Parallel netCDF: A Scientific High-Performance I/O Interface

Jianwei Li Wei-keng Liao Alok Choudhary
ECE Department, Northwestern University
{jianwei, wkliao, choudhar}@ece.northwestern.edu

Robert Ross Rajeev Thakur William Gropp Rob Latham
MCS Division, Argonne National Laboratory
{rross, thakur, gropp, robl}@mcs.anl.gov

Abstract

Dataset storage, exchange, and access play a critical role in scientific applications. For such purposes netCDF serves as a portable and efficient file format and programming interface, which is popular in numerous scientific application domains. However, the original interface does not provide a efficient mechanism for parallel data storage and access.

In this work, we present a new parallel interface for writing and reading netCDF datasets. This interface is derived with minimal changes from the serial netCDF interface but defines semantics for parallel access and is tailored for high performance. The underlying parallel I/O is achieved through MPI-IO, allowing for dramatic performance gains through the use of collective I/O optimizations. We compare the implementation strategies with HDF5 and analyze both. Our tests indicate programming convenience and significant I/O performance improvement with this parallel netCDF interface.

1. Introduction

Scientists have recognized the importance of portable and efficient mechanisms for storing large datasets created and used by their applications. The Network Common Data Form (netCDF) [9, 8] is one such mechanism used by a number of applications.

NetCDF intends to provide a common data access method for atmospheric science applications to deal with a variety of data types that encompass single-point observations, time series, regularly spaced grids, and satellite or radar images [8]. Today several organizations have adopted netCDF as a data access standard [19].

The netCDF design consists of both a portable file format and an easy-to-use application programming interface (API) for storing and retrieving netCDF files across multiple platforms. More and more scientific applications choose netCDF as their output file format. While these applications become computational and data intensive, they tend to be parallelized on high-performance computers. Hence, it is highly desirable to have an efficient parallel programming interface to the netCDF files. Unfortunately, the original design of the netCDF interface is proving inadequate for parallel applications because of its lack of a parallel access mechanism. In particular, there is no support for concurrently writing to a netCDF file. Therefore, parallel applications operating on netCDF files must serialize access. Traditionally, parallel applications write to netCDF files through one of the allocated processes which easily becomes a performance bottleneck. The serial I/O access is both slow and cumbersome to the application programmer.

To facilitate parallel I/O operations, we have defined a parallel API for concurrently accessing netCDF files. With minimal changes to the names and argument lists, this interface maintains the look and feel of the serial netCDF interface while the implementation underneath incorporates well-known parallel I/O techniques such as collective I/O to allow high-performance data access. We implement this work on top of MPI-IO, which is specified by the MPI-2 standard [3, 7, 2] and is freely available on most platforms. MPI has become the de facto parallel mechanism for communication and I/O on most parallel environments, making this approach portable across different platforms.

Hierarchical Data Format version 5 (HDF5) [5] also provides a portable file format and programming interfaces for storing multidimensional arrays together with ancillary data in a single file. It supports parallel I/O and its implementation is also built on top of MPI-IO. Similar to HDF5, our goal on designing the parallel netCDF is to make the programming interface a data access standard for parallel scientific applications and provide more optimization opportu-

nities for I/O performance enhancement.

We ran a set of benchmarks using regular data access patterns commonly seen in scientific applications as well as the ones from a production astrophysics application called FLASH [1]. We study the scalability of our parallel netCDF implementation and compare the performance results between using parallel netCDF and parallel HDF5 in the FLASH I/O benchmark [18]. In this benchmark, our experiments show significant I/O performance improvement when using parallel netCDF.

The rest of this paper is organized as follows. Section 2 reviews some related work. Section 3 presents the design background of netCDF and points out its potential usage in parallel scientific applications. The design and implementation of our parallel netCDF is described in Section 4. Experimental performance results are given in Section 5. Section 6 concludes the paper.

2. Related Work

Considerable research has been done on data access for scientific applications. The work has focused on data I/O performance and data management convenience. Two projects, MPI-IO and HDF, are most closely related to our research.

MPI-IO is a parallel I/O interface specified in the MPI-2 standard. It is implemented and used on a wide range of platforms. The most popular implementation, ROMIO [17] is implemented portably on top of an abstract I/O device layer [14, 16] that enables portability to new underlying I/O systems. One of the most important features in ROMIO is collective I/O operations, which adopt a two-phase I/O strategy [11, 12, 13, 15] and improve the parallel I/O performance by significantly reducing the number of I/O requests that would otherwise result in many small, noncontiguous I/O requests. However, MPI-IO reads and writes data in a raw format without providing any functionality to effectively manage the associated metadata. Nor does it guarantee data portability, thereby making it inconvenient for scientists to organize, transfer, and share their application data.

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 [4] and HDF5 [5]. The design goal of HDF4 is mainly to deal with sequential data access and its API is consistent with its earlier versions. On the other hand, HDF5 is a major revision in which its API is completely re-designed. Both versions store multidimensional arrays together with ancillary data in portable, self-describing file formats. The support for parallel data access in HDF5 is built on top of MPI-IO, which ensures its portability since MPI-IO has become a de facto standard for parallel I/O. However, the fact that HDF5 file format is not compatible with HDF4 can be inconvenient for existing HDF4 programmers to migrate their applications to HDF5. Furthermore, HDF5 adds several new features, such as a hierarchical file structure, that give the programmer more power to describe metadata while making it more difficult for the implementation to optimize parallel data access. And the overhead involved may make HDF5 perform much worse than its underlying MPI-IO. By using a number of scientific applications, this problem is addressed in [6, 10].

3. NetCDF Background

NetCDF is an abstraction that supports a view of data as a collection of self-describing, portable, array-oriented objects that can be accessed through a simple interface. It defines a file format as well as a set of programming interfaces for storing and retrieving data in the form of arrays in netCDF files. We first describe the netCDF file format and its serial API and then consider various approaches to access netCDF files in parallel computing environments.

3.1. File Format

NetCDF stores data in an array-oriented dataset, which contains dimensions, variables, and attributes. Physically, the dataset file is divided into two parts: file header and array data. The header contains all information (or metadata) about dimensions, attributes, and variables except for the variable data itself, while the data part contains arrays of variable values (or raw data).

The netCDF file header first defines a number of dimensions, each with a name and a length. These dimensions are used to define the shapes of variables in the dataset. One dimension can be unlimited and is used as the most significant dimension (record dimension) for growing-size variables.

Following the dimensions, a list of named attributes are used to describe the properties of the dataset (e.g., data range, purpose, associated applications). These are called global attributes and are separate from attributes associated with individual variables.

The basic units of named data in a netCDF dataset are variables, which are multidimensional arrays. The header part describes each variable by its name, shape, named attributes, data type, array size, and data offset, while the data part stores the array values for one variable after another, in their defined order.

To support variable-size arrays (e.g., data growing with time stamps), netCDF introduces record variables and uses a special technique to store such data. All record variables share the same unlimited dimension as their most significant dimension and are expected to grow together along that dimension. The other, less significant dimensions all

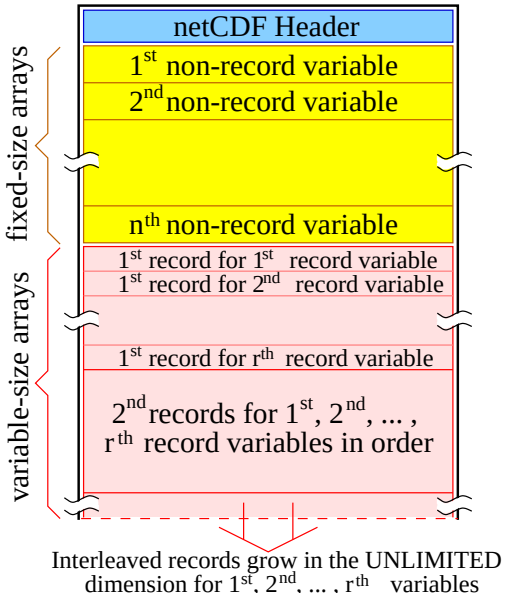


Figure 1. NetCDF file structure: there is a file header containing metadata of the stored arrays, then the fixed-size arrays are laid out in the following contiguous file space in a linear order, with variable-size arrays appending at the end of the file in an interleaved pattern.

together define the shape for one record of the variable. For fixed-size arrays, each array is stored in a contiguous file space starting from a given offset. For variable-size arrays, netCDF first defines a *record* of an array as a subarray comprising all fixed dimensions and the records of all such arrays are stored interleaved in the arrays' defined order. Figure 1 illustrates the storage layouts for fixed-size and variable-size arrays in a netCDF file.

In order to achieve network transparency (machine-independence), both the header and data parts of the file are represented in an well-defined format similar to XDR (eXternal Data Representation) but extended to support efficient storage of arrays of non-byte data.

3.2. Serial NetCDF API

The original netCDF API was designed for serial codes to perform netCDF operations through a single process. In the serial netCDF library, a typical sequence of operations to write a new netCDF dataset is to create the dataset; define the dimensions, variables and attributes; write variable data; and close the dataset. Reading an existing netCDF dataset involves first opening the dataset; inquiring about dimensions, variables, and attributes; reading variable data; and closing the dataset.

These netCDF operations can be divided into the following five categories. Refer to [8] for details of each function in the netCDF library.

(1) **Dataset Functions**: create/open/close a dataset, set the dataset to define/data mode, and synchronize dataset changes to disk

(2) **Define Mode Functions**: define dataset dimensions and variables

(3) **Attribute Functions**: manage adding, changing, and reading attributes of datasets

(4) **Inquiry Functions**: return dataset metadata: dim(id, name, len), var(name, ndims, shape, id)

(5) **Data Access Functions**: provide the ability to read/write variable data in one of the five access methods: single value, whole array, subarray, subsampled array (strided subarray) and mapped strided subarray

The I/O implementation of the serial netCDF API is built on the native I/O system calls and has its own buffering mechanism in user space. Its design and optimization techniques are suitable for serial access but are not efficient or even not possible for parallel access, nor do they allow further performance gains provided by modern parallel I/O techniques.

3.3. Using NetCDF in Parallel Environments

Today most scientific applications are programmed to run in parallel environments due to the increasing requirements on data amount and computational resources. It is highly desirable to develop a set of parallel APIs for accessing netCDF files that employs appropriate parallel I/O techniques. In the meantime, programming convenience is also important, since scientific users may desire to spend minimal effort on dealing with I/O operations. Before presenting our design on parallel netCDF, we would like to discuss current approaches for using netCDF in parallel programs in a message-passing environment.

The first and most straightforward approach is described in the scenario of Figure 2(a) in which one process is in charge of collecting/distributing data and performing I/O to a single netCDF file using the serial netCDF API. The I/O requests from other processes are carried out by shipping all the data through this single process. The drawback of this approach is that collecting all I/O data on a single process can easily cause an I/O performance bottleneck and may overwhelm its memory capacity.

To avoid unnecessary data shipping, an alternative approach is to have all processes perform their I/O independently using the serial netCDF API, as shown in Figure 2(b).

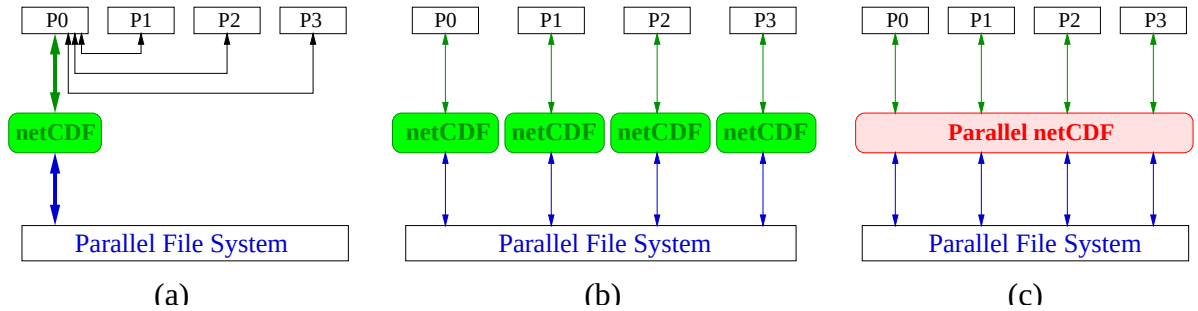


Figure 2. Using netCDF in parallel programs: (a) use serial netCDF API to access single files through a single process; (b) use serial netCDF API to access multiple files concurrently and independently; (c) use new parallel netCDF API to access single files cooperatively or collectively.

In this case, all netCDF operations can proceed concurrently, but over multiple files, one for each process. However, it is more difficult to manage a netCDF dataset when it is spread across multiple files. This approach also violates the netCDF design goal of easy data integration and management.

A third approach introduces a new set of APIs with parallel access semantics and optimized parallel I/O implementation such that all processes perform I/O operations cooperatively or collectively through the parallel netCDF library to access a single netCDF file. This approach, as shown in Figure 2(c), both frees the users from dealing with details of parallel I/O and provides more opportunities for employing various parallel I/O optimizations in order to obtain higher performance. We discuss the details of this parallel netCDF design and implementation in the next section.

4. Parallel NetCDF

To facilitate convenient and high-performance parallel access to netCDF files, we define a new parallel interface and provide a prototype implementation. Since a large number of existing users are running their applications over netCDF, our parallel netCDF design retains the original netCDF file format (version 3) and introduces minimal changes from the original interface. We distinguish the parallel API from the original serial API by prefixing the C function calls with "ncmpi_" and the Fortran function calls with "nfmpi_".

4.1. Interface Design

Our parallel netCDF API is built on top of MPI-IO. The parallel netCDF built on MPI-IO can benefit from several well-known optimizations already used in existing MPI-IO implementations, such as data sieving and two-phase I/O strategies [11, 12, 13, 15] in ROMIO. Figure 3 describes the overall architecture for our design.

In parallel netCDF, a file is opened, operated, and closed by the participating processes in a communication group. In order for these processes to operate on the same file space, especially upon the structural information contained in the file header, a number of changes have been made to the original serial netCDF API.

For the function calls that create/open a netCDF file, an MPI communicator is added in the argument list to define the participating I/O processes within the file's open and close scope. An MPI_Info object is also added to pass user access hints to the MPI-IO for further optimizations. By describing the collection of processes with a communicator, we provide the underlying implementation with information that can be used to ensure file consistency. The MPI_Info hint provides users the ability to deliver the high level access information to netCDF and MPI-IO libraries, such as file access patterns and file system specifics to direct optimization.

We keep the same syntax and semantics for the parallel netCDF define mode functions, attribute functions, and inquiry functions as the original ones. These functions are also made collective to guarantee consistency of dataset structure among the participating processes in the same MPI communication group. For instance, all processes must call the define mode functions with the same values.

The major effort of this work is the parallelization of the data access functions. We provide two sets of data access APIs: a *high-level API* that mimics the serial netCDF data access functions and serves an easy path for original netCDF users to migrate to the parallel interface, and a *flexible API* that provides a more MPI-like style of access. Specifically, the flexible API uses more MPI functionality in order to provide better handling of internal data representations and to more fully expose the capabilities of MPI-IO to the application programmer. The major difference be-

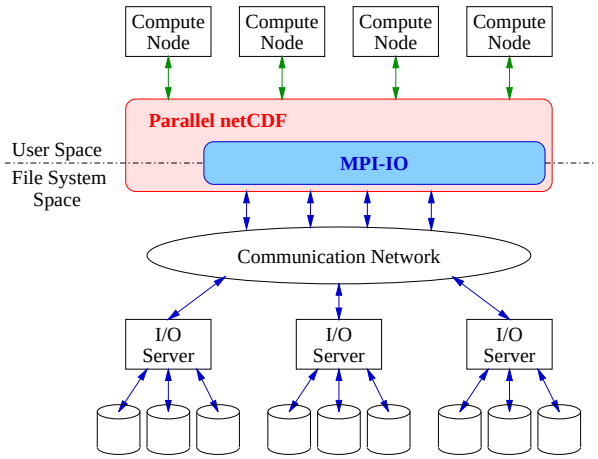


Figure 3. Design of parallel netCDF on a parallel I/O architecture. Parallel netCDF runs as a library between user space and file system space. It processes parallel netCDF requests from user compute nodes and, after optimization, passes the parallel I/O requests down to MPI-IO library, and then the I/O servers receive the MPI-IO requests and perform I/O over the end storage on behalf of the user.

(a) WRITE:

```
1  ncmpi_create(mpi_comm, filename, 0, mpi_info,   &file_id);
2  ncmpi_def_var(file_id, ...);
   ncmpi_enddef(file_id);
3  ncmpi_put_vara_all(file_id, var_id,
                      start[], count[],
                      buffer, bufcount,
                      mpi_datatype);
4  ncmpi_close(file_id);
```

(b) READ:

```
1  ncmpi_open(mpi_comm, filename, 0, mpi_info,   &file_id);
2  ncmpi_inq(file_id, ... );
3  ncmpi_get_vars_all(file_id, var_id,
                      start[], count[], stride[],
                      buffer, bufcount,
                      mpi_datatype);
4  ncmpi_close(file_id);
```

Figure 4. Example of using parallel netCDF. Typically there are 4 main steps: 1. collectively create/open the dataset; 2. collectively define the dataset by adding dimensions, variables and attributes in WRITE, or inquiry about the dataset to get metadata associated with the dataset in READ; 3. access the data arrays (collective or non-collective); 4. collectively close the dataset.

tween the two is the use of MPI derived data types. We believe using MPI derived datatypes can better illustrate the access patterns than the subarray mapping methods used in the original API.

The most important change from the original netCDF interface with respect to data access functions is the split of data mode into two distinct modes: collective and non-collective data modes. To make it obvious the functions involve all processes, collective function names end with "_all". Similar to MPI-IO, the collective functions are synchronous across the processes in the communicator associated to the opened netCDF file, while the non-collective functions are not. Using collective operations can provide the underlying parallel netCDF implementation an opportunity to further optimize access to the netCDF file. These optimizations are performed without further intervention by the application programmer and have been proven to provide dramatic performance improvement in multidimensional dataset access [15]. Figure 4 shows example code of using our parallel netCDF API to write and read a dataset using collective I/O.

4.2. Parallel Implementation

The parallel API implementation is discussed in two parts: header I/O and parallel data I/O. We first describe our implementation strategies for dataset functions, define mode functions, attribute functions, and inquiry functions that access the netCDF file header.

4.2.1. Access to File Header

Internally, the header is read/written only by a single process, although a copy is cached in local memory on each process. The define mode functions, attribute functions, and inquiry functions all work on the local copy of the file header. Since they are all in-memory operations not involved in any file I/O, they bear few changes from the serial netCDF API. They are made collective, but this feature does not necessarily imply inter-process synchronization. In some cases, however, when the header definition is changed synchronization is needed to verify that the values passed in by all processes match. In all possible cases we allow inter-process communications.

The dataset functions, unlike the other functions cited, need to be completely reimplemented because they are in charge of collectively opening/creating datasets, performing header I/O and file synchronization for all processes, and managing inter-process communication. We build these functions over MPI-IO so that they have better portability and provide more optimization opportunities. The basic idea is to let the ROOT process fetch the file header,

broadcast it to all processes when opening a file, and write the file header at the end of define mode if any modification occurs in the header part. Since all define mode and attribute functions are collective and require all processes in the communicator to provide the same arguments when adding/removing/changing definitions, the local copies of the file header shall be the same across all processes once the file is collectively opened and until it is closed.

4.2.2. Parallel I/O for Array Data

Since the majority of time spent accessing a netCDF file is in data access, the data I/O must be efficient. By implementing the data access functions above MPI-IO, we enable a number of advantages and optimizations.

For each of the five data access methods in the flexible data access functions, we represent the data access pattern as an MPI file view (a set of data visible and accessible from an open file [7]), which is constructed from the variable metadata (shape, size, offset, etc.) in the netCDF file header and start[], count[], stride[], imap[], mpi_datatype arguments provided by users. 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.

The high-level data access functions are implemented in terms of the flexible data access functions, so that existing users migrating from serial netCDF can also benefit from the MPI-IO optimizations. However, the flexible data access functions are closer to MPI-IO and hence incur less overhead. They accept a user-specified MPI derived datatype and pass it directly to MPI-IO for optimal handling of in-memory data access patterns.

In some cases (for instance, in record variable access) the data is stored interleaved by record and the contiguity information is lost, so the existing MPI-IO collective I/O optimization may not help. In that case, we need more optimization information from users, 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. This kind of information is passed in as an MPI_Info hint when a user opens or creates a netCDF dataset. We implement our user hints in parallel netCDF for all such specific optimization points, while a number of standard hints are passed down for MPI-IO to take control of optimal parallel I/O behaviors. Thus experienced users have the opportunity to tune their applications for further performance gains.

4.3. Advantages and Disadvantages

There are a number of advantages within the design and implementation of our parallel netCDF, as compared to other related work, like HDF5.

First, the parallel netCDF design and implementation is optimized for the netCDF file format so that the data I/O performance is as good as the underlying MPI-IO implementation. The NetCDF file chooses linear data layout, in which the data arrays are either stored in contiguous space and in a predefined order or interleaved in a regular pattern. This regular and highly predictable data layout enables the parallel netCDF data I/O implementation to simply pass the data buffer, metadata (fileview, mpi_datatype, etc.), and other optimization information to MPI-IO, and all parallel I/O operations are carried out in the same manner as when MPI-IO alone is used. Thus, there is very little overhead, and the parallel netCDF performance should be nearly the same as MPI-IO if only raw data I/O performance is compared. On the other hand, parallel HDF5 uses tree-like file structure that are similar to the UNIX file system and the data is irregularly laid out using super block, header blocks, data blocks, extended header blocks and extended data blocks. This irregular layout pattern may make it difficult to pass user access patterns directly to MPI-IO especially for the case of variable-size arrays. Instead, parallel HDF5 uses dataspace and hyperslabs to define the data organization, map and transfer data between memory space and the file space and does buffer packing/unpacking in a recursive way, while these can otherwise be directly handled by MPI-IO in a more efficient and optimized way.

Secondly, the parallel netCDF implementation manages to keep the overhead involved in header I/O as low as possible. In the netCDF file, there is only one header that contains all necessary information for direct access of each data array and each array is associated with a predefined, numerical ID that can be efficiently inquired when it is needed to access the array. By maintaining a local copy of the header on each process, our implementation saves a lot of interprocess synchronization as well as avoids repeated access of the file header each time the header information is needed to access a single array. All header information can be accessed directly in local memory and inter-process synchronization is needed only during the definition of the dataset. Once the definition of the dataset is created, each array can be identified by its permanent ID and accessed at any time by any process, without any collective open/close operation. On the other hand, in HDF5 the header metadata is dispersed in separate header blocks for each object and, in order to operate on an object, it has to iterate through the entire namespace to get the header information of that object and then open, access and close it. This kind of access method may be inefficient for parallel access, since the

parallel HDF5 designs the open/close of each object as collective operations, which forces all participating processes to communicate when accessing one single object, not to mention the cost of file access to locate and fetch the header information of that object.

Finally, the programming interface of the parallel netCDF is concise and designed for easy usage, and the file format is fully compatible with serial netCDF. Porting existing serial netCDF application to parallel netCDF should be straightforward because the parallel API contains nearly all functions of the serial API with parallel semantics but with minimal change of function names and argument lists.

However, there are also limitations in parallel netCDF. Unlike HDF5, netCDF does not support hierarchical group based organization of data objects. Since it lays out the data in a linear order, adding fixed-size array or extending the file header may be very costly once the file is created and has existing data stored, though moving the existing data to the extended area is performed in parallel. Also, parallel netCDF does not provide functionality to combine two or more files in memory through software mounting, as HDF5 does. Nor does netCDF support data compression within its file format. Fortunately, these features can all be achieved by external software, sacrificing some manageability of the files.

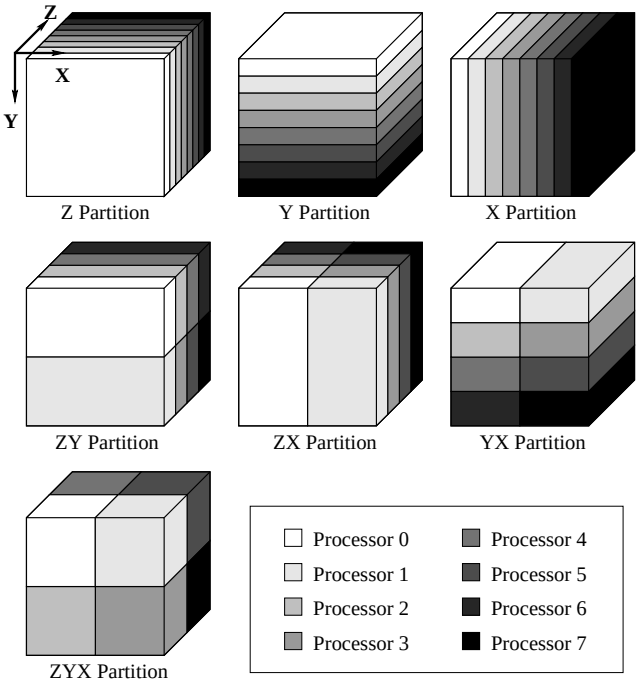


Figure 5. Various 3-D array partitions on 8 processors

5. Performance Evaluation

To evaluate the performance and scalability of our parallel netCDF with that of serial netCDF, we ran some experiments and compared the results. We also compared the performance of parallel netCDF with that of parallel HDF5, using the FLASH I/O benchmark.

The experiments were run on an IBM SP-2 machine. This system is a teraflop-scale clustered SMP with 144 compute nodes. Each compute node has 4 GB of memory shared among its eight 375 MHz Power3 processors. All the compute nodes are interconnected by switches and also connected via switches to the multiple I/O nodes running the GPFS parallel file system. There are 12 I/O nodes, each with dual 222 MHz processes. The aggregate disk space is 5 TB and the peak I/O bandwidth is 1.5 GB/s.

5.1. Scalability Analysis

We wrote a test code (in C) to evaluate the performance of the current implementation of parallel netCDF. This test code was originally developed in Fortran by Woo-sun Yang and Chris Ding at Lawrence Berkeley National Laboratory (LBL). Basically it reads/writes a three-dimensional array field tt(Z,Y,X) from/into a single netCDF file, where Z=level is the most significant dimension and X=longitude is the least significant dimension. The test code partitions the three dimensional array along Z, Y, X, ZY, ZX, YX, and ZYX axes, respectively, as illustrated in Figure 5. All data I/O operations in these tests used collective I/O. For comparison purpose, we prepared the same test using the original serial netCDF API and ran it in serial mode, in which a single processor reads/writes the whole array.

Figure 6 shows the performance results for reading and writing 64 MB and 1 GB netCDF datasets. Generally, the parallel netCDF performance scales with the number of processes. Because of collective I/O optimization, the performance difference made by various access patterns is small, although partitioning in the Z dimension generally performs better than in the X dimension because of the different access contiguity. The overhead involved is inter-process communication, which is negligible compared to the disk I/O when using a large file size. The I/O bandwidth does not scale in direct proportion because the number of I/O nodes (and disks) is fixed so that the dominating disk access time at I/O nodes is almost fixed. As expected, the parallel netCDF outperforms the original serial netCDF as the number of processes increases. The difference between the serial netCDF performance and the parallel netCDF performance with one processor is because of their different I/O implementations and different I/O caching/buffering strategies. In the serial netCDF case, if, as in Figure 2(a), multiprocessors were used and the ROOT processor needed to

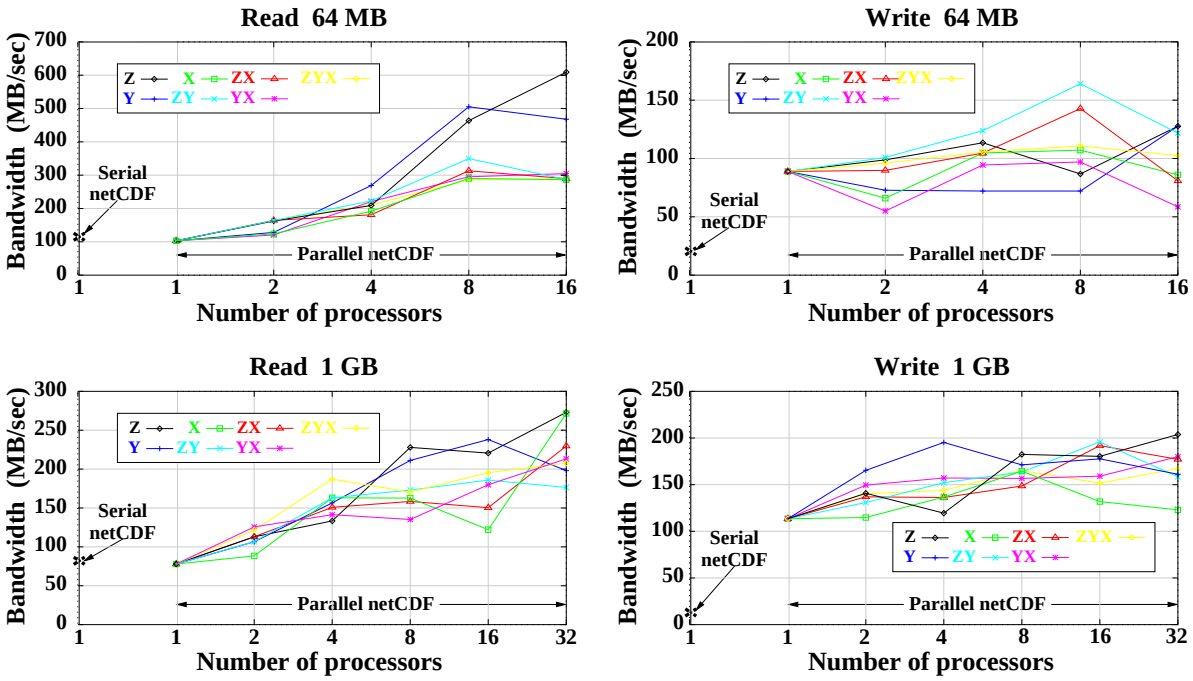


Figure 6. Serial and parallel netCDF performance for 64 MB and 1 GB datasets. The first column of each chart shows the I/O performance of reading/writing the whole array through a single processor using serial netCDF; the rest of the columns show the results using parallel netCDF.

collect partitioned data and then perform the serial netCDF I/O, the performance would be much worse and decrease with the number of processors because of the additional communication cost and division of a single large I/O request into a series of small requests.

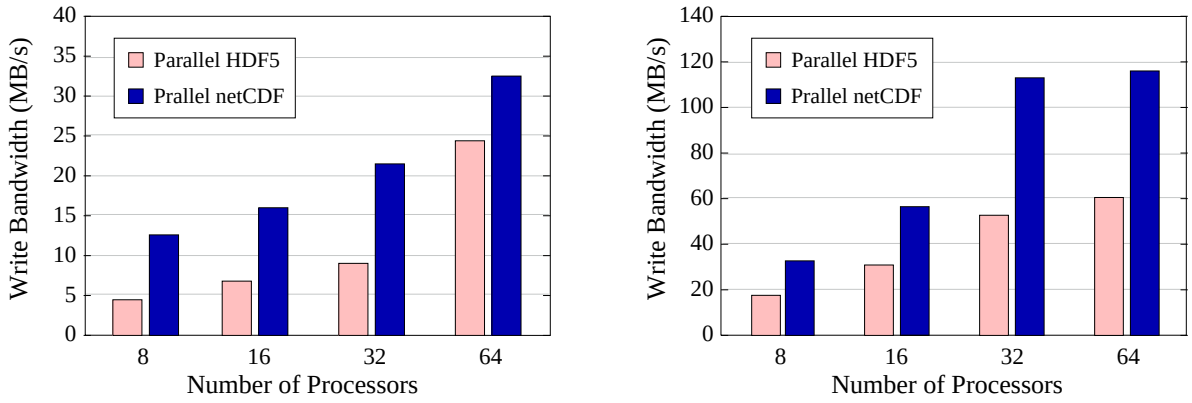
5.2. FLASH I/O Performance

The FLASH I/O benchmark simulates the I/O pattern of an important scientific application called FLASH [1]. It recreates the primary data structures in the FLASH code and produces a checkpoint file, a plotfile with centered data, and a plotfile with corner data, using parallel HDF5. Basically, these three output files contains a series of multidimensional arrays, and the access pattern is simple (Block, *, ...), which is similar to the Z partition in Figure 5. In each of the files, the benchmark writes the related arrays in a fixed order from contiguous user buffers, respectively. The I/O routines in the benchmark are identical to the routines used by FLASH, so any performance improvements made to the benchmark program will be shared by FLASH. In our experiments, in order to focus on the data I/O performance, we modified this benchmark, removed the part of code writing attributes, ported it to parallel netCDF, and observed the effect of our new parallel I/O approach.

Figure 7 shows the performance results of the FLASH I/O benchmark using parallel netCDF and parallel HDF5. We tested both small data size and large data size. The parameters used in these two experiments are: (a) nxb = nyb = nzb = 8, nguard = 4, number of blocks = 80, and nvar = 24; (b) nxb = nyb = nzb = 16, nguard = 8, number of blocks = 80, and nvar = 24. Although both I/O libraries are built above MPI-IO, the parallel netCDF has much less overhead and outperforms parallel HDF5 by almost doubling the overall I/O rate. The extra overhead involved in parallel HDF5 includes inter-process synchronizations and file header access performed internally in parallel open/close of every dataset (analogous to a netCDF variable) and recursive handling of the hyperslab used for parallel access, which makes the packing of the hyperslabs into contiguous buffers take a relatively long time.

6. Conclusion and Future Work

In this work, we extend the serial netCDF interface to facilitate parallel access, and we provide an implementation for a subset of this new parallel netCDF interface. By building on top of MPI-IO, we gain a number of interface advantages and performance optimizations users can benefit from by using this parallel netCDF package, as shown by our test



(a) Small Data Size

(b) Large Data Size

Figure 7. Performance of FLASH I/O benchmark using parallel HDF5 and parallel netCDF. The two experiments use different parameters so that the file sizes are different. Also the file sizes are varies with the number of processors. The I/O amount is 3MB * Number of Processors in (a), and 24MB * Number of Processors in (b).

results. So far, we have released our parallel netCDF library at the website *http://www.mcs.anl.gov/parallel-netcdf/*, and a number of users from LBL, ORNL, and University of Chicago are using our parallel netCDF library.

Future work involves developing a production-quality parallel netCDF API (for C, C++, Fortran, and other programming languages) and making it freely available to the high-performance computing community. Moreover, we need to develop a mechanism for matching the file organization to access patterns, and we need to develop cross-file optimizations for addressing common data access patterns.

Acknowledgements

This work is sponsored by Scientific Data Management Center of DOE SciDAC ISICs program and jointly conducted at Northwestern University and Argonne National Laboratory. This research was also supported in part by NSF cooperative agreement ACI-9619020 through computing resources provided by the National Partnership for Advanced Computational Infrastructure at the San Diego Supercomputer Center.

We thank Woo-Sun Yang from LBL for providing us the test code for performance evaluation and Nagiza F. Samatova and David Bauer at ORNL for using our library and for giving us feedback and valuable suggestions.

References

[1] 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*, 2000, pp. 131-273.

[2] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. "A high-performance, portable implementation of the MPI Message-Passing Interface standard," *Parallel Computing*, 22(6):789-828, 1996.

[3] W. Gropp, E. Lusk, and R. Thakur. *Using MPI-2: Advanced Features of the Message Passing Interface*, MIT Press, Cambridge, MA, 1999.

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

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

[6] J. Li, W. Liao, A. Choudhary, and V. Taylor. "I/O Analysis and Optimization for an AMR Cosmology Application," in *Proceedings of IEEE Cluster 2002*, Chicago, September 2002.

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

[8] 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/*.

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

[10] R. Ross, D. Nurmi, A. Cheng, and M. Zingale, "A Case Study in Application I/O on Linux Clusters", in *Proceedings of SC2001*, Denver, November 2001.

[11] J.M. Rosario, R. Bordawekar, and A. Choudhary. "Improved Parallel I/O via a Two-Phase Run-time Access Strategy," *IPPS '93 Parallel I/O Workshop*, February 9, 1993.

[12] R. Thakur, R. Bordawekar, A. Choudhary, R. Ponnusamy, and T. Singh. "PASSION Runtime Library for Parallel I/O", *Scalable Parallel Libraries Conference*, Oct. 1994.

[13] R. Thakur and A. Choudhary. "An Extended Two-Phase Method for Accessing Sections of Out-of-Core Arrays," *Scientific Programming*, 5(4):301-317, Winter 1996.

[14] R. Thakur, W. Gropp, and E. Lusk. "An Abstract-Device interface for Implementing Portable Parallel-I/O Interfaces"(ADIO), in *Proceedings of the 6th Symposium on the Frontiers of Massively Parallel Computation*, October 1996, pp. 180-187.

[15] R. Thakur, W. Gropp, and E. Lusk. "Data Sieving and Collective I/O in ROMIO," in *Proceeding of the 7th Symposium on the Frontiers of Massively Parallel Computation*, February 1999, pp. 182-189.

[16] R. Thakur, W. Gropp, and E. Lusk. "On implementing MPI-IO portably and with high performance," in *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, May 1999, pp. 23-32.

[17] R. Thakur, R. Ross, E. Lusk, and W. Gropp, "Users Guide for ROMIO: A High-Performance, Portable MPI-IO Implementation," Technical Memorandum No. 234, Mathematics and Computer Science Division, Argonne National Laboratory, Revised January 2002.

[18] M. Zingale. FLASH I/O benchmark. *http://flash.uchicago.edu/~zingale/flash_benchmark_io/*.

[19] Where is NetCDF Used? Unidata Program Center. *http://www.unidata.ucar.edu/packages/netcdf/usage.html*.