Using Subfiling to Improve Programming Flexibility and Performance of Parallel Shared-file I/O

Kui Gao, Wei-keng Liao, Arifa Nisar, Alok Choudhary
Electrical Engineering and Computer Science Department
Northwestern University
Evanston, IL 60208, USA
kgao, wkliao, ani662, 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*—There are two popular parallel I/O programming styles used by modern scientific computational applications: unique-file and shared-file. Unique-file I/O usually gives satisfactory performance, but its major drawback is that managing a large number of files can overwhelm the task of post-simulation data processing. Shared-file I/O produces fewer files and allows arrays partitioned among processes to be saved in the canonical order. As the number of processors on modern parallel machines increases into thousands and more, the problem size and in turn the global array size also increase proportionally. It is not practical to manage files of size each larger than a few hundreds of GB. Hence, to seek a middle ground between these two I/O styles, we propose a subfiling scheme that divides a large multi-dimensional global array into smaller subarrays, each saved in a smaller file, named subfile. Subfiling is implemented on top of MPI-IO. We also incorporate it into the parallel netCDF library in order to preserve the partitioning information in the netCDF file header, so that the global array can later be reconstructed. In addition, since the subfiling scheme decreases the number of processes sharing a file, it can reduce the overhead of file system's data consistency control. Our experimental results with several I/O benchmarks show that subfiling can provide improved I/O performance.**

***Keywords*-Parallel netCDF; MPI-IO; subfiling;**

I. Introduction

Modern parallel computers are increasingly used to solve large, data-intensive applications, such as climate modeling, fusion, fluid dynamics, and computational biology. The amount of produced data can be very large and requires effective I/O libraries and fast storage system [1]. There are two I/O programming styles commonly used by today's parallel applications. One is termed as unique I/O, also known as unique-file-per-process style, in which each process accesses files that are unique to the process. This approach may create a management nightmare for file systems when a parallel job running on thousands of processes produces hundreds of thousands or millions of files. Furthermore, accessing millions of files can be a daunting task for post-run data analysis. To avoid these problems, one solution is to adopt the shared-file I/O programming style. Shared-file I/O refers to a single file being accessed by multiple processes concurrently. It can be used to save global partitioned data structures in the canonical order and is often achieved by using MPI-IO [2]. Shared-file I/O performance often suffers from significant file system overheads due to conflicted I/O requests among processes [3], [4]. Such overheads include the system's data consistency control for cache coherency and I/O atomicity. As the number of concurrent accesses to a shared file increases, the overhead can increase dramatically.

Furthermore, shared file I/O can result in files too large for some systems to archive. The High Performance Storage System (HPSS) [5] is a modern, flexible, performance-oriented mass storage system. It has been used for archival storage and supports several hierarchies with different service characteristics, such as access time, maximum file size, number of data copies, and data transfer rate. For example, the default max file size is 10GB in IU's TeraGrid-accessible HPSS archival storage system [6]. If the amount of scientific data generated from one application is more than the default file size, the scientific data must be split into more than one file. In order to seek a middle ground between the two I/O programming styles, we propose a subfiling scheme that allows a large multi-dimensional global array to be split into a number of smaller subarrays, each saved in a separate file. The subfiling scheme reduces the file system control overhead by decreasing the number of processes concurrently accessing a shared file. We develop the subfiling scheme as a standalone I/O library using MPI-IO.

The Network Common Data Form (netCDF) [7] provides a set of software libraries and machine-independent data formats that support the creation, access, and sharing of array-oriented scientific data and is widely used by many scientific applications. The original netCDF API is designed for serial data access and does not provide an efficient mechanism for parallel data storage and access. Parallel netCDF (PnetCDF) [8] provides a parallel API to access netCDF files with significantly better performance on top of MPI-IO and is popular used by many scientific applications to save metadata along with data, and provides high-performance parallel I/O. In order to preserve the array partitioning information by the subfiling, we also incorporated it in parallel netCDF

0190-3918/09 $26.00 © 2009 IEEE
DOI 10.1109/ICPP.2009.68

(PnetCDF). By saving the subfiling information in the netCDF files, the structure of global arrays can be portably reconstructed. In the standalone library, the data partitioning of a global array among subfiles is user customizable. For instance, a 3D array can be partitioned along one of the three axes. For PnetCDF, the data partitioning is always along the most significant axis and the only customizable parameter is the number of subfiles. Through the subfiling application programming interfaces (APIs), the partitioned global array still appears to users as a single netCDF array and access to it is kept the same as without subfiling. In order to achieve this goal, the subfiling implementation automatically generates the MPI fileviews for each subfile for a given I/O request. Mapping information of a global array to subfiles is stored as netCDF attributes which are duplicated in all subfiles. Each netCDF subfile is self-contained with sufficient information to describe the layout of local array mapped to the global array, including the names of all subfiles and partitioning of each subfile. Therefore, accessing the metadata from one of the subfiles is enough to understand the subfiling structure of a global array.

We evaluate the subfiling scheme using the ROMIO benchmark for collective I/O and two application I/O kernels, FLASH-IO and S3D-IO. We compare the approaches of unique I/O and single shared-file I/O on two parallel file systems, Lustre and GPFS. Our measurements are performed using from 16 to 512 processes. In most of the cases, subfiling performs as expected in between the two I/O approaches. Subfiling can even outperform both approaches when the number of processes becomes large, causing significant increase in the file open cost. For example, up to 239.589% improvement relative to one shared-file I/O was observed on Lustre and 97.654% on Mercury in the 512-process case of ROMIO benchmark. The rest of the paper is organized as follows. Section 2 discusses background and related work. The design and implementation of the subfiling scheme is presented in Section 3. The implementation for parallel netCDF is discussed in Section 4. Section 5 gives the performance results and analysis. The paper is concluded in Section 6.

II. Background and Related Work

Programming for unique-file I/O usually uses POSIX I/O APIs [9] as illustrated in Figure 1(a). The POSIX standard also defines a set of I/O consistency semantics that have been abided by many modern file systems, such as Lustre [10], IBM GPFS [11], Sun ZFS [12], and SGI XFS [13]. Unique-file I/O can also be programmed using MPI-IO, which is done by using the MPI_COMM_SELF communicator in the file open. However, the design goal of MPI-IO is mainly for parallel I/O operations on shared files.

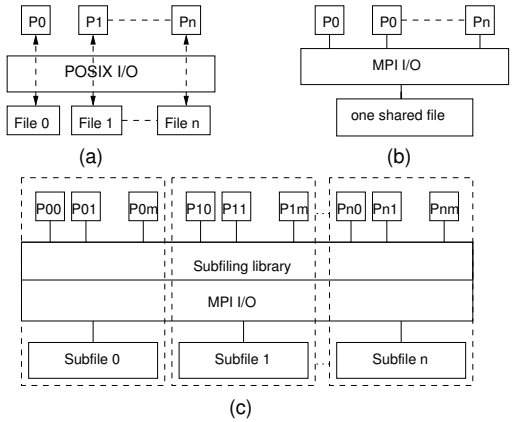

Fig. 1 I/O accessing Pattern Style

Figure 1. I/O accessing Pattern Style

A. MPI-IO

MPI-IO inherits two important MPI features: MPI communicators defining a set of processes for performing group operations and MPI derived data types for describing non-contiguous memory layouts [2]. A communicator specifies the processes that can participate in a collective operation for both inter-process communication and file I/O. When opening a file, the MPI communicator is a required argument to indicate the group of processes sharing the file. MPI collective I/O operations require all the processes in the communicator to participate, as shown in Figure 1(b). Independent I/O functions, in contrast, requiring no synchronization. MPI derived data types are also used to describe non-contiguous layouts in file space. A process can specify the visible file ranges by setting its fileview through a derived data type.

B. PnetCDF

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 supports the creation, access, and sharing of array-oriented scientific data [7]. NetCDF stores metadata in the file header that describes the structures of the arrays and their layout in the file. Additional information, such as user annotations, can also be saved as attributes. Parallel netCDF [8] is developed to support parallel I/O operations and access files exceeding 4GB in size. The PnetCDF library is built on the top of MPI-IO for portability. Its I/O functions take an additional argument of an MPI communicator to indicate the processes participating in the shared-file I/O operations. PnetCDF constructs MPI derived data types to define a process's file view based on its request to a subarray. MPI file hints supplied by the users are passed to the underlying MPI-IO library, so that PnetCDF can take advantage of the I/O optimizations available in MPI-IO.

There are many optimizations to improve shared-file I/O performance, including two-phase I/O [1], [14], disk-directed I/O [16], server-directed I/O [17], persistent file domain [18], active buffering [19], and collaborative caching [20]. The two-phase I/O has been adopted by ROMIO, a popular MPI-IO implementation developed at Argonne National Laboratory [15]. However, even with these improvements, the shared-file I/O performance is still far from the unique-file I/O. A major obstacle comes from file system locking overhead due to the data consistency control. Such overhead does not exist if a file is only accessed by a unique process. Therefore, reducing the number of processes sharing a file can alleviate lock conflict and improve the performance.

```
int  subfile_to_global_view(int ndims, int gsizes[],
                    MPI_Datatype etype, int order,
                      int num_subfiles, subfile_struct *sfp);
int local_to_global_view(int ndims, int starts[],
                    int lengths[], subfile_struct *sfp);
int subfile_open(char *filename, int cmode,
                    int subfile_struct *sfp);
int subfile_write(subfile_struct *sfp, int count,
                    MPI_Datatype etype, void *buf);
int subfile_read(subfile_struct*sfp, int count,
                    MPI_Datatype etype, void *buf);
int subfile_close(subfile_struct *sf);
```

Figure 2. API of subfiling scheme on top of MPI-IO

III. Design and implementation

The subfiling scheme defines a set of APIs, as shown in Figure 2, for users to customize the partitioning of an array among subfiles. The implementation of these APIs is built on top of MPI-IO, as depicted in Figure 1(c).

A. Subfiling APIs

A data structure named subfile_struct is created to store the information about the subfiling of an array as shown in Figure 3. Used as an argument in the APIs, its contents are filled with the dimensionalities of the array, dimensionalities of subarrays in subfiles, and the partitioning order. The subfile partitioning order specifies the dimensions of a multi-dimension array that are partitioned among subfiles. The partitioning can be done along one or more axes. The partitioning order argument is an integer variable whose values of 1 or 0 in the bit locations indicate if the axis is partitioned or not. For instance, the integer value of 6 (110 in binary bit form) means the partitioning order is along Z and Y axes. Two mapping functions are used to specify the I/O buffer's layout in the subfiles. One is a local-to-global array mapping, which is the same as defining MPI-IO file view. The other defines the mapping from the subarrays in subfiles to the global array, which describes how the global array is split into subfiles. Through these two functions, we can construct a process's file views to each subfile. Given a generic file name in function subfile_open(), the subfile are named by appending with a unique number. A subfile is only opened by the processes that have valid file view to it. Depending on the partitioning, a process may open more than one subfile.

```
typedef struct {
    int ndims;
    int sf_num;
    MPI_Comm sf_comm;
    int *global_sizes;
    int *global_starts;
    int *subsizes;
    int *sfsizes;
    char *filename;
    MPI_File sf_fh;
    MPI_Datatype sf_type;
    MPI_Datatype buf_type;
    subfile_struct *next;
} subfile_struct;
```

Figure 3. The subfile_struct structure

B. Implementation

Two additional pieces of subfiling metadata stored in the subfile_struct object (as shown in Figure 3.) are MPI file handlers and communicators. Given the global array dimensionality, number of subfiles, and the partitioning order, we first calculate the dimensionalities of the subarrays for each subfile. Conceptually, a subfile is like an MPI process which has a file view mapping from its subarray to the global array. This mapping information is saved in a text file that later can be used to reconstruct the global array. The subarray dimensionalities are also used to calculate a process's file view to each subfile. In other words, a process's local-to-global array file view is decomposed into several local-to-subarray file views, each corresponding to a subfile. Meanwhile, the derived data type describing memory layout of a process's I/O buffer is also decomposed in to several ones, each for a subfile. Once the local-to-subarray mapping is known, the MPI communicators can be created from the group of processes sharing the same subfiles. This approach avoids the processes with zero-length file view for a subfile to open it. Otherwise, if a subfile is opened by all processes, those processes with zero-length file view would wait idly for other process to complete a collective I/O, as MPI collective I/O is synchronous. Therefore, our implementation improves the process efficiency by ensuring a file is only opened by the processes with valid accesses to it.

In function subfile_open(), each process calls MPI_File_open() for the subfiles which have a non-zero length file view, followed by setting the file view through MPI_File_set_view() using the subfile view calculated earlier. Note that one process may open more than one file, if it accesses data across the subfile partitioning boundaries. Similarly, in subfile_write() or subfile_read(), each process calls MPI_File_write_all() or MPI_File_read_all() for to access data in the subfiles. At the end, subfile_close() closes all subfiles. Figure 4 shows an example for the file views

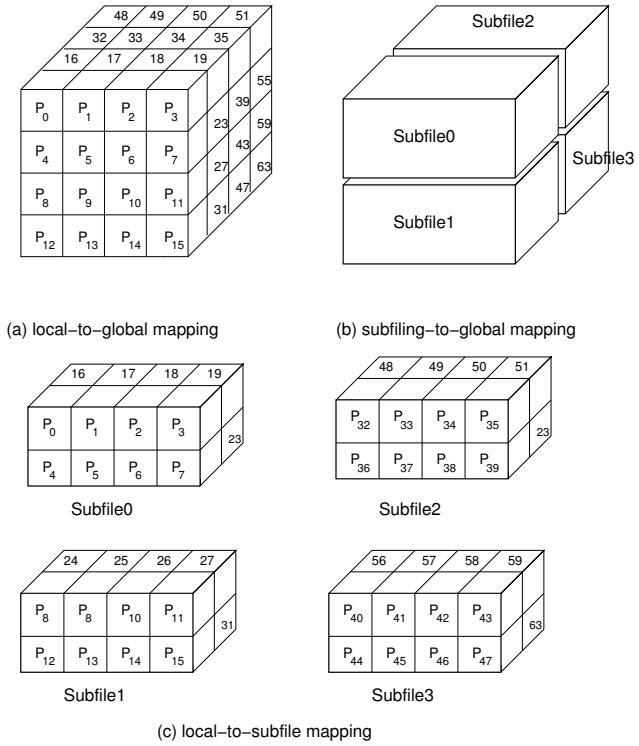

Figure 4. Data partitioning and Mapping

and array mappings used in the subfiling. A 3-D array partitioned among 64 processes and split into 4 subfiles. The mapping of local subarrays to global array is illustrated in Figure 4(a). The global array is partitioned along both Y and Z axes into four subfile datasets, as shown as Figure 4(b). The subfile views are given in Figure 4(c). A new communicator is created for each subfile, which includes only the processes that have an accessible view in the file. Each subfile is opened collectively by the processes in this communicator. Subfiling is in fact a superset of the unique-file and shared-file I/O styles. It can behave like the unique-file I/O if the number of subfiles is equal to the number of application processes. Subfiling is equivalent to the shared-file I/O, when the number of subfiles is set to one.

IV. Incorporating Subfiling into PnetCDF

The stand-alone subfiling library saves the subfiling information in a separate text file, which may not be a favorable strategy. A better choice is to save the metadata together with array data in a self-describing file format, such as netCDF. As parallel netCDF supports parallel I/O, it has been adopted by many scientific applications. Therefore, we augment the PnetCDF functionalities by incorporating the subfiling scheme. The software architecture is shown in Figure 5.

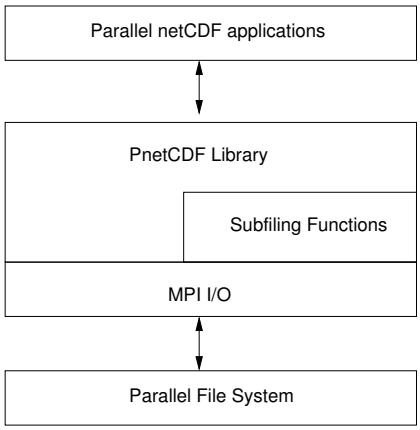

Figure 5. The subfiling component in the pnetCDF software architecture

A. Subfiling Attributes

We define the subfiling metadata as netCDF attributes to the array variable that is partitioned by the subfiling scheme. The names of these attributes have the common prefix "subfiling_". They include subfiling_enabled (a flag indicating if subfiled), subfiling_gndims (number of dimensions in the global array), subfiling_gsize (global array size), subfiling_nfiles (number of subfiles), and subfiling_porder (partitioning order). There are also attributes describing the subarrays in each subfile, such as subfile name, subarray dimensions, and subarray size. These attributes are duplicated in all subfiles so that reading one subfile is enough to obtain the information about the partitioning of a global array among the subfiles.

The subfiling feature is completely hidden in PnetCDF without adding new APIs and is enabled through two MPI hints: subfiling_enabled and subfiling_nfiles. Due to the netCDF format and programming characteristics, the array partitioning cannot be as flexible as the stand-alone subfiling library.

B. Implementation

Since arrays saved in a netCDF file must be organized in the canonical order, we limit our subfile partitioning in PnetCDF to along only the array's most significant dimension. The only user controllable parameter is the number of subfiles. To allow subfiling to be enabled on a per variable basis, we add an MPI hint as a new argument of type MPI_Info to the variable define API, ncmpi_def_var(). When defining variables, users can enable subfiling for a variable and disable it for another. Although PnetCDF file create function also takes an MPI hint argument, since subfiling is per variable basis, enabling subfiling at file create time has no effect. The file name supplied by the user is used as a base name for the subfiles. We refer this file as the base file. A subfile name appends the variable name and subfile number to the based name. The base file contains

all variables that are not subfiled. The subfiling metadata described in Section 2 are saved in the base file as global attributes and in each subfile.

When using the stand-alone subfiling library, a process's file view must be defined before opening the subfiles. The library uses the mappings of local-to-global array and subfile-to-global array to determine the group of processes for each subfile. However, in PnetCDF, a process's file view can only be known when it calls PnetCDF write/read functions. Therefore, in our implementation, the subfiling partitioning, mapping, MPI communicator creation, and file opening are all carried out in the write/read functions. All subfiles are closed before the function returns.

During read operations, the root process retrieves the subfiling metadata from the base file and broadcasts to all other processes. Each process uses this information to construct its file view for each subfiles. MPI communicators are also created based on the same information. Subfiling implementations are transparent to users.

An example program that enables subfiling for one variable is given in Figure 6. Lines 1 and 2 create an MPI hint. Line 3 creates the base netCDF file. Line 4 defines a variable and uses the MPI hint to enable subfiling for this variable. Line 5 declares the end of variable/attribute define mode. Line 6 writes the variable and Line 7 closes the base file.

```
1. MPI_Info_set(sf_into, "subfiling", "enable");
2. MPI_Info_set(sf_info, "subfiling_nfiles", "4");
3. ncmpi_create(comm, filename, mode,
                null_info, ncid);
4. ncmpi_def_var(ncid, type, ndims, dimids, varid, sf_info);
5. ncmpi_enddef(ncid);
6. ncmpi_put_vara_all(ncid, varid, starts, counts, buf);
7. ncmpi_close(ncid).
```

Figure 6. Example of using subfiling scheme in PnetCDF

V. Experimental Results

The performance results of the subfiling scheme were collected from two parallel machines: Mercury at the National Center for Supercomputing Applications (NCSA) and Franklin at Lawrence Berkeley National Laboratory. Mercury is an IA-64 Linux cluster with 887 nodes where each node contains two Intel 1.3/1.5 GHz Itanium II processors sharing 4 GB of memory. Mercury is using Myrinet interconnected network and the GPFS parallel file system. On this system, GPFS has 54 I/O servers. Franklin is a 9660-node SuSE Linux cluster where each node contains two 2.6 GHz dual-core AMD Opteron processor with a theoretical peak performance of 5.2 GFlop/sec. Each compute node has 4 GBytes of memory. The parallel file system on Franklin is Lustre [10] with 80 I/O servers in total.

We evaluate and compare the performance of subfiling with the unique-file and shared-file I/O. We first define β = nproc/nsubfile as the parameter to indicate the ratio of number of processes sharing a subfile. For performance evaluation, we use three benchmarks: a three-dimensional block I/O test named coll_perf from the PnetCDF test suite and two application I/O kernels, FLASH-IO and S3D-IO. We measured the performance of all benchmarks by timing the MPI-IO open, write, and close functions separately. The I/O 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. PnetCDF Collective I/O Test

Based on the collective I/O test program from ROMIO test suite, PnetCDF test suite contains a program named coll_perf.c using a three-dimensional block-partitioning pattern. We chose this test to report boh the write bandwidths for unique-file, one shared-file, and subfiling I/O methods. The partitioning of data is done through the assignment of a number of processes on each Cartesian dimension. In our experiments, we set the subarray size in each process to 256 x 256 x 256 4-byte integers.

Figure 7 and Figure 8 show the results for using the unique-file I/O, one shared-file I/O, and subfiling method with β=8 and β=16. The subarry in each subfile is of size 512x512x512 and 512x512x1024, when β=8 and 16, respectively. On Lustre, the two subfiling methods perform as expected in between the unique-file and shared-file methods. However, on GPFS, the two subfilings outperform the other two I/O methods when the number of processes is very large. The unique-file I/O has a significantly higher open cost than the subfiling on GPFS. The performance improvement is attributed to the reduced file system control overhead due to the decreased number of processes concurrently accessing a subfile.

B. FLASH I/O Benchmark

The FLASH I/O benchmark suite [22] 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 [23]. 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 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. We use the PnetCDF version of FLASH I/O in our experiments.

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 16x16x16, which produces approximately 64 MB of data per process. There are 24 collective write calls, one for each of the 24 variables. In each write operation, 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 performance for writing the checkpoint file. Figure 9 shows the results of unique-file I/O, one-shared-file I/O, subfiling method with β=8 and β=16. The two subfiling methods perform as expected in between the unique-file and shared-file methods.

C. S3D I/O Benchmark

The S3D I/O benchmark is the I/O kernel of the S3D application, a parallel turbulent combustion application using a parallel direct numerical simulation (DNS) solver designed at Sandia National Laboratories [21]. It is parallelized using a three dimensional domain decomposition and MPI communication. Each MPI process is in charge of a piece of the three dimensional domain. A checkpoint is performed at regular intervals, and its data consist primarily of the solved variables in 8-byte three-dimensional arrays, corresponding to the values at the three-dimensional Cartesian mesh points. Each aggregate checkpoint stores four global arrays, which represent mass, velocity, pressure, and temperature, respectively. The mass and velocity arrays are four-dimensional and the pressure and temperature arrays are three-dimensional. All four arrays share the same size for the lowest three spatial dimensions X, Y, and Z, which are partitioned among MPI processes in a block-block-block fashion. The fourth dimension sizes of the mass and velocity data are 11 and 3, respectively, and are not partitioned. In our experiments, we keep the size of partitioned X-Y-Z dimensions a constant 50x50x50 in each process. Thus, each run produces about 15.26 MB of write data per process per checkpoint.

Figure 10 and Figure 11 show the results of unique-file I/O, one-shared-file I/O, subfiling method with β=8 and β=16. When β=8, the subarray in each subfile is of size 100x100x100. When β=16, the subarray size is 200x100x100. Compared to one shared-file I/O scheme, the timing of MPI collective write is effectively reduced by the subfiling scheme. On Lustre, the subfiling's write bandwidths are in between the unique I/O and shared-file I/O, as expected. However, on GPFS, the two subfilings outperform the other two I/O methods. By examining the file open time, the unique-file I/O has a significantly higher open cost than the subfiling and share-file methods. On the other hand, Lustre handles large-scale concurrent file open much better than GPFS and hence the unique-file I/O is still the best. The performance improvement of subfiling scheme over the shared-file I/O is attributed to reduced write time by decreasing the number of processes concurrently accessing each subfile.

VI. Conclusions

In this work we propose a subfiling scheme to extend the flexibility of parallel shared-file I/O for large multi-dimensional arrays to allow split into a number of smaller subfiles. Subfiling's implementation is built on top of MPI-IO. We evaluate its performance under PnetCDF. The subfiling scheme demonstrated it can outperform single shared-file I/O and reduce file open cost by decreasing the number of files opened. Reducing file open cost is important as some parallel file systems may not handle large number of files efficiently.

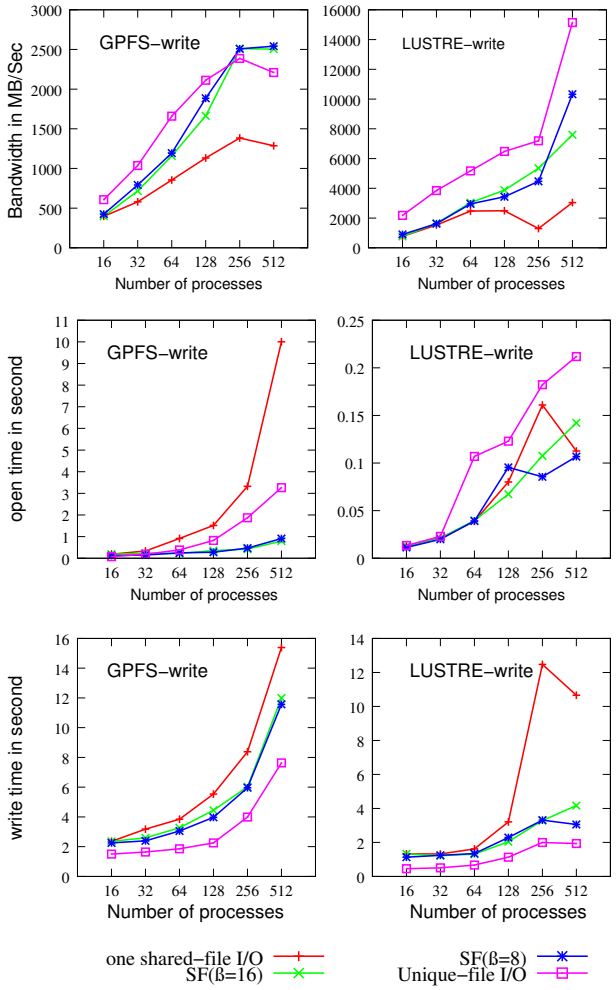

Figure 7. Write Performance results of ROMIO collective I/O test

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

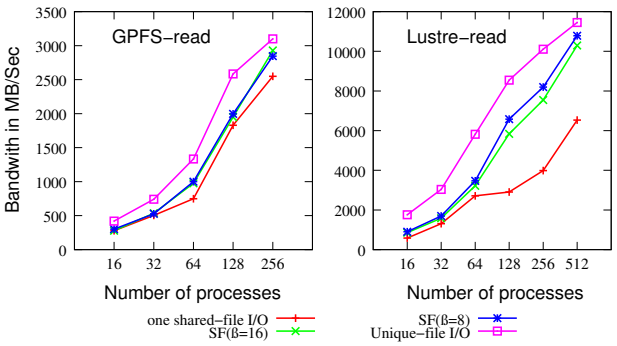

Figure 8. Read Performance results of ROMIO collective I/O test

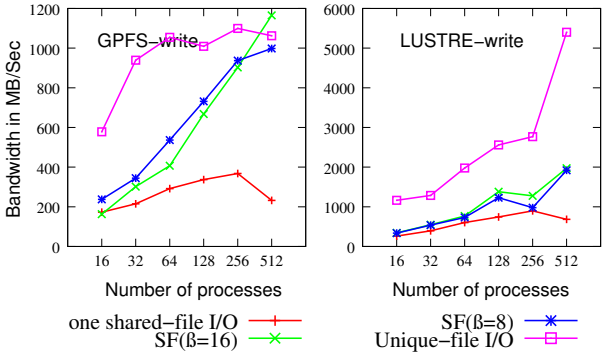

Figure 9. Write Performance results of FLASH I/O benchmark

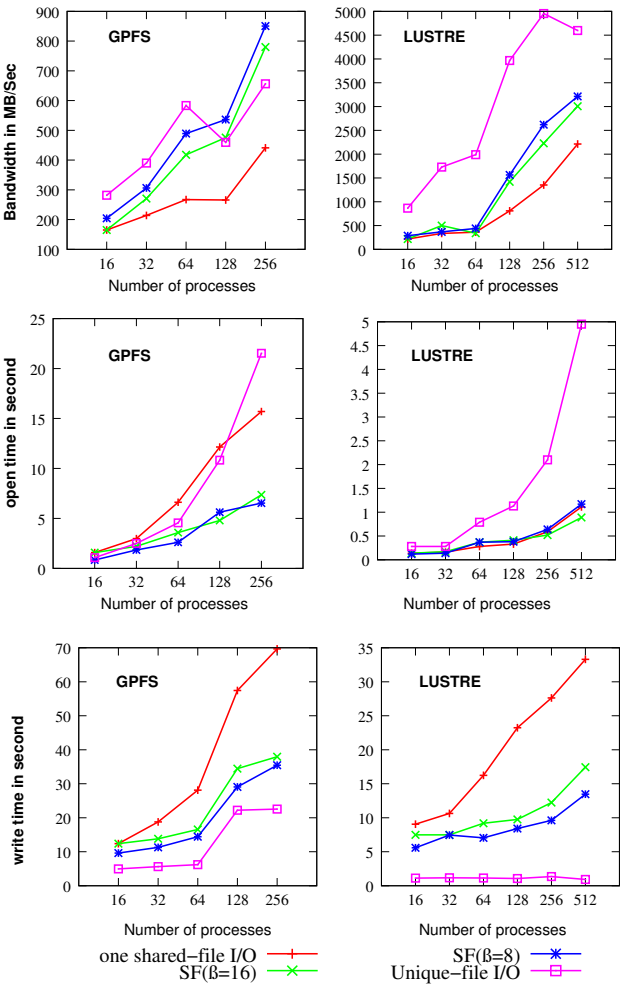

Figure 10. Write Performance results of S3D I/O benchmark

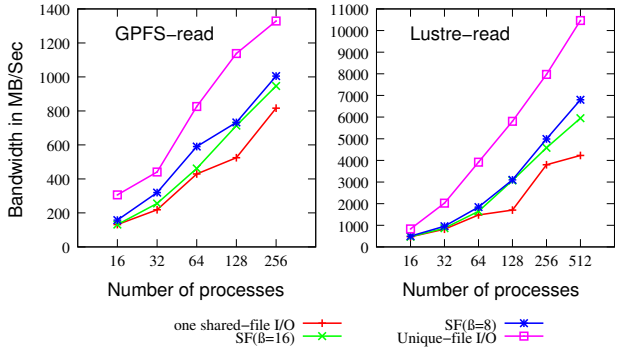

Figure 11. Read Performance results of S3D I/O benchmark

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. 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.

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

[3] R. Ross, R. Latham, W. Gropp, R. Thakur, and B. Toonen. Implementing MPI-IO Atomic Mode Without File System Support. In Proceedings of the 5th IEEE/ACM International Symposium on Cluster Computing and the Grid, May 2005.

[4] W. Liao, A. Ching, K. Coloma, A. Choudhary, and L. Ward. An Implementation and Evaluation of Client-Side File Caching for MPI-IO. In Proceedings of the International Parallel and Distributed Processing Symposium, March 2007.

[5] HPSS User's Guide, High Performance Storage System Release 7.1, February 2009 (Revision 1.0). http://www.hpss-collaboration.org/hpss/.

[6] Anurag Shankar, IU's TeraGrid-accessible HPSS archival storage system, http://racinfo.indiana.edu/hpc /workshops/includes/HPSS_TG_slides.ppt.

[7] R. Rew, 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.

[8] J. Li, W. Liao, A. Choudhary, Robert Ross, Rajeev Thakur, William Gropp, Rob Latham, Andrew Siegel, Brad Gallagher, and Michael 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.

[9] IEEE/ANSI Std. 1003.1. Portable operating system interface (POSIX) - Part 1: System application program interface (API) [C language], 1996.

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

[11] 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.

[12] Sun Microsystems. ZFS: the last word in file systems. http://www.sun.com/2004-0914/feature/, 2004.

[13] S.G. Inc. XFS: A High-Performance Journaling Filesystem. http://oss.sgi.com/projects/xfs, 2006.

[14] 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 Carallel Computer Systems at IPPS '93, pages 56-60, April 1993.

[15] 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.

[16] D. Kotz, Disk-directed I/O for MIMD Multiprocessors. In proceedings of th e1994 Symposium on Operating Systems Design and Implementation, pages 61-74, 1994.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] R. Sankaran, E. R. Hawkes, J. H. Chen, T. Lu and C. K. Law. Direct numerical simulations of turbulent lean premixed combustion. Journal of Physics: Conference Series, 46:38-42, 2006.

[22] M. Zingale, "FLASH I/O Benchmark Routine - Parallel HDF 5," Mar. 2001, http://flash.uchicago.edu/?zingale/ flash benchmark io.

[23] 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.