Supporting Data Compression in PnetCDF

Kaiyuan Hou*, Qiao Kang*, Sunwoo Lee*, Ankit Agrawal*, Alok Choudhary*, Wei-keng Liao*
*Northwestern University
{khl7265,qiao.kang,slz839,ankitag,choudhar,wkliao}@ece.northwestern.edu

2021 IEEE International Conference on Big Data (Big Data) | 978-1-6654-3902-2/21/$31.00 ©2021 IEEE | DOI: 10.1109/BigData52589.2021.9671998

***Abstract*—Recently, the dramatic increase of the data amounts drives up the demand for data compression among HPC applications. Although many file systems and I/O middlewares have incorporated compression features, few high-level parallel I/O libraries support data compression due to the challenges of achieving scalable performance on HPC systems. This paper presents the design and implementation of 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 on performance and scalability. 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. Our result suggests that handling multiple requests at once can significantly improve the parallel I/O performance on chunked and compressed data.**

***Index Terms*—Compression, Chunked Storage Layout, NetCDF, I/O Aggregation**

I. INTRODUCTION

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 [1]. 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 [2], [3] 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 [4] 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 work, we introduce a compression feature for variables in the PnetCDF [5] library. Scientific applications in various domains, including much of the climate applications, store their data in classic NetCDF format [6], [7], [8]. The classic NetCDF format [9] 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[4]. 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 [10].

We evaluated our implementation on Cori [11] 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 [12] simulation framework, as well as the Pandana module [13] in the NuMI Off-axis ν_e Appearance (NOvA) experiment [14].

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.

II. 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 [15]. Filgueira et al. use compression to reduce the communication time between compute nodes and I/O aggregators in MPI-IO [16]. Islam et al. utilize compression to reduce checkpointing overhead [17]. Bui et al. proposed several techniques, including compression, to improve the I/O performance on IBM Blue Gene/Q supercomputers [18].

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 [19] 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 [20]. 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 [21]. Other than a workaround to access the compressed dataset in parallel, chunked storage layout is also used to store data growing along multiple dimensions [22], [23]. Zarr is a python package that implements chunked multi-dimensional arrays that can be accessed concurrently by all threads or processes [24]. Zarr allows chunks to be transformed using the Numcodecs [25] package. N5 is a library providing primitive operations to store chunked n-dimensional arrays [26]. It stores every chunk as a single file under a directory representing the array. ADIOS [27] supports data compression by compressing individual data blocks in a log-based storage layout.

A. Data compression in parallel HDF5

HDF5 [4] 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.

B. NetCDF

NetCDF (Network Common Data Form) [9], [28] 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 [29] 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 work, we focus on supporting variable compression in the classic NetCDF format.

C. PnetCDF

PnetCDF [5] 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.

III. 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.

A. 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 [30].

- **_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.

B. 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.

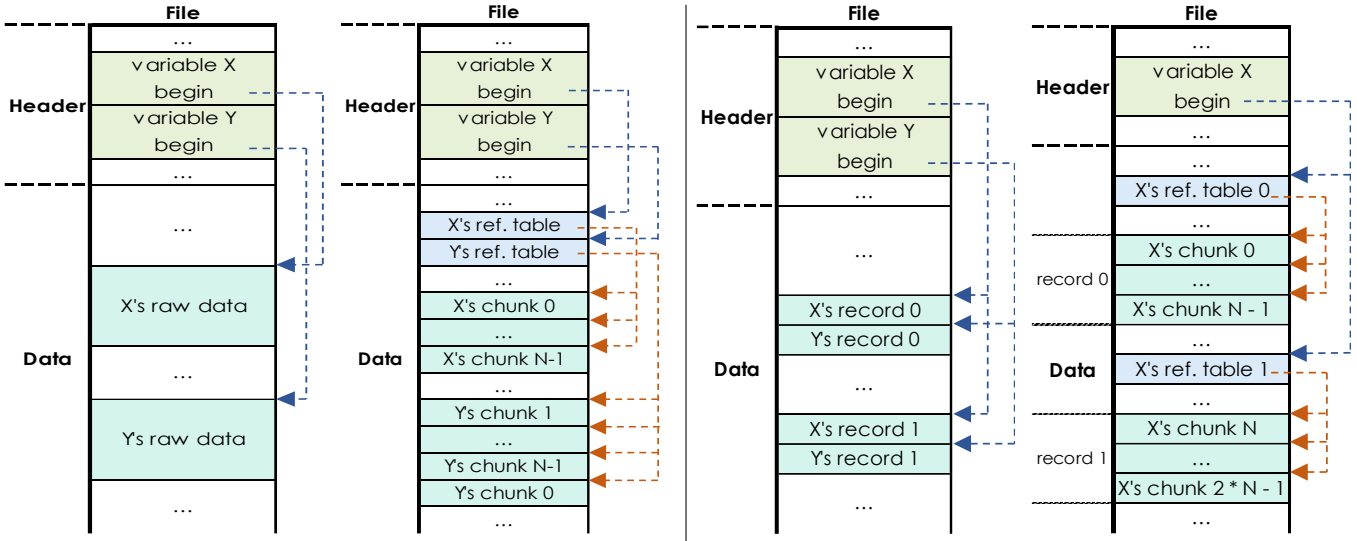
Without violating the NetCDF file format specification, we store the new data objects in the "free space" between variables. The free spaces are paddings between the end of a variable and the beginning of the next variable. Paddings are required to comply with the NetCDF format specification in which 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 [31], [32]. 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 [33], [34], zstd [35], SZ [36], ZFP [37] ... etc.

C. Chunk data layout for fixed-size variables

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



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

Fig. 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.

Fig. 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.

D. Chunk data layout for record variables

Fig. 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 [38]. 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. 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.

E. 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.

F. 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.

G. 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.

IV. 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 [10] 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 III-F and Section III-G. 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.

V. EXPERIMENT

We ran experiments on Cori, a Cray XC40 supercomputer at National Energy Research Scientific Computing Center (NERSC) [11], [39] 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 [39] and about 1 GiB/s per compute node [40].

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 [12] with a fragment and near-random I/O pattern. Another is the Pandana I/O [14] 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. 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 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.

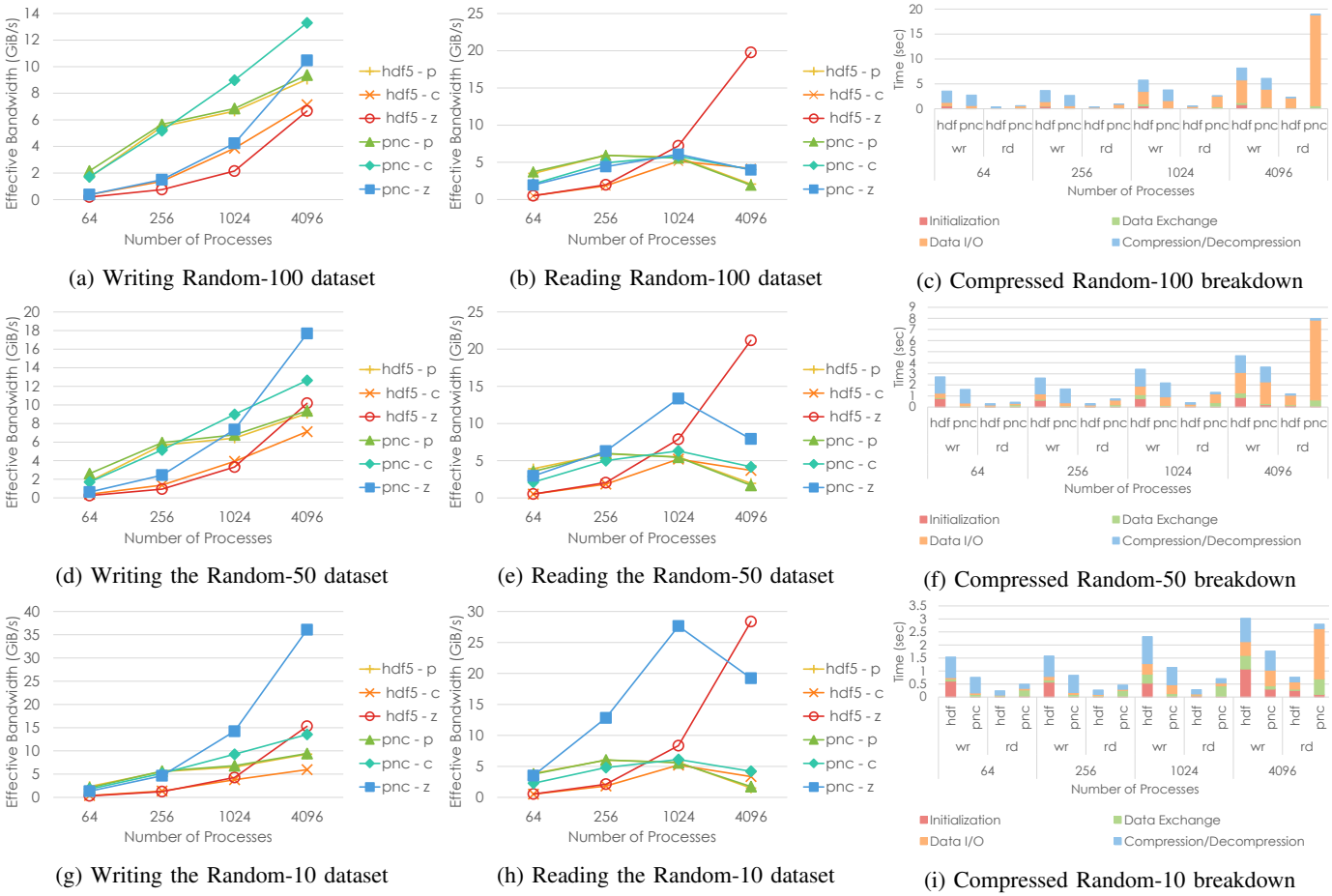
Chunked and compressed layout out-performs the contiguous storage layout even on the non-compressible dataset due to more efficient I/O pattern [41], [42]. While compression cannot improve write performance compared to chunked storage layout except on the highly compressible dataset, it can boost read performance on the 50% compressible dataset in both PnetCDF and HDF5. The main reason is that inflate (decompression) runs significantly faster than deflate (compression) [43]. We need more size reduction in I/O to compensate for compression cost than that to compensate for decompression cost.

B. 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) [44], such as the FLASH code [45] and AMReX [46].

We used the data generated by the Galaxy Cluster Merger simulation, a FLASH [45] application, from the sample datasets of the yt project [47]. 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 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



(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

Fig. 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.

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.

C. Pandana I/O

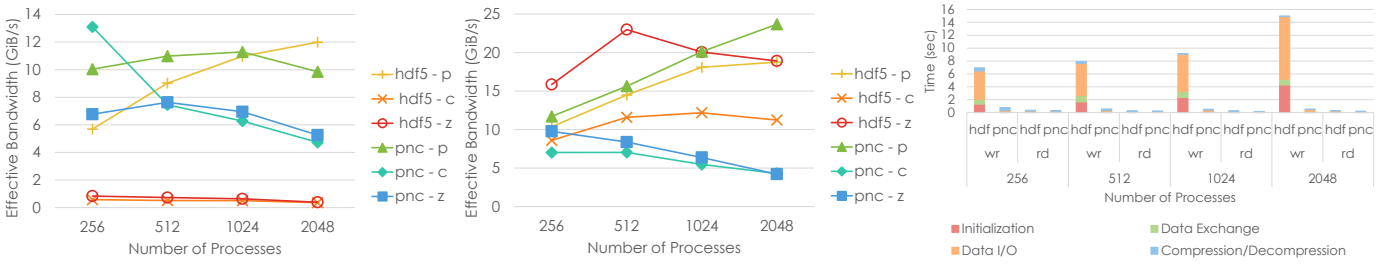
We implemented an I/O benchmark to simulate the I/O patterns in the Pandana framework [13] used in the NuMI Off-axis ν_e Appearance (NOvA) experiment designed to study neutrino oscillations [14]. In the NoVA experiment, sensors are set up to monitor particle collision events and other events of interest. 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 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.

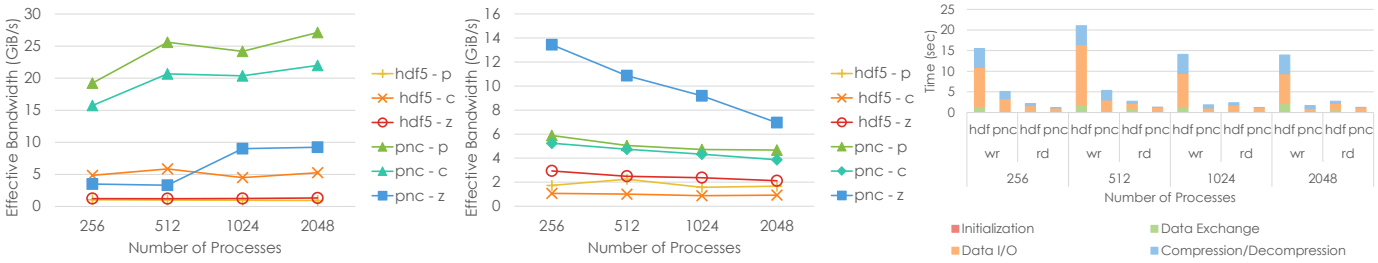
We gathered the data from 1951 rounds of NoVA experiments. We conducted the experiment using a subset of events (groups) used in one of the NOvA experiment's analysis tasks. The subset contains 108 variables organized in 15 groups totaling 7.09 GiB. We set the chunk size to 1 MiB. In this experiment, 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 4 shows the result on the NoVA dataset. Despite a



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

Fig. 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.



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

Fig. 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.

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.

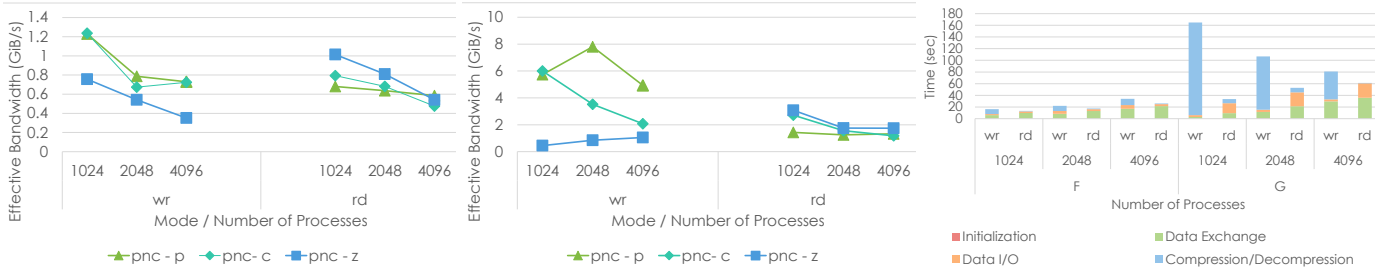
With the advantage of I/O aggregation, PnetCDF outperform HDF5 by a huge margin on all storage layouts for both reading and writing. PnetCDF performs 5 and 14 times faster than HDF5 on writing and reading, respectively, when using a compressed storage layout. Unlike that in FLASH I/O, HDF5 does not perform 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.

D. 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 [12]. 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 [48]. 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 [49]. 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



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

Fig. 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.

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. 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 [50].

VI. Conclusion

In this paper, 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.

Acknowledgment

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) program under Award Numbers DE-SC0021399 and DE-SC0019358. This work is partially supported by the National Institute of Standards and Technology award number 70NANB19H005. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility located at Lawrence Berkeley National Laboratory, operated under Contract No. DE-AC02-05CH11231.

REFERENCES

[1] M. Paterno, J. Kowalkowski, and S. Sehrish, "Parallel event selection on hpc systems," in *EPJ Web of Conferences*, vol. 214. EDP Sciences, 2019, p. 04059.

[2] R. Agarwal, A. Khandelwal, and I. Stoica, "Succinct: Enabling queries on compressed data," in *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, 2015, pp. 337–350.

[3] F. Zhang, J. Zhai, X. Shen, D. Wang, Z. Chen, O. Mutlu, W. Chen, and X. Du, "Tadoc: Text analytics directly on compression," *The VLDB Journal*, vol. 30, no. 2, pp. 163–188, 2021.

[4] "The hdf5® library & file format." [Online]. Available: https://www.hdfgroup.org/solutions/hdf5/

[5] J. Li, W.-k. Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale, "Parallel netcdf: A high-performance scientific i/o interface," in *SC'03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*. IEEE, 2003, pp. 39–39.

[6] G. Peng, W. Meier, D. Scott, and M. Savoie, "A long-term and reproducible passive microwave sea ice concentration data record for climate studies and monitoring," 2013.

[7] T. W. Estilow, A. H. Young, and D. A. Robinson, "A long-term northern hemisphere snow cover extent data record for climate studies and monitoring," *Earth System Science Data*, vol. 7, no. 1, p. 137, 2015.

[8] "Where is netcdf used?" [Online]. Available: https://www.unidata.ucar.edu/software/netcdf/usage.html

[9] R. Rew and G. Davis, "Netcdf: an interface for scientific data access," *IEEE computer graphics and applications*, vol. 10, no. 4, pp. 76–82, 1990.

[10] K. Gao, W.-k. Liao, A. Choudhary, R. Ross, and R. Latham, "Combining i/o operations for multiple array variables in parallel netcdf," in *2009 IEEE International Conference on Cluster Computing and Workshops*. IEEE, 2009, pp. 1–10.

[11] K. Antypas, N. Wright, N. P. Cardo, A. Andrews, and M. Cordery, "Cori: a cray xc pre-exascale system for nersc," *Cray User Group Proceedings. Cray*, 2014.

[12] E3SM Project, "Energy Exascale Earth System Model (E3SM)," [Computer Software] https://dx.doi.org/10.11578/E3SM/dc.20180418.36, Apr. 2018. [Online]. Available: https://dx.doi.org/10.11578/E3SM/dc.20180418.36

[13] M. Groh, N. Buchanan, D. Doyle, J. B. Kowalkowski, M. Paterno, and S. Sehrish, "Pandana: A python analysis framework for scalable high performance computing in high energy physics," Fermi National Accelerator Lab.(FNAL), Batavia, IL (United States), Tech. Rep., 2021.

[14] Fermi National Accelerator Laboratory. NOvA experiment. Https://novaexperiment.fnal.gov/.

[15] B. Welton, D. Kimpe, J. Cope, C. M. Patrick, K. Iskra, and R. Ross, "Improving i/o forwarding throughput with data compression," in *2011 IEEE International Conference on Cluster Computing*. IEEE, 2011, pp. 438–445.

[16] R. Filgueira, D. E. Singh, J. C. Pichel, and J. Carretero, "Exploiting data compression in collective i/o techniques," in *2008 IEEE International Conference on Cluster Computing*. IEEE, 2008, pp. 479–485.

[17] T. Z. Islam, K. Mohror, S. Bagchi, A. Moody, B. R. De Supinski, and R. 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*. IEEE, 2012, pp. 1–11.

[18] H. Bui, H. Finkel, V. Vishwanath, S. Habib, K. Heitmann, J. Leigh, M. Papka, and K. 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*. IEEE, 2014, pp. 107–111.

[19] U. Rasthofer, F. Wermelinger, P. Hadijdoukas, and P. Koumoutsakos, "Large scale simulation of cloud cavitation collapse," *Procedia Computer Science*, vol. 108, pp. 1763–1772, 2017.

[20] P. Hadjidoukas and F. Wermelinger, "A parallel data compression framework for large scale 3d scientific data," *arXiv preprint arXiv:1903.07761*, 2019.

[21] T. Bicer, J. Yin, and G. 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*. IEEE, 2014, pp. 1–10.

[22] E. Otoo, G. Nimako, and D. Ohene-Kwofie, "Using chunked extendible array for physical storage of scientific datasets," in *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. IEEE, 2012, pp. 1315–1321.

[23] G. Nimako, E. J. Otoo, and D. Ohene-Kwofie, "Chunked extendible dense arrays for scientific data storage," in *2012 41st International Conference on Parallel Processing Workshops*. IEEE, 2012, pp. 38–47.

[24] "Zarr¶." [Online]. Available: https://zarr.readthedocs.io/en/stable/

[25] "Numcodecs¶." [Online]. Available: https://numcodecs.readthedocs.io/en/stable/

[26] "saalfeldlab/n5." [Online]. Available: https://github.com/saalfeldlab/n5

[27] Q. Liu, J. Logan, Y. Tian, H. Abbasi, N. Podhorszki, J. Y. Choi, S. Klasky, R. Tchoua, J. Lofstead, R. Oldfield *et al.*, "Hello adios: the challenges and lessons of developing leadership class i/o frameworks," *Concurrency and Computation: Practice and Experience*, vol. 26, no. 7, pp. 1453–1473, 2014.

[28] R. 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.

[29] 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.

[30] R. Rew, E. Hartnett, D. Heimbigner, E. Davis, and J. Caron, "Netcdf classic and 64-bit offset file formats," Open Geospatial Consortium (OGC), Geneva, CH, NASA Earth Science Data System (ESDS) community standard, August 2008.

[31] "Pnetcdf user guide." [Online]. Available: https://parallel-netcdf.github.io/wiki/Documentation.html

[32] "Netcdf user's guide." [Online]. Available: https://www.unidata.ucar.edu/software/netcdf/guide_toc.html

[33] P. Deutsch and J.-L. Gailly, "Zlib compressed data format specification version 3.3," RFC 1950, May, Tech. Rep., 1996.

[34] J.-l. Gailly and M. Adler, "Zlib compression library," 2004.

[35] Y. Collet and E. Kucherawy, "Zstandard-real-time data compression algorithm," 2015.

[36] S. Di and F. Cappello, "Fast error-bounded lossy hpc data compression with sz," in *2016 ieee international parallel and distributed processing symposium (ipdps)*. IEEE, 2016, pp. 730–739.

[37] P. Lindstrom, "Fixed-rate compressed floating-point arrays," *IEEE transactions on visualization and computer graphics*, vol. 20, no. 12, pp. 2674–2683, 2014.

[38] "Appendix b. file format specifications." [Online]. Available: https://www.unidata.ucar.edu/software/netcdf/docs/file_format_specifications.html

[39] T. Declerck, K. Antypas, D. Bard, W. Bhimji, S. Canon, S. Cholia, H. Y. He, D. Jacobsen, and N. J. W. Prabhat, "Cori-a system to support data-intensive computing," *Cray User Group*, 2016.

[40] J. Liu, Q. Koziol, H. Tang, F. Tessier, W. Bhimji, B. Cook, B. Austin, S. Byna, B. Thakur, G. Lockwood *et al.*, "Understanding the i/o performance gap between cori knl and haswell," Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), Tech. Rep., 2017.

[41] M. Howison, "Tuning hdf5 for lustre file systems," 2010.

[42] M. Folk, G. Heber, Q. Koziol, E. Pourmal, and D. Robinson, "An overview of the hdf5 technology suite and its applications," in *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*. ACM, 2011, pp. 36–47.

[43] J. Alakuijala, E. Kliuchnikov, Z. Szabadka, and L. Vandevenne, "Comparison of brotli, deflate, zopfli, lzma, lzham and bzip2 compression algorithms," *Google Inc*, 2015.

[44] W. Zhang, A. Almgren, V. Beckner, J. Bell, J. Blaschke, C. Chan, M. Day, B. Friesen, K. Gott, D. Graves, M. Katz, A. Myers, T. Nguyen, A. Nonaka, M. Rosso, S. Williams, and M. Zingale, "AMReX: a framework for block-structured adaptive mesh refinement," *Journal of Open Source Software*, vol. 4, no. 37, p. 1370, May 2019. [Online]. Available: https://doi.org/10.21105/joss.01370

[45] A. Dubey, K. Antypas, A. Calder, B. Fryxell, D. Lamb, P. Ricker, L. Reid, K. Riley, R. Rosner, A. 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*. IEEE Press, 2013, pp. 1–8.

[46] W. Zhang, A. Almgren, V. Beckner, J. Bell, J. Blaschke, C. Chan, M. Day, B. Friesen, K. Gott, D. Graves *et al.*, "Amrex: a framework

for block-structured adaptive mesh refinement," *Journal of Open Source Software*, vol. 4, no. 37, 2019.
[47] "Get yt: all-in-one script." [Online]. Available: https://yt-project.org/
[48] J. Sturtevant, M. Christon, P. D. Heermann, and P.-C. Chen, "Pds/pio: Lightweight libraries for collective parallel i/o," in *SC'98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*. IEEE, 1998, pp. 3–3.
[49] 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*.
[50] 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 *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020, pp. 1–13.