Parallel netCDF: A High-Performance Scientific 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 Andrew Siegel
MCS Division, Argonne National Laboratory
{rross, thakur, gropp, robl, siegela}@mcs.anl.gov

Brad Gallagher
ASCI Flash Center
University of Chicago
jbgallag@flash.uchicago.edu

Michael Zingale
UCO/Lick Observatory
University of California, Santa Cruz
zingale@ucolick.org

Abstract

Dataset storage, exchange, and access play a critical role in scientific applications. For such purposes netCDF serves as a portable, efficient file format and programming interface, which is popular in numerous scientific application domains. However, the original interface does not provide an 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 substantial performance gains through the use of collective I/O optimizations. We compare the implementation strategies and performance with HDF5. Our tests indicate programming convenience and significant I/O performance improvement with this parallel netCDF (PnetCDF) interface.

SC'03, November 15-21, 2003, Phoenix, Arizona, USA Copyright 2003 ACM 1-58113-695-1/03/0011...$5.00

1. Introduction

Scientists have recognized the importance of portable and efficient mechanisms for storing large datasets that are created and used by their applications. The Network Common Data Form (netCDF) [10, 9] is one such mechanism used by a number of applications.

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. NetCDF provides applications with a common data access method for storage of structured datasets. Atmospheric science applications, for example, use netCDF to store a variety of data types that encompass single-point observations, time series, regularly spaced grids, and satellite or radar images [9]. Many organizations, including much of the climate community, rely on the netCDF data access standard [21] for data storage.

Unfortunately, the original design of the netCDF interface is proving inadequate for parallel applications because of its lack of a parallel access mechanism. Because there is no support for concurrently writing to a netCDF file, parallel applications writing netCDF files must serialize access. This serialization is usually performed by passing all data to a single process that then writes all data to netCDF files. The serial I/O access is both slow and cumbersome to the application programmer.

To provide the broad community of netCDF users with a high-performance, parallel interface for accessing netCDF files, we have defined an alternative parallel API for concurrently accessing netCDF files. This interface maintains the look and feel of the serial netCDF interface while providing flexibility under the implementation to incorporate 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]. Because MPI has become the de facto parallel mechanism for communication and I/O on most parallel environments, this approach provides portability across most platforms. More important, our use of MPI-IO allows us to benefit from the optimizations built into the MPI-IO implementations, such as data shipping in the IBM implementation [13] and data sieving and two-phase I/O in ROMIO [17], which we would otherwise need to implement ourselves or simply do without.

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

In this paper we describe the design of our parallel netCDF (PnetCDF) interface and discuss preliminary benchmarking results using both a synthetic benchmark accessing a multidimensional dataset and the I/O kernel from the FLASH astrophysics application [20]. This simulation of the FLASH checkpoint and visualization data generation process is now available for both PnetCDF and HDF5.

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. Section 4 describes the design and implementation of our PnetCDF. Section 5 gives experimental performance results. Section 6 concludes the paper with some ideas for future research.

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 [19] is implemented portably on top of an abstract I/O device layer [16, 18] 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 [12, 14, 15, 17] 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]. 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. The support for parallel data access in HDF5 is built on top of MPI-IO, which ensures its portability. This move undoubtedly inconvenienced users of HDF4, but it was a necessary step in providing parallel access semantics. HDF5 also adds several new features, such as a hierarchical file structure, that provide application programmers with a host of options for organizing how data is stored in HDF5 files. Unfortunately this high degree of flexibility can sometimes come at the cost of high performance, as seen in previous studies [6, 11].

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 variables of growing-size.

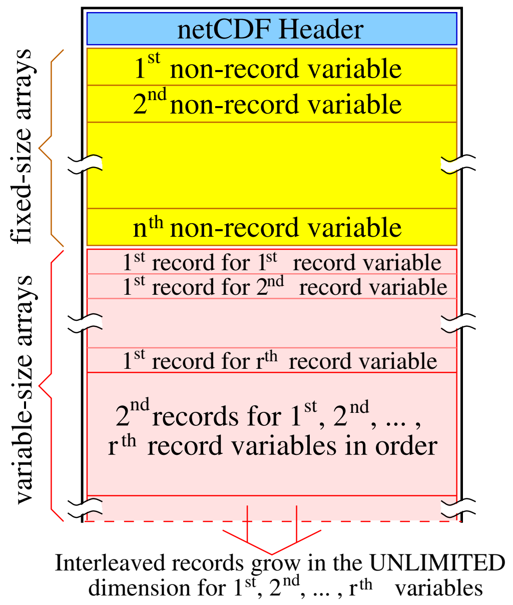
netCDF Header
1st non-record variable
2nd non-record variable
nth non-record variable
1st record for 1st record variable
1st record for 2nd record variable
1st record for rth record variable
2nd records for 1st, 2nd, ... , rth record variables in order
fixed-size arrays
variable-size arrays
Interleaved records grow in the UNLIMITED dimension for 1st, 2nd, ... , rth variables

Figure 1. NetCDF file structure: A file header contains 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-sized arrays appended at the end of the file in an interleaved pattern.

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-sized 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 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-sized arrays, netCDF first defines a *record* of an array as a subarray comprising all fixed dimensions; the records of all such arrays are stored interleaved in the arrays' defined order. Figure 1 illustrates the storage layouts for fixed-sized and variable-sized 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 a well-defined format similar to XDR (eXternal Data Representation) but extended to support efficient storage of arrays of nonbyte 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 [9] for details of each function in the netCDF library.

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

(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(id, name, ndims, shape, datatype), number of dims/vars/attributes, unlimited dimension, etc.

(5) **Data Access Functions**: 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

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 because of the increasing requirements on data amount and computational resources. It

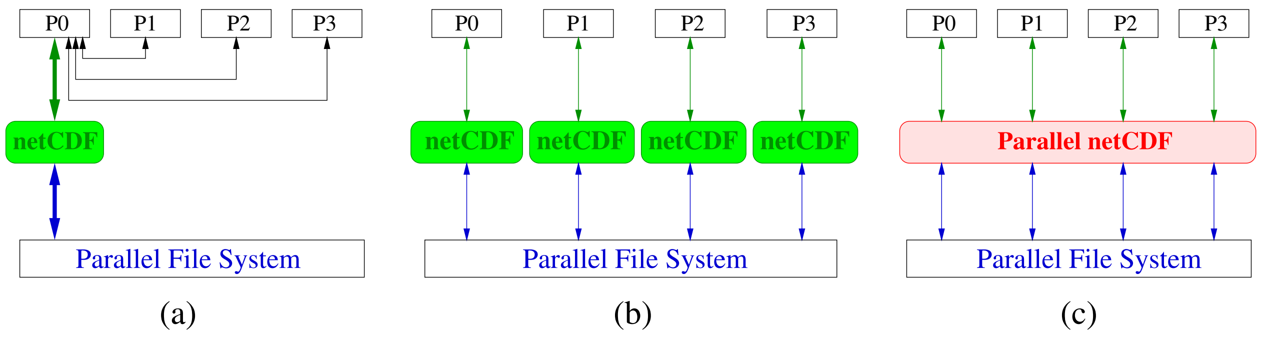

Figure 2. Using netCDF in parallel programs: (a) use serial netCDF API to access single files through a single process; (b) use serial netCDF API to access multiple files concurrently and independently; (c) use new parallel netCDF API to access single files cooperatively or collectively.

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 PnetCDF design, we 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.

In order 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). In this case, all netCDF operations can proceed concurrently, but over multiple files, one for each process. However, managing a netCDF dataset is more difficult 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 PnetCDF API is built on top of MPI-IO, allowing users to benefit from several well-known optimizations already used in existing MPI-IO implementations, such as data sieving and two-phase I/O strategies [12, 14, 15, 17] in ROMIO. Figure 3 describes the overall architecture for our design.

In PnetCDF 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 on 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. By describing the collection of processes with a communicator, we provide the underlying implementation with information that can be used to ensure file consistency during parallel access. An MPI_Info object is also added to pass user access hints to the implementation for further optimizations. Using hints is not mandatory (`MPI_INFO_NULL`

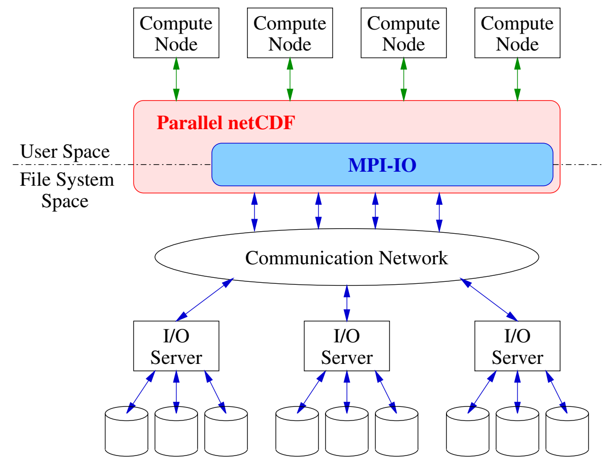

Figure 3. Design of parallel netCDF on a parallel I/O architecture. Parallel netCDF runs as a library between user space and file system space. It processes parallel netCDF requests from user compute nodes and, after optimization, passes the parallel I/O requests down to MPI-IO library, and then the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

can be passed in, indicating no hints). However, hints provide users the ability to deliver the high-level access information to PnetCDF and MPI-IO libraries. Traditional MPI-IO hints tune the MPI-IO implementation to the specific platform and expected low-level access pattern, such as enabling or disabling certain algorithms or adjusting internal buffer sizes and policies. These are passed through the PnetCDF layer to the MPI-IO implementation. PnetCDF hints can be used to describe expected access patterns at the netCDF level of abstraction, in terms of variables and records. These hints can be interpreted by the PnetCDF implementation and either used internally or converted into appropriate MPI-IO hints. For example, given a hint indicating that only a certain small set of variables were going to be read an aggressive PnetCDF implementation might initiate a nonblocking read of those variables at open time so that the values were available locally at read time. For applications that pull a small amount of data from a large number of separate netCDF files, this type of optimization could be a big win, but is only possible with this additional information.

We keep the same syntax and semantics for the PnetCDF 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 to get consistent dataset definitions.

The major effort of this work is the parallelization of the data access functions. We provide two sets of data access APIs. The *high-level API* closely follows the original netCDF data access functions and serves an easy path for original netCDF users to migrate to the parallel interface. These calls take a single pointer for a contiguous region in memory, just as the original netCDF calls, and allow for the description of single elements (`var1`), whole arrays (`vara`), strided arrays (`vars`), and multiple noncontiguous regions (`varm`) in file.

One drawback of the original netCDF interface, and our high-level one, is that only contiguous memory regions may be described to the API. The *flexible API* provides a more MPI-like style of access and relaxes this constraint. Specifically, the flexible API provides the user with the ability to describe noncontiguous regions in memory, which is missing from the original interface. These regions are described using MPI datatypes. For application programmers that are already using MPI for message passing, this approach should be natural. The file regions are still described by using the original parameters. All our high-level data access routines are actually written using this interface.

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 noncollective data modes. In order to make it obvious that the functions involve all processes, collective function names end with "_all". Similar to MPI-IO, the collective functions must be called by all the processes in the communicator associated to the opened netCDF file, while the noncollective functions do not have this constraint. Using collective operations provides the underlying PnetCDF implementation an opportunity to further optimize access to the netCDF file. While users can choose whether to use collective I/O, these optimizations are performed without further intervention by the application programmer and have been proven to provide dramatic performance improvement in multidimensional dataset access [17]. Figure 4 shows example code of using our PnetCDF API to write and read a dataset by using collective I/O.

4.2. Parallel Implementation

Based on our parallel interface design, we provide an implementation for a major subset of this new parallel API. The 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.

(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 PnetCDF. 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 noncollective); 4. collectively close the dataset.

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 interprocess 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.

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 interprocess 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, or changing definitions, the local copies of the file header are guaranteed to 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.

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 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. 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 PnetCDF as extensions to the MPI hint mechanism, while a number of standard hints may still be passed down to MPI-IO to control optimal parallel I/O behaviors at that level. Thus experienced users have the opportunity to tune their applications for further performance gains.

4.3. Advantages and Disadvantages

Our design and implementation of PnetCDF offers a number of advantages, as compared with related work, such as HDF5.

First, the PnetCDF design and implementation are 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 PnetCDF data I/O implementation to simply pass the data buffer, metadata (file view, 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 PnetCDF 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 a tree-like file structure that is similar to the UNIX file system: the data is irregularly laid out using super block, header blocks, data blocks, extended header blocks, and extended data blocks. This is a very flexible system and might have advantages for some applications and access patterns. However, this irregular layout pattern can make it difficult to pass user access patterns directly to MPI-IO, especially for variable-sized 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. MPI-IO is used under this, but this additional overhead can result in significant performance loss.

Second, the PnetCDF implementation manages to keep the overhead involved in header I/O as low as possible. In the netCDF file, only one header 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 interprocess 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 before accessing it. This kind of access method may be inefficient for parallel access, particularly because parallel HDF5 defines the open/close of each object to be a collective operation, which forces all participating processes to communicate when accessing a single object, not to mention the cost of file access to locate and fetch the header information of that object. Further, HDF5 metadata is updated *during* data writes in some cases. Thus additional synchronization is necessary at write time in order to maintain synchronized views of file metadata.

However, PnetCDF also has limitations. Unlike HDF5, netCDF does not support hierarchical group based organization of data objects. Since it lays out the data in a linear order, adding a fixed-sized 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, PnetCDF 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 (although compressed writes must be serialized in HDF5, limiting their usefulness). Fortunately, these features can all be achieved by external software such as netCDF Operators [8], with some sacrifice of manageability of the files.

5. Performance Evaluation

To evaluate the performance and scalability of our PnetCDF with that of serial netCDF, we compared the two with a synthetic benchmark. We also compared the performance of PnetCDF with that of parallel HDF5, using the FLASH I/O benchmark.

Both sets of experiments were run on IBM SP-2 machines. The system on which the serial/parallel comparison was run is a teraflop-scale clustered SMP with 144 compute nodes at the San Diego Supercomputer Center. 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.

The FLASH I/O comparison of HDF5 and PnetCDF was performed on ASCI White Frost, a 68 compute node system with 16 Power3 processors per node. This system is attached to a 2-node I/O system running GPFS. Results in these tests are the average of 20 runs.

Version 0.8.4 of the PnetCDF package was used, as was version 1.4.5-post2 of HDF5. No hints were passed to PnetCDF. HDF5 runs were executed both without hints and with `sieve_buf_size` and `alignment` hints; these have been helpful in some previous runs on this system but did not appear to be useful in this particular case.

5.1. Scalability Analysis

We wrote a test code (in C) to evaluate the performance of the current implementation of PnetCDF. This test code was originally developed in Fortran by Woo-sun Yang and Chris Ding at Lawrence Berkeley National Laboratory. 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

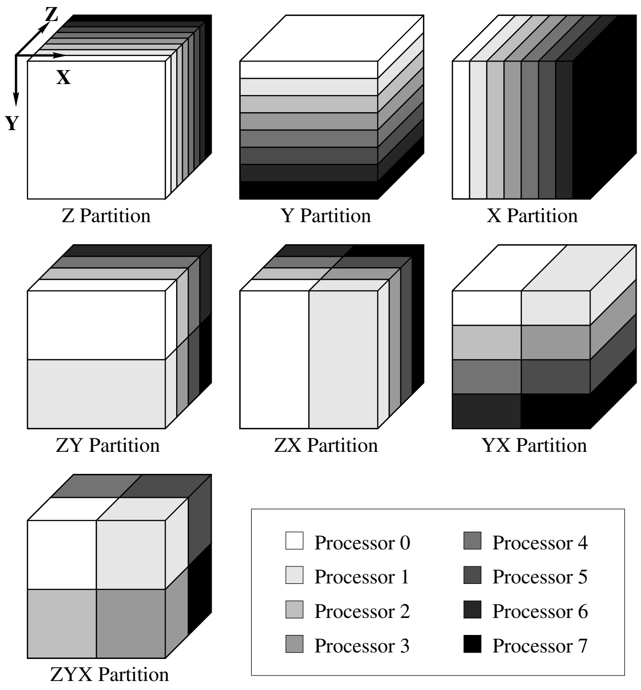

Figure 5. Various 3-D array partitions on 8 processors

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, PnetCDF 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 interprocess communication, which is negligible compared with 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, PnetCDF outperforms the original serial netCDF as the number of processes increases. The difference between serial netCDF performance and PnetCDF 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), multi-processors were used and the root processor needed to 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 code [1] is an adaptive mesh, parallel hydrodynamics code developed to simulate astrophysical thermonuclear flashes in two or three dimensions, such as Type Ia supernovae, Type I X-ray bursts, and classical novae. It 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.

FLASH is primarilty written in Fortran 90 and uses MPI for interprocess communication. The target platforms are the ASCI machines (Frost and Blue Pacific at LLNL, QSC and Red at SNL, and Nirvana at LANL) and Linux clusters (Jazz and Chiba City at ANL and C-plant at SNL). The code scales well up to thousands of processors, has been ported to over a dozen platforms, and has been used for numerous production runs.

The FLASH I/O benchmark simulates the I/O pattern of FLASH [20]. 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. The in-memory data structures are 3D AMR sub-blocks of size 8x8x8 or 16x16x16 with a perimeter of four guard cells that are left out of the data written to file. In the simulation 80 of these blocks are held by each processor. Basically, these three output files contain a series of multi-dimensional 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 PnetCDF, and observed the effect of our new parallel I/O approach.

Figure 7 shows the performance results of the FLASH I/O benchmark using PnetCDF and parallel HDF5. Checkpoint files are the largest of the three output data sets. In the 8x8x8 case each processor outputs approximately 8 MB and in the 16x16x16 case approximately 60 MB. In the plotfile cases processors write approximately 1 MB in the 8x8x8 case and 6 MB in the 16x16x16 cases.

Although both I/O libraries are built above MPI-IO, PnetCDF has much less overhead and outperforms paral-

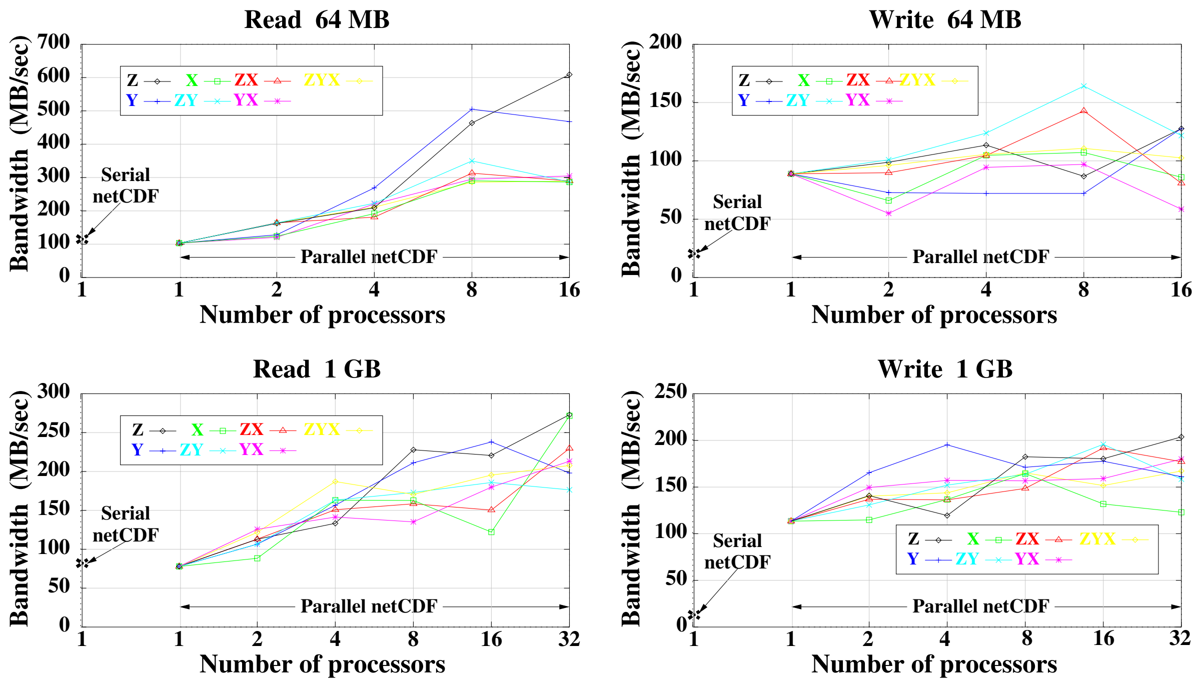

Figure 6. Serial and parallel netCDF performance for 64 MB and 1 GB datasets. The first column of each chart shows the I/O performance of reading/writing the whole array through a single processor using serial netCDF; the rest of the columns show the results using PnetCDF.

lel HDF5 in every case, more than doubling the overall I/O rate in many cases. The extra overhead involved in parallel HDF5 includes interprocess 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. Preliminary test results show that the somewhat simpler netCDF file format coupled with our parallel API combine to provide a very high-performance solution to the problem of portable, structured data storage. So far, we have released our PnetCDF library at the website *http://www.mcs.anl.gov/parallel-netcdf/*, and a number of users from Argonne National Laboratory, Lawrence Berkeley National Laboratory, Oak Ridge National Laboratory, and the University of Chicago are using our PnetCDF library.

Future work involves completing the production-quality parallel netCDF API (for C, C++, Fortran, and other programming languages) and making it freely available to the high-performance computing community. Testing on alternative platforms and with additional benchmarks is also ongoing. In particular we are interested in seeing how read performance compares between PnetCDF and HDF5; perhaps without the additional synchronization of writes the performance is more comparable. We also 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.

Acknowledgments

This work was supported in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, SciDAC Program, U.S. Department of Energy, under Contract W-31-109-ENG-38. Also it was 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.

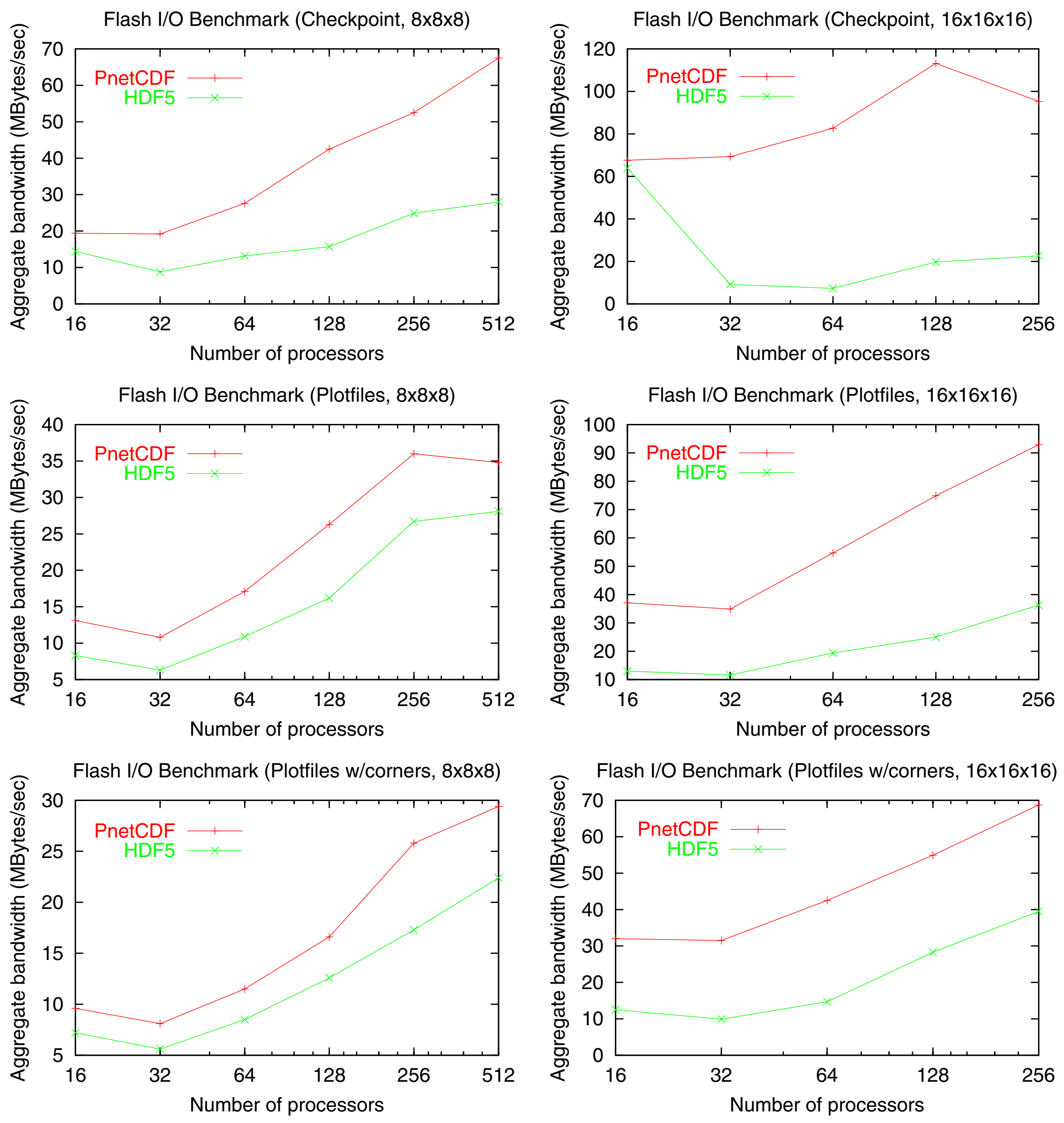

Figure 7. Performance of the FLASH I/O benchmark on the ASCI White Frost platform.

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] C. Zender. The NetCDF Operators (NCO). *http:// nco.sourceforge.net/.*

[9] 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/.*

[10] 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.

[11] 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.

[12] 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.

[13] F. Schmuck and R. Haskin. "GPFS: A Shared-Disk File System for Large Computing Clusters," in *Proceedings of FAST'02*, January 2002.

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

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

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