Combining I/O Operations for Multiple Array Variables in Parallel NetCDF

Kui Gao, Wei-keng Liao, Alok Choudhary
Electrical Engineering and Computer Science Department
Northwestern University
Evanston, IL 60208, USA
{*kgao,wkliao,choudhar*}*@eecs.northwestern.edu*

Robert Ross and Robert Latham
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439, USA
{*rross,robl*}*@mcs.anl.gov*

***Abstract*—Parallel netCDF (PnetCDF) is a popular library used in many scientific applications to store scientific datasets. It provides high-performance parallel I/O while maintaining file-format compatibility with Unidata's netCDF. Array variables comprise the bulk of the data in a netCDF dataset, and for accesses to large regions of single array variables, PnetCDF attains very high performance. However, the current PnetCDF interface only allows access to one array variable per call. If an application instead accesses a large number of small-sized array variables, this interface limitation can cause significant performance degradation, because high end network and storage systems deliver much higher performance with larger request sizes. Moreover, the record variables data is stored interleaved by record, and the contiguity information is lost, so the existing MPI-IO collective I/O optimization can not help. This paper presents a new mechanism for PnetCDF to combine multiple I/O operations for better I/O performance. This mechanism can be used in a new function that takes arguments for reading/writing multiple array variables, allowing application programmers to explicitly access multiple array variables in a single call. It can also be used in the implementation of asynchronous I/O functions, so that the combination is carried out implicitly, without changes to the application. Our performance results demonstrate significant improvement using well-known application benchmarks.**

***Keywords*-parallel netCDF; MPI-IO; parallel-IO**

Many large-scale scientific applications are data-intensive, processing large array data sets, such as climate modeling, fusion, fluid dynamics, and computational biology. These applications produce enormous amounts of data and require effective I/O libraries and fast storage systems in order to operate effectively. The Network Common Data Format (netCDF) [1], [2] defines a set of I/O functions for serial file access and a machine-independent data format to support the creation, access, and sharing of array-oriented scientific data. Data stored in netCDF format are described as array variables with well-defined attributes, such as dimensions, data types, and annotations. NetCDF has become widely used by many scientific applications to store data in files, along with their metadata, and in a portable file format.

In order to handle increasing size of scientific data and enable efficient use of netCDF files in parallel programs, parallel netCDF (PnetCDF) provides a set of application programming interfaces for concurrent file access [3]. Its implementation is built on top of MPI-IO [4] and hence can take advantage of MPI's portability and already-optimized I/O features to achieve high performance. However, as with the serial netCDF interface, the variable ID is one input parameter in the I/O opertion functions of current PnetCDF library. The PnetCDF application interface only allows reading or writing one array variable at a time. There are two kinds of array variables in netCDF, namely record and non-record array variables. The dimensionalities of a non-record variable are fixed and a record variable is not. The most significant dimension of a record variable can be increased. In a netCDF file (as shown in Figure 1 [3]), a non-record variable is stored in a contiguous file space and record array variables are stored interleaved with each other at the end of file. If applications have a large number of small-sized array variables, reading or writing one variable at a time could result in poor I/O performance, due to the under-utilized network bandwidth. Record variables are interleaved and the contiguity information is lost, so accessing one record array variable has a poor performance due to noncontiguous accesses.

This paper presents a new mechanism for combining I/O operations of multiple array variables in PnetCDF for better I/O performance. It can be used to define new I/O functions that explicitly ask for a list of arguments of multiple I/O buffers and corresponding variable IDs. It can also be used implicitly in applications using the asynchronous PnetCDF write operations without modification of the existing application program. The implementation involves defining a new combined MPI file view for array variables' file layout as well as a combined MPI derived datatype for the I/O buffers. We describe the challenge of multiple file views combination, as MPI-IO semantics require that the displacements of a file view to be used in the collective I/O must be in monotonically non-decreasing order. We adopt different I/O schemes for non-record array variables and records in order to fulfill MPI-IO's requirements.

The rest of this paper is organized as follows. Section 2 provides the relevant background for this work. Section 3 our mechanism for combining I/O operations for multiple array variables in PnetCDF. Section 4 describes the results of our experiments. Section 5 concludes the paper.

978-1-4244-5012-1/09/$25.00

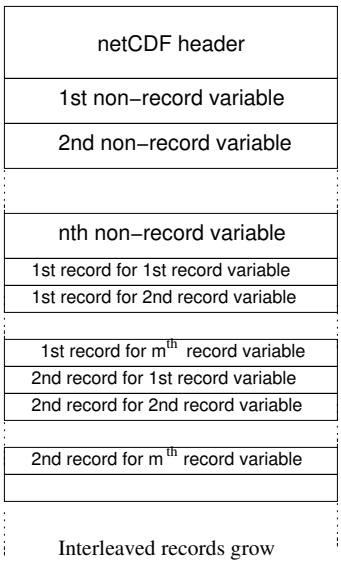

Figure 1. NetCDF file structure

I. Related Work

Before we discuss our new mechanism and enhancements to PnetCDF, we cover the related work and background of this work, namely MPI-IO, HDF5 and PnetCDF.

A. MPI I/O

The majority of large-scale scientific parallel applications are written using the message passing model and the Message Passing Interface (MPI) [5]. MPI-IO is an important feature of the MPI-2 standard [4], which allows multiple processes of a parallel program to access data in a shared file simultaneously. MPI-IO inherits two important MPI features: MPI communicators defining a set of processes for group operations, and MPI derived datatypes describing complex memory layouts. A communicator specifies the processes that participate in a collective operation for both interprocess communication and file I/O. When opening a file, the MPI communicator is a required argument to indicate the group of processes accessing the file. MPI collective I/O functions also require all processes in the communicator to participate. Such an explicit coordination allows a collective I/O implementation to exchange access information among all processes and reorganize I/O requests for better performance. Independent I/O functions, in contrast, require no coordination but make any collaborative optimization very difficult.

A success story of using process collaboration to improve shared-file I/O performance is two-phase I/O [6]. This collective I/O optimization assumes that file systems handle large contiguous requests much better than small noncontiguous ones. Two-phase I/O first calculates the aggregate access region, a contiguous file region starting from the minimal access offset among the requesting processes and ending at the maximal offset among the processes. The aggregate access region is then divided into non-overlapping, contiguous sub-regions denoted as file domains, and each file domain is assigned to a unique process. A process makes read/write calls on behalf of all processes for the requests located in its file domain. The two-phase I/O is adopted by ROMIO, a popular MPI-IO implementation developed at Argonne National Laboratory [7].

In addition to the two-phase I/O, many collaboration strategies have been proposed and demonstrated their success, including disk-directed I/O [8], server-directed I/O [9], persistent file domain [10], active buffering [11], and collaborative caching [12]. Many of these optimizations are implemented under the MPI-IO interface, meaning that libaries using MPI-IO can benefit from these optimizations without any additional coding effort.

B. HDF5

Hierarchical Data Format (HDF) is a file format and software, developed at NCSA, for storing, retrieving, analyzing, visualizing, and converting scientific data. The most popular versions of HDF are HDF4 [13] and HDF5 [14]. Both versions store multidimensional arrays together with ancillary data in portable, self-describing file formats. HDF4 was designed with serial data access in mind, much like the current netCDF interface. HDF5 is a major revision in which its API is completely redesigned and now includes parallel I/O access, much like the current pnetCDF.

HDF5 can store large numbers of large data objects, such as multidimensional arrays, tables, and computational meshes, and these can be mixed together in any way that suits a particular application. HDF5 supports cross platform portability of the interface and corresponding file format, as well as ease of access for scientists and software developers. The main conceptual building blocks of HDF5 are the "dataset" and the "group". An HDF5 dataset is a multidimensional array of elements of a specified datatype. Datatypes can be atomic (integers, floats, and others) or compound. A compound datatype is similar to a struct in C or a common block in FORTRAN. It is a collection of one or more atomic types or small arrays of such types. However, compound datatype must be fixed total size and fixed variable number. This is a limit in applications.

C. Parallel netCDF

Dataset storage, exchange, and access play a critical role in scientific applications. For such purposes, netCDF serves as a software library and self-describing machine-independent data format that support the creation, access, and sharing of array-oriented scientific data. NetCDF stores data in an array-oriented dataset which contains dimensions, array variables, and attributes. As illustrated in Figure

1, a netCDF file is divided into three parts: file header, non-record array variables and record array variables. The netCDF file header stores metadata, such as array dimensions, names and sizes of dimensions, data types, and character strings for annotations. The dimension metadata are used to define the shapes and attributes of array variables. Non-record array variables must be defined with fixed sizes for all dimensions. Record array variables allow the most significant dimension to be defined as "unlimited". All record array variables are expected to grow together along that dimension. Non-record variable arrays are stored in the first section, followed by the second section for record variable arrays. For variable-sized record arrays, netCDF first defines a record of an array as a subarray comprising all fixed dimensions and the records of all arrays are stored interleaved in the arrays' defined order.

The netCDF API is originally designed for serial data access, and it does not define semantics for concurrent access, particularly concurrent writing to a netCDF dataset. PnetCDF has been developed to support parallel I/O operations and large file size (greater than 4 GB). Some of its I/O functions take an additional argument for an MPI communicator to indicate the processes participating the shared-file I/O operations, but are otherwise very similar to the serial netCDF API. Internally, PnetCDF constructs MPI derived datatypes from each process's requests to a subarray and defines the process's file view. Any MPI file hints supplied by the user, often used for performance tuning, are passed to the underlying MPI-IO library, so PnetCDF can fully take advantage of all I/O optimizations available in the MPI-IO layer.

In PnetCDF a file is opened, operated, and closed by the participating processes in an MPI communication group. Internally, the header is read/written only by a single process, although a copy is cached in local memory on each process. Header modifications are made in "define mode", the mode used to describe the contents of a netCDF file. The root process fetches the file header, broadcasts it to all processes when opening a file, and writes the file header at the end of the define mode if any modifications occur in the header. The define mode functions, attribute functions, and inquiry functions all work on the local copy of the file header. All define mode and attribute functions are made collectively and require all the processes to provide the same arguments when adding, removing, or changing definitions so the local copies of the file header are guaranteed to be the same across all processes from the time the file is collectively opened until it is closed.

The APIs provide users with the ability to use MPI derived datatypes to describe noncontiguous memory regions for the I/O buffers, meaning that users do not need to copy data into a contiguous buffer prior to making a PnetCDF call. The data partitioning of a variable and the corresponding file regions in a process is internally converted to an MPI file view. The file views are calculated from the variable's metadata (shape, size, offset, etc.) and the arguments of starts, counts, strides, and datatype provided by the user. These file views are passed to MPI-IO calls to allow noncontiguous regions in the netCDF file to be accessed with a single call. Below are two example collective functions for writing a variable:

```
ncmpi_put_vara_int_all(file_id,
          var_id, start[],
          count[], buffer)

ncmpi_put_vara_all(file_id, var_id,
          start[], count[], buffer,
          bufcount, buftype)
```

II. Design and Implementation

Data files in the netCDF file formats are called self-describing files because the data and metadata are packaged together in the same file. PnetCDF provides two categories I/O functions: header I/O and parallel data I/O. Since the majority of time spent accessing a netCDF file is in data access, the data I/O must be efficient. PnetCDF provides the implementation of parallel I/O for array data above MPI-IO. PnetCDF library offers a number of advantages. For parallel access, particularly for collective access, each process has a different file view. All processes in combination can make a single MPI-IO request to transfer large contiguous data as a whole, thereby preserving useful semantic information that would otherwise be lost if the transfer were expressed as per process noncontiguous requests. However, the record array variables are stored interleaved by record, and the contiguity information is lost, so the existing MPI-IO collective I/O optimization may not help. In such cases, more optimization information from users can be beneficial, such as the number, order, and record indices of the record variables they will access consecutively. With such information we can collect multiple I/O requests over a number of record variables and optimize the file I/O over a large pool of data transfers, thereby producing more contiguous and larger transfers.

Current PnetCDF does not provide functionality for read or write multiple array variables in a single call. Data Access Functions of PnetCDF provide the ability to read/write variable data in one of the five access methods: single element, whole array, subarray, subsampled array (strided subarray) and mapped strided subarray. Programs must access one variable at a time. As this limitation may hamper the I/O performance for accessing a large number of small-sized array variables. We propose a new mechanism to combine multiple requests into a single request. Two important implementation issues are to create a combined buffer datatype for multiple write buffers and, similarly, a combined file datatype for the array variables in the file. The combined buffer datatype and file datatype will be used in a single MPI collective I/O call. Since record and non-record array variables are stored in different ways, the construction for

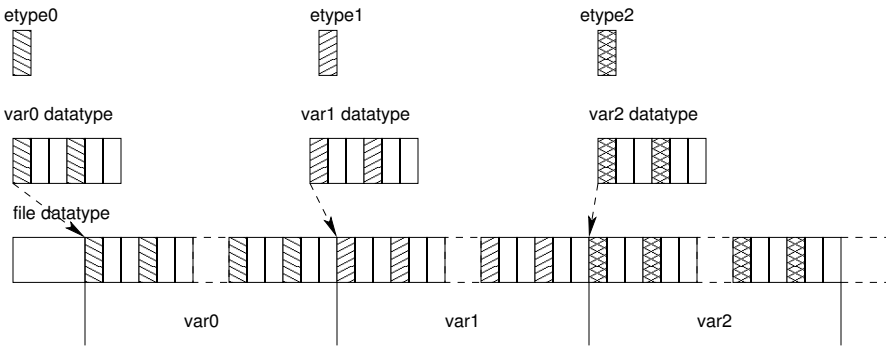

Figure 2. Combining derived datatypes for non-record array variables

the new combined MPI derived datatypes must treat these two kinds of array variables differently. This functionality is implemented in a subroutine that takes inputs from the variable metadata and user buffers to prepare the combined arguments for a single collective I/O call.

A. Connecting to Applications

This functionality can potentially be adopted by applications in two ways: (1) explicitly using a new PnetCDF function that allow users to provide lists of array variables and I/O buffers as the arguments; and (2) implicitly using this functionality by making multiple asynchronous I/O operations that accumulate read/write requests for multiple array variables.

For the explicit approach, new calls are provided:

```
int ncmpi_put_mvara_all(int file_id,
         int nvars,
         int varid_list[],
         const MPI_Offset *start_list[],
         const MPI_Offset *count_list[],
         const void **buf_list,
         int *bufcount_list,
         MPI_Datatype *datatype_list).

int ncmpi_get_mvara_all(int file_id,
         int nvars,
         int varid_list[],
         const MPI_Offset *start_list[],
         const MPI_Offset *count_list[],
         const void **buf_list,
         int *bufcount_list,
         MPI_Datatype *datatype_list).
```

In the implicit approach, when an asynchronous write request is posted, the arguments parameters are simply stored. During the wait call, normally used to test for completion of asynchronous I/O completion, the saved arguments are used to produce the combined file views and datatypes, and the I/O is performed. The implicit approach benefits the applications without modifying their source codes; a new PnetCDF hint is simply used to enable this feature. For the implicit approach, a write example is as follows.

```
MPI_Info_set(info,
             "multi-variable I/O",
             "enable");
ncmpi_create(... info);
...
ncmpi_iput_vara(ncid,
             varid1, buf, bufcount,
             datatype, ncmpi_request);
...
ncmpi_iput_vara(ncid,
             varid2, buf, bufcount,
             datatype, ncmpi_request);
...
ncmpi_iput_vara(ncid,
             varidm, buf, bufcount,
             datatype, ncmpi_request);
...
ncmpi_wait();
...
```

When the applications call *ncmpi_iput_vara_all()* function, the input parameters infomation only are stored into *ncmpi_request* struct and nothing is done. When *ncmpi_wait_all()* is called, the program performs the write operation.

1. Input start[], count[], stride[] datatype arguments provided by users
2. Get variables metadata (shape, size, offset, etc.)
3. Create datatype for each variable
4. Construct a filetype from multiple non−record variables datatype
5. Setup fileview, each process has a different file view
6. Create buffer datatype for each variable's buffer
7. Construct a filetype from multiple non−record variables buffer datatype
8. Call MPI_FILE_Write_all() or MPI_FILE_Write_AT_all() functions

Figure 3. Procedure for writing combined non-record array variables

B. Implementation for Non-Record array variables

All non-record array variables are stored contiguously starting after the file header and header padding. When reading/writing a variable, each process defines an MPI file view to specify the file locations it is accessing to. In our new system, the file views for multiple array variables are concatenated into a new file view, using an MPI derived

datatype constructor, *MPI_Type_hindexed()*. Each process first calculates from its I/O request a list of offsets and lengths for each variable. Since non-record array variables are stored in a contiguous space in the file, the lists of offset-length pairs will be combined in the same order as the array variables are defined; this may require combining in a different order than originally specified by the user. The combination process is depicted in Figure 2. The memory buffers are combined in an identical manner and their absolute memory addresses are used to construct the new buffer derived datatype. The newly created file datatype is used in *MPI_File_set_view()* to define the file view and the buffer datatype are used in *MPI_File_write_all()* for the write buffers and *MPI_File_read_all()* for the read buffers. The write procedure is shown in Figure 3.

C. Implementation for Record array variables

Record array variables are stored in an interleaved manner at the end of the so that they can grow along the unlimited dimension. The datatype combination procedure for non-record array variables cannot be directly used for record array variables, because MPI-IO requires that the offsets of a file view be monotonically non-decreasing [4]. Because the record array variables are stored interleaved, simply concatenating the individual variable's file views will violate the MPI-IO requirement. For instance, the file view for the first record variable consists of non-contiguous regions that interleaved with all other record array variables.

Recall that in the netCDF terminology the collection of variable values that are stored contiguously for a record variable is called a record. Because these records capture all data from n-1 dimensions of the variable, they can be quite large. Our approach is to write one record for all the array variables as a single I/O operation. We'll call a group of records with the same index across a set of array variables a record collection. As shown in Figure 4(a), the collection of all array variables' first record is located at the beginning of the file's record variable section. It is followed by the collection of all array variables' second record, and so on. We construct a derived datatype for each record collection to set up the file view, and we repeat this process for every record collection accessed by the user. Since the records are organized in the order of array variables defined, the monotonic non-decreasing requirement will be abided. Further, these I/O operations exhibit very good spatial locality, because the data items in each call are located near one another in the netCDF file.

In order to write one record at a time, a process's buffer datatype must be decomposed into smaller buffers the size of a record of that variable. The construction of new buffer datatypes is depicted in Figure 4(b). Again, the absolute memory addresses of write buffers are used to construct the new buffer derived datatypes. The number of new combined buffer datatypes is equal to the number of records. By using MPI derived datatypes for setting the file views and buffer memory layouts, data can be directly written from the user buffers, without allocating intermediate buffers for packing or unpacking the write data.

III. Experiment Results

Our experiments were run on two parallel machines: Mercury at the National Center for Supercomputing Applications and Franklin at Lawrence Berkeley National Laboratory. Mercury is an IA-64 Linux cluster and has 887 nodes, where each node contains two Intel 1.3/1.5 GHz Itanium II processors sharing 4 GB of memory. Mercury is running a SuSE Linux operating system and employs a Myrinet network. The parallel file system on Mercury is an IBM GPFS [15] configured in the Network Shared Disk server model with 54 I/O servers and 512 KB file stripe size. Franklin is a 9660-node SuSE Linux cluster where each compute node consists of a 2.3 GHz single socket, quad-core AMD Opteron processor with a theoretical peak performance of 9.2 GFlop/sec per core. Each compute node has 8 GBytes of memory. The parallel file system is Lustre [16] with 80 I/O servers (OSTs). Lustre allows users to customize the striping configuration of a directory and all new files created in that directory inherit the striping configuration. In our experiment, we configure our directory to store all output files with 1024 KB stripe size, 64 I/O servers, and the start server to be randomly chosen by the file system.

For performance evaluation, we use an I/O benchmark named coll-perf from the ROMIO test suite, modified to use PnetCDF, and the I/O kernel from the FLASH application. The bandwidth numbers were obtained by dividing the aggregate I/O amount by the total run time measured from the beginning of file open until after file close, a conservative value.

A. ROMIO Collective I/O Test

The ROMIO's coll-perf.c has been ported to PnetCDF for testing purpose. Its data partitioning pattern is commonly used by scientific applications. In our modified version, users are provided options to select multiple non-record or record array variables. Each variable is three-dimensional block-distributed array. The partitioning of each variable is accomplished through the assignment of a number of processes on each Cartesian dimension. The three-dimensional array has been split into blocks on each process. An example of data partitioning pattern on 64 processes is illustrated in Figure 5(a). The subarray size in each process is kept constant, independent from the number of processes used, and hence the total I/O amount is proportional to the number of processes. In our experiments, each subarray is a $32 \times 32 \times 32$ integers. We evaluate the reading and writing performance with 10 and 20 array variables separately. The buffer datatypes are all contiguous in memory.

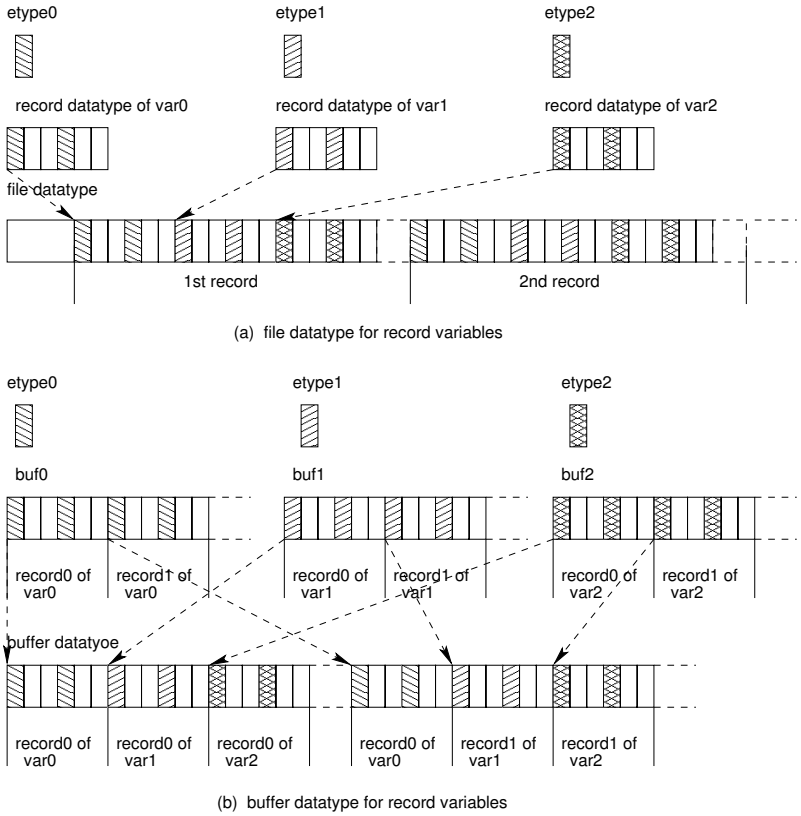

Figure 4. Combining derived datatypes for record array variables

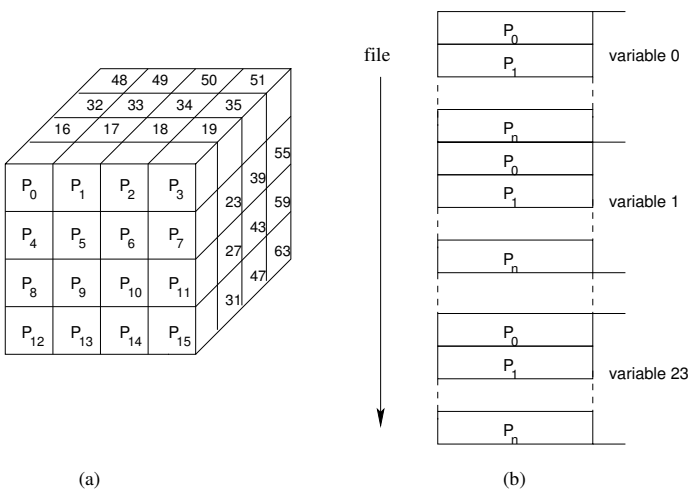

Figure 5. (a) Data partitioning pattern in the ROMIO collective I/O test. A three-dimensional array is partitioned among 64 processes in a block-block-block fashion. (b) I/O pattern of the FLASH I/O benchmark.

Our measurements are performed using from 8 to 512 processes on Mercury and from 8 to 2048 processes on Franklin. Figures 6 and 7 show the performance results of multiple non-record and record array variables with and without our I/O combining scheme. The performance of the I/O combining scheme is better than the per-variable I/O approach. In addition, the more array variables, the wider the performance gap.

The timings for file open and write operations are also broken out in Figures 6 and 7. File open time increases dramatically with the number of processes, and at a moderate number of processes open time consumes a serious fraction of execution time. For example, the open time for writing 10 non-record array variables on Mercury in the 512-process case is 4.191437 second, hiding much of the multi-variable write approach's benefit (0.46285 sec) over the original per-variable write approach (2.083309 sec) and the multi-variable read approach's benefit (0.198982 sec) over the original per-variable read approach (0.295532 sec) . The open time alone causes the large write bandwidth dip on Mercury. Similar degradation also appears on Franklin.

In the ROMIO MPI-IO implementation used on both of these systems, by default only one process from a compute node will be picked as an I/O aggregator. Aggregators are a subset of the MPI processes that act as I/O proxies for the rest of the processes. Hence, half of the MPI processes are I/O aggregators on Mercury, and a quarter of processes are aggregators on Franklin. In the 512-process case on Franklin, the number of aggregators is 128, twice the number of the

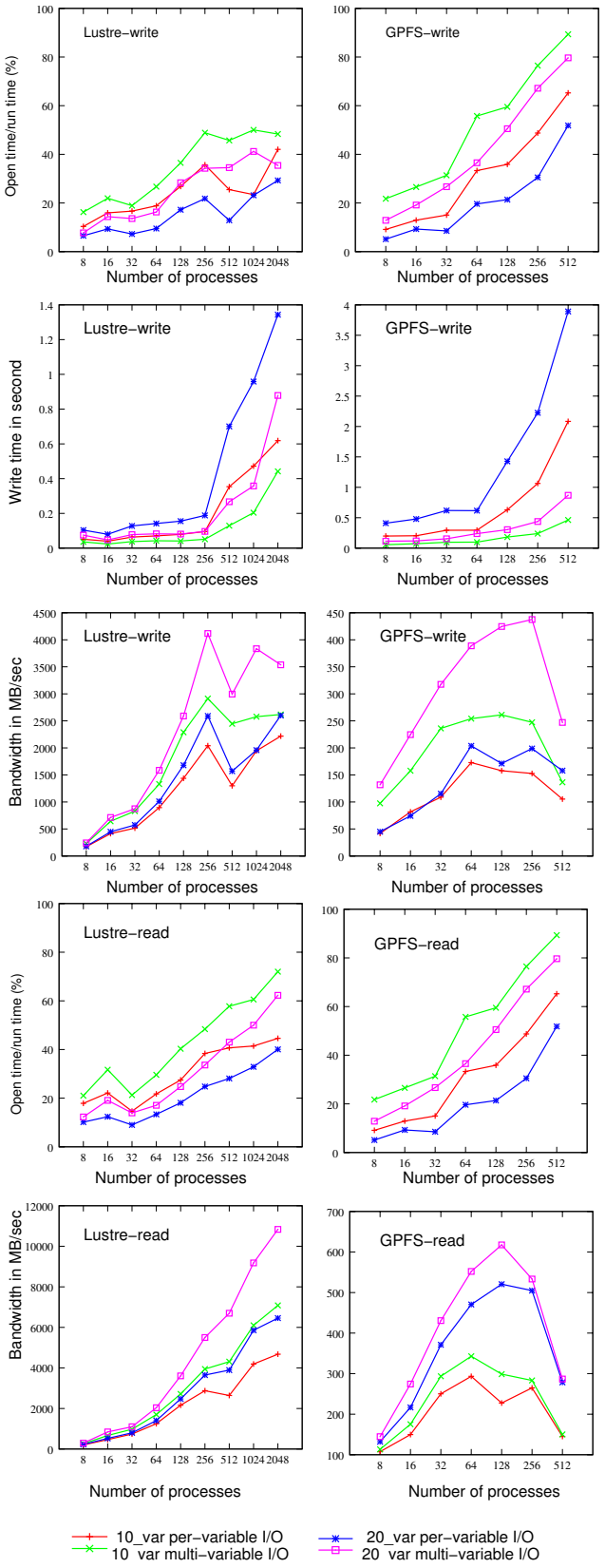

Figure 6. Performance results of ROMIO collective I/O test for non-record array variables

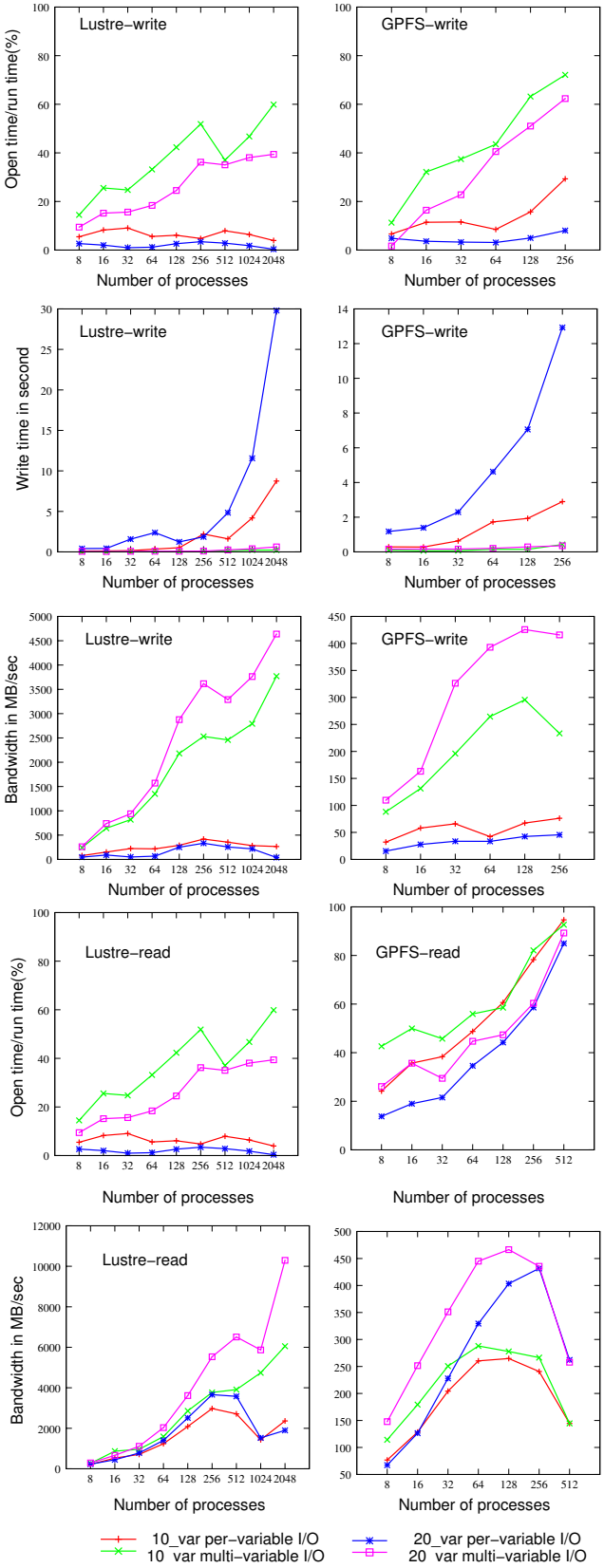

Figure 7. Performance results of ROMIO collective I/O test for record array variables

I/O servers used in our experiments. Contention on the I/O servers slows down the I/O performance on the 512-process case. Its performance becomes even worse than the 256-process case. Similar degradation also appears on Mercury. Previous research has identified MPI-IO optimizations that can improve performance in these scenarios for both of these file systems [17], and our optimizations in the PnetCDF library are complementary to this prior work.

The performance gap is even more significant for record array variables (Figure 7). I/O bandwidths for the per-variable write approach drop as compared to the non-record variable case, while the multi-variable write approach remains about the same bandwidth. As the record array variables are stored interleaved in file, the per-variable write approach of writing data causes many small interleaved accesses, which leads to much worse performance. This characteristic is highlighted by the fact that multi-variable I/O approach results in lower performance for the per-variable write approach on both test systems, in contrast to the higher performance obtained for the multi-variable I/O approach.

As the size of the array variables increases, more data is available to the per-variable write and multi-variable write calls during writes. We would expect the benefit of our optimization to decrease for non-record array variables because of this increase in the per-variable call performance. Table 1 explores the performance of our optimization for larger. In the $32 \times 32 \times 32$ case, the total variable size is 671 MBytes, and in the $50 \times 50 \times 50$ case, the total variable size is 2.56 GBytes. We see that our optimization continues to provide a significant performance increase: 63.53% faster for $32 \times 32 \times 32$ sub-regions and 27.37% faster for $50 \times 50 \times 50$ sub-regions for non-record array variables. The same sitution is appeared as well during reads.

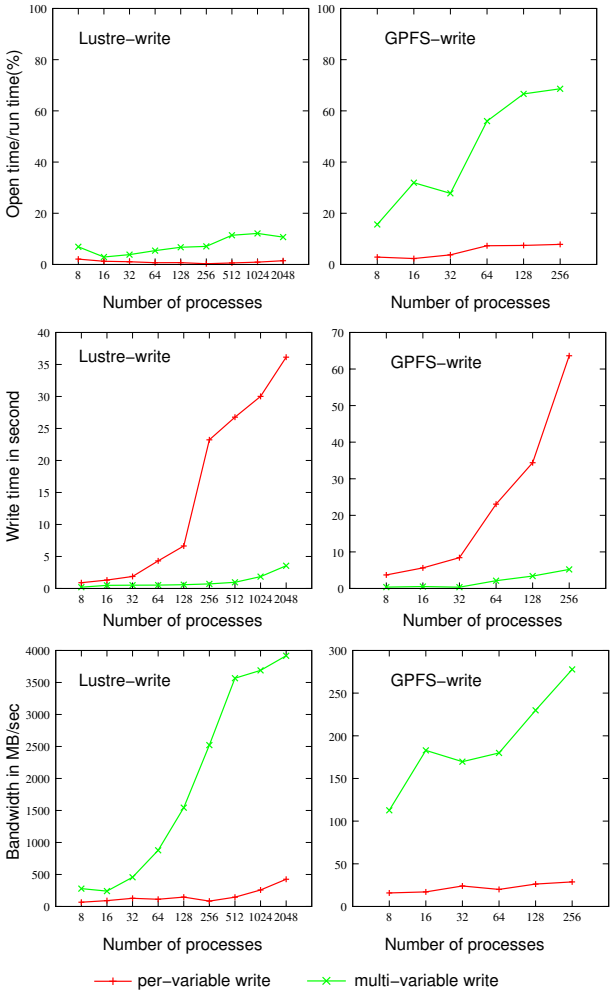

Figure 8. Performance results of FLASH I/O benchmark

B. FLASH I/O Benchmark

The FLASH I/O benchmark suite [18] is the I/O kernel of a block-structured adaptive mesh hydrodynamics code that solves the compressible Euler equations on a block structured adaptive mesh and incorporates the necessary physics to describe the environment, including the equation of state, reaction network, and diffusion [19]. The computational domain is divided into blocks that are distributed across a number of MPI processes. A block is a three-dimensional array with an additional 4 elements as guard cells in each dimension on both sides to hold information from its neighbors. There are 24 data array variables per array element, and about 80 blocks on each MPI process. A variation in block numbers per MPI process is used to generate a slightly unbalanced I/O load. Since the number of blocks is fixed for each process, an increase in the number of processes linearly increases the aggregate I/O amount as well. A PnetCDF version of FLASH I/O is used in our experiments, modified to use asynchronous PnetCDF calls.

FLASH I/O produces a checkpoint file and two visualization files containing centered and corner data. Checkpoint files are the largest of the three output data sets, the I/O time of which dominates the entire benchmark. We set the block size to be $8 \times 8 \times 8$, which produces approximately 8 MB of data per process. Figure 5(b) depicts the I/O pattern of FLASH I/O. There are 24 collective write calls, one for each of the 24 array variables. In each MPI collective write, every MPI process writes a contiguous chunk of a variable, appended to the data written by the previous ranked MPI process.

In our experiments, we only evaluated the write performance for checkpoint files. We define every FLASH variable as a non-record variable PnetCDF. We use implicit approach for the combining write operations. The test program calls asynchronous write functions and adds a call to ncmpi_wait() function before closing the file. Each time a asynchronous collective write function is called for a variable, the internal implementation only stores the

Table I
TIME TO WRITE FOR 512 PROCESSES ON FRANKLIN SYSTEM

size of subarray	multi-variable write(second)		per-variable write(second)		percentage improvement	
	non-record	record	non-record	record	non-record	record
$8 \times 8 \times 8$	0.0408	0.0162	0.3026	0.0844	86.52%	80.80%
$32 \times 32 \times 32$	0.1290	0.1518	0.3537	1.6132	63.53%	90.59%
$50 \times 50 \times 50$	0.5444	0.5377	0.7497	25.004	27.37%	97.84%

request metadata and does no I/O. All procedures for combing parameters and write operations are executed when ncmpi_wait() is called. There is only one collective write call for the checkpoint write. Figure 8 compares the performance results of the FLASH I/O benchmark using the multi-variable write approach and the original per-variable write approach. The multi-variable write approach is significant better than the per-variable approach in all cases on both Franklin and Mercury. Similar performance dips are observed at 512-process case on Franklin. The dip starts at 128-process case on Mercury. This is owing to the increasing contention on the I/O servers.

IV. CONCLUSIONS AND FUTURE WORK

In this work, we propose a new mechanism for PnetCDF to combine multiple I/O operations for better I/O performance. The mechanism uses MPI derived datatypes to specify file views and buffer's memory layout for multiple array variables. It can be used to define a new function that takes arguments for a list of multiple array variables to be written, allowing the application to explicitly access multiple array variables in a single function call. It can also be used in the implementation of asynchronous I/O functions, so that the combination is carried out implicitly, without changing the application source code. Our performance results demonstrate significant improvement for two well-known I/O benchmarks.

In the future, we will extend the approach to allow for writing multiple distinct sub-regions of the same variable with a single call, which will benefit from this approach as well. This type of access is common for applications that do not distribute data in a simple row-block, column-block, or block-block distribution, but rather use some more complex distribution such as a space filling curve. Currently these applications must perform many separate PnetCDF calls to perform I/O, even for a single variable.

ACKNOWLEDGMENT

This work was supported in part by DOE SCIDAC-2: Scientific Data Management Center for Enabling Technologies (CET) grant DE-FC02-07ER25808, DOE FASTOS award number DE-FG02-08ER25848, NSF HECURA CCF-0621443, NSF SDCI OCI-0724599, and NSF ST-HEC CCF-0444405. This work was supported by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Dept. of Energy, under Contract DE-AC02-06CH11357, and in part by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy award DE-FG02-08ER25835. This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. This research was supported in part by the National Science Foundation through TeraGrid resources provided by NCSA, under TeraGrid Projects TG-CCR060017T, TG-CCR080019T, and TGASC080050N.

REFERENCES

[1] R. Rew and G. Davis, "The Unidata netCDF: Softwarefor Scientific Data Access," Sixth International Conference on Interactive Information and Processing Systems for Meteorology, Oceanography and Hydrology, February 1990.

[2] R. Rew, G. Davis, S. Emmerson and H. Davies, "NetCDF User's Guide for C," Unidata Program Center, June 1997. http://www.unidata.ucar.edu/packages/netcdf/guidec/.

[3] J. Li, W. Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher and M. Zingale, "Parallel netCDF: A Scientific High-Performance I/O Interface," In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC)*, pages 39-49, November 2003.

[4] Message Passing Interface Forum, "MPI-2: Extensions to the Message-Passing Interface," 1997, [Online] Available: http://www.mpi-forum.org/docs/docs.html.

[5] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard," 1995. http://www.mpi-forum.org/docs/docs.html.

[6] R. Thakur and A. Choudhary, "An Extended Two-Phase Method for Accessing Sections of Out-of-Core Arrays," Journal of Scientific Programming, 5(4):301, Winter 1996.

[7] R. Thakur, W. Gropp and E. Lusk, "Users Guide for ROMIO: A High-Performance, Portable MPI-IO Implementation," Technical Report ANL/MCS-TM-234, Mathematics and Computer Science Division, Argonne National Laboratory, October 1997.

[8] D. Kotz, "Disk-directed I/O for MIMD Multiprocessors," In *proceedings of the 1994 Symposium on Operating Systems Design and Implementation*, pages 61-74, 1994.

[9] K. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett, "Server-directed collective I/O in Panda," In *Proceedings of Supercomputing '95*, December 1995.

[10] W. K. Liao, K. Coloma, A. Choudhary, L. Ward, E. Russel, and N. Pundit, "Scalable Design and Implementations for MPI Parallel Overlapping I/O," *IEEE Transactions on Parallel and Distributed Systems*, 17 (11) pages1264-1276, 2006.

[11] X. Ma, M. Winslett, J. Lee, and S. Yu, "Improving MPI-IO Output Performance with Active Buffering Plus Threads," In *the International Parallel andDistributed Processing Symposium (IPDPS)*, April 2003.

[12] W. K. Liao, K. Coloma, A. Choudhary, L. Ward, E. Russel, and S. Tideman, "Collective Caching: Application-Aware Client-Side File Caching," In *Proceedings of the 14th International Symposium on High Performance Distributed Computing (HPDC)*, July 2005.

[13] HDF4 Home Page. The National Center for Supercomputing Applications. http://hdf.ncsa.uiuc.edu/hdf4.html.

[14] HDF5 Home Page. The National Center for Supercomputing Applications. http://hdf.ncsa.uiuc.edu/HDF5/.

[15] F. Schmuck and R. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters," In *Proceedings of the File and Storage Technologies (FAST '02)*, pp. 231-244, Jan. 2002.

[16] Cluster File System, Inc. "Lustre: A Scalable, High Performance File System," http://www.Lustre.org/docs.html.

[17] W. K. Liao, A. Choudhary "Dynamically Adapting File Domain Partitioning Methods for Collective I/O Based on Underlying Parallel File System Locking Protocols," In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC)*, pages 313-344, November 2008.

[18] M. Zingale, "FLASH I/O Benchmark Routine - Parallel HDF 5," Mar. 2001, http://flash.uchicago.edu/~zingale/flash_benchmark_io.

[19] B. Fryxell, K. Olson, P. Ricker, F. X. Timmes, M. Zingale, D. Q. Lamb, P. MacNeice, R. Rosner, and H. Tufo, "FLASH: An Adaptive Mesh Hydrodynamics Code for Modelling Astrophysical Thermonuclear Flashes," *Astrophysical Journal Suppliment*, pp. 131-273, 2000.

[20] R. Thakur, W. Gropp and E. Lusk, "Data sieving and collective I/O in ROMIO," In *Proceedings of the Seventh Symposium on the Frontiers of Massively Parallel Computation*, pages 182-189. IEEE Computer Society Press, 1999.

[21] R. Thakur, W. Gropp, and E. Lusk, "A Case for Using MPI's Derived Datatypes to Improve I/O Performance," in *Proc. of SC98: High Performance Networking and Computing*, November 1998.

[22] J. del Rosario, R. Bordawekar and A. Choudhary, "Improved Parallel I/O via a Two-phase Run-time Access Strategy," In *Proceedings of the workshop on I/O in Parallel Computer Systems at IPPS '93* , pages 56-60, April 1993.