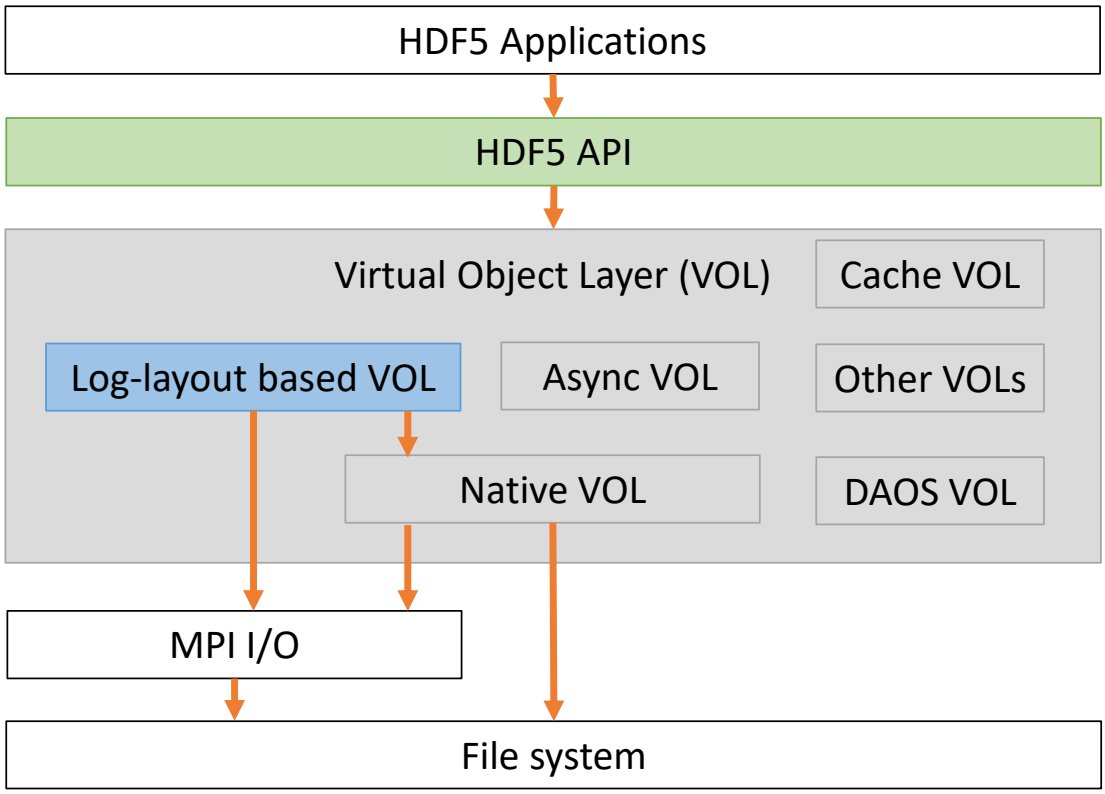


Figure 6.2. Archtecture of log-layout based VOL. The HDF5 API is colored green. The log-layout based VOL is colored blue. The VOl layer and other VOLs are colored gray. Orange arrows indicate the path of I/O requests from applications to the file system. log-layout based VOL is a hybrid VOL, so different types of requests may take a different path.

as hidden attributes of the dataset. When the application queries the dataspace, the log-layout based VOL reconstructs it using the attributes. Each dataset is assigned a unique ID to identify them in the log.

6.2.2. Writing HDF5 Datasets

On each HDF5 H5Dwrite call, the log-layout based VOL appends an entry in the log to represent the H5Dwrite operation. A metadata log entry includes: (1) the ID of the

dataset involved in the operation; (2) the selection in the dataset data space; (3) the offset of the data in the file; (4) the size of the data in the file; (5) the size of the metadata entry; and (6) flags that indicates the endianness of the entry and other information for future extension. The dataset dataspace selection is stored as a list of hyper-slabs (subarrays). If the selection has an irregular shape, the log-layout based VOL decomposes it into disjoint hyper-slabs. Element selections are treated as hyper-slab selections with unit-sized hyper-slabs.

A data log entry contains the data of the part of the dataset selected in the operation. If the dataset space selection contains multiple hyper-slabs, the data is ordered according to the order of the hyper-slabs in the corresponding metadata log entry, ie., the data of the first hyper-slab, followed by the data of the second hyper-slab. If the memory space selection is not contiguous, the data is packed into a contiguous buffer. If the memory data type does not match the dataset data type, the log-layout based VOL converts the data into the type of the target dataset. After type conversion, the data went through the filter pipeline associated with the dataset before being written to the data log.

Appending log entries to the shared log structure requires communication between processes to avoid overwriting each other. Such communication can become a bottleneck if performed on every H5Dwrite call. We tackle this problem using I/O aggregation. On an H5Dwrite call, the log-layout based VOL does not record the operation immediately into the log. It caches the log entries in the memory. The aggregation is transparent to applications. log-layout based VOL stores a copy of the data internally so the application can modify the buffer after H5Dwrite returns. If the application does not reuse the data buffer, data copying can be disabled in the dataset transfer property list (dxpl) to improve

performance. The size of the buffer to store the data is also configurable in the file access property list (fapl).

The data log is flushed when the file is closing, when the datasets are being read, and when the application calls H5Fflush. On each data log flush, log-layout based VOL creates a new dataset for the aggregated data. Each process writes a contiguous block of its data. The metadata log is only flushed when the file is closing or when the metadata is needed by a read operation. On each metadata log flush, log-layout based VOL creates a new dataset for the cached data. In a metadata dataset, each process writes a contiguous block of metadata entries referred to as a metadata section. A section contains exclusively the metadata written by one process. It can be decoded independently no matter how the metadata was encoded. log-layout based VOL records the number of metadata sections and the boundary of each section at the beginning of the metadata dataset.

6.2.3. Metadata I/O

A major drawback of log-based storage layouts is the additional metadata required to describe the unorganized data. The metadata overhead increases the total I/O amount, increasing storage demand and negating the performance gain from the improved I/O pattern. The log-layout based VOL generates a metadata entry per write operation per process. A metadata entry has a fixed size irrelevant to the size of the data. While the size of the metadata is usually negligible, in applications that make many small and fragment write requests, the size of the metadata can become significant, sometimes even exceeding the size of the data. We incorporate several techniques in the log-layout based VOL to minimize the size of the metadata entries.

A direct contributing factor to the size of a metadata entry is the way it is encoded. A log-layout based VOL metadata entry is divided into 2 parts, the header followed by the dataset dataspace selection. The header records the size of the metadata entry, the ID of the dataset, the file offset and size of the data, and a flag indicating the type of encoding used to encode dataspace selection. It allows log-layout based VOL to select proper encoding for different types of I/O patterns.

The most simple form of dataspace selection is a single block (hyper-slab). We record its starting coordinate in the dataspace and the size of the block along each dimension. If the selection consists of more than one block, we record the number of blocks followed by the blocks.

For high-dimensional datasets, representing the blocks by their bounding box is a waste of space. Although HDF5 dataspace allows dimensions that are as large as 2^{64}, in practice, the dataset must fit in the file which is less than 2^{64} bytes. Given the dimensions of a dataspace, there exists a one-to-one mapping between points in the dataspace and 64-bit integers. Instead of recording the high-dimensional coordinates, we record their canonical order when flattened into 1-dimensional space. In other words, if we store the corresponding dataset in a contiguous layout, the file offset of each element represents its coordinate in the dataspace. The size of the block is encoded in the same way. Resizing the dataset, except along the most significant dimension, changes the mapping. We record the current size of the dataset before the blocks so they can be decoded correctly. This encoding method is only used on datasets with at least 2 dimensions and when there is more than one block in the selection.

Compression is a straightforward way to reduce the size of any data. When many blocks are selected in a metadata entry, log-layout based VOL will try to compress the blocks. If compression yields a smaller size, the selection blocks are replaced with the compressed stream. To make the blocks more compressible, we store the offset of the blocks and the size of the blocks in 2 separate arrays. While the offsets are always different, the sizes may be the same in some applications. Storing them together makes the redundancy easier to detect by the compression algorithm.

Another intuitive yet effective method to reduce the metadata size is deduplication. We observed that many applications store their data across multiple datasets. Each dataset represents a variable or property in the problem domain. The dataspace selections on those datasets are highly repetitive as they are based on the same decomposition of the problem domain.

The log-layout based VOL maintains a hash table of all I/O patterns the application used so far. For every new write request, the dataspace selection is compared with known patterns in the hash table. If there is a match, log-layout based VOL replace the blocks with a 64-bit reference to the metadata entry sharing the same I/O pattern. Otherwise, the blocks are recorded as is, and the I/O pattern is inserted into the hashtable.

A special type of write operation is writing records. Some simulation applications run for multiple iterations (time steps) and write out the result at the end of each iteration. The output of each iteration is usually stored in datasets in which a unit slice along the most significant dimension contains the output of an iteration. If all selected blocks in a write request fall within a unit slice of the dataset along its most significant dimension, we consider it a record write. Similar to the case across datasets, the I/O

pattern across records is likely to be identical except for the most significant dimension (iteration number). We store the record number (iterations) separately and eliminate the most significant dimension from the blocks. It does not reduce the size of the blocks but allows us to deduplicate across records. If the I/O pattern of a write request matches a previously seen entry, the log-layout based VOL stores the record number followed by the reference.

6.2.4. Reading HDF5 Datasets

The log-layout based VOL is designed for trading read performance for write performance, so we did not apply any optimizations on dataset read. While there are works that focus on read performance in log-based storage layout [**116**], they are beyond the scope of this work. The log-layout based VOL handles read requests by iterating through the log for entries that contain the requested data and then stitching the data together into the user buffer. It can be very inefficient if the read request involves too many entries. For applications that are sensitive to read performance, we provide a tool to convert log-layout based VOL files into traditional HDF5 files.

H5Dread calls are handled in the following steps. First, the log-layout based VOL flushes all pending write requests to the file so they can be seen by every process. Processes then collectively read and independently decode the metadata entries in the metadata log by iterating through the metadata datasets. The decoded entries are stored in an array-based index grouped by dataset ID. Each process maintains its own copy of the index.

After the index is ready, the log-layout based VOL iterates through the index for any block of the data written to the dataset that intersects with the dataspace selection in

the read request. For each intersection, the log-layout based VOL constructs an MPI datatype to represent the selected part of the data in the data log entry. If the data is filtered, the entire data log entry is selected, and a temporary buffer is allocated to store the unfiltered data. The datatypes of each intersection are then combined into a structure type (MPI_Type_create_struct) for setting the MPI file view to read the data. MPI requires that the file offsets accessed in a file view be monotonically increasing. If the data accessed by two intersections interleave each other, that is, the region between their first byte accessed and their last byte accessed overlaps, their datatypes are broken down into file offset and length pairs and sorted into increasing order.

Another MPI datatype is generated in a similar way to cover the corresponding location in the memory buffer. Data that needs to be unfiltered or type converted is read to the temporary buffer. Data that does not need any post-processing is read directly into the user buffer. The log-layout based VOL then uses the combined structure types to collectively read the data from the file. The final step is to un-filter the data and copy the selected part into the user buffer.

For datasets that are associated with a fill value, the user buffer is pre-initialized to the fill value of the dataset before being overwritten by the data gathered in the log. If no log entries intersect a region in the selection, the corresponding user buffer will not be touched, leaving the fill value in the user buffer. This approach supports fill value without any I/O operation by taking advantage of the way read requests are handled under log-based storage layouts. Canonical storage layouts, on the other hand, do not have this opportunity and must actually write the fill value to the file.

A potential problem of this design is that the size of the index may be large. Unlike in the case of writing where each process only needs to store the metadata generated by itself, the index for reading must contain all metadata entries in the file, effectively caching the entire metadata log in the memory of each process. As log-layout based VOL records a metadata entry per process per write operation, the size of the metadata increases proportionally to the number of processes under the same I/O pattern. On a larger scale, there may not be enough memory to index all metadata entries. We allow the user to specify the amount of memory that can be used on the index. If the available memory is not enough to index all metadata entries, log-layout based VOL will perform index searching on one section of metadata at a time.

6.3. Experiment Setup

We ran experiments on Summit at Oak Ridge Leadership Computing Facility (OLCF) [**99, 100, 4, 101**] and Cori at National Energy Research Scientific Computing Center (NERSC) [**30, 31**]. Summit is comprised of approximately 4608 IBM Power System AC922 nodes connected by Mellanox Dual-rail EDR InfiniBand in a Non-blocking Fat Tree topology providing a node injection bandwidth of 23 GiB/s. Each node has 2 IBM POWER9 processors providing 44 cores/176 threads and 512 GB DDR4 memory. We ran 84 processes per node on Summit. Should the number of processes used not divide 84, we allow some nodes to host fewer processes. The test file system on Summit is an IBM Spectrum Scale GPFS consisting of 77 Elastic Storage System units [**117**] with a theoretical peak performance of 2.5 TiB/s or around 8 GiB/s per node [**118**].

Cori is comprised of 2388 Haswell nodes and 9688 KNL nodes [**119**]. The interconnect is Cray Aries with Dragonfly topology providing 5.625 TiB/s global bandwidth. A Haswell node has 2 Intel® Xeon™ E5-2698 v3 processors providing 32 cores/64 threads and 128 GB DDR4 memory. A KNL node has 1 Intel® Xeon Phi™ 7250 processors providing 68 cores/272 threads and 96 GB DDR4 memory and 16 GB MCDRAM. We ran 64 processes per node on Cori. Should the number of processes used not divide 64, we allow some nodes to host fewer processes. The theoretical peak parallel I/O bandwidth under this setup is around 115 GiB/s for the lustre file system [**31**], or about 1 GiB/s per Haswell node and 350 MiB/s per KNL node [**88**].

Since log-based storage layouts produce additional metadata, the I/O size differs from canonical layouts and varies across different designs. For a fair comparison, we introduced a measurement called effective I/O bandwidth. The effective I/O bandwidth is defined as the size of the data written by the application divided by the total time of I/O operation. The effective bandwidth accounts for the metadata overhead of log-based storage layouts.

We evaluate our designs in three different I/O patterns, a block-appending pattern, a checkerboard partitioned subarray pattern, and the I/O kernel of the E3SM simulation framework [**65**]. For each I/O pattern, we compared the effective bandwidth between our two different designs of log-based storage layout along with ADIOS2. We also include canonical storage layouts of PnetCDF and HDF5 as a reference. As the file system is shared, we report the highest measured bandwidth across at least 3 runs to mitigate the interference from other applications. Experiments of different configurations are interleaved to reduce the effect of file system caching.

We used the latest version of PnetCDF (1.12.3), HDF5 (1.13.0), and ADIOS (2.7.1) at the time of our experiment. Both libraries were built with the default toolchain on the experiment machine and ran with their default configurations.

6.4. E3SM Case Study

Table 6.1. E3SM I/O Dataset Properties and Configurations

Property\File	**F-H0**	**F-H1**	**G**	**I-H0**	**I-H1**
Number of processes	21600	21600	9600	1344	1344
Total size of data (GiB)	14.09	6.68	79.69	86.11	0.36
Number of fixed sized variables	15	15	11	18	10
Number of record variables	399	36	41	542	542
Number of records	1	25	1	240	1
Number of partitioned vars	25	27	11	14	14
Number of non-partitioned vars	389	24	41	546	538
Number of non-contig requests	174953	83261	20888	9248875	38650

Energy Exascale Earth System Model (E3SM) is a coupled model used for modeling, simulation, and prediction of the Earth's climate [**65**]. We evaluated our implementation with a benchmark program that mimics E3SM's I/O kernel using the I/O pattern captured by the PIO library [**96**]. The problem domain is represented by cubed sphere grids, which produce long lists of small and non-contiguous I/O requests across MPI processes. Also, E3SM writes a large number of variables, generating a high amount of metadata. It represents one of the most challenging I/O patterns for high-level I/O libraries.

We used the I/O pattern collected from a high-resolution simulation of E3SM [**97**]. E3SM contains specific models for each component in the earth system. Each model writes its own output file with a different file structure and I/O pattern. We take the output from the atmospheric component (F case), the oceanic component (G case), and

the land component (I case). The properties of their output and configurations are shown in table 6.1. The F case produces two files - H0, and H1. It generates a large amount of small non-contiguous requests. The G case writes one file. Compared to the F case, the G case generates fewer but larger requests. The I case produces two files - H0, and H1. It generates a lot of fragment requests like the F case. The difference is that the I case's requests do not fully cover the logical space of datasets. Also, the requests span across many records (time steps).

The E3SM I/O pattern presents a great challenge to I/O libraries. Its data span hundreds of variables and hundreds of records. The space-filling curve used to divide the problem space cause the processes to issue a large number of fragment I/O requests and a near-random (from the perspective of the underlying I/O library) I/O pattern. Canonical storage layout performs poorly under this pattern due to tremendous communication overhead to rearrange fragments and near-random requests. Log-based storage layouts do not perform well either. Although it does not rearrange the data, a log-based storage layout requires additional metadata to describe the I/O operation in every log entry. With hundreds of thousands of fragment requests, the metadata size can easily grow beyond the size of the data, increasing the write amount and I/O time by many folds. To the best of our knowledge, no existing I/O library can achieve satisfactory performance on raw E3SM I/O pattern.

To improve I/O performance, the E3SM team developed a tailored I/O module called Scorpio which writes E3SM data using the ADIOS library. Scorpio stores E3SM data in ADIOS's local variables in which ADIOS does not record any metadata. The canonical location of the data is stored in separate variables referred to as the **decomposition**

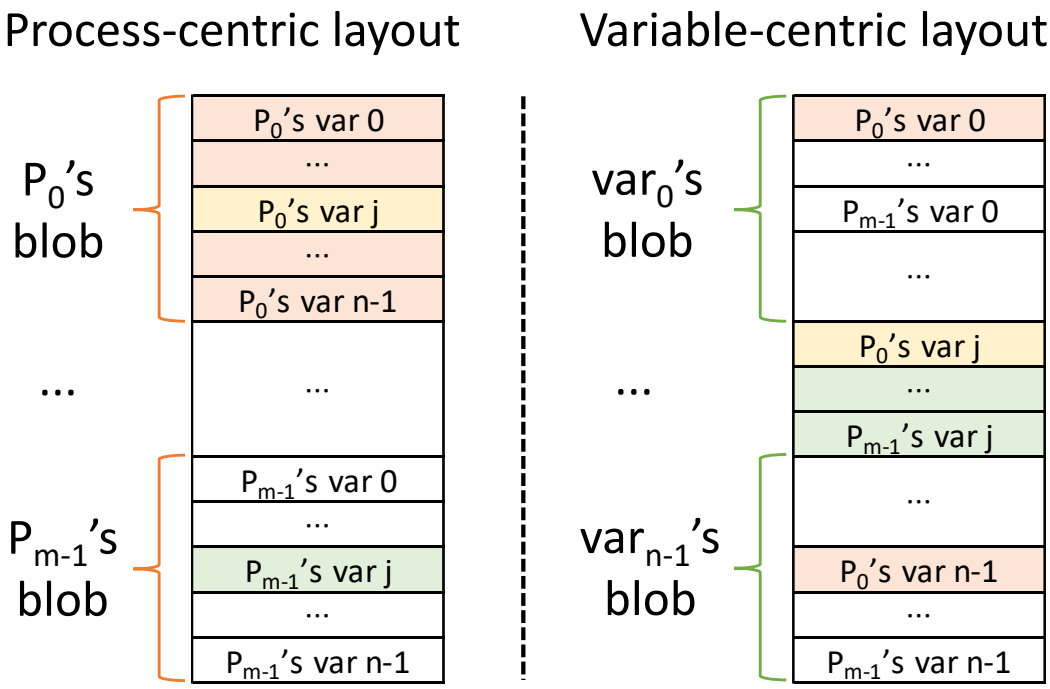


Figure 6.3. Process-centric log-based layout vs variable-centric log-based layout. Rank 0's blobs are shaded red. Variable j's blobs are shaded green. The blob belonging to both rank 0 and variable j is shaded yellow.

map. Each local variable is linked to a decomposition map so that the canonical order of the data can be reconstructed if needed. With knowledge about the I/O pattern and application need, the size of the decomposition map can be reduced significantly compared to what ADIOS would have generated. In this way, the Scorpio module can take advantage of ADIOS's log-based storage layout while avoiding most of the metadata overhead.

6.4.1. Supporting log-based data layout in PnetCDF

Inspired by Scorpio's approach, we designed a similar log-based storage layout module inside PnetCDF [**9**]. For simplicity, we refer to this I/O method **PnetCDF-log**. We

introduce new APIs in the PnetCDF library that allow the application to define NetCDF variables in a log-based storage layout. Unlike log-layout based VOL which encodes the selection in each I/O operation into metadata, the metadata is defined a prior by the application.

Similar to other log-based storage layout designs, we adopt sub-filing to reduce lock contention in the parallel file system. The processes are divided into groups and processes in each group access only one subfile. By default, PnetCDF will create a NetCDF file for each compute node. In each subfile, we store as a file attribute the total number of processes (across all subfiles), the number of decomposition maps, and the number of subfiles. There is also a new dimension, `nproc`, representing the number of processes sharing the NetCDF file.

6.4.1.1. The decomposition map. The concept of **deomposition map** is first introduced in the PIO library [**120**]. A decomposition map defines the region of responsibility of each process within a global variable's data space. For each process, the decomposition map records a list of file offsets accessed by that process relative to the beginning of the variable. When writing to variables associated with a decomposition map, the application does not specify the location to write and the location defined in the decomposition map is assumed. The role of decomposition maps is comparable to the metadata login the log-layout based VOLas they both hold the metadata of variable write operations.

We borrow the concept of a decomposition map into PnetCDF. In PnetCDF, a decomposition map records the high-dimensional subarrays (represented by start and count) subarrays accessed by each process instead of flattened file offsets used in PIO. This design allows the coalescing of adjacent accesses to reduce the size of the decomposition map.

Despite the difference, we also refer to its decomposition map (PIO's terminology) as they serve the same purpose.

A new API named `ncmpi_def_decom` is used to define a decomposition map in a NetCDF file. The user specifies the dimensions to decompose in a way similar to defining variables. The subarrays are specified by a list of starting coordinates and a list of edge lengths similar to that in `ncmpi_put_varn` APIs. Each process calls `ncmpi_def_decom` with the subarrays they are responsible for. A decomposition map can be analogized to an HDF5 dataspace with a predefined selection for each process. Each decomposition map is assigned a unique ID which is used to associate them with variables.

We store the decomposition map as well as other metadata as NetCDF variables and attributes. A decomposition map is stored as two dimensions and five variables. The first dimension, `Dx.nelems` represents the total number of elements in the decomposition map, which is equal to the product of the size of the decomposed dimensions. The second dimension, `Dx.max_nreqs` represents the maximum number of subarrays accessed among processes sharing the NetCDF file. Variable `Dx.nreqs` is a 1-dimensional variable of dimensions same as the number of processes writing to the file ((P)). It indicates the number of subarrays accessed by each process. The variable also stores the ID of the decomposed dimensions of the decomposition map as its attribute. Variable `Dx.blob_start` and `Dx.blob_count` is a 1-dimensional variable of dimensions (P). They represent the offset (relative to the beginning of the variable) and size (in the unit of variable elements) of the data from each process in a decomposed variable. Variable `Dx.offsets` and `Dx.lengths` is a 2-dimensional variable of dimensions (P)x Dx.max_nreqs. They are the starting coordinates and size of subarrays accessed by each process encoded into 64-bit

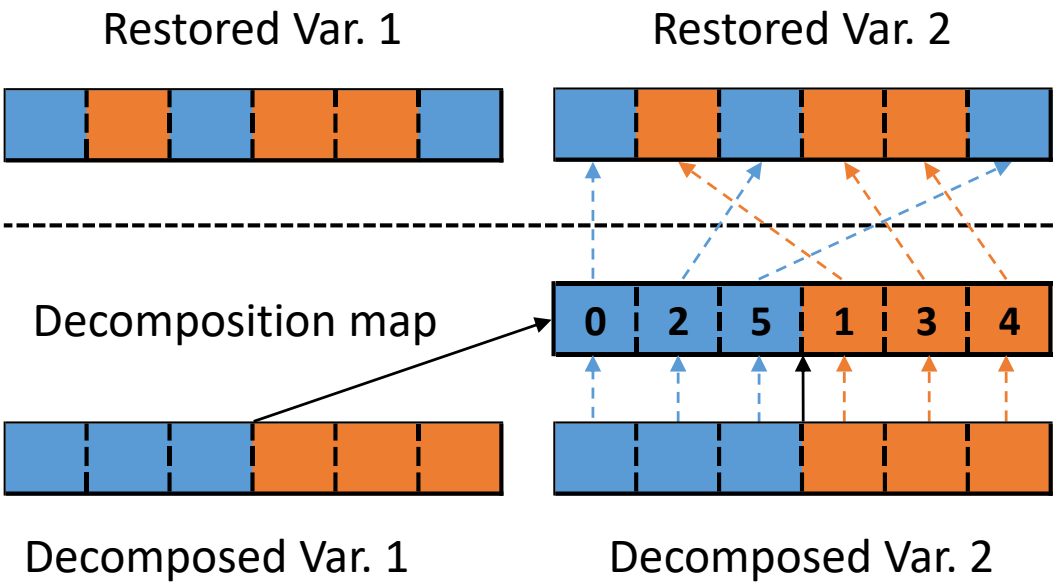


Figure 6.4. Two decomposed 1-dimensional NetCDF variables sharing the same decomposition map. The variables are shared by two processes. Blue and orange cells are written by rank 0 and 1 respectively. The lower part shows the decomposed variables and their decomposition map. For simplicity, we only show variable `Dx.offsets` in the decomposition map. The solid black arrow indicates the association between the variables and the decomposition map. The numbers in the decomposition map indicate the original position of each element in the decomposed variables. The upper part demonstrates the restored traditional variables and the original I/O pattern. The dashed arrows demonstrate the reconstruction of the original data layout from the decomposed variable using the decomposition map. It is only shown for variable 2 to make the figure clearer.

integers in a way similar to that in section 6.2.3. Note that processes access a different number of subarrays, so some elements of the variables are left empty. All decomposition map variables contain an attribute describing their content to make them easier to understand by third-party developers.

6.4.1.2. Decomposed variables. The variable stored in a log-based storage layout is called a **decomposed variable**. To define a decomposed variable, the application first defines its decomposition map and then associates the variable with the decomposition map. A variable can have at most one decomposition map, but a decomposition map can be associated with multiple variables. A decomposed variable must be accessed with the same number of processes defining the decomposition map. Variables without a decomposition map are considered traditional variables.

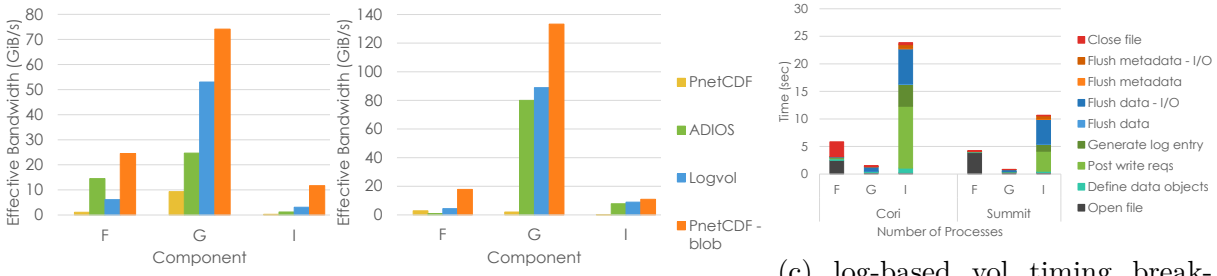
A new API named `ncmpi_def_decom_var` is used to define a decomposed variable in the log-based storage layout. Its interface is similar to `ncmpi_def_var` except for an additional argument to specify the decomposition map. A variable can be partially decomposed, meaning that it contains non-decomposed dimensions, such as the time (record) dimension. The non-decomposed dimensions precede the dimensions defined in the decomposition map. Note that a NetCDF dimension can be used multiple times in a variable, so a dimension can be a variable's non-decomposed dimension while being part of its decomposition map. A decomposed variable has the dimension representing the size of its decomposition map attached to the end of its non-decomposed dimensions. For example, a 2-D variable of dimensions row x col decomposed by decomposition map 1 is created as a 3-D variable of dimensions row x col x `Dx.nelems`. The additional dimension is hidden from applications. In a decomposed variable, the data along the decomposed dimensions are reordered into a log-based storage layout in which each process writes a single block of data after the blocks of processes of smaller rank. The original order is defined in the decomposition map so that a traditional variable can be reconstructed later on.

fig. 6.4 demonstrates two processes writing two decomposed variables sharing the same decomposition map. Rank 0 writes 3 elements at positions 0, 2, and 5. Rank 1 writes 3 elements at positions 1, 3, and 4. The first half of the decomposition map represents the position of the elements written by rank 0, and the second half represents that for rank 1. When writing a decomposed variable, processes always write all of its data to a contiguous region regardless of the I/O pattern. In this figure, rank 0 writes the first 3 elements, and rank 1 writes the later 3 elements. To reconstruct the original variable, we read each element in the decomposed variable and write them to the position indicated by the number in the decomposition map at the corresponding index.

The same API for accessing traditional variables is used to access decomposed variables. Applications specify the selection in the non-decomposed dimensions, ie., dimensions other than the least significant one. The selection within the decomposed dimensions is defined by the decomposition map. At each selected coordinate, the data of the decomposed dimensions is packed into a blob and written to the corresponding 1-D subspace at the offset indicated by `Dx.blob_start`.

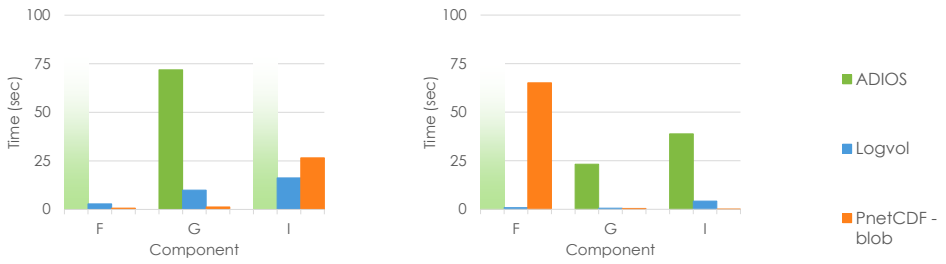
6.4.1.3. Comparison with log-layout based VOL and Scorpio. This design differs from log-layout based VOL and Scorpio in the way data are organized in the file. Under this design, each process writes one blob per variable per process instead of a single contiguous space in the file no matter how many variables are defined and written in the file. In other words, there is only one blob per MPI process in the file in contrast to one blob per variable per process in PnetCDF. We refer to it as `variable-centric` data layout, and the design in log-layout based VOL `process-centric` data layout. ADIOS's BP file format and our log-layout based VOL also use the `process-centric` data layout.

fig. 6.3 demonstrates the different between `variable-centric` data layout and `process-centric` layout. Compared to `variable-centric` data layout, `process-centric` layout results in a simpler I/O pattern. The data is flushed in a single MPI collective write call to a contiguous space in the file. It also creates fewer internal data objects that the underlying library has to manage because all data are stored in a single variable. However, the memory footprint can be large, as all write requests during the entire file open-to-close session must be cached internally. It is also very expensive to append new records to record variables after the file is written. To maintain the `process-centric` layout, the blobs of the later process must be relocated to accommodate growing previous blobs. Processes have to read back their blob, pack it with newly written data, and then repeat the flushing procedure, The `variable-centric` layout does not incur such penalties as the blobs are stored inside each variable. As long as the underlying library supports the expansion of variables, no further steps are required to append new records to the file.



(a) Cori (b) Summit (c) log-based vol timing breakdown

Figure 6.5. E3SM I/O effective write bandwidth comparison. (c) shows the time log-layout based VOLspent in each step in E3SM I/O. We use 'F' for the atmospheric component, 'G' for the oceanic component, and 'I' for the land component.



(a) Cori (b) Summit

Figure 6.6. E3SM I/O log-based layout replay utility read time comparison. ADIOS cannot finish in a reasonable time in some cases. They are represented by bars that fade out gradually.

6.4.2. E3SM I/O performance evaluation

We compare E3SM performance between PnetCDF, ADIOS with Scorpio, log-layout based VOL, and PnetCDF-log. PnetCDF stores the data in a single shared file while other I/O methods write one file per compute node (subfiling). For PnetCDF that uses a shared file setup, the Lustre on Cori is configured to 128 stripes and a 16 MiB stripe size. For other I/O methods that employed subfiling, the Lustre on Cori is configured to 8 stripes and a 1 MiB stripe size. The reason for using 8 stripes per subfile is to use more than one aggregator in an MPI collective write call as a single core cannot saturate the bandwidth of the NIC and the I/O capacity of an OST [**88**]. Although the ADIOS community suggests setting Lustre to one stripe (one OST per node)[**121**], we use the same setting as other log-based I/O methods for a fair comparison. We also tested ADIOS using one stripe, but the difference in performance is negligible.

The write performance numbers are shown in fig. 6.5. The advantage of log-based storage layout and subfiling over a single shared file canonical storage layout is obvious. Conventional HDF5 with canonical storage layout cannot finish in a reasonable time, so

it was not presented. PnetCDF was able to finish due to its ability to aggregate the fragment I/O requests [**64**] into large and more efficient requests. Still, log-layout based VOLis 5.4, 5.7, and 10.9 times faster than PnetCDF on F, G and I case respectively.

Log-layout based VOLperforms better than ADIOS in almost all write cases. Note that ADIOS enjoy the assistance of the Scorpio module while log-layout based VOLdoes not. We tried to run ADIOS without the Scorpio module but it took too long to finish. Despite handling a more complex I/O pattern (metadata), log-layout based VOLstill out-perform ADIOS. The reason is the difference in I/O methods used by the two libraries. Log-layout based VOLwrites log entries using MPI collective IO while ADIOS by default has a single process writing the log [**122**] using POSIX I/O directly. On Cori, an output file is striped across 8 OSTs. Under this setup, MPI (ROMIO) will use 8 processes (aggregators) per node (subfile) to handle a collective write request compared to only 1 in ADIOS. The gap is much narrower on Summit because MPI adopts a one aggregators per node policy similar to ADIOS on GPFS.

PnetCDF-logperforms the best among all log-based I/O methods, surpassing log-layout based VOLby a noticeable margin thanks to a tailored I/O rearrangement layer combined with the lightweight NetCDF file format.

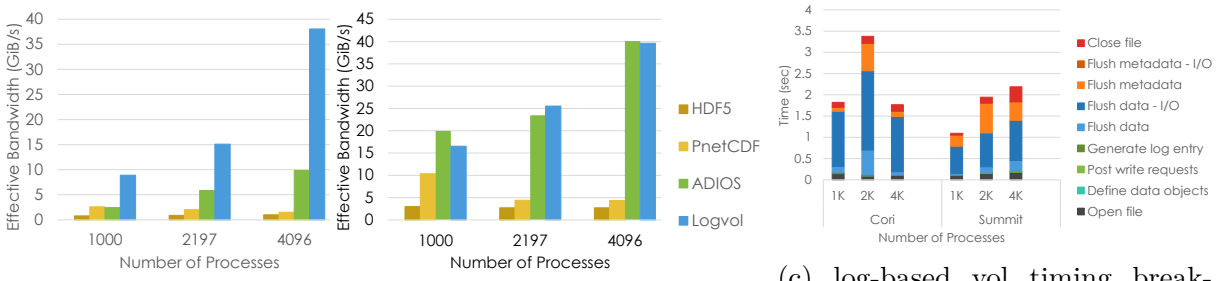
As described in section 6.2.4, reading from datasets stored in a log-based layout directly is very inefficient and unscalable. Due to the data being unstructured, the library has to compare the selection of the read request with the selection in each log entry one by one. E3SM's fragment I/O pattern results in millions of log entries across hundreds of output subfiles. In this case, reading directly from the output file becomes infeasible.

A common workaround is to convert the data from a log-based layout to a canonical layout when read performance is a concern. The conversion process is often referred to as **replay** as it replays the write request in each log entry using an I/O method that stores data in a canonical storage layout. Most I/O libraries using a log-based storage layout come with a replay utility in case read performance is required. Unlike handling read requests, the conversion process does not need to compute the intersection and the workload is fully scalable. Log entries are evenly distributed to processes and each process only has to read the entries it is assigned.

Instead of evaluating read performance directly on the log-based files, we evaluate read operation in the conversion process. section 6.4.1.3 compares the time spent on reading the data and the metadata by the replay utility of the Scorpio module, log-layout based VOL, and PnetCDF-log. The writing time is comparable to writing E3SM data directly in a canonical storage layout. It is not presented as they are several magnitudes higher than the reading time, and will make reading time too small to read.

PnetCDF-logachieves the fastest read time on the F and the G case NetCDF file format is simple and efficient. log-layout based VOLperforms better on the I case because the conversion tool of PnetCDF-logconverts one record (time steps) at a time. The I case contains 240 records, making each read request small. Scorpio performs poorly on this task. In many cases, it cannot finish in a reasonable time and thus was not presented. While the exact cause requires further investigation, our preliminary study found that Scorpio's replay tool handles one variable for one record at a time. It cannot finish in F and I cases because the F case has hundreds of variables, and the I case has hundreds of records.

6.5. S3D I/O



(a) Cori (b) Summit (c) log-based vol timing breakdown

Figure 6.7. S3D I/O effective write bandwidth comparison. (c) shows the time log-based vol spent in each step handling the S3D output file.

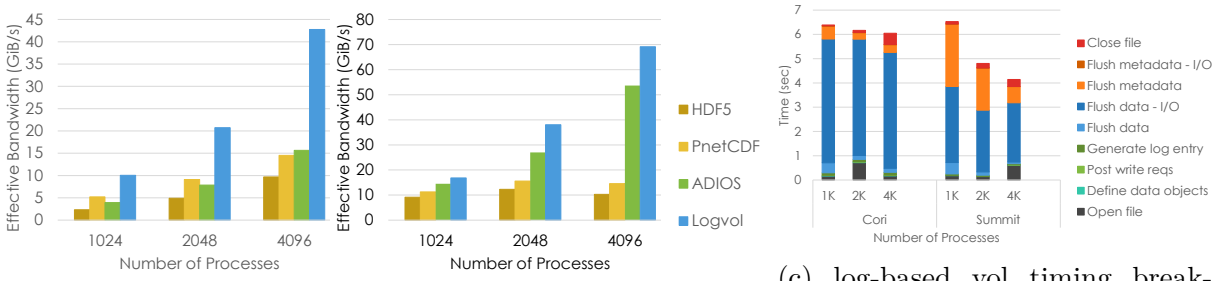
The S3D combustion simulation [**123, 124**] produces a subarray I/O pattern in which the high dimensional problem space is sliced perpendicular to its dimensions into rectangular blocks. Each process handles one or more blocks. The data is usually stored as datasets (arrays) of the same shape as the problem space in which every process writes its data to subarrays corresponding to their blocks. In MPI-IO, it is equivalent to setting the file view with one subarray datatype (MPI_Type_create_subarray) for each block.

Subarray patterns are commonly seen in applications using fixed-sized and fixed-resolution grids in their solution space. Applications that produce subarray I/O patterns including S3D[**123, 124**], the VORPAL plasma simulation code [**125, 126**], the Global Cloud Resolving Model [**127, 128**]... etc.

The data contains two four-dimensional datasets and two three-dimensional datasets of double type. Datasets are sliced along their lower three dimensions with an even number of slices along each dimension. The blocks are 50 x 50 x 50 along the lower three dimensions, and 11 and 3 in the fourth dimension respectively. Each process handles one block, implying that the number of processes is cubic.

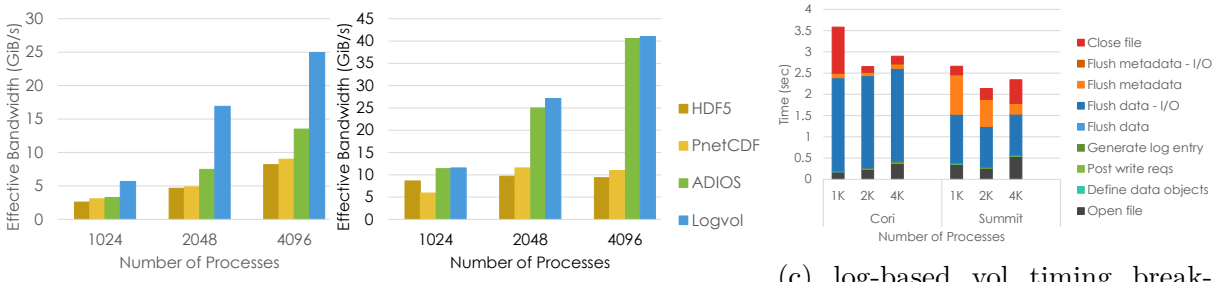
The result is shown in fig. 6.7 The relative performance between different I/O methods is similar to that in the E3SM case study. Log-based storage layouts still out-perform canonical storage layouts, but the gap is smaller than that in E3SM It is because S3D's I/O requests contain fewer and larger non-contiguous blocks. It also allows us to measure the performance of HDF5 in the canonical layout within the allocated time limit. Its performance is lower than PnetCDF due to the lack of an I/O aggregation feature.

6.6. FLASH I/O



(a) Cori (b) Summit (c) log-based vol timing breakdown

Figure 6.8. FLASH checkpoint file effective write bandwidth comparison. (c) shows the time log-based vol spent in each step handling the checkpoint file.



(a) Cori (b) Summit (c) log-based vol timing breakdown

Figure 6.9. FLASH plot files effective write bandwidth comparison. (c) shows the time log-based vol spent in each step handling the plot files.

FLASH is a modular and extensible set of physics solvers, maintained by Flash Center for Computational Science [**35**]. In the FLASH I/O pattern, the data to write are high-dimensional arrays called blocks. A block usually represents the value of a variable of a rectangular region (subarray) within the problem space. When storing the blocks in the file, blocks of the same size are stacked one after another to form a dataset that is one dimensional higher than the blocks. Each unit slice along the most significant dimension is a block. Blocks handled by a process are usually stored adjacent to each other and after the blocks of processes with smaller ranks.

This type of block-appending I/O pattern is commonly seen in applications that adopt Adaptive Mesh Refinement (AMR) in their solution space. They include FLASH [**33, 34, 20, 35**], OpenPMD [**129, 126**], QMCPack [**130**], AMReX [**94, 131**] ... etc.

We evaluated log-layout based VOL using the I/O kernel of a FLASH application running a block-structured adaptive mesh hydrodynamics simulation [**33, 34**]. The application produces 3 files: a checkpoint file containing 24 variables, and two plot files containing 4 variables each [**20**]. The data are 3-dimensional blocks of size 16 x 16 x 16. Each process writes 80 to 82 blocks in each variable. Because FLASH I/O writes larger blocks, we increase the Lustre stripe size on Cori to 64 MiB and 128 MiB for subfile setup and shared file setup respectively.

The result is shown in fig. 6.8 and fig. 6.9. Although log-based storage layouts still hold an advantage over canonical layouts, the gap is much narrower than that in S3D and E3SM. On Cori, PnetCDF can match or exceed the performance of ADIOS in a few cases. The reason is that FLASH's block appending I/O pattern resembles the I/O pattern of variable-centric log-based storage layout. In this case, PnetCDF also produces

an efficient I/O pattern as log-based storage layouts but does not bear any metadata overhead, leaving subfiling the only advantage of ADIOS.

6.7. Summary

In this chapter, we introduced log-layout based VOLwhich enables a log-based storage layout for HDF5 applications. We designed the data structure to store log entries in an HDF5 file. We proposed several techniques to improve the metadata efficiency of log entries. We compared log-layout based VOLwith other libraries and applications using a long-based storage layout and discussed different design choices.

We evaluated log-layout based VOLon supercomputer Cori [**30, 31**] and Summit [**99, 100, 4, 101**]. We compared the performance of log-layout based VOLwith other I/O methods on the I/O kernel of E3SM, FLASH, and S3D applications. The experiment result demonstrates the advantage of a log-based storage layout on complex I/O patterns. It also proves the effectiveness of our metadata reduction techniques and the efficiency of log-layout based VOL.

In the era of an increasing performance gap between computation and storage, log-layout based VOLprovides HDF5 users an option to store their data in a more efficient log-based storage layout and I/O pattern without the need to modify the code.

CHAPTER 7

COMCLUSION AND FUTURE WORKS

In this dissertation, we look for solutions to achieve efficient and scalable parallel I/O for exascale HPC environments. We discussed three commonly used ideas to improve parallel I/O performance on modern parallel file systems. We presented four projects that experiment with applying those ideas to various applications. We demonstrated the use of a burst buffer to perform I/O aggregation. We showed that data compression can effectively reduce the I/O time and file system workload for some applications. We proposed a solution to support compressed variables in classic NetCDF files that can be accessed in parallel efficiently. We design and developed a log-based storage layout for HDF5 datasets and verified its efficiency. We conclude that the three ideas mentioned above provide a roadway to scalable I/O in the exascale era. The results of our projects have been or are being incorporated into parallel I/O libraries. We hope they can help mitigate the I/O bottlenecks for applications running on exascale supercomputers.

In following subsections, we discuss some potential directions for future work.

7.1. Extending the Classic NetCDF File Format to Support Chunked Storage Layout and Variable Compression

The classic NetCDF file format, while being lightweight and efficient, lacks the means to support advanced data layouts such as chunked and compressed variables, compound datatypes ...etc. In chapter 3, we presented our solution to store chunked and compressed variables in classic NetCDF files. We were forced to eliminate record variables so that the fixed data section of the file can expand whenever we need more space for the chunks. The HDF5 format supports those features, however, the file structure is very complex and is very inefficient to navigate and maintain.

We can extend the classic NetCDF format to support variable chunking and compression with a small tweak. Inspired by ADIOS's BP5 file format, we divide a NetCDF file into multiple files so that the directory containing the files serves as the new NetCDF file. In the NetCDF directory, there is a classic NetCDF file (metadata file) that stores all metadata, variables that are not chunked, as well as chunk reference tables of chunked variables. Data chunks are stored separately in data files besides the metadata file. The number of data files and their relationship to processes can be configured by the user. Users can have a single data file shared by all processes, or have one file per computer node or process (subfiling). Other setups, such as a data file per variable, are also possible. Since data chunks are removed from the metadata file, record variables are allowed. There is no need to handle record variables especially as we did in section 5.2.4.

The chunk reference table can be extended to record the name of the data file in addition to the offset of the data. We store the chunk reference table in classic NetCDF variables. For fixed-sized variables, the chunk reference table is stored as a fixed-sized

NetCDF variable. For record variables, the chunk reference table is stored as a record NetCDF variable. This approach eliminates the need to expand the chunk reference table for record variables as they are automatically extended when new records as appended to the metadata file.

7.2. Support Asynchronous I/O Operations in Log-layout Based VOL

One idea not studied in this dissertation is to perform I/O operations in the background. Synchronous I/O operations block the application until the I/O operation completes. Asynchronous I/O operations, on the other hand, return immediately with a handle that can be used by the application to check the progress of the I/O operation. Most applications run for multiple iterations, alternating between computation and I/O operation. Using asynchronous I/O operations allows applications to hide the time spent on I/O operations by overlapping them with computation.

In the past, asynchronous I/O was not widely adopted by I/O libraries due to fear of contending the limited system resource with the application. Nowadays, computing nodes usually contain multiple CPU cores that are capable of running multiple hardware threads with enough memory to perform multiple tasks at the same time. The improvement in hardware capability opens an opportunity for asynchronous I/O. HDF5's newly published Virtual Object Layer (VOL) contains an interface for asynchronous I/O, paving the way to support asynchronous I/O for HDF5 applications. Tang et al. developed an HDF5 VOL for asynchronous I/O [**132**]. Zheng et al. developed an HDF5 VOL that allows existing applications to utilize the asynchronous I/O VOL[**132**] transparently [**133**].

We can extend log-layout based VOLto support asynchronous I/O. We create separate threads to manage I/O operations so the calling thread can proceed with computation. For each process, we create an I/O thread to handle I/O requests. The requests are stored in a queue shared by the I/o thread and the main thread. The I/O thread does not spawn until an asynchronous API is called so that applications using synchronous I/O do not bear additional overhead. When an asynchronous API is called, we append the request

to the queue and return a token to the application. When the application waits on the token, we wait until the job in the queue finishes before returning the result.

7.3. Distributed Index for Log-layout Based VOL Read Operations

In chapter 6, we discussed some challenges in reading datasets in log-based storage layouts. One of them is the number of log entries that have to be examined. To handle a write request, a process only needs to write its own log entry. A read request, on the other hand, requires iterating through all log entries from all processes. This limitation not only makes read operations unscalable but also makes it memory intensive due to the sheer number of log entries that have to be indexed. In many cases, we do not have enough memory to index all log entries. The current log-layout based VOLimplementation work around this issue by reading (and storing) log entries in one subfile at a time and then combining the data gathered from all subfiles. This approach can be inefficient when the number of subfiles is large.

An alternative solution is to employ distributed index. In a distributed index setup, each process holds an index for a subset of the log entries. There are many ways to divide the log entries. One way is to have one node holding the index for one dataset. To look up intersections for a read request, a process sends an index lookup request to all processes holding the relevant index. The remote processes calculate and reply with the intersections between the read request and their index. A protocol similar to that used in chapter 5 to exchange chunk data can be used. The only difference is that the remote process does not return the data directly. They return the metadata required to locate the data.

This solution avoids repeated metadata read by different processes. A process only needs to read the metadata it is assigned. However, the communication overhead for

index lookup may be high at a larger scale. Whether they outweigh the cost to switch between the subfiles requires further study to find out.

7.4. Generalize Log-based Storage Layout in PnetCDF

In section 6.4.1, we proposed a log-based storage layout solution for NetCDF files called PnetCDF-log. While achieving a decent performance, it has some limitations that can hinder its compatibility with some NetCDF applications. In this section, we discuss some of the limitations and potential solutions that can be future research topics.

One of the limitations is that a decomposed variable must be written at once. Recall that in PnetCDF-log, applications do not specify any selections when writing to a decomposed variable. It is assumed that every process writes to all of its subarrays in the decomposition map. Filling up a decomposed variable in multiple stages is not supported under the current design. This limitation makes asynchronousI/O difficult to implement. The application may want to write out part of the data at the same time it processes the remaining part. It is not possible without manually dividing the data into multiple variables, making the file hard to read and manage. There are two ways to support partial writing to decomposed variables. One is to allow the application to specify a set of subarrays in the decomposition map to write to. Another one is to support all existing PnetCDF APIS (such as put_vara) on decomposed variables. The former is a lot easier as the location of individual subarrays in the decomposition map can be calculated efficiently. The latter requires the library to remap each subarray selection into the rearranged file space of the decomposed variables. It requires additional data structure to maintain the mapping and can be computationally infeasible.

Another limitation is that the variable cannot be overwritten by different processes. Consider an I/O pattern in which a variable is being written entirely by a process and then overwritten by another process at a later time. While our current design allows defining

decomposition maps that write to more elements than the size of the decomposed variable, the decomposition map does not record the order the write operations are taken. When the selection of two processes in the decomposition map intersects each other, we will not be able to tell which process's data should be taken when reconstructing the canonical variables. log-layout based VOLdoes not have this issue because the order of the log entries implies the order of the write operations. A potential solution is to record a timestamp for each subarray in the decomposition map and update them each time the subarray is written.

References

[1] Frank B Schmuck and Roger L Haskin. Gpfs: A shared-disk file system for large computing clusters. In *FAST*, volume 2, 2002.

[2] Peter J Braam and Michael J Callahan. Lustre: A san file system for linux. *white paper*, 1999.

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

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

[5] Peter Corbett, Dror Feitelson, Yarsun Hsu, Jean-Pierre Prost, Marc Snir, Sam Fineberg, Bill Nitzberg, Bernard Traversat, and Parkson Wong. Mpi-io: a parallel file i/o interface for mpi version 0.3. 1995.

[6] Peter Corbett, Dror Feitelson, Sam Fineberg, Yarsun Hsu, Bill Nitzberg, Jean-Pierre Prost, Marc Snir, Bernard Traversat, and Parkson Wong. Overview of the mpi-io parallel i/o interface. In *IPPS'95 Workshop on Input/Output in Parallel and Distributed Systems*, pages 1–15, 1995.

[7] Russ Rew and Glenn Davis. Netcdf: an interface for scientific data access. *IEEE computer graphics and applications*, 10(4):76–82, 1990.

[8] Russell K Rew. The unidata netcdf: Software for scientific data access. In *Sixth International Conference on Interactive Information and Processing Systems for Meteorology, Oceanography, and Hydrology*, 1990.

[9] Jianwei Li, Wei-keng Liao, Alok Choudhary, Robert Ross, Rajeev Thakur, William Gropp, and Rob Latham. Parallel netcdf: A scientific high-performance i/o interface. *arXiv preprint cs/0306048*, 2003.

[10] Bin Dong, Xiuqiao Li, Limin Xiao, and Li Ruan. A new file-specific stripe size selection method for highly concurrent data access. In *Proceedings of the 2012 ACM/IEEE 13th International Conference on Grid Computing*, pages 22–30. IEEE Computer Society, 2012.

[11] Wahid Bhimji, Debbie Bard, Melissa Romanus, David Paul, Andrey Ovsyannikov, Brian Friesen, Matt Bryson, Joaquin Correa, Glenn K Lockwood, Vakho Tsulaia, et al. Accelerating science with the nersc burst buffer early user program. 2016.

[12] 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 *012 ieee 28th symposium on mass storage systems and technologies (msst)*, pages 1–11. IEEE, 2012.

[13] 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. In *2015 IEEE INTERNATIONAL CONFERENCE ON CLUSTER COMPUTING-CLUSTER 2015*. Oak Ridge National Laboratory, Oak Ridge Leadership Computing Facility (OLCF), 2015.

[14] Teng Wang, Kathryn Mohror, Adam Moody, Kento Sato, and Weikuan Yu. An ephemeral burst-buffer file system for scientific applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 69. IEEE Press, 2016.

[15] Teng Wang, Sarp Oral, Yandong Wang, Brad Settlemyer, Scott Atchley, and Weikuan Yu. Burstmem: A high-performance burst buffer system for scientific applications. In *2014 IEEE International Conference on Big Data (Big Data)*, pages 71–79. IEEE, 2014.

[16] Dries Kimpe, Rob Ross, Stefan Vandewalle, and Stefaan Poedts. Transparent log-based data storage in mpi-io applications. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pages 233–241. Springer, 2007.

[17] Mike Folk, Albert Cheng, and Kim Yates. Hdf5: A file format and i/o library for high performance computing applications. In *Proceedings of supercomputing*, volume 99, pages 5–33, 1999.

[18] Rajeev Thakur, William Gropp, and Ewing Lusk. Data sieving and collective i/o in romio. In *Proceedings. Frontiers' 99. Seventh Symposium on the Frontiers of Massively Parallel Computation*, pages 182–189. IEEE, 1999.

[19] Llnl. Llnl/ior.

[20] Flash i/o benchmark routine – parallel hdf 5.

[21] Parkson Wong, Rob F VanderWijngaart, and Bryan A Biegel. Nas parallel benchmarks i/o version 2.4. 2.4. 2002.

[22] Bryce Hicks. Improving i/o bandwidth with cray dvs client-side caching. *Concurrency and Computation: Practice and Experience*, 30(1):e4347, 2018.

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

[24] Anthony Kougkas, Hariharan Devarajan, and Xian-He Sun. Hermes: a heterogeneous-aware multi-tiered distributed i/o buffering system. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, pages 219–230. ACM, 2018.

[25] Ning Liu, Christopher Carothers, Jason Cope, Philip Carns, Robert Ross, Adam Crume, and Carlos Maltzahn. Modeling a leadership-scale storage system. In *International Conference on Parallel Processing and Applied Mathematics*, pages 10–19. Springer, 2011.

[26] Kento Sato, Kathryn Mohror, Adam Moody, Todd Gamblin, Bronis R De Supinski, Naoya Maruyama, and Satoshi Matsuoka. A user-level infiniband-based file system and checkpoint strategy for burst buffers. In *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 21–30. IEEE, 2014.

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

[28] Mendel Rosenblum and John K Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, 1992.

[29] Hui Dai, Michael Neufeld, and Richard Han. Elf: An efficient log-structured flash file system for micro sensor nodes. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 176–187, 2004.

[30] Katie Antypas, Nicholas Wright, Nicholas P Cardo, Allison Andrews, and Matthew Cordery. Cori: a cray xc pre-exascale system for nersc. *Cray User Group Proceedings. Cray*, 2014.

[31] Tina Declerck, Katie Antypas, Deborah Bard, Wahid Bhimji, Shane Canon, Shreyas Cholia, Helen Yun He, Douglas Jacobsen, and Nicholas J Wright Prabhat. Cori-a system to support data-intensive computing. *Cray User Group*, 2016.

[32] Hongzhang Shan and John Shalf. Using ior to analyze the i/o performance for hpc platforms. 2007.

[33] Bruce Fryxell, Kevin Olson, Paul Ricker, FX Timmes, Michael Zingale, DQ Lamb, Peter MacNeice, Robert Rosner, JW Truran, and H Tufo. Flash: An adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes. *The Astrophysical Journal Supplement Series*, 131(1):273, 2000.

[34] The flash code.

[35] Rob Latham, Chris Daley, Wei-keng Liao, Kui Gao, Rob Ross, Anshu Dubey, and Alok Choudhary. A case study for scientific i/o: improving the flash astrophysics code. *Computational Science & Discovery*, 5(1):015001, 2012.

[36] Nas parallel benchmarks.

[37] Ronald M Errico. What is an adjoint model? *Bulletin of the American Meteorological Society*, 78(11):2577–2592, 1997.

[38] Jean Utke, Uwe Naumann, Mike Fagan, Nathan Tallent, Michelle Strout, Patrick Heimbach, Chris Hill, and Carl Wunsch. Openad/f: A modular open-source tool for automatic differentiation of fortran codes. *ACM Transactions on Mathematical Software (TOMS)*, 34(4):1–36, 2008.

[39] Dougal Maclaurin, David Duvenaud, and Ryan P Adams. Autograd: Effortless gradients in numpy. In *ICML 2015 AutoML Workshop*, volume 238, 2015.

[40] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[41] Andreas Griewank and Andrea Walther. *Evaluating derivatives: principles and techniques of algorithmic differentiation*, volume 105. Siam, 2008.

[42] Andreas Griewank and Andrea Walther. Algorithm 799: revolve: an implementation of checkpointing for the reverse or adjoint mode of computational differentiation. *ACM Transactions on Mathematical Software (TOMS)*, 26(1):19–45, 2000.

[43] Guillaume Aupy, Julien Herrmann, Paul Hovland, and Yves Robert. Optimal multistage algorithm for adjoint computation. *SIAM Journal on Scientific Computing*, 38(3):C232–C255, 2016.

[44] Michel Schanen, Oana Marin, Hong Zhang, and Mihai Anitescu. Asynchronous two-level checkpointing scheme for large-scale adjoints in the spectral-element solver nek5000. *Procedia Computer Science*, 80:1147–1158, 2016.

[45] Kurt Brian Ferreira, Ronald B Brightwell, Dewan Ibtesham, Dorian Arnold, and Patrick Bridges. On the viability of compression for reducing the overheads of checkpoint/restart-based fault tolerance. Technical report, Sandia National Laboratories, 2012.

[46] Kurt Brian Ferreira, Ronald B Brightwell, Dewan Ibtesham, Dorian C Arnold, and Patrick G Bridges. Checkpoint compression for extreme scale fault tolerance. Technical report, Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2011.

[47] Dewan Ibtesham, Kurt B Ferreira, and Dorian Arnold. A checkpoint compression study for high-performance computing systems. *The International Journal of High Performance Computing Applications*, 29(4):387–402, 2015.

[48] Naoto Sasaki, Kento Sato, Toshio Endo, and Satoshi Matsuoka. Exploration of lossy compression for application-level checkpoint/restart. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 914–922. IEEE, 2015.

[49] Mitgcm.

[50] John Marshall, Alistair Adcroft, Chris Hill, Lev Perelman, and Curt Heisey. A finite-volume, incompressible navier stokes model for studies of the ocean on parallel computers. *Journal of Geophysical Research: Oceans*, 102(C3):5753–5766, 1997.

[51] Jean-loup Gailly and Mark Adler. Zlib compression library. 2004.

[52] Llnl/zfp.

[53] Peter Lindstrom. Fixed-rate compressed floating-point arrays. *IEEE transactions on visualization and computer graphics*, 20(12):2674–2683, 2014.

[54] Sheng Di and Franck Cappello. Fast error-bounded lossy hpc data compression with sz. In *2016 ieee international parallel and distributed processing symposium (ipdps)*, pages 730–739. IEEE, 2016.

[55] Dingwen Tao, Sheng Di, Zizhong Chen, and Franck Cappello. Significantly improving lossy compression for scientific data sets based on multidimensional prediction and error-controlled quantization. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1129–1139. IEEE, 2017.

[56] Daniel Goldberg, Sri Hari Krishna Narayanan, Laurent Hascoet, and Jean Utke. An optimized treatment for algorithmic differentiation of an important glaciological fixed-point problem. 2016.

[57] Marc Paterno, Jim Kowalkowski, and Saba Sehrish. Parallel event selection on hpc systems. In *EPJ Web of Conferences*, volume 214, page 04059. EDP Sciences, 2019.

[58] Rachit Agarwal, Anurag Khandelwal, and Ion Stoica. Succinct: Enabling queries on compressed data. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, pages 337–350, 2015.

[59] Feng Zhang, Jidong Zhai, Xipeng Shen, Dalin Wang, Zheng Chen, Onur Mutlu, Wenguang Chen, and Xiaoyong Du. Tadoc: Text analytics directly on compression. *The VLDB Journal*, 30(2):163–188, 2021.

[60] The hdf5® library & file format.

[61] G Peng, WN Meier, DJ Scott, and MH Savoie. A long-term and reproducible passive microwave sea ice concentration data record for climate studies and monitoring. 2013.

[62] Thomas W Estilow, Alisa H Young, and David A Robinson. A long-term northern hemisphere snow cover extent data record for climate studies and monitoring. *Earth System Science Data*, 7(1):137, 2015.

[63] Where is netcdf used?

[64] Kui Gao, Wei-keng Liao, Alok Choudhary, Robert Ross, and Robert Latham. Combining i/o operations for multiple array variables in parallel netcdf. In *2009 IEEE International Conference on Cluster Computing and Workshops*, pages 1–10. IEEE, 2009.

[65] E3SM Project. Energy Exascale Earth System Model (E3SM). [Computer Software] `https://dx.doi.org/10.11578/E3SM/dc.20180418.36`, April 2018.

[66] Micah Groh, Norman Buchanan, Derek Doyle, James B Kowalkowski, Marc Paterno, and Saba Sehrish. Pandana: A python analysis framework for scalable high performance computing in high energy physics. Technical report, Fermi National Accelerator Lab.(FNAL), Batavia, IL (United States), 2021.

[67] Fermi National Accelerator Laboratory. NOvA experiment. https://novaexperiment.fnal.gov/.

[68] Benjamin Welton, Dries Kimpe, Jason Cope, Christina M Patrick, Kamil Iskra, and Robert Ross. Improving i/o forwarding throughput with data compression. In *2011 IEEE International Conference on Cluster Computing*, pages 438–445. IEEE, 2011.

[69] Rosa Filgueira, David E Singh, Juan C Pichel, and Jesús Carretero. Exploiting data compression in collective i/o techniques. In *2008 IEEE International Conference on Cluster Computing*, pages 479–485. IEEE, 2008.

[70] Tanzima Zerin Islam, Kathryn Mohror, Saurabh Bagchi, Adam Moody, Bronis R De Supinski, and Rudolf Eigenmann. Mcrengine: A scalable checkpointing system using data-aware aggregation and compression. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, 2012.

[71] Huy Bui, Hal Finkel, Venkatram Vishwanath, Salma Habib, Katrin Heitmann, Jason Leigh, Michael Papka, and Kevin Harms. Scalable parallel i/o on a blue gene/q supercomputer using compression, topology-aware data aggregation, and subfiling. In *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 107–111. IEEE, 2014.

[72] Ursula Rasthofer, Fabian Wermelinger, P Hadijdoukas, and Petros Koumoutsakos. Large scale simulation of cloud cavitation collapse. *Procedia Computer Science*, 108:1763–1772, 2017.

[73] Panagiotis Hadjidoukas and Fabian Wermelinger. A parallel data compression framework for large scale 3d scientific data. *arXiv preprint arXiv:1903.07761*, 2019.

[74] Tekin Bicer, Jian Yin, and Gagan Agrawal. Improving i/o throughput of scientific applications using transparent parallel compression. In *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 1–10. IEEE, 2014.

[75] Ekow Otoo, Gideon Nimako, and Daniel Ohene-Kwofie. Using chunked extendible array for physical storage of scientific datasets. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 1315–1321. IEEE, 2012.

[76] Gideon Nimako, Ekow J Otoo, and Daniel Ohene-Kwofie. Chunked extendible dense arrays for scientific data storage. In *2012 41st International Conference on Parallel Processing Workshops*, pages 38–47. IEEE, 2012.

[77] Zarr¶.

[78] Numcodecs¶.

[79] saalfeldlab/n5.

[80] Qing Liu, Jeremy Logan, Yuan Tian, Hasan Abbasi, Norbert Podhorszki, Jong Youl Choi, Scott Klasky, Roselyne Tchoua, Jay Lofstead, Ron Oldfield, et al. Hello adios: the challenges and lessons of developing leadership class i/o frameworks. *Concurrency and Computation: Practice and Experience*, 26(7):1453–1473, 2014.

[81] R Rew, E Hartnett, J Caron, et al. Netcdf-4: Software implementing an enhanced data model for the geosciences. In *22nd International Conference on Interactive Information Processing Systems for Meteorology, Oceanograph, and Hydrology*, 2006.

[82] R. Rew, E. Hartnett, D. Heimbigner, E. Davis, and J. Caron. Netcdf classic and 64-bit offset file formats. Nasa earth science data system (esds) community standard, Open Geospatial Consortium (OGC), Geneva, CH, August 2008.

[83] Pnetcdf user guide.

[84] Netcdf user's guide.

[85] Peter Deutsch and Jean-Loup Gailly. Zlib compressed data format specification version 3.3. Technical report, RFC 1950, May, 1996.

[86] Yann Collet and EM Kucherawy. Zstandard-real-time data compression algorithm, 2015.

[87] Appendix b. file format specifications.

[88] Jialin Liu, Quincey Koziol, Houjun Tang, Francois Tessier, Wahid Bhimji, Brandon Cook, Brian Austin, Suren Byna, Bhupender Thakur, Glenn Lockwood, et al. Understanding the i/o performance gap between cori knl and haswell. Technical report, Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), 2017.

[89] Mark Howison. Tuning hdf5 for lustre file systems. 2010.

[90] Mike Folk, Gerd Heber, Quincey Koziol, Elena Pourmal, and Dana Robinson. An overview of the hdf5 technology suite and its applications. In *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, pages 36–47. ACM, 2011.

[91] Jyrki Alakuijala, Evgenii Kliuchnikov, Zoltan Szabadka, and Lode Vandevenne. Comparison of brotli, deflate, zopfli, lzma, lzham and bzip2 compression algorithms. *Google Inc*, 2015.

[92] Weiqun Zhang, Ann Almgren, Vince Beckner, John Bell, Johannes Blaschke, Cy Chan, Marcus Day, Brian Friesen, Kevin Gott, Daniel Graves, Max Katz, Andrew Myers, Tan Nguyen, Andrew Nonaka, Michele Rosso, Samuel Williams, and Michael Zingale. AMReX: a framework for block-structured adaptive mesh refinement. *Journal of Open Source Software*, 4(37):1370, May 2019.

[93] Anshu Dubey, Katie Antypas, Alan Calder, Bruce Fryxell, Don Lamb, Paul Ricker, Lynn Reid, Katherine Riley, Robert Rosner, Andrew Siegel, et al. The software development process of flash, a multiphysics simulation code. In *Proceedings of the 5th International Workshop on Software Engineering for Computational Science and Engineering*, pages 1–8. IEEE Press, 2013.

[94] Weiqun Zhang, Ann Almgren, Vince Beckner, John Bell, Johannes Blaschke, Cy Chan, Marcus Day, Brian Friesen, Kevin Gott, Daniel Graves, et al. Amrex: a framework for block-structured adaptive mesh refinement. *Journal of Open Source Software*, 4(37), 2019.

[95] Get yt: all-in-one script.

[96] Judy Sturtevant, Mark Christon, Philip D Heermann, and Pang-Chieh Chen. Pds/pio: Lightweight libraries for collective parallel i/o. In *SC'98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, pages 3–3. IEEE, 1998.

[97] Peter M Caldwell, Azamat Mametjanov, Qi Tang, Luke P Van Roekel, Jean-Christophe Golaz, Wuyin Lin, David C Bader, Noel D Keen, Yan Feng, Robert Jacob, et al. The doe e3sm coupled model version 1: Description and results at high resolution. *Journal of Advances in Modeling Earth Systems*.

[98] Qiao Kang, Robert Ross, Robert Latham, Sunwoo Lee, Ankit Agrawal, Alok Choudhary, and Wei-keng Liao. Improving all-to-many personalized communication in two-phase i/o. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–13, 2020.

[99] James A Kahle, Jaime Moreno, and Dan Dreps. 2.1 summit and sierra: designing ai/hpc supercomputers. In *2019 IEEE International Solid-State Circuits Conference-(ISSCC)*, pages 42–43. IEEE, 2019.

[100] Jack Wells, Buddy Bland, Jeff Nichols, Jim Hack, Fernanda Foertter, Gaute Hagen, Thomas Maier, Moetasim Ashfaq, Bronson Messer, and Suzanne Parete-Koon. Announcing supercomputer summit. Technical report, Oak Ridge National Lab.(ORNL), Oak Ridge, TN (United States), 2016.

[101] Verónica G Vergara Larrea, Wayne Joubert, Michael J Brim, Reuben D Budiardja, Don Maxwell, Matt Ezell, Christopher Zimmer, Swen Boehm, Wael Elwasif, Sarp Oral, et al. Scaling the summit: deploying the world's fastest supercomputer. In *International Conference on High Performance Computing*, pages 330–351. Springer, 2019.

[102] Jay F Lofstead, Scott Klasky, Karsten Schwan, Norbert Podhorszki, and Chen Jin. Flexible io and integration for scientific codes through the adaptable io system (adios). In *Proceedings of the 6th international workshop on Challenges of large applications in distributed environments*, pages 15–24, 2008.

[103] Kaiyuan Hou, Reda Al-Bahrani, Esteban Rangel, Ankit Agrawal, Robert Latham, Robert Ross, Alok Choudhary, and Wei-keng Liao. Integration of burst buffer in high-level parallel i/o library for exa-scale computing era. In *2018 IEEE/ACM 3rd International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems (PDSW-DISCS)*, pages 1–12. IEEE, 2018.

[104] Soumyadeb Mitra, Rishi Rakesh Sinha, Marianne Winslett, and Xiangmin Jiao. An efficient, nonintrusive, log-based i/o mechanism for scientific simulations on clusters. In *2005 IEEE International Conference on Cluster Computing*, pages 1–10. IEEE, 2005.

[105] Ryusuke Konishi, Yoshiji Amagai, Koji Sato, Hisashi Hifumi, Seiji Kihara, and Satoshi Moriai. The linux implementation of a log-structured file system. *ACM SIGOPS Operating Systems Review*, 40(3):102–107, 2006.

[106] Margo I Seltzer, Keith Bostic, Marshall K McKusick, Carl Staelin, et al. An implementation of a log-structured file system for unix. In *USENIX Winter*, pages 307–326, 1993.

[107] H Dai, M Neufeld, and R Han. Elf: An efficient log-structured flash file system. In *Proceedings of the 2nd Conference on Embedded Networked Sensor Systems (SenSys)*, pages 176–187.

[108] high level introduction to hdf5.

[109] Jingqing Mu, Jerome Soumagne, Suren Byna, Quincey Koziol, Houjun Tang, and Richard Warren. Interfacing hdf5 with a scalable object-centric storage system on hierarchical storage. *Concurrency and Computation: Practice and Experience*, 32(20):e5715, 2020.

[110] Suren Byna, M Scot Breitenfeld, Bin Dong, Quincey Koziol, Elena Pourmal, Dana Robinson, Jerome Soumagne, Houjun Tang, Venkatram Vishwanath, and Richard Warren. Exahdf5: delivering efficient parallel i/o on exascale computing systems. *Journal of Computer Science and Technology*, 35(1):145–160, 2020.

[111] Houjun Tang, Quincey Koziol, Suren Byna, John Mainzer, and Tonglin Li. Enabling transparent asynchronous i/o using background threads. In *2019 IEEE/ACM Fourth International Parallel Data Systems Workshop (PDSW)*, pages 11–19. IEEE, 2019.

[112] vol-cache.

[113] Jerome Soumagne, Jordan Henderson, Mohamad Chaarawi, Neil Fortner, Scot Breitenfeld, Songyu Lu, Dana Robinson, Elena Pourmal, and Johann Lombardi. Accelerating hdf5 i/o for exascale using daos. *IEEE Transactions on Parallel and Distributed Systems*, 33(4):903–914, 2021.

[114] adios 1.6.0 developer's manual.

[115] William F Godoy, Norbert Podhorszki, Ruonan Wang, Chuck Atkins, Greg Eisenhauer, Junmin Gu, Philip Davis, Jong Choi, Kai Germaschewski, Kevin Huck, et al. Adios 2: The adaptable input output system. a framework for high-performance data management. *SoftwareX*, 12:100561, 2020.

[116] Milo Polte, Jay Lofstead, John Bent, Garth Gibson, Scott A Klasky, Qing Liu, Manish Parashar, Norbert Podhorszki, Karsten Schwan, Meghan Wingate, et al. ... and eat it too: High read performance in write-optimized hpc i/o middleware file formats. In *Proceedings of the 4th Annual Workshop on Petascale Data Storage*, pages 21–25. ACM, 2009.

[117] Thomas Liebsch. Concurrent installation and acceptance of summit and sierra supercomputers. *IBM Journal of Research and Development*, 64(3/4):6–1, 2020.

[118] Sarp Oral, Sudharshan S Vazhkudai, Feiyi Wang, Christopher Zimmer, Christopher Brumgard, Jesse Hanley, George Markomanolis, Ross Miller, Dustin Leverman, Scott Atchley, et al. End-to-end i/o portfolio for the summit supercomputing ecosystem. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2019.

[119] Yun He, Brandon Cook, Jack Deslippe, Brian Friesen, Richard Gerber, Rebecca Hartman-Baker, Alice Koniges, Thorsten Kurth, Stephen Leak, Woo-Sun Yang, et al. Preparing nersc users for cori, a cray xc40 system with intel many integrated cores. *Concurrency and Computation: Practice and Experience*, 30(1):e4291, 2018.

[120] Edward Hartnett and Jim Edwards. The parallelio (pio) c/fortran libraries for scalable hpc performance. In *37th Conference on Environmental Information Processing Technologies, American Meteorological Society Annual Meeting*, pages 10–15, 2021.

[121] Lipeng Wan, Matthew Wolf, Feiyi Wang, Jong Youl Choi, George Ostrouchov, and Scott Klasky. Comprehensive measurement and analysis of the user-perceived i/o performance in a production leadership-class storage system. In *2017 IEEE 37th*

International Conference on Distributed Computing Systems (ICDCS), pages 1022–1031. IEEE, 2017.

[122] Aggregation — adios2 2.8.3 documentation.

[123] Jacqueline H Chen, Alok Choudhary, Bronis De Supinski, Matthew DeVries, Evatt R Hawkes, Scott Klasky, Wei-Keng Liao, Kwan-Liu Ma, John Mellor-Crummey, Norbert Podhorszki, et al. Terascale direct numerical simulations of turbulent combustion using s3d. *Computational Science & Discovery*, 2(1):015001, 2009.

[124] David Lignell, C Yoo, Jacqueline Chen, Ramanan Sankaran, and M Fahey. S3d: Petascale combustion science, performance, and optimization. In *Proceedings of the Cray Scaling Workshop, Oak Ridge National Laboratory, TN*, 2007.

[125] Chet Nieter and John R Cary. Vorpal: a versatile plasma simulation code. *Journal of Computational Physics*, 196(2):448–473, 2004.

[126] A Huebl, F Poeschel, F Koller, and J Gu. openpmd-api: C++ & python api for scientific i/o with openpmd. 2021. *URL: https://github. com/openPMD/openPMD-api, doi*, 10, 2018.

[127] David Randall, Marat Khairoutdinov, Akio Arakawa, and Wojciech Grabowski. Breaking the cloud parameterization deadlock. *Bulletin of the American Meteorological Society*, 84(11):1547–1564, 2003.

[128] Masaki Satoh, Bjorn Stevens, Falko Judt, Marat Khairoutdinov, Shian-Jiann Lin, William M Putman, and Peter Düben. Global cloud-resolving models. *Current Climate Change Reports*, 5(3):172–184, 2019.

[129] Axel Huebl, Remi Lehe, Jean-Luc Vay, David P Grote, I Sbalzarini, Stephan Kuschel, and M Bussmann. openpmd: A meta data standard for particle and mesh based data. *URL https://doi. org/10.5281/zenodo*, 591699, 2015.

[130] Paul RC Kent, Abdulgani Annaberdiyev, Anouar Benali, M Chandler Bennett, Edgar Josué Landinez Borda, Peter Doak, Hongxia Hao, Kenneth D Jordan, Jaron T Krogel, Ilkka Kylänpää, et al. Qmcpack: Advances in the development, efficiency, and application of auxiliary field and real-space variational and diffusion quantum monte carlo. *The Journal of chemical physics*, 152(17):174105, 2020.

[131] Weiqun Zhang, Andrew Myers, Kevin Gott, Ann Almgren, and John Bell. Amrex: Block-structured adaptive mesh refinement for multiphysics applications. *The International Journal of High Performance Computing Applications*, 35(6):508–526, 2021.

[132] Houjun Tang, Quincey Koziol, John Ravi, and Suren Byna. Transparent asynchronous parallel i/o using background threads. *IEEE Transactions on Parallel and Distributed Systems*, 33(4):891–902, 2021.

[133] Huihuo Zheng, Venkatram Vishwanath, Quincey Koziol, Houjun Tang, John Ravi, John Mainzer, and Suren Byna. Hdf5 cache vol: Efficient and scalable parallel i/o through caching data on node-local storage. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 61–70. IEEE, 2022.