Optimizing Performance of Parallel I/O Accesses to Non-contiguous Blocks in Multiple Array Variables

Qiao Kang[1], Scot Breitenfeld[2], Kaiyuan Hou[3], Wei-keng Liao[3], Robert Ross[4], and Suren Byna[1]

[1]Lawrence Berkeley National Laboratory, [2]The HDF Group,
[3]Northwestern University, [4]Argonne National Laboratory

{qkang, sbyna}@lbl.gov,
brtnfld@hdfgroup.org,
{khl7265, wkliao}@ece.northwestern.edu,
rross@mcs.anl.gov

2021 IEEE International Conference on Big Data (Big Data) | 978-1-6654-3902-2/21/$31.00 ©2021 IEEE | DOI: 10.1109/BigData52589.2021.9671638

***Abstract*—Accessing non-contiguous blocks in multiple array variables is a challenging I/O pattern for parallel applications to obtain good I/O performance. High-level I/O libraries such as HDF5 allow users to implement this pattern conveniently, but users have observed significant performance bottlenecks in the two-phase I/O implementation of MPI-IO. Recent studies have advanced the two-phase I/O performance by novel communication algorithms, but such improvements still have limitations. Two-phase I/O has to faithfully process inputs from high-level I/O libraries, so that implementation overheads can accumulate for improper usage of high-level I/O libraries. In this paper, we propose approaches for efficient usage of high-level I/O libraries that can circumvent major collective I/O overheads. We adopt a multi-dataset implementation of HDF5 dataset I/O to aggregate non-contiguous requests for array blocks and provide corresponding parameter assignment strategies. These approaches reduce the overheads caused by communication straggler effects in two-phase I/O. We show that our proposed methods can improve the parallel I/O performance up to 8× on two supercomputing systems for the HDF5 implementations of an I/O kernel extracted from climate simulation code compared with its baseline implementations.**

I. Introduction

Achieving superior parallel I/O performance on high-performance computing (HPC) systems is challenging. This challenge gets even more difficult when large-scale scientific applications have complex I/O patterns. For instance, modern workflow applications such as the Energy Exascale Earth System Model (E3SM) [1] have high volumes of parallel I/O to access (reads and writes) non-contiguous blocks from all processes to and from many variables. These patterns have a mismatch in the data representations in memory and file spaces. Non-contiguous block access pattern poses a scalability challenge because processes frequently compete for communication and I/O resources. Another reason why this I/O access pattern often obtains poor performance is due to the inter-dependencies between I/O software layers.

High-level I/O libraries handle parallel I/O by calling collective I/O functions implemented by I/O middleware libraries. For example, Hierarchical Data Format version 5 (HDF5) [2] implements parallel dataset read and write functions using collective Message Passing Interface I/O (MPI-IO) functions. Major MPI implementations, such as MPICH and OpenMPI, implement MPI-IO functions with the ROMIO drivers that adopt the two-phase I/O approach [3]. The two-phase I/O design selects a subset of processes called *I/O aggregators*. The rest of the MPI processes send data in their I/O requests to the aggregators in the communication phase. In the I/O phase, the aggregators perform I/O operations with the underlying file systems using the aggregated I/O requests. The coordination of I/O requests from all processes to the aggregators can reduce contention at the file system level and other shared communication resources.

Although HDF5 provides convenient interfaces for implementing complicated I/O patterns, the parallel I/O performance exhibits scalability challenges for these patterns. Recent studies have found bottlenecks in the communication phase of two-phase I/O [4]. High-dimensional block access patterns from high-level I/O libraries are translated into a large number of small and non-contiguous file access requests [5] for middleware libraries. As a result, the communication overheads can be significant. Recent research efforts focus on improving the communication phase performance in two-phase I/O. For example, two-layered aggregation method (TAM) [6] can reduce two-phase I/O communication cost by reducing the number of concurrent communications. These approaches focus mainly on improving the performance of MPI-IO collective functions with novel communication strategies by assuming fixed input patterns. However, collective MPI-IO functions have to faithfully perform parallel I/O with the given inputs from the high-level I/O library called by

users. For the same I/O pattern, different implementations of parallel I/O with a high-level library, along with MPI-IO hint parameters, can significantly change the MPI-IO collective performance. Applications have high-level abstract views of metadata and data, so it is possible to circumvent some two-phase I/O overhead by properly rearranging and aggregating collective function calls.

In this paper, we generalize the solutions for processing multi-dimensional blocks of I/O accesses on an arbitrary number of datasets. We identified that the bottleneck of ROMIO was the communication straggler effect caused by multiple iterations of two-phase I/O implementations. Without modifying the MPI implementations, we have studied strategies to reduce this bottleneck at the HDF5 level. The first part of the solutions is a design for data space and memory space aggregation at the HDF5 dataset level. This design allows applications to trigger a single collective I/O function call for any multi-dimensional block access patterns on a single dataset. The next step is to aggregate I/O requests from all datasets and perform a single MPI-IO collective call. This approach is called *multi-dataset parallel I/O* [7], which hides the complexity in non-contiguous parallel I/O accesses to array blocks. The multi-dataset I/O operations of HDF5 allow reading (or writing) multiple HDF5 datasets (i.e., data objects) from (or to) a file without a collective call between each dataset read (or write) operation. This multi-dataset I/O strategy at the HDF5 level improves the performance of ROMIO implementations. With multi-dataset implementation, we propose parameter tuning strategies for Lustre settings and MPI-IO hints that can further improve the I/O performance.

We conducted our performance evaluations on Cori, a Cray XC40 supercomputer with Intel KNL processors in the KNL partition and the Lustre file system, and on Summit, an IBM system equipped with POWER9 CPUs and GPFS. We used the state-of-art two-layer aggregation method for the Lustre and GPFS drivers in ROMIO that gives better performance than the native MPI-IO libraries deployed on Cori and Summit. The first part of the experiments is performed using the I/O kernels of E3SM-IO F and G cases [8] implemented with the HDF5 library. Since the multi-dataset feature is still unavailable in the official releases of HDF5, we implement the same I/O pattern using APIs provided by the HDF5 multi-dataset branch. Consequently, we can make direct comparisons between independent dataset I/O and multi-dataset I/O. The multi-dataset implementation can improve the I/O performance of the F case up to 8 times on both Cori and Summit for the E3SM-IO F case. In addition to E3SM-IO, we also present results from a synthetic benchmark, called *non-contiguous HDF5 I/O*, which is a part of the *h5bench* [9] benchmark suite. This benchmark allows us to generate I/O patterns that are more complex than E3SM-IO at different scales, so we can further study the limitations and advantages of our proposed approaches.

II. Background

This section discusses motivations and related R&D efforts in parallel I/O.

We describe I/O in the E3SM climate simulation code in Section II-A, which is a use case that motivates the use of the complex I/O patterns of writing and reading non-contiguous blocks of data from multiple variables. In Section II-B, we briefly describe HDF5 dataset I/O, which can be used to store multiple high-dimensional variables in parallel for applications like E3SM. Section II-C presents two-phase I/O concept and some implementation details for different file systems. Finally, in Section II-D we discuss related efforts targeting parallel I/O performance improvement.

A. E3SM I/O

The Energy Exascale Earth System Model (E3SM) is a large-scale workflow that simulates, models, and predicts oceans, ice land, and atmosphere components on Earth [1]. The simulations for high-resolution attributes such as surface temperatures consist of a large number of computation steps, so E3SM has a high demand in computation power. Therefore, applications in E3SM usually run in parallel on supercomputers such as NERSC Cori and OLCF Summit.

To recover from any failures during the computational phase, E3SM applications perform checkpointing periodically for storing intermediate data. E3SM applications can recover their computational state from the checkpointed files. The checkpointing operations from multiple parallel processes generate high volumes of non-contiguous I/O requests that can cause I/O performance bottlenecks [10]. The performance of saving and retrieving intermediate computation data to and from parallel file systems depends on the choice and utilization of I/O libraries. For example, the current HDF5 drivers directly store data from all processes into a single HDF5 file with collective I/O to maintain the canonical ordering of the data from the processes. The Adaptable Input Output (ADIOS) [11] driver of E3SM, on the other hand, stores data from different nodes in different files without the canonical ordering. As a result, a post-processing program joins these files into a single file for analysis applications to use. Such offline file coalescing can slow down the performance of I/O intensive workflows that frequently perform read-modify-write. This paper targets optimizing the single shared file accesses with collective I/O.

B. HDF5 Dataset I/O

HDF5 is a portable high-level I/O library for managing and storing heterogeneous data [2]. Data representations in HDF5 files are self-describing objects, namely *datasets* for multi-dimensional arrays of data with the same data type and groups that organize objects. The concept of data space in HDF5 allows users to access any non-contiguous portions of a dataset elegantly. The design advantages of

HDF5 allow users to manage large-scale scientific data such as experimental data and observation data. E3SM checkpointing modules store variables with homogeneous types in one to three-dimensional arrays, naturally mapping to the HDF5 data format when saving computational progress. This paper investigates the potential of using HDF5 data format for E3SM checkpointing data.

The official HDF5 releases support parallel I/O on shared file access. Multiple processes can access one HDF5 dataset with either `H5DWrite` or `H5Dread` functions. By default, the current HDF5 release (1.12.0) translates data space and memory space views for parallel I/O into file offset/length pairs and passes these metadata along with data buffer down to the lower-level MPI-IO functions. We discuss implementations of MPI-IO in the next section.

C. Two-phase I/O for MPI-IO

Two-phase I/O is a design for parallel I/O that consists of two parts: communication and I/O phases. A subset of processes called I/O aggregators are assigned with file access regions, denoted as file domains. In the communication phase, the I/O aggregators collect data from I/O requests that fall into their file domain from the rest of the processes. In the I/O phase, the I/O aggregators perform read/write operations with the underlying file systems using the data gathered in the communication phase.

ROMIO is an implementation of the MPI-IO standard that adopts a two-phase I/O strategy. Popular production libraries such as MPICH [12] and OpenMPI [13] adopt ROMIO as backbones for implementing MPI-IO. To optimize the I/O performance on different file systems, ROMIO adopts a driver selection mechanism, abstracted by the ADIO layer. During the library configuration time, users can specify I/O drivers optimized for local file systems such as the Lustre file system, GPFS, etc.

Implementation of collective buffering in file read and write functions perform two-phase I/O in multiple iterations. For each iteration, a limited file domain is processed. This implementation choice is motivated by memory footprint concerns. I/O aggregators gather metadata and data from the rest of the processes, so accessing large files can result in a large memory buffer allocated if one round (i.e., iteration) of two-phase I/O processes the entire file domain. Limiting the file domain size per this iteration can constrain temporary buffer allocation size.

D. Related Work

Published literature on improving two-phase I/O performance is mainly in two domains: One track focused mainly on improving the I/O phase performance because file system I/O was much slower compared with data communication in the past. For instance, Ma et al. applied a combination of active buffering and threads approaches for improving the throughput of I/O performance [14]. Liao et al. proposed file domain alignment protocols for Lustre and GPFS at I/O aggregators in ROMIO [15], [16]. Zhang et al. proposed and implemented a resonance I/O in ROMIO that rearranges I/O requests for efficient utilizations of underlying file systems [17]. LACIO is another strategy that improves I/O performance by taking the logical I/O access pattern among processes and physical layouts of file access into account [18].

As the file I/O in HPC systems becomes faster, the communication phase cost of the two-phase I/O is no longer negligible compared with the I/O phase. Communication and I/O phase overlapping has been a popular research area over the past decade. One direct overlapping example is the two phases I/O pipelining with the help of asynchronous MPI communication functions [19], [20]. Multithreading is another approach that hides the communication cost in the background during I/O processing [21]. TAPIOCA is a topology-aware two-phase I/O algorithm using double-buffering and one-sided communication to reduce data aggregation overhead [22], [23].

Recently, the communication cost generated by high volumes of non-contiguous I/O requests becomes larger than the I/O cost for certain applications. Algorithms designed for reducing the communication cost are necessary. The state-of-the-art design for two-phase I/O that has shown significant performance improvement is a two-layered aggregation method (TAM) [6]. This approach assigns a subset of processes called local aggregators that aggregate I/O requests from processes within the same compute node. Later, the local aggregators carry out two-phase I/O with I/O aggregators using these aggregated I/O requests. The main advantage of this approach is the reduction of communication contention by replacing the all-to-many communication patterns with many-to-one and many-to-many patterns in the communication phase of two-phase I/O. A subsequent study [4] for bottlenecks of all-to-many communication patterns to further support the advantages of TAM.

III. I/O Optimization Strategies

We describe various strategies for efficient processing of I/O requests to multi-dimensional blocks of data from multiple variables in this section. We assume that an MPI application runs in parallel with p processes. The application performs I/O operations on d datasets (i.e., array variables) in a single HDF5 file shared by all MPI processes. Each of the datasets represents the storage of data in an n dimensional space. Process i accesses a list of sub-arrays (n-dimensional blocks) from each dataset. We denote the number of sub-arrays for a dataset j by process i as $c_{i,j}$. For example, in the E3SM F case, there are 402 datasets (i.e., $d = 402$) and the number of processes (p) is 21632. E3SM applications access two-dimensional and three-dimensional datasets with sub-array blocks.

As mentioned earlier, parallel HDF5 translates its data structures to an MPI datatype before calling MPI-IO collective I/O functions. The end-to-end I/O performance can vary significantly depending on the HDF5-layer tuning

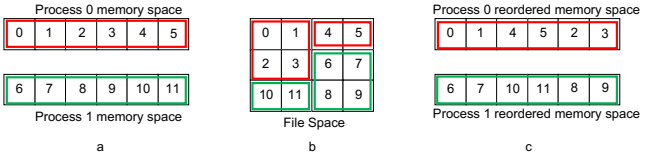

Figure 1: This figure illustrates an example of how a contiguous memory buffer is rearranged between two MPI processes according to the locations of sub-arrays placed in a two-dimensional data space of a dataset in parallel I/O. Array elements on the same row from different sub-arrays are concatenated into a new contiguous memory buffer. Row buffers are concatenated into a single contiguous memory buffer in ascending order.

parameters that are passed to the MPI-IO layer. Also, as mentioned in Section II-C, ROMIO implements two-phase I/O in multiple iterations for processing collective MPI I/O function calls. One factor that is negatively proportional to the end-to-end I/O performance is the total number of two-phase I/O iterations summed up across all collective MPI I/O functions. A large number of two-phase I/O iterations leads to more overheads spent on the communication phase. In addition, processing a small file domain per two-phase I/O iteration is likely to result in imbalanced workloads among processes so that some processes may stay idle during the execution. We refer to these scenarios to the communication straggler effect of multiple two-phase I/O iterations.

The end-to-end execution time for processing the multi-dimensional blocks of I/O accesses on HDF5 datasets consists of the following costs: Two-phase I/O file access time ($t_{\mathrm{I/O}}$), two-phase I/O communication time (t_{comm}), two-phase I/O overheads (t_{o}), and HDF5 overhead (t_{h5}). Equation 1 summarizes these factors.

$$\text{total I/O time} = t_{\mathrm{I/O}} + t_{\mathrm{comm}} + t_{\mathrm{o}} + t_{\mathrm{h5}} \qquad (1)$$

In the rest of this section, we present incremental tuning options for reducing the two-phase I/O overhead.

A. Sub-array Aggregation

To access multi-dimensional blocks in multiple variables (i.e., HDF5 datasets), the independent I/O implementation can let each process independently access the sub-arrays for each dataset. This strategy results in $\sum_{j=1}^{d} c_{i,j}$ independent HDF5 dataset I/O calls per process. Independent I/O calls from each MPI process can result in resource contention on file systems at large scales due to a large number of requests and the lack of coordination among processes. Thus, the I/O cost ($t_{\mathrm{I/O}}$) will be large. On parallel file systems shared by several applications on a HPC system, this solution is inefficient due to contention for I/O resources.

Collective I/O allows process coordination for accessing shared resources. With collective I/O, the number of collective HDF5 dataset I/O calls is $\max_{i\in[0,p-1]}\left(\sum_{j=1}^{d} c_{i,j}\right)$. Processes that access (read or write) fewer sub-arrays than $\max_{i\in[0,p-1]}\left(\sum_{j=1}^{d} c_{i,j}\right)$ contribute 0 write size for some collective calls. A disadvantage of this method is the potentially high number of collective function calls, which is determined by the process that accesses the largest number of sub-arrays. For applications with large number of non-contiguous blocks per process, collective I/O overhead resulting from the function calls is not negligible. If the number of sub-arrays is not balanced across all processes, overheads of collective function calls can significantly slow down the overall performance because the majority of processes can stay idle.

Coalescing sub-arrays at the dataset level reduces the number of collective calls. HDF5 provides `H5Sselect_hyperslab` functions that can coalesce multiple sub-arrays into one data space using the `H5S_SELECT_OR` operator. This implementation needs to reorder the corresponding memory buffers, which is a part of the HDF5 overhead (t_{h5}). The number of collective I/O calls becomes d after coalescing data space and memory space.

In Figure 1, we illustrate an example for memory reordering. We show the contiguous input memory buffers of two MPI processes in ascending order of array elements (Figure 1-(a)). We differentiate the memory buffers for each of the processes by different border colors. In Figure 1-(b), we show the non-contiguous 2D blocks for each of the processes in the file space of a dataset. For example, process 0 reads or writes array elements circled by the two red blocks in the file space. From Figure 1 (b), the first block of process 0 consists of 4 array elements, and the second block of process 0 consists of 2 array elements. In Figure 1-(c), buffers are rearranged along with the dataset dimensions after block coalescing. In the case of the two-dimensional dataset, the array elements on the same row from different blocks are rearranged to become contiguous. For example, process 0 concatenates array elements 0 and 1 from the 2×2 block with the elements 4 and 5 from the 1×2 in the first row. The elements 2 and 3 from the second row of the file space are then appended to the memory space of process 0. Thus, the array element order for process 0 after rearrangement is {0, 1, 4, 5, 2, 3}.

B. HDF5 Multi-dataset

In the previous sub-section, we discussed how sub-array coalescing reduces the number of collective I/O calls from $\max_{i\in[0,p-1]}\left(\sum_{j=1}^{d} c_{i,j}\right)$ to d. As mentioned earlier, the E3SM F case performs I/O using 402 variables. Each of these variables is represented as one dataset in HDF5. Thus, writing all variables into a file triggers enormous collective I/O calls. The overheads of d number of collective I/O operations are still not trivial. We demonstrate this fact in the experimental evaluation section.

The `H5DWrite` and `H5Dread` functions in the released versions of HDF5 (including 1.12.x) can store one dataset at a time, which we call *independent dataset I/O (IDIO)*. To further reduce the number of collective calls, we use the multi-dataset branch of HDF5 implementation [24], where new APIs and features are developed. The new APIs include `H5Dwrite_multi` and `H5Dread_multi` for multi-dataset write and read operations, respectively. `H5Dwrite_multi` and `H5Dread_multi` take arrays of parameters for `H5DWrite` and `H5Dread` functions, respectively. With multi-dataset API (MDIO), the parallel I/O implementation wraps the pairs of file offset and length for all datasets into a *MPI_Datatype*. Later, the multi-dataset library calls a single collective MPI I/O function.

The multi-dataset approach only takes a single MPI collective I/O function call for a parallel application program to perform sub-array I/O on multiple datasets in a shared HDF5 file. In the ROMIO's implementation, each MPI collective I/O function call distributes metadata from all processes to I/O aggregators with an all-to-many personalized communication pattern. In addition, all processes synchronize their error codes at the end of an MPI collective I/O function. Therefore, reducing the number of MPI collective I/O calls can reduce the collective I/O function overheads t_o.

HDF5's MDIO implementation sometimes reduces the number of two-phase I/O iterations compared to those with IDIO, especially for small files with large numbers of variables. The number of two-phase I/O iterations is lower bounded by the number of collective I/O calls. With IDIO, the number of two-phase I/O iterations is at least the number of variables. The number of two-phase I/O iterations for MDIO is independent of the number of variables. Instead, it is proportional to the file size. As a result, MDIO has only a few two-phase I/O iterations for all small files, regardless of the number of variables.

In most cases, implementations with fewer two-phase I/O iterations are expected to have better performance because overheads accumulated by the communication straggler effects in each of the two-phase I/O iterations are less. We will elaborate on these overheads in the next subsection.

C. Tuning MPI-IO

As mentioned in II-C, ROMIO performs two-phase I/O in multiple iterations depending on the file domain partitioning. Processes exchange data with I/O aggregators using `MPI_Issend` and `MPI_Irecv` functions. `MPI_Waitall` is placed at the end of each two-phase I/O iteration to wait for the completion of the asynchronous communication functions. I/O aggregators thus cannot start to receive data for the next iteration of two-phase I/O until all the `MPI_Irecv` requests in the current iteration are finished. Processes that send data with `MPI_Issend` cannot exit `MPI_Waitall` until the corresponding receivers in the current iteration start to receive data, so senders cannot advance to the next iteration of two-phase I/O until the corresponding receivers in the previous iteration have finished all communications in the previous iteration. Communication time for each process per iteration can vary due to imbalanced data size, network stability, and contentions for local node hardware resources. As a result, a sender and a receiver can frequently wait for each other before starting their data. Such communication straggler effect accumulates as the number of two-phase I/O iterations increases. Therefore, a large number of two-phase I/O iterations can slow down the overall performance.

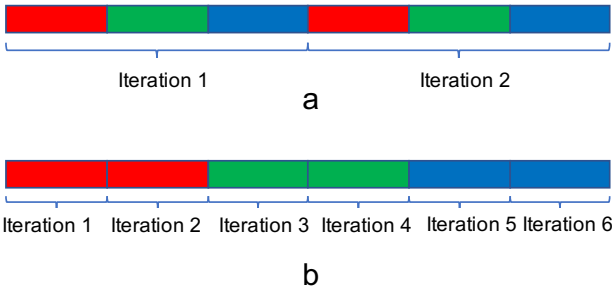

Figure 2: This figure illustrates data domain partitioning approach of ROMIO in (a) Lustre and (b) GPFS.

Tuning the file domain size per two-phase I/O iteration changes the total number of iterations. Figure 2 illustrates the file domain partitioning strategies for Lustre and GPFS. In general, increasing the file domain size per two-phase I/O iterations reduces the number of iterations. On the Lustre file system, ROMIO set the number of I/O aggregators equal to Lustre stripe size by default. This setting can avoid the Lustre lock contention issue, which can cause prolonged I/O phase performance. An I/O aggregator handles one contiguous Lustre stripe size amount of data at a time, so each two-phase I/O iteration processes a complete Lustre stripe as shown in Figure 2 (a). To reduce the number of iterations, we can increase the Lustre stripe size. In the experimental result section, we demonstrate this approach is helpful.

On GPFS, the number of I/O aggregators equals the number of compute nodes by default. The entire file access region is partitioned to the number of I/O aggregators contiguous regions as even as possible with alignments to the block boundaries. Each I/O aggregator handles one contiguous region in the collective function call as shown in Figure 2 (b). For each of the two-phase I/O iterations, an I/O aggregator processes the next `cb_buffer_size` of the contiguous block assigned to it. As a result, increasing `cb_buffer_size` can reduce the number iterations.

IV. Experimental Evaluation

We evaluate the strategies discussed in the previous section on two supercomputing systems. Cori is a Cray XC40 supercomputer with Intel KNL processors and a Lustre file system at the National Energy Research Scientific Computing Center (NERSC). We use Lustre stripe counts 64 and 128, which are two typical stripe counts used by

this application's production runs. Each Cori KNL node contains one CPU with 68 CPU cores. In our experiments, we allocate 64 processes per node in the KNL partition. Summit is a supercomputer located at the Oak Ridge Leadership Computing Facility (OLCF), with IBM Power System AC922 nodes equipped with IBM POWER9 CPUs and IBM GPFS. Each Summit node has two CPUs with 21 CPU cores per socket. In our experiments, we allocated 42 processes per node. We use the two-layered aggregation(TAM) implementation of ROMIO proposed in [6] for all experiments, which can yield better performance results than the system's default MPI-IO libraries.

We select two workloads to demonstrate the advantages of the proposed strategies. The first is E3SM-IO [25], an I/O kernel extracted from the E3SM application. The second is the HDF5 non-contiguous benchmark from H5Bench [9] that emulates the multi-dimensional block I/O access pattern on multiple HDF5 datasets collectively. We use this non-contiguous benchmark to generate random 3D block access patterns that are more complex than E3SM-IO.

A. E3SM-IO Results

We evaluate two data decomposition patterns used in the E3SM production simulation runs, namely F and G cases [8]. E3SM-IO [25] suite extracts the I/O kernels from E3SM production runs. The F case simulates atmosphere, land, and runoff models components, represented with 402 variables. The total number of I/O blocks accumulated across all variables and MPI processes for the F case is 1.37 billion. An HDF5 dataset denotes each variable, i.e., an HDF5 file generated by the F case would have 402 datasets per time stamp of checkpointed data. The total file size of the F case is ≈15GB and each I/O request is for a small amount of data. Small HDF5 blocks per dataset are distributed across 21,632 MPI processes in the production run. The G case simulates active ocean and sea-ice components, represented with 41 variables. The total number of I/O blocks accumulated across all variables and processes for the G case is 176 million. The G case decomposition file has the model for prediction across scales (MPAS) grid data structure that consists of pentagons and hexagons on top of a spherical surface. Small HDF5 blocks per dataset are distributed across 9,600 processes in the production run and the total file size is ≈85GB.

1) Optimizing with Multi-dataset I/O: In Figure 3, we summarize the time breakdown of running the E3SM-IO F and G cases on Lustre and GPFS with different parameter settings. In each plot, the first two stacked bars correspond to the F case, and the latter two to the G case. The system and the configuration used are listed at the top of each plot. We compare the implementation of HDF5 multi-dataset I/O (MDIO) with the original implementation that performs I/O each dataset separately, i.e., independent dataset I/O (IDIO) for E3SM-IO. In Table I, we show the number of two-phase I/O iterations at the ROMIO ADIO layer with different file system and I/O settings. The numbers of two-phase I/O iterations are reported from our profiling tool. However, it is also possible to roughly compute them by dividing the file domain per collective I/O function by the file domain size per iteration. Figures 3 (a)-(d) illustrate the results on Lustre stripe count 64. Performance with using a stripe count of 128 is shown in Figures 3 (e)-(h). In Figures 3 (i)-(l), we show these results for write and read operations on Summit. For the E3SM-IO F case, it is evident that the use of HDF5 MDIO significantly reduces the end-to-end time compared with IDIO. The major improvements are the two-phase I/O communication time (t_{comm}) and two-phase I/O overhead (t_{o}). The number of two-phase I/O iterations is inversely proportional to the file domain processed per two-phase I/O iterations. In addition, the number of two-phase I/O iterations is lower-bounded by the number of collective I/O calls. Independent dataset write has to trigger collective function calls for a least the number of variables times. MDIO, on the other hand, has the number of two-phase I/O iterations mainly dependent on the total data size, instead of the number of variables. The F case has a large number of variables and relatively small data size, so we expect a large difference in the number of two-phase I/O iterations between IDIO and MDIO. For the F case write with Lustre configuration of 64 stripes and 1MB per stripe, the total two-phase I/O iterations using IDIO is 591, where as that for MDIO is 229. The corresponding end-to-end timings are 22.7s for MDIO and 55.8s for IDIO as shown in Figure 3 (a). The same pattern holds for Lustre configuration of 128 stripes of 1MB reduce the total number of two-phase I/O iterations from 465 to 116. Similarly, on Summit, the total two-phase I/O iterations with IDIO from 402, and that with MDIO is 5. With fewer MPI-IO collective function calls, MDIO reduces collective function overheads, including metadata exchanges and error code synchronization. Moreover, two-

Table I: The number of iterations in two-phase I/O with different Lustre and GPFS configurations for the E3SM F and G cases. The column label the number of iterations for write/read when multi-dataset I/O (MDIO) or independent dataset IO (IDIO) operations are used.

Dataset & Setting	IDIO writes	MDIO writes	IDIO reads	MDIO reads
F case LFS 1MB 64	591	229	387	4
F case LFS 1MB 128	465	116	387	4
F case LFS 16MB 64	402	18	387	4
F case LFS 16MB 128	402	11	387	4
F case GPFS 16MB	402	5	387	3
F case GPFS 256MB	402	4	387	2
G case LFS 1MB 64	1302	1276	72	45
G case LFS 1MB 128	656	638	45	35
G case LFS 16MB 64	113	80	72	45
G case LFS 16MB 128	76	45	45	35
G case GPFS 16MB	43	23	387	23
G case GPFS 256MB	41	2	41	2

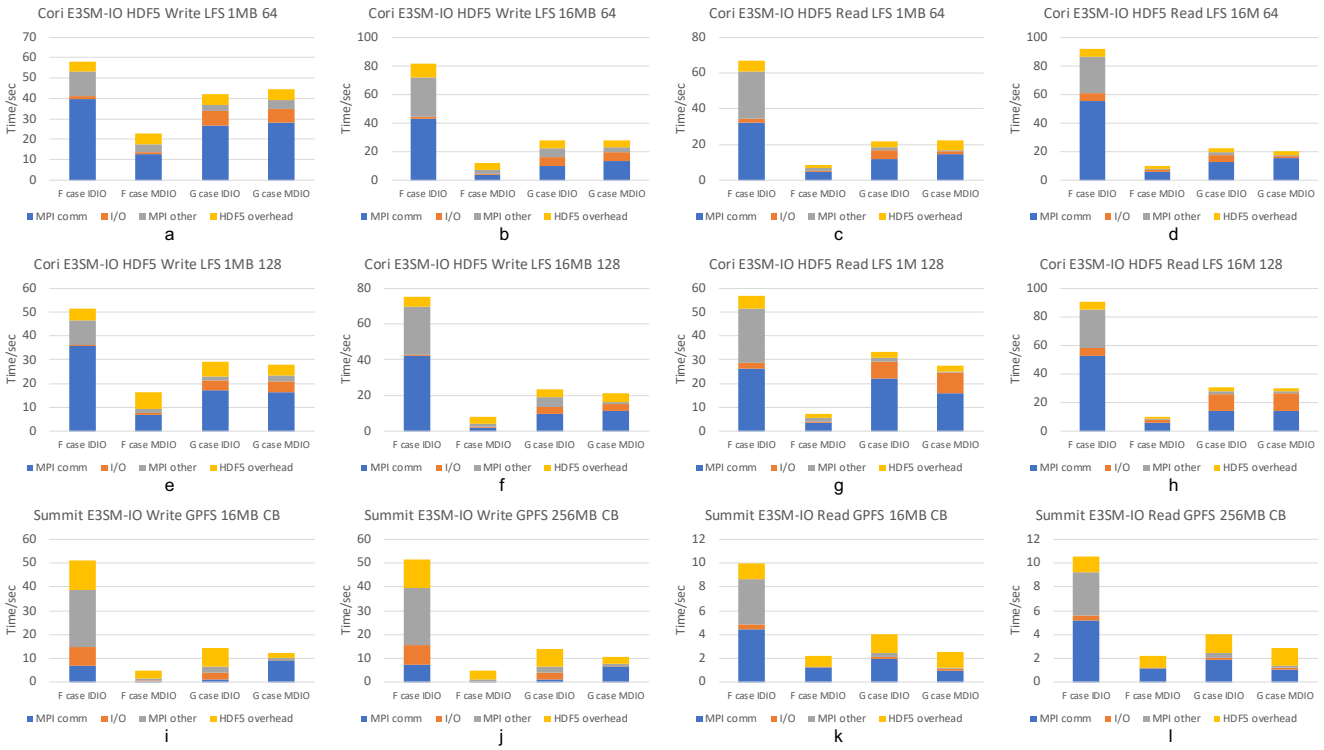

Figure 3: E3SM-IO end-to-end time breakdown for the F and G cases on Cori and Summit. We show the timings for MPI communication time (t_{comm}), I/O time ($t_{\text{I/O}}$), two-phase I/O overhead (t_{o}), and HDF5 overhead (t_{h5}) (from Eq. 1) in different colors. The x-axis indicates whether independent dataset I/O (IDIO) or multi-dataset I/O (MDIO) is used for the F or G cases. (a)-(d): E3SM-IO results on Cori Lustre (LFS) with a stripe count of 64. (e)-(h): E3SM-IO results on Cori Lustre with Lustre count 128. (i)-(l): E3SM-IO results on Summit GPFS. The LHS plots in each of these sets are for write operations and the RHS plots are for read operations. For each set of write and read plots, 1MB and 16MB stripe sizes are used on LFS, and 16MB and 256MB collective buffer sizes on GPFS.

phase I/O overhead caused by communication straggler effects also reduces as the number of two-phase I/O iterations decreases. Reducing communication straggler effects for two-phase I/O communication also reduces the receiver idle time per iteration, hence the communication overhead, i.e., t_{comm} also reduces. The same reasoning and conclusions apply to performance improvement for the F case read operations.

The G case does not benefit from HDF5 MDIO in the same way as the F case for Lustre stripe size 1MB, as shown in the RHS two bars in Figure 3. The total number of collective calls for IDIO is 41, one-tenth of the F case collective calls. However, the G case has a much larger data size. With a fewer number of variables and a larger data size, each G case variable triggers a large number of two-phase I/O iterations. From Table I, the total number of iterations is much larger than the number of variables for the G case, so the numbers of two-phase I/O iterations depend mainly on the data size instead of the number of variables for both IDIO and MDIO. Therefore, aggregation of all 41 large variables does not result in a significant difference for the number of two-phase I/O iterations. As a result, MDIO does not have a noticeable advantage according to Figures 3 for the G case runs with 1MB stripe size. A similar conclusion applies to experiments on Summit GPFS with a 16MB collective buffer size.

2) Tuning file domain size for Multi-dataset I/O: To further reduce the number of two-phase I/O iterations in ROMIO, we increase the Lustre stripe and collective buffer sizes as discussed in Section III-C. As shown in Table I, with the same Lustre stripe count 128, increasing the stripe size from 1MB to 16MB reduces the number of two-phase I/O iterations for MDIO write from IDIO's 229 to 18. Thus, t_{comm} and t_{o} are reduced by comparing the "F_case with multi" bars in Figures 3 (a) and (b). For stripe count 128, the pattern is similar by observing Figures 3 (e) and (f). For the G case write on Cori Lustre, the observations are similar to the F case.

On Summit, increasing the collective buffer size from 16MB to 256MB for MDIO does not have a significant performance improvement for the F case write according to Figures 3 (i) and (j). With a 16MB collective buffer size, the number of two-phase I/O iterations is only 5 for the F case, so two-phase I/O overhead is expected to be small. As a result, a larger collective buffer size does not improve performance in the same way as on Cori. On the other hand, the G case write can benefit from the increased collective buffer size on Summit. The G case has a larger data size than the F case, so increasing the collective buffer size from 16MB to 256MB can effectively

Table II: The number of iterations in two-phase I/O with different settings on Lustre (LFS) and GPFS for 3D non-contiguous I/O accesses. The columns label the numbers for write/read when multi-dataset I/O (MDIO) and independent dataset (IDIO) implementations are used.

# of procs	Setting	IDIO write	MDIO write	IDIO read	MDIO read
4096	LFS 1MB 64	800	627	400	40
4096	LFS 1MB 128	400	314	400	40
4096	LFS 16MB 64	400	40	400	40
4096	LFS 16MB 128	400	20	400	40
4096	GPFS 16MB	400	26	400	26
4096	GPFS 256MB	400	2	400	2
16384	LFS 1MB 64	800	627	400	10
16384	LFS 1MB 128	400	314	400	10
16384	LFS 16MB 64	400	40	400	11
16384	LFS 16MB 128	400	20	400	10
16384	GPFS 16MB	400	7	400	7
16384	GPFS 256MB	400	1	400	1

reduce the number of two-phase I/O iterations. From the third bars in Figures 3 (i) and (j), the write performance is reduced from 12.3 seconds to 10.7 seconds as we increase the collective buffer size from 16MB to 256MB for MDIO.

The number of two-phase I/O iterations for collective read is independent of Lustre stripe settings as shown in Table I because ROMIO adopts the ADIO common driver Lustre file system by default. On Summit, the collective buffer size can reduce the number of two-phase I/O iterations for the multi-dataset read in the same way as the multi-dataset write. However, the end-to-end read timings are less than 3 seconds for multi-dataset read on Summit, so performance improvement for increasing the collective buffer size is not significant.

With IDIO implementation, the numbers of two-phase I/O iterations are not reduced for the F case with different collective buffer sizes and Lustre stripe settings because these numbers are lower bounded by the numbers of collective I/O calls, which is the number of datasets. Most datasets of the F case only have one two-phase I/O iteration. As shown in the third and fourth bars in Figures 3 (i)-(l), MDIO improves the performance of the G case I/O. The G case has fewer numbers of datasets and a larger data size compared with the F case. I/O for one of the G case variables takes many two-phase I/O iterations. Consequently, increasing the file domain per two-phase I/O iterations reduces the number of two-phase I/O iterations per G case variable, which in turn reduces the total number of two-phase I/O iterations and improves the performance.

B. Evaluation of Non-contiguous Block I/O Accesses

As illustrated in the previous section, the E3SM-IO F and G cases are real-world I/O storage in 2-dimensional space. The F case has a smaller data size (15GB), but with a larger number of datasets (402) and more non-contiguous blocks. The G case has a larger data size (85GB), but with a smaller number of datasets (41) and less number of non-contiguous blocks. A more challenging pattern for non-contiguous I/O consists of both a large number of datasets and a large number of non-contiguous blocks. In this section, we use an I/O benchmark with three-dimensional non-contiguous block accesses with 400 datasets, 200K non-contiguous 8^3 blocks per dataset, and a total I/O size of 42GB. The non-contiguous 3D blocks are randomly distributed on all MPI processes, so the access pattern consists of highly non-contiguous small requests. The F and G cases only run on fixed numbers of processes (21632 and 9600). For this benchmark, we run strong-scaling experiments on both small scales (4096 processes) and large scales (16834 processes).

1) Optimizing with Multi-dataset I/O: Figures 4 (a) and (e) illustrate the write results with 4096 processes and Cori Lustre 64/128 stripe counts. MDIO has a worse write performance than IDIO as compared respectively in the first two bars in both plots. From Table II, the number of two-phase I/O iterations for IDIO write is 800 for Lustre stripe count 64 and size 1MB. MDIO write has 627 two-phase I/O iterations. If the stripe count increases from 64 to 128, the numbers of two-phase I/O iterations are approximately halved. According to the number of two-phase I/O iterations, MDIO should have better performance than that of IDIO, which is contradictory to the results. The cause of this performance degradation for MDIO is the heavy memory footprint on Cori KNL nodes, which results in unstable MPI pack and unpack operations. MPI pack reorders data in a contiguous memory buffer to another buffer in the order described by MPI data type. MPI unpack is the reverse operation. The presented results are with strong scaling, so data size per process with 4096 processes is four times the data size per process with 16384 processes. I/O aggregators gather all the metadata describing file accessing offsets and sizes at the beginning of collective I/O function calls in ROMIO. The total number of non-contiguous I/O requests is 5.2 billion across all processes. Offset and length pairs, parsed from MPI datatype passed from HDF5 layer, are stored in arrays of 8-byte integers in ROMIO. With MDIO, all non-contiguous offset/length pairs for 400 datasets are aggregated to I/O aggregators in a single collective function call. With IDIO, metadata for only one dataset is aggregated per collective function call. Thus, MDIO has a high memory footprint at the ROMIO layer for a small number of processes. MPI pack and unpack operations for the communication phase are memory-intensive operations. The performance becomes unstable because the high memory footprint occupies most of the memory of KNL nodes. As a result, communication straggler effects accumulate for a large number of two-phase I/O iterations. To reduce the memory footprint per node, we can increase the number of compute nodes. With 16384 nodes, MDIO retrieves its advantages as shown in Figures 4 (b) and (f).

Another solution for this scenario is to place the two-phase I/O aggregators on designated I/O nodes [26], so I/O aggregators can have exclusive access to hardware

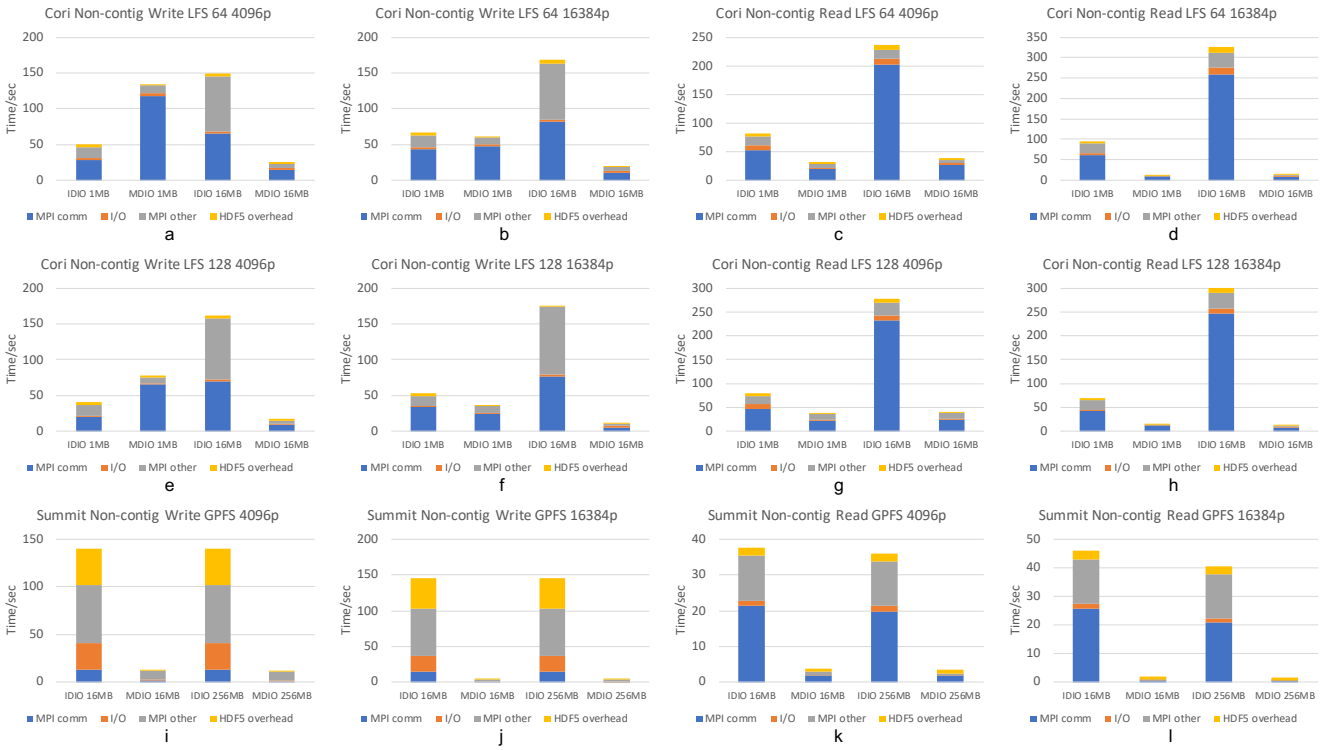

Figure 4: Performance results for non-contiguous I/O benchmark on Cori and Summit. We use a total number of 400 datasets. A dataset has 200K 3-D blocks of shape 8^3 randomly distributed over all processes. Total data size is 42GB.

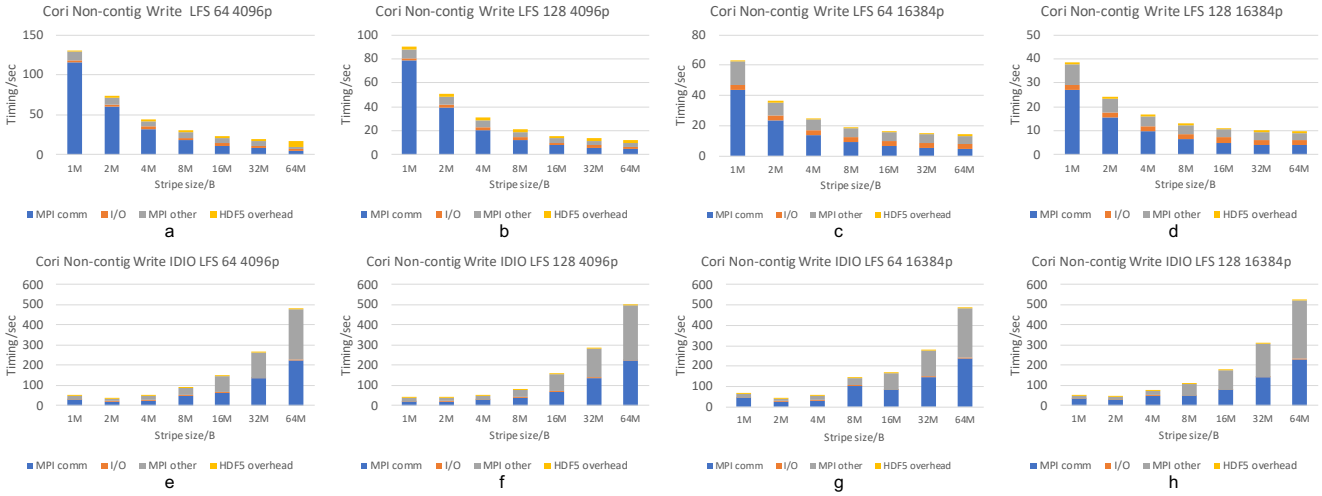

Figure 5: Non-contiguous I/O benchmark write results on Cori for Lustre striping sizes ranged from 1MB to 64MB. (a)-(d) Multi-dataset write. (e)-(h) Independent dataset write.

resources, instead of sharing these resources with other processes. This alternative solution is suitable for applications that require a specific number of processes.

On Summit GPFS, the performance improvements of MDIO are more evident than the results on Cori Lustre. From Table II, the number of two-phase I/O iterations is reduced from 400 to 26 with MDIO on 4096 processes. With 16384 processes and a default collective buffer size of 16MB, this number is further reduced to 7 because the number of I/O aggregators is increased. A larger collective buffer size of 256MB further reduces the number of two-phase I/O iterations to 2 and 1 on 4096 and 16384 processes. Consequently, we expect significant performance improvements on Summit GPFS.

2) Tuning file domain size for Multi-dataset I/O: Increasing file domain per two-phase I/O iterations can improve MDIO write performance. If we increase the stripe size from 1MB to 16MB, the numbers of two-phase I/O iterations with MDIO reduce to approximately $\frac{1}{16}$. IDIO write, on the other hand, has the same number of two-phase I/O iterations because this number is lower bounded by the number of datasets. Moreover, IDIO write

performs worse using 16MB stripe size compared with 1MB stripe size as shown in the first and the third bars in Figures 4 (a), (b), (e), and (f). As shown in Figure 2, the Lustre driver handles an entire stripe per two-phase I/O iterations. For every two-phase I/O iteration, each I/O aggregator gathers data for a contiguous file domain. In our non-contiguous benchmark settings, a dataset has approximately 128MB data size. Therefore, less than 8 I/O aggregators are active for one collective I/O function call when independent dataset write is used for Lustre stripe size 16MB. On the other hand, most I/O aggregators are active for the Lustre stripe size of 1MB configuration because a full Lustre stripe (64MB or 128MB) is not larger than a dataset size. With fewer active I/O aggregators, both communication and I/O phases become slower due to limited resources. Consequently, IDIO writes are expected to have worse performance with 16MB stripe size. The MDIO does not have the same issue since the entire 42GB file is processed with a single collective call.

Figures 4 (a)-(h) show that a larger Lustre stripe size benefits the multi-dataset implementation. We study the relationship between the number of two-phase I/O iterations and end-to-end I/O performance using the write results on Cori Lustre. Figures 5 (a)-(d) illustrates how the performance of multi-dataset write varies with different Lustre stripe sizes on Cori for 4096 and 16384 processes. The MPI-IO communication cost reduces as the stripe size increases. In theory, a large stripe size should reduce the number of two-phase I/O iterations down to the number of MPI-IO collective calls. For MDIO that writes all datasets in a single collective I/O function call, the number of two-phase I/O iterations is 1. However, infinitely large stripe size results in high memory footprints. For large files, the out-of-memory issue can occur at the ROMIO layer. In addition, Lustre file systems do not recommend a very large stripe size because it can reduce I/O performance. Thus, choosing a reasonably large stripe size depending on the memory budget is necessary.

Figures 5 (e)-(h) illustrates how the performance of independent dataset write varies with different Lustre stripe sizes on Cori for 4096 and 16384 processes. Different from the multi-dataset write, the performance of independent dataset write decreases as the stripe size increases in general. As mentioned earlier, a larger stripe size (16MB) can result in the idle of I/O aggregators with IDIO. Some I/O aggregators are idle during collective I/O if the stripe size is greater than the dataset size divided by the stripe count. To minimize the number of two-phase I/O iterations without resulting in I/O aggregators staying idle, the maximum stripe size should be the dataset size divided by the stripe count. In Figures 5, we can observe that the 2MB and 1MB stripe sizes yield the best performance for stripe counts 64 and 128 respectively, regardless of the number of processes. Therefore, if applications decide to use IDIO, this stripe size setting is the optimal choice. On GPFS, the same conclusion holds. The stripe size is replaced with collective buffer size and the stripe count is replaced with the number of I/O aggregators, which is usually the number of compute nodes.

V. Conclusion

Large-scale scientific workflow applications adopt high-level I/O libraries to abstract complex data layouts in storage for parallel I/O. Since these complex patterns are common in scientific applications that have a mismatch in data representations in memory and file spaces, optimizing I/O for them is of high importance. This paper identifies the bottlenecks of MPI-IO's two-phase I/O implementations for non-contiguous blocked parallel I/O on multiple variables. Instead of modifying middleware level implementation, we design optimization strategies for using high-level I/O libraries to avoid overheads in I/O software layers.

We have shown that sub-array aggregation and HDF5 multi-dataset collective I/O can effectively reduce the number of two-phase I/O iterations in the ROMIO's ADIO implementation using E3SM-IO and HDF5 non-contiguous benchmarks on Cori and Summit. Two-phase I/O can further reduce overheads by increasing the file domain size per two-phase I/O iteration. We plan to integrate the multi-dataset implementation into the HDF5 releases in the future. Rearranging I/O patterns submitted to the MPI-IO layer that can further reduce MPI-IO overheads are interesting future studies.

A. Acknowledgment

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. This work was supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, under Contract DE-AC02-06CH11357.

References

[1] "E3sm model." https://e3sm.org/model/e3sm-model-description. Accessed: 2021-06-26.

[2] The HDF Group, "Hierarchical Data Format, version 5," 1997-. http://www.hdfgroup.org/HDF5.

[3] R. Thakur, W. Gropp, and E. Lusk, "Data sieving and collective i/o in romio," in *Proceedings. Frontiers' 99. Seventh Symposium on the Frontiers of Massively Parallel Computation*, pp. 182–189, IEEE, 1999.

[4] Q. Kang, R. Ross, R. Latham, S. Lee, A. Agrawal, A. Choudhary, and W.-k. Liao, "Improving all-to-many personalized communication in two-phase i/o," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–13, IEEE, 2020.

[5] A. Ching, A. Choudhary, K. Coloma, W.-k. Liao, R. Ross, and W. Gropp, "Noncontiguous i/o accesses through mpi-io," in *CCGrid 2003. 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid, 2003. Proceedings.*, pp. 104–111, IEEE, 2003.

[6] Q. Kang, S. Lee, K. Hou, R. Ross, A. Agrawal, A. Choudhary, and W.-k. Liao, "Improving mpi collective i/o for high volume non-contiguous requests with intra-node aggregation," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 11, pp. 2682–2695, 2020.

[7] P. Cao, Q. Koziol, and J. Kim, "RFC: Read/Write Multiple Datasets in a HDF5 file." https://github.com/HDFGroup/hdf5doc/tree/master/RFCs/HDF5_Library/HPC_H5Dread_multi_H5Dwrite_multi, 2013.

[8] P. M. Caldwell, A. Mametjanov, Q. Tang, L. P. Van Roekel, J.-C. Golaz, W. Lin, D. C. Bader, N. D. Keen, Y. Feng, R. Jacob, *et al.*, "The doe e3sm coupled model version 1: Description and results at high resolution," *Journal of Advances in Modeling Earth Systems*, 2019.

[9] T. Li, S. Byna, H. Tang, Q. Koziol, J. Ravi, *et al.*, "H5bench: a benchmark suite for parallel hdf5 (h5bench) v0. 1," tech. rep., North Carolina State Univ., Raleigh, NC (United States), 2021.

[10] Q. Kang, S. Lee, K.-y. Hou, R. Ross, A. Agrawal, A. Choudhary, and W.-k. Liao, "Improving mpi collective i/o performance with intra-node request aggregation," *arXiv preprint arXiv:1907.12656*, 2019.

[11] W. F. Godoy, N. Podhorszki, R. Wang, C. Atkins, G. Eisenhauer, J. Gu, P. Davis, J. Choi, K. Germaschewski, K. Huck, *et al.*, "Adios 2: The adaptable input output system. a framework for high-performance data management," *SoftwareX*, vol. 12, p. 100561, 2020.

[12] "Mpich3." http://www.mpich.org/downloads/, 2017.

[13] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Proceedings, 11th European PVM/MPI Users' Group Meeting*, (Budapest, Hungary), pp. 97–104, September 2004.

[14] X. Ma, M. Winslett, J. Lee, and S. Yu, "Improving mpi-io output performance with active buffering plus threads," in *Proceedings International Parallel and Distributed Processing Symposium*, pp. 10–pp, IEEE, 2003.

[15] W.-k. Liao and A. Choudhary, "Dynamically adapting file domain partitioning methods for collective i/o based on underlying parallel file system locking protocols," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, p. 3, IEEE Press, 2008.

[16] W.-k. Liao, "Design and evaluation of mpi file domain partitioning methods under extent-based file locking protocol," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 2, pp. 260–272, 2011.

[17] X. Zhang, S. Jiang, and K. Davis, "Making resonance a common case: A high-performance implementation of collective i/o on parallel file systems," in *2009 IEEE International Symposium on Parallel & Distributed Processing*, pp. 1–12, IEEE, 2009.

[18] Y. Chen, X.-H. Sun, R. Thakur, P. C. Roth, and W. D. Gropp, "Lacio: A new collective i/o strategy for parallel i/o systems," in *2011 IEEE International Parallel & Distributed Processing Symposium*, pp. 794–804, IEEE, 2011.

[19] S. Sehrish, S. W. Son, W. keng Liao, A. Choudhary, and K. Schuchardt, "Improving collective i/o performance by pipelining request aggregation and file access," in *the 20th EuroMPI Conference*, September 2013.

[20] Y. Tsujita, H. Muguruma, K. Yoshinaga, A. Hori, M. Namiki, and Y. Ishikawa, "Improving collective i/o performance using pipelined two-phase i/o," in *Proceedings of the 2012 Symposium on High Performance Computing*, HPC '12, (San Diego, CA, USA), pp. 7:1–7:8, Society for Computer Simulation International, 2012.

[21] Y. Tsujita, K. Yoshinaga, A. Hori, M. Sato, M. Namiki, and Y. Ishikawa, "Multithreaded two-phase i/o: Improving collective mpi-io performance on a lustre file system," in *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pp. 232–235, IEEE, 2014.

[22] F. Tessier, V. Vishwanath, and E. Jeannot, "Tapioca: An i/o library for optimized topology-aware data aggregation on large-scale supercomputers," in *International Conference on Cluster Computing*, pp. 70–80, September 2017.

[23] V. Vishwanath, M. Hereld, V. Morozov, and M. E. Papka, "Topology-aware data movement and staging for i/o acceleration on blue gene/p supercomputing systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, (New York, NY, USA), pp. 19:1–19:11, ACM, 2011.

[24] M. S. Breitenfeld, K. Chadalavada, R. Sisneros, and S. Byna, "Recent progress in tuning performance of large-scale i/o with parallel hdf5," 2020. Accessed: 2021-08-25.

[25] "Parallel i/o kernel case study – e3sm." https://github.com/Parallel-NetCDF/E3SM-IO. Accessed: 2021-09-05.

[26] A. Nisar, W.-k. Liao, and A. Choudhary, "Scaling parallel i/o performance through i/o delegate and caching system," in *SC'08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, pp. 1–12, IEEE, 2008.