A Transparent Collective I/O Implementation

Yongen Yu, Jingjin Wu, Zhiling Lan
Department of Computer Science
Illinois Institute of Technology
Chicago, USA
{yyu22, jwu45, lan}@iit.edu

Douglas H. Rudd
Research Computing Center
University of Chicago
Chicago, USA
drudd@uchicago.edu

Nickolay Y. Gnedin
Theoretical Astrophysics Group
Fermi National Accelerator Laboratory
Batavia, IL
gnedin@fnal.gov

Andrey Kravtsov
Department of Astronomy and Astrophysics
The University of Chicago
Chicago, IL
andrey@oddjob.uchicago.edu

***Abstract*—I/O performance is vital for most HPC applications especially those that generate a vast amount of data with the growth of scale. Many studies have shown that scientific applications tend to issue small and noncontiguous accesses in an interleaving fashion, causing different processes to access overlapping regions. In such scenario, collective I/O is a widely used optimization technique. However, the use of collective I/O deployed in existing MPI implementations is not trivial and sometimes even impossible. Collective I/O is an optimization based on a single collective I/O access. If the data reside in different places (e.g. in different arrays), the application has to maintain a buffer to first combine these data and then perform I/O operations on the buffer rather than the original data pieces. The process is very tedious for application developers. Besides, collective I/O requires the creating of a file view to describe the noncontiguous access patterns and additional coding is needed. Moreover, for the applications with complex data access using dynamic data sizes, it is hard or even impossible to use the file view mechanism to describe the access pattern through derived data types. In this study, we develop a user-level library called transparent collective I/O (TCIO) for application developers to easily incorporate collective I/O optimization into their applications. Preliminary experiments by means of a synthetic benchmark and a real cosmology application demonstrate that the library can significantly reduce the programming efforts required for application developers. Moreover, TCIO delivers better performance at large scales as compared to the existing collective functionality provided by MPI-IO.**

Keywords-component; Transparent Collective I/O, Collective I/O, Parallel I/O, MPI, One-sided communication, I/O intensive applications, HPC

I. INTRODUCTION

Studies have shown that the processes of parallel applications tend to access a large number of small and noncontiguous pieces of data from a file, leading to the access of overlapping regions by different processes [1] [2] [3] [4]. Many applications need to map their multidimensional computing volume to one-dimensional file blocks in the eventual file order before performing I/O. For example, Scalable Earthquake Simulation (SCEC) partitions the 3D computing volume into a set of slices and assigns each slice to a core [5]; both S3D and Pixie3D divide their computing volumes into small cubes and assign each small cube to one core [6]. If mapping all the cells of the computing volume one-by-one in the order of x, y, and z, each process would access many small noncontiguous data blocks in an interleaving fashion (see Figure 1). Such I/O access patterns lead to poor parallel I/O performance and optimizations are necessary. Collective I/O [7] is a common optimization mechanism that is used to improve parallel I/O performance with such access patterns. That is, collective I/O is used to improve IO performance when each process accesses several noncontiguous portions of a file and the requests of different processes are interleaved and together span large contiguous portions of the file [7].

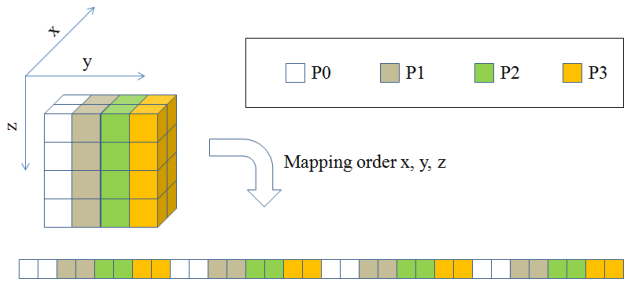

Figure 1. An example to illustrate the mapping from multiple dimensional computing volume to one dimensional file blocks, where each slice of the computing volume is assigned to a process. For writes, each process outputs four noncontiguous blocks with the stride distance equal to eight cells.

Despite the compelling advantage of collective I/O, studies have shown that some applications prefer to use POSIX (or POSIX stream) rather than using collective functionality

provided by MPI-IO [8]. The existing collective functionality provided by MPI-IO (denoted as "the original collective I/O (OCIO)" in the rest of this paper) is not transparent to applications, and requires extra coding from application developers. We argue there are three issues with OCIO.

First, an application may use multiple in-memory data structures to store their data. Since OCIO is an optimization for a single collective I/O call, data blocks from multiple data structures must be first combined and cached into an application level buffer before issuing a single collective MPI-IO call [9] [10] [11]. Maintaining such a buffer within each process requires additional programming efforts. Further, a poorly designed buffer can lead to a waste of memory. Hence, the first question is: ***can we let application developers focus on their I/O operations and free them from explicitly manipulating an extra application-level buffer to use collective I/O?***

Second, OCIO requires users to define a file view in their code to handle noncontiguous I/O accesses from multiple application processes. Each file view consists of two parts: the elementary data types to describe individual data elements and the file data types to describe data distribution in the file. Again, creating a file view requires extra coding. The second question is: ***can we free application developers from writing extra file view code for using collective I/O?***

Finally, many parallel applications perform computations using complex, dynamic data structures that change during the course of execution. As a result, the noncontiguous data blocks are of different sizes. It is hard or even impossible for users to use a single derived data type instance to describe these data blocks. Hence the third question is: ***can we use collective I/O to boost parallel I/O performance of the applications whose data blocks are of different sizes and varying distances?***

To address the above problems, in this paper we design and develop a user-level library, called transparent collective I/O (TCIO), to facilitate the use of collective I/O for parallel applications with random noncontiguous access patterns. The library exposes POSIX-like interfaces for applications to perform parallel I/O operations. Application developers are freed from writing derived data types to describe the noncontiguous access patterns of their codes. The library is built on two key elements. First is a 2-level buffer approach. When an application calls the library, the library transparently creates two levels of buffers per application process. The level-1 buffers are responsible for combining small data blocks within each process, and the level-2 buffers rearrange the I/O requests from different processes in a file-offset order. Second is the use of one-sided communication for data exchange among processes.

TCIO is a new implementation of collective I/O, which differs from the existing implementation provided by MPI-IO (i.e. OCIO) at four key aspects. First, TCIO frees application developers from explicit management of application-level buffers for achieving collective I/O. Second, by using TCIO, application developers do not need to write extra codes to describe file view. Both features can be easily observed by comparing Program 2 and 3 listed in Section V. Consequently, the amount of programming efforts needed by TCIO is significantly less than that required by OCIO. Third, TCIO facilities the use of collective I/O for the applications using complex, dynamic data structures like the cosmology application presented in Section V. Finally, TCIO adopts several optimization techniques to improve I/O performance including one-sided communication for data exchange among processes and lazy-loading for read operations.

We evaluate the library by means of both a synthetic benchmark and a real cosmology application. The synthetic benchmark is used to extensively compare TCIO as against ROMIO (an implementation of OCIO) [12] in terms of both programming efforts and I/O performance. The cosmology application highlights the benefits of TCIO in the case where OCIO cannot be used. Together, these case studies demonstrate that TCIO library can significantly reduce programming efforts from application developers, while providing comparable or even better I/O performance as against OCIO.

The remainder of this paper is organized as follows. Section II discusses the related work. Section III introduces the background of collective I/O. We describe the design methodology of TCIO in Section IV. Experiments are listed in Section V. Finally, we draw the conclusions in Section VI.

II. RELATED WORK

Recognizing that some scientific applications access multiple files simultaneously for different array data, G. Memik et al. introduce Multicollective I/O (MCIO) to extend Collective I/O by taking the inter-file access patterns into consideration [13]. Their study shows that determining the optimal MCIO access pattern is an NP-complete problem. Therefore, they propose two heuristics (Greedy Heuristic and Maximal Matching Heuristic) to determine the MCIO access patterns.

Overlapping computation with communication is a widely used optimization to reduce the overhead associated with parallel I/O in the field of HPC. V. Venkatesan et al. present the challenges associated with developing non-blocking collective I/O operations [14]. They extend the libNBC library in conjunction with Open MPI's OMPIO framework to handle non-blocking collective I/O operations.

W. Yu et al. claim that the time spent in the global process synchronization dominates the actual IO time and point out that there exists a "collective wall" in the performance of collective I/O [15]. To address the issue, the authors introduce a novel technique called partitioned collective I/O (ParColl) to augment the collective I/O protocol with new mechanisms for file domain partitioning, I/O aggregator distribution and intermediate file views. By using ParColl, a group of processes and their corresponding files are divided into a collection of small groups and each group performs I/O aggregation in a disjointed manner.

In [16], J. Blas et al. propose an alternative implementation of collective I/O, namely view-based collective I/O. It improves the performance of collective I/O by reducing the cost of data scatter-gather operations, file metadata transfer, consecutive collective communication and synchronization operations.

There are several studies on the improvement of collective I/O by exploring parallelism and physical locality. Y. Chen et al. propose a new collective I/O strategy, called Layout Aware Collective I/O (LACIO) [17]. This new collective I/O strategy explores on the physical data layout of the parallel file system instead of the logical file layout for performance optimization. Basically, LACIO incorporates the physical data distribution and information from parallel file systems with parallel I/O middleware. Requests from aggregators and file domain's partitions are rearranged in a fashion that matches with the physical data layout on storage servers of the parallel file system.

By considering the pattern of file stripping over multiple I/O nodes in the parallel file system, Zhang et al. designed a new Collective I/O implementation, named resonant I/O [18], which rearranges requests from multiple processes by the presumed on-disk data layout so as to turn non-sequential accesses into sequential accesses. Resonant I/O allows I/O requests to an I/O node to be from the same agent process or coordinates the requests from multiple processes to each I/O node in a preferred order.

Many modern parallel file systems maintain data consistency rules via locking mechanisms, which assign a process to exclusive access the requested file region in case of concurrent I/O requests on the shared file. Due to the potential serialization caused by locking, W. Liao et al. develop three file domain partition methods (i.e., partitioning aligned with lock boundaries, static-cyclic partitioning, and group-cyclic partitioning) for collective I/O optimization [19].

Unlike the aforementioned studies that focus on improving collective I/O from the performance perspective, this work is intended to provide a new collective I/O implementation that conducts collective I/O optimization transparently without leveraging knowledge from the applications. The resulting user-level TCIO library frees application developers from writing I/O optimization code. It also allows the applications with complex access patterns to use collective I/O.

Collective buffering is often used in MPI-IO implementations to boost parallel I/O performance at large scale. It selects a subset of nodes to communicate with IO servers for the purpose of reducing IO contention [20] [21]. While collective buffering can optimize a single collective call, TCIO is a new implementation of collective I/O targeting non-contiguous requests of a file from multiple processes. Note that in this study we do not enable collective buffering in the experiments.

III. BACKGROUND OF COLLECTIVE I/O

In this section, we first briefly describe collective I/O, and then demonstrate how to use collective I/O through an example.

A. Basic Design

MPI derived data types are a key feature of the MPI specification. They provide an elegant and efficient way to express noncontiguous, mixed types of data. OCIO, as a subset of the MPI specification, inherits this feature. It requires each process to use the MPI derived data type instances to describe the noncontiguous access patterns and pass them to the library by laying out a "view" of the file via the "MPI_File_set_view" subroutine.

OCIO divides the I/O operations into a data exchange phase and an I/O phase [7]. When an application invokes OCIO subroutines to output the data in application level buffers, OCIO calculates the file domain accessed by the application via the minimum and maximum file offsets. The aggregate file domain is then divided into equal, disjointed file regions, and each region is assigned to a temporary buffer per process (a.k.a. aggregator). The data from the application level buffers are shuffled among the computing processes according to the file offset and placed in the temporary buffers of aggregators. The aggregators then perform write calls on behalf of all the processes to output data to the file system. When an application invokes MPI-IO read operations, the aggregators serve as I/O delegators to move the data from files to their temporary buffers and then distribute them to the target processes.

B. An Example

In order to clearly describe the programming efforts necessitated by using OCIO, we introduce a simple example here. Consider an application that performs computation based on two in-memory arrays of type int and double, respectively. At write, the application first interleaves variables of the two types at the same array location, and then places variables in a single, shared file in a round-robin manner.

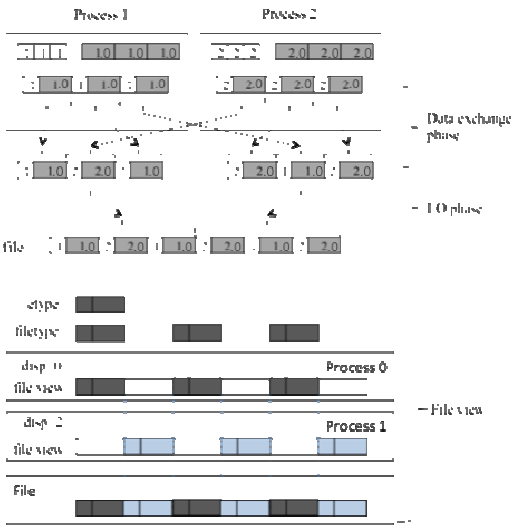

Figure 2. An example to illustrate OCIO, where there are two application processes, and each process accesses two arrays (white and grey).

Figure 2 shows the write operations of the application by using OCIO. Here we assume that the number of processes is two and the array length is three. Each application process first has to copy and combine the data of the two in-memory arrays into one application level buffer, because OCIO is an optimization for one single MPI-I/O call and each MPI call can only operate on one in-memory data structure. In this application, the buffer combines the variables in round-robin

fashion: The first position of the buffer holds the first element of the *int* array; the second position records the first element of the *double* array; the third and fourth slots hold the second elements of the *int* array and *double* array; finally, all the variables are combined in the application level buffer.

After combining the data, each application process creates a file view to define the noncontiguous access patterns. The bottom of Figure 2 illustrates the file views describing the I/O access regions of different processes. Each file view consists of three parameters: displacement, etype and filetype. In this example, "etype" is a contiguous derived data type consisting of two numbers: one integer number and one double number; "filetype" is a vector with the stride equals the number of processes times the size of etype; the displacements of the process one and process two are 0 and sizeof(etype) respectively. All the information is passed to the collective I/O library through the "MPI_File_set_view" function.

When the application invokes the collective MPI-IO writing subroutine to output the data buffered in the application level buffer, the I/O operations are divided into two phases. In the data exchange phase, all the noncontiguous data blocks are ordered by the logical file offsets to form one large contiguous data block. After that, the block is evenly partitioned into two parts. The first part is assigned to the first process, and the second part is assigned to the second process. Therefore, each process only needs to issue one contiguous access instead of three small accesses during the I/O phase. Moreover, the regions accessed by different processes are disjoint.

IV. TRANSPARENT COLLECTIVE I/O DESIGN AND IMPLEMENTATION

In this section, we present the design and implementation of TCIO, which is capable of transparently boosting the I/O performance of parallel applications with noncontiguous accesses.

A. Main Design

Figure 3 depicts the layered architecture of TCIO. At the top layer, it exposes POSIX-like interfaces for MPI applications to interact with the library. Beneath that, the library provides two levels of buffers to expedite I/O operations. The level-1 buffers are for combining in-memory data blocks within the same process locally, and the level-2 buffers are used to reorganize I/O accesses from different processes. The level-1 buffers are private to each process, while the level-2 buffers are shared among all application processes through MPI-2 one-sided communication.

Since TCIO, similar to OCIO, is an optimization for MPI-IO, it uses basic MPI-IO routines to move data between the level-2 buffers and the file system.

TCIO exposes POSIX-like interfaces for parallel applications to perform IO operations based on each piece of data. It is capable of performing collective I/O optimization across multiple I/O requests. The level-1 buffer is an indispensible component to deliver such a feature. The level-1 buffer is used to combine data blocks of a sequential I/O accesses within the same process locally.

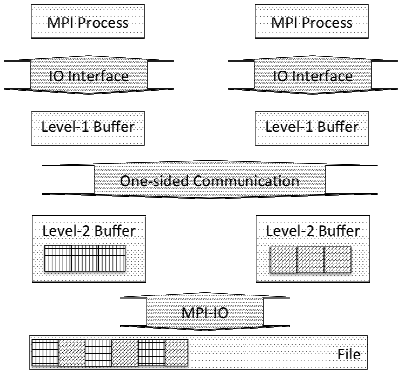

Figure 3. The archetecture of transparent collective I/O

The level-2 buffers are used to coordinate I/O requests among application processes so as to improve parallel I/O performance. Since the library does not leverage any information from the application regarding the file domain accessed by the application, it does not know in advance what size it should allocate for the level-2 buffers. In TCIO, each level-2 buffer consists of multiple equal sized segments and the segments of different processes are mapped to the file regions in a round-robin fashion according to the logical file offset (see Figure 3). This design achieves good load balance in terms of the buffered data from different processes. As for lookup operations, the application has to know the rank id of the remote MPI process that holds the required data (ID_{rank}), the segment id ($ID_{segment}$), and the displacement from the starting of the segment ($DISP_{block}$) of the desired data block. These values can be calculated using the following equations:

$$\mathrm{ID}_{\mathrm{rank}} = \frac{\mathrm{OFFSET}}{\mathrm{SIZE}_{\mathrm{segment}}} \,\%\, \mathrm{NUM}_{\mathrm{processes}} \quad (1)$$

$$\mathrm{ID}_{\mathrm{segment}} = \frac{\mathrm{OFFSET}}{\mathrm{SIZE}_{\mathrm{segment}}} \,/\mathrm{NUM}_{\mathrm{processes}} \quad (2)$$

$$\mathrm{DISP}_{\mathrm{block}} = \mathrm{OFFSET} \,\%\, \mathrm{SIZE}_{\mathrm{segment}} \quad (3)$$

where *OFFSET* is the logical file offset of the target data block, $SIZE_{segment}$ is the size of one level-2 buffer segment and $NUM_{processes}$ is the number of processes of MPI application. The library can calculate these three values in O (1) time given the logical file offset of a data block.

The level-2 segment size ($SIZE_{segment}$) is a crucial parameter for TCIO. If the segment size is smaller than the lock granularity of the underlying file system, MPI processes might compete with each other for the privilege to access a locked region, leading to poor performance. A large segment size might render an extremely unbalanced data distribution among MPI processes. Based on these facts, we set segment size as the stripe size (the locking granularity) of underlying file system.

TCIO uses the level-1 buffers to combine multiple pieces of data together. The combined data are placed in the level-2 buffers as segments. If a combined data block were larger than the size of one level-2 buffer segment, it has to be subdivided and placed in different segments of the level-2 buffers. Since there is no benefit to setting the size of level-1 buffer larger than the segment size of the level-2 buffer, we set them to be

equal, and each level-1 buffer is aligned with one level-2 buffer segment.

For write operations, TCIO buffers the data blocks in the level-1 buffer. It also retains the $DISP_{block}$ together with the length of the data blocks. Processes individually send out level-1 buffered data to the level-2 buffer when either the file domain of cached data blocks exceeds the size of the level-1 buffer or the application explicitly invokes the "flush" function. During the reading phase, TCIO moves the data in the other direction. Instead of using a preloading technique, TCIO uses a lazy-loading strategy for read operations. In particular, when the application issues read calls, rather than loading the data, the library stores the address, the length and the $DISP_{block}$ of the target data blocks. The real data-loading tasks take place when either the file domain of cached reads exceeds the size of the level-1 buffer, or the applications explicitly request the library to load data.

OCIO uses non-blocking communication to shuffle data across the computing processes at the data exchange phase in the all-to-all manner. Non-blocking communication (Isend/Irecv) is a two-sided communication model, which requires a matching pair on both sender and receiver sides. OCIO first issues MPI_Irecv to receive data from all processes, then issues MPI_Isend to send data to all processes, and then waits until all communication complete [22]. TCIO, however, cannot use two-sided communication. It allows processes to issue I/O calls for each piece of data individually, and as a result, different processes may issue a different number of I/O requests. TCIO instead relies on one-sided communication, which removes the requirement of a matching pair in both sender and receiver sides and allows the processes to initiate an end-to-end data movement across computing nodes from either the sender or receiver side. During writes, TCIO initiates data movements from the sender side. During reads, the receivers pull over the data from the level-2 buffer of the remote process.

In one-sided communication, "MPI_Win_fence" is the simplest approach to allow all processes to synchronize. However, "MPI_Win_fence" is a collective call, which by nature would break the TCIO design, which allows all the I/O accesses to be performed independently. Therefore, we use the lock-request paradigm ("MPI_Win_lock" and "MPI_Win_unlock") in the TCIO implementation.

When TCIO moves data between level-1 and level-2 buffers, these data consist of multiple disjointed data blocks. If it were to move each piece of data with its own one-sided communication call (MPI_Get, MPI_Put), this would cause a large number of network connections, which would in turn degrade the performance. We use "MPI_Type_indexed" to combine multiple data blocks as one derived data type instance. The library can then transfer the newly created data type instance by a single one-sided communication call.

B. Implementation

TCIO library is written in C language. It consists of about a thousand lines of code. We distribute it as a user-level library. Program 1 is the API definition of the library. It exposes POSIX-like I/O interfaces for parallel applications. It also allows applications to perform I/O operations based on MPI data types. "tcio_flush" function allows the application to explicitly move data from the level-1 buffers to the level-2 buffers. It is a collective call, which invokes "MPI_Barrier" to synchronize among processes. Since the library uses a lazy-loading strategy for reading operations, the actual data are not loaded into the target places after the read calls return. The library provides "tcio_fetch" function to enable applications to inform the library to load the desired data blocks to the target locations explicitly. "tcio_close" function issues "MPI_barrier" to synchronize among processes before outputting data from the level-2 buffers to file system.

Program 1: API Definition

```
tcio_file * tcio_open(char * fname, int mode)

tcio_write (tcio_file *fh , void * data, int count,
MPI_Datatype type)

tcio_write_at (tcio_file *fh , MPI_Offset offset,void *
data, int count, MPI_Datatype type)

tcio_read(tcio_file * fh, void * data, int count,
MPI_Datatype type)

tcio_read_at(tcio_file * fh, MPI_Offset offset , void * data,
int count, MPI_Datatype type)

tcio_seek(tcio_file * fh, MPI_Offset offset, int whence)

tcio_flush(tcio_file * fh)

tcio_fetch(tcho_file * fh)

tcio_close(tcho_file * fh)
```

To use TCIO, a user needs to specify the segment size and the number of segments per process.

C. An Example

Figure 4 uses the same example as shown in Figure 2 to demonstrate the algorithm of TCIO. For simplicity, we assume that each process holds one segment of the level-2 buffer.

At the first step, process 1 outputs the first element of the *int* array via a TCIO call. Since this piece of data will be stored at the beginning of the file, process 1 aligns its level-1 buffer with the first segment of the level-2 buffer and places the *int* value at the beginning of the level-1 buffer. After that, process 1 issues another TCIO call to output the first element of the *double* array, which is also stored in the level-1 buffer. Similarly, process 2 aligns its level-1 buffer with the first segment of the level-2 buffer and places the first element of the two in-memory arrays in its level-1 buffer.

At the second step, process 1 outputs the second element of the two in-memory arrays by invoking another two write calls. Since these two pieces of data fall into the same segment of the level-2 buffer with the previous writes, they can be placed in the current level-1 buffer. At this stage, process 2 cannot place the second elements of its two in-memory arrays in its level-1 buffer because these data blocks fall into a different segment of the level-2 buffer. It must first flush the data in its level-1 buffer to the global level-2 buffer.

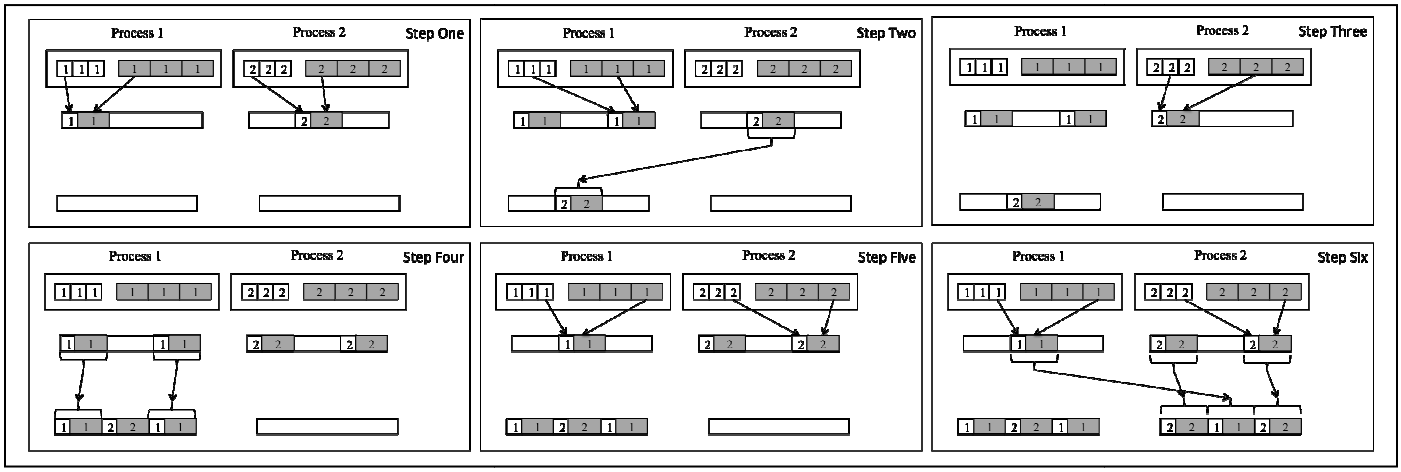

Figure 4. The work flow of TCIO. Step 1 shows the contents of the level-1 and level-2 buffers after the application outputs the first elements of each array. Step 3 presents the contents of level-1 and level-2 buffers after the application outputs the second elements of each array. Step 5 shows the content of the level-1 and level-2 buffers after the application outputs the third elements of each array. In step 6,the application outputs all the data in the level-1 buffers to level-2 buffers. Finally, all the data are transferred to a file.

At the third step, process 2 aligns its level-1 buffer with the second segment of the level-2 buffer and places the second elements of the two in-memory arrays in level-1 buffer.

At the fourth step, process 1 attempts to output the third pair of elements. Since these writes fall outside the first segment that the level-1 buffer is aligned with, the library flushes the level-1 buffer and moves these data blocks to the level-2 buffer.

At the fifth step, process 1 aligns the level-1 buffer with the second segment of the level-2 buffer and places the third elements of the two arrays in the level-1 buffer. Process 2 also places the third elements of its two in-memory arrays in its level-1 buffer.

At the sixth step, both the processes flush all the buffered data from the level-1 buffers to the level-2 buffers.

By comparing Figure 4 and Figure 2, it is clear that the actual I/O operations adopted by TCIO and OCIO are different. As we will show in the next section, the amount of programming efforts needed by TCIO is significantly less than that required by OCIO. Further, TCIO offers comparable or better I/O performance as against OCIO.

V. EXPERIMENT

A. Testbed

We evaluate TCIO library on the production Lonestar machine at TACC [23]. Lonestar is a 1,888-node cluster and each node features two 6-Core processors. Centos 5.5 is installed on the computing nodes and these nodes are connected by Mellanox InfiniBand network in a fat-tree topology with a 40Gbit/sec point-to-point bandwidth. Each node holds up to 24GB memory. The parallel file system Lustre [24] is installed on this machine and provides up to 1PB storage capability. Lonestar is configured with 30 object storage targets (OST). The stripe size is 1MB. By default, each file is stored on a single OST. We use the default setting in the following experiments. SGE is used to provide batch services.

We evaluate the library by means of a synthetic benchmark and a real parallel application. We compare TCIO as against OCIO with regard to programming efforts and I/O performance by using the benchmark. The application is used to illustrate the case where OCIO cannot be used, while TCIO can be used to boost application I/O performance. All these experiments were conducted during the production mode, meaning other applications coexist in the system. To minimize the noise in the performance results, a minimum of three runs were conducted for each experiment, and we present the average values.

B. Synthetic Benchmark

We create a benchmark to simulate the I/O accesses of the example application as shown in Figure 2. The benchmark has the special pattern --- small noncontiguous data blocks accessed by parallel processes in an interleaving fashion --- where collective I/O can optimize I/O performance over the vanilla MPI-IO. As mentioned earlier, the main goal of our benchmark experiments is to compare programming efforts and IO performance by using TCIO and OCIO respectively. Hence, in the rest of this subsection, we list the results achieved by TCIO and OCIO. Before presenting the experimental results, we list the configuration parameters of the benchmark in Table I. The following configuration parameters are used:

$NUM_{array} = 2$

$TYPE_{array}$ =i, d

$LEN_{array} = 3$

$SIZE_{access} = 1$

TABLE I. CONFIGURATION PARAMETERS

Symbol	Description
method	0: OCIO; 1: TCIO; 2 MPI-IO
NUM_{array}	The number of arrays within each process
$TYPE_{array}$	The data types of arrays separated by a comma. c: char; s: short; i: integer; f: float; d: double e.g. "i,d" means the first array is of integer type and the second array is of double type
LEN_{array}	The length of arrays
$SIZE_{access}$	The number of array elements per I/O access. We can use it to adjust the I/O access size. For example, if $SIZE_{access}$ equals 4, the benchmark access four array elements for each I/O call.

1) Programming Efforts

Freeing application developers from writing extra code is a key motivation of this work. Before comparing TCIO and OCIO implementations on performance, we list their respective codes. Program 2 is the OCIO code and Program 3 is the TCIO code.

Program 2: Programming efforts by using OCIO

1. Create an application level buffer
2. //Combine data in the buffer by two for loops
   for each i ∈ [1,LEN_{array}] increase $SIZE_{access}$
   for each j ∈[1, NUM_{array}]
   append the [i*th* ~ i+$SIZE_{access}$) elements of array j to the end of the buffer
3. //Open file
   MPI_File_open(MPI_COMM_WORLD, file_name, mode, MPI_INFO_NULL, &handle)
4. //Set out file view
   block_size ← (sizeof(int)+sizeof(double))* $SIZE_{access}$
5. disp ← my_rank *block_size
6. MPI_Type_contiguous (block_size, MPI_BYTE, &eType)
7. MPI_Type_commit(&eType)
8. MPI_Type_vector(LEN_{array}/ $SIZE_{array}$, 1 , num_procs , eType, &fileType)
9. MPI_Type_commit(&fileType)
10. MPI_File_set_view(handle, disp, eType, fileType, "native", MPI_INFO_NULL)
11. MPI_File_write_all(handle, <address of the buffer>, LEN_{array}/$SIZE_{access}$*block_size, MPI_BYTE, &status)
12. MPI_File_close(&handle)
13. Release the buffer.

Program 2 shows the implementation by using OCIO library. First, each benchmark process has to create an application level buffer to combine data, and then appends segments of each array to the buffers in a round-robin fashion through two for loops. Finally, all the array values are combined within a single application level buffer. After that, each benchmark process creates two derived data types to describe the noncontiguous access patterns and passes such information to library by setting out the view of the file. A single collective write call is issued to output all the data in the buffer. At last, release the occupied memory space for further use. For simplicity, we just describe the buffer operations with a single sentence in Program 2. In practice, however, creating and maintaining these application level buffers requires significant programming efforts by the application developers.

Program 3 presents the code of the implementation using TCIO. Each benchmark process only needs to output the elements of each array through two for loops in POSIX I/O fashion. It first calculates the file offset of the data block, then seeks the file handle to that position and then places the data block there. Application developers do not have to manipulate application level buffers, create derived data types or set out file views. Applications can benefit from collective I/O optimization by using fewer lines of code in a simpler way.

Program 3: Programming efforts by using TCIO

1. block_size ← (sizeof(int)+sizeof(double))*$SIZE_{access}$
2. handle ←tcio_open(file_name, mode)
3. for each i ∈ [1,LEN_{array}] increase $SIZE_{access}$
   a. pos ←my_rank*block_size +i*block_size*num_procs
   b. for each j ∈[1, NUM_{array}]
      i. tcio_write_at(handle,pos,$array_j$[i], $SIZE_{access}$, MPI_BYTE)
      ii. pos ←pos +<type size of array i> * $SIZE_{access}$
4. tcio_close(handle)

2) I/O Performance

a) Impact of the Number of Processes

In this subsection, we evaluate the performance of TCIO against OCIO with different numbers of processes. Table II shows the configurations of the benchmark. We vary the number of processes from 64 to 1024. Each process holds two in-memory arrays of integer and double types, respectively. The length of each local array is 4M.

TABLE II. EXPERIMENT CONFIGURATION

	Parameters				
	NUM_{array}	$TYPE_{array}$	LEN_{array}	$SIZE_{access}$	NUM_{proc}
Value	2	i,d	4M	1	64~1024

The left subfigure of Figure 5 shows the write throughput as a function of the number of processes. Collective I/O improves parallel I/O performance by aggregating large numbers of small and noncontiguous accesses into large fewer ones. Hence, the improvement of collective I/O for large I/O

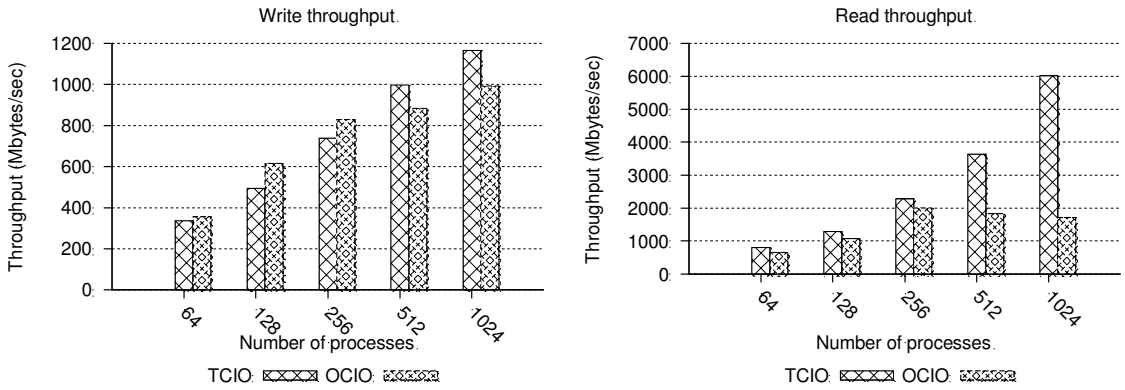

Figure 5. I/O throughput of the synthetic benchmark with varying access sizes

accesses is not evident. In this set of experiments, we set the access size to 1. From this figure, we observe that OCIO delivers better performance at small scales (<=256). TCIO, however, outperforms OCIO at large scale. OCIO exchanges data among computing nodes in all-to-all fashion. Each process receives data from all processes and then broadcast data to all processes through non-blocking I/O. The number of network connections increases quickly with the growth of computing nodes. TCIO uses one-sided communication to transfer data in end-to-end fashion. Each process sends or receives data from a single process each time. Thus the number of connections increases slower than that of OCIO. Moreover, OCIO performs all the communication at the same time, which might cause heavy traffic bursting in the network. TCIO, however, performs each communication individually. Therefore, TCIO achieves better writing performance as against OCIO at large scales (>=512).

The right subfigure of Fig. 5 presents the read throughput for the same set of experiments. In this figure, we can see that TCIO is better than OCIO. Moreover, we can observe that the gap between TCIO and OCIO is widened with the growth of compute nodes.

b) Impact of File Size

In this set of experiments, we evaluate TCIO as against OCIO with different file sizes. We use the same configuration parameters listed in Table II except that we fix the number of processes at 64 and vary the LEN_{array} from 1M to 64M, leading to the file size varies from 768MB to 48GB.

Figure 6 shows the write throughput of the benchmark with different dataset sizes. The key observation of this figure is that when the size of dataset is 48GB, the benchmark with OCIO fails to work. If the size of the entire dataset is 48GB and the number of process is 64, each process has to hold up to 0.75GB of data. By using OCIO, these data are first combined and cached in the application level buffers and then copied to the temporary buffers of the library. Therefore, each process has to provide 1.5GB (0.75*2) memory space for I/O operations. On Lonestar, the memory space is not sufficient for the benchmark to perform I/O operations with the code listed in Program 2. In TCIO, the benchmark does not have to combine all the data together in order to output them with a single call. Each process just has one reusable level-1 buffer and the size of which equals one segment size of the level-2 buffer. The size of the level-2 buffer equals the size of the temporary buffer in OCIO. Therefore, only 0.751GB(0.75GB+1MB) memory space is requires. TCIO outperforms OCIO in terms of memory utilization.

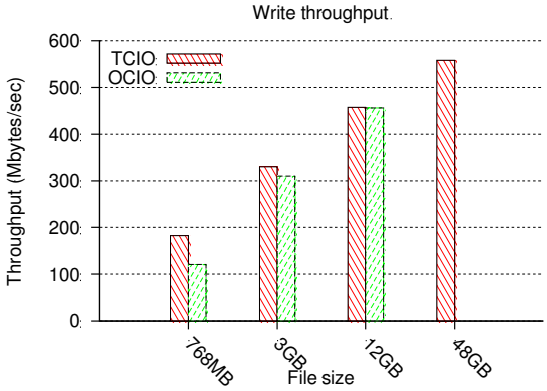

Figure 6. Write throughput

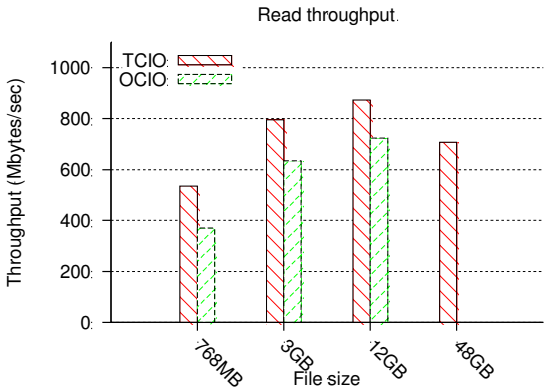

Figure 7. Read throughput

Fig. 7 shows the experimental results of read throughput for the same set of experiments. As for reads, TCIO delivers better performance than OCIO. Also, the benchmark fails to work

with the code listed in Program 2 when the size of dataset is 48GB.

3) Summary of Benchmark Results

Table III summarizes the differences between OCIO and TCIO. OCIO requires applications to create an additional level of buffers while TCIO does not. OCIO uses a file view mechanism, whereas TCIO exposes POSIX-like interfaces for applications to perform I/O operations in a transparent manner. The number of lines of code for I/O operations by using OCIO is more than that of TCIO. In OCIO, the total size of the application level buffers from different processes should be large enough to hold all the output data, while the size of the corresponding buffers in TCIO, the level-1 buffers, equals one segment size of the level-2 buffers. The memory utilization of TCIO is more efficient than that of OCIO. Restricted by the file view mechanism, OCIO is suitable for those applications with easy-to-describe access patterns, while TCIO is a generic library that can be used by a broad range of parallel applications.

TABLE III. COMPARISION BETWEEN OCIO AND TCIO

	Table Column Head	
	Original collective I/O	***Transparent collective I/O***
Application-level buffer	Yes	No
File view	Yes	No
Lines of code	Many	Few
Memory efficency	Poor	High
Restriction	access patterns that can be easily described by MPI derived data types	Any POSIX-like access patterns

C. Cosmology Application

In this subsection, we evaluate TCIO by means of a real cosmology simulation code called ART (Adaptive Refinement Tree) [25]. ART is a cell-based AMR application, which divides the whole 3D space computing volume into uniform cells, so-called root cells. Each root cell is an individual computing unit. If higher spatial resolution is required, a root cell can be refined into eight finer cells. The finer cells can be further refined and are organized in an octal tree. ART uses a fully threaded tree (FTT) [26] to represent refinement cells and their relationship. The structures of these trees change dynamically throughout the course of the computation, which causes these trees to have different structures and sizes.

In order to write the data into a file, ART also must map the 3D computing volume to one-dimensional on-disk blocks. When the mapping is done by allocating the cells in the order of x, y, and z, each process would divide its cells within its computing volume into multiple segments and place these segments on disk in an interleaving fashion. Such I/O access patterns can benefit from the use of collective I/O optimization.

Figure 8 shows the data layout of one FTT. It is a self-described file data format [27]. Both the variable values and tree structure information are recorded in the file. If one FTT holds two variables, the depth of the tree equals 6, and the numbers of nodes of each level are {1,2,4,8,16,32}, one FTT will consist of 129 arrays of different types and sizes. Since these arrays are adjacent in the file, a buffer is needed to combine these arrays together. OCIO requires the application to explicitly manage the buffer. TCIO, on the other head, performs the aggregation implicitly through its level-1 buffer.

Since these FTT differ in the number of cells they contain, they represent different amount of computational work. Processes will contain various numbers of FTT in order to maintain a reasonable load balance. In our tests, we assume that the lengths of the segments assigned to each process follows the normal distribution and use the following parameters to generate 1024 random numbers to represent the lengths of these segments. These segments are in turn assigned to the processes in a round-robin fashion.

TABLE IV. SEGMENTS GENERATION

	Parameters			
	Distribution	***Mu***	***Sigma***	***Seed***
Value	Normal	2048	128	5

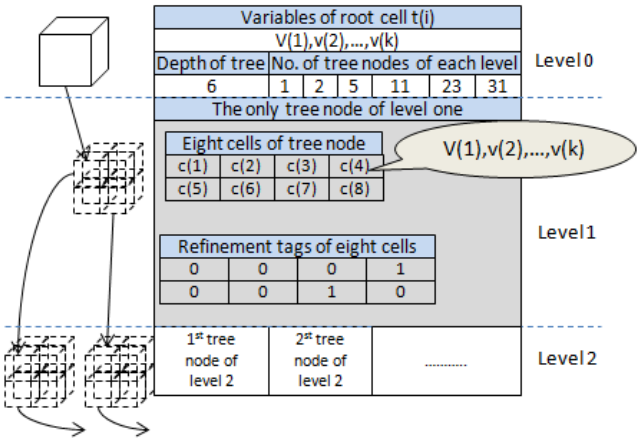

Figure 8. Data layout of one FTT

a) Programming Efforts

In order to use OCIO, ART first has to set out the view of the file. Since the lengths of these segments are different, the application cannot use a single elementary data type to describe the data block of each segment. Moreover, FTT is a complex data structure, which is represented by many small arrays of different sizes and types. Even we can use derived data types (e.g. MPI_Type_create_struct) to describe the structure of the FTT, we still have to create an application level buffer to combine these arrays together. Manipulating an application level buffer to combine and buffer these data is also an arduous work, not to mention that these FTT instances are of different sizes, and we have to create different derived data type instances for different FTT. In short, it is hard to use OCIO for the application, at least with the similar programming efforts by using TCIO.

As for TCIO, ART code does not have to inform the library of the noncontiguous access pattern of the application via file view by using derived data types. Also, the application does not have to create application level buffers to combine data blocks. The only thing that the application needs to do is to output each piece of data individually and TCIO will handle collective I/O operations transparently to the application.

b) I/O Performance

In this set of experiments, we evaluate the parallel I/O performance of the ART code with TCIO as against vanilla MPI-IO. Both allow applications to perform I/O operations based on each piece of data individually except that the former incorporates collective I/O optimization. In the experiments, we let the simulation first dump the intermediate data and then restart from this snapshot.

Figure 9 and 10 show the write and read throughputs of the ART code by using TCIO as against vanilla MPI-IO with a variety of scales. It is evident that TCIO is much better, up to 100X faster than the vanilla MPI-IO. When the number of processes is equal to or larger than 512, ART with vanilla MPI-IO takes more than 90 minutes to complete. That is why the figures only present TCIO data when the number of processes is equal or larger than 512.

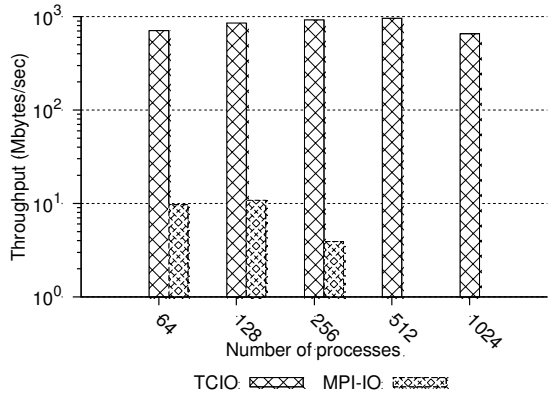

Figure 9. Write throughout of ART code

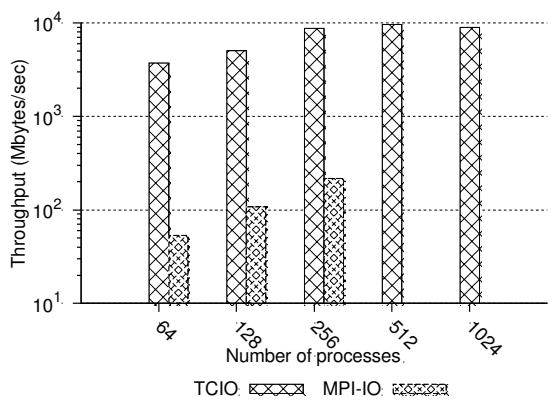

Figure 10. Read throughput of ART code

Another observation is that both the write and read throughputs of TCIO first increase with the increasing number of processes and then drop slightly. In this set of experiments, we test strong-scaling, meaning that the total number of root cells is fixed and the size of entire dataset is the same. When the computing scale is small, the aggregate I/O throughputs of compute nodes are the performance bottleneck. With the increasing number of processes, there will be more compute nodes to perform I/O operations. Hence, both the write and read throughputs grow with the increasing number of processes. On Lonestar, the centralized parallel file system Lustre is used to manage data. The number of I/O servers determines the bandwidth of the file system. If data-outputting rate overwhelms the bandwidth of the file system, application I/O throughputs stop increasing. Even worse, the competition among computing nodes will bring down the I/O performance. Therefore, the I/O throughputs of TCIO stop increasing and drop with the increasing of processes. Such a phenomenon indicates that the I/O performance of parallel large-scale applications subjects to the bandwidth of the underlying centralized parallel file system.

D. Experiment Summary

In summary, our experimental results with the synthetic benchmark and the cosmology application indicate that:

- TCIO can greatly reduce user's programming efforts for using collective I/O in their applications (see Program 2 and Program 3). Moreover, the applications with complex dynamic access patterns like ART can benefit from collective I/O by using TCIO.
- Experimental results indicate that TCIO outperforms OCIO at large scales. A key reason is that TCIO utilizes one-sided communication for data exchange among processes, which can significantly reduce the network traffic. This is essential for those large-scale applications where the network bandwidth is the performance bottleneck.
- TCIO uses less memory than OCIO, thereby being appropriate for those memory-intensive applications.

VI. CONCLUSION

Collective I/O is a powerful technique for parallel applications to improve I/O performance in the case that applications perform small, noncontiguous I/O accesses in an interleaving fashion. Existing collective I/O implementations require application developers to explicitly describe the noncontiguous access patterns through derived data types and inform the library by setting out the file view. In the case of the application having multiple data structures, each application process must first combine all the data from different places into a single application level buffer in order to perform I/O operations by issuing a single call. All these require significant programming efforts from application developers. In addition, due to the limitation of derived data types, some applications with complex dynamic access patterns may not be able to use the existing collective I/O implementation.

In this paper, we have presented a user-level library, namely TCIO to address the issues described above. TCIO exposes POSIX-like interfaces for parallel applications to conduct collective I/O optimization. Our case studies have

shown that the library can significantly reduce user's programming efforts. Moreover, it delivers better throughput as against the OCIO at large scales.

ACKNOWLEDGMENT

This work is supported in part by National Science Foundation grants OCI-0904670. This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number OCI-1053575.

REFERENCES

[1] P. Crandall, R. Aydt, A. Chien and D. Reed, "Input-Ouput Characteristics of Scalable Parallel Applications," in *Proceedings of Supercomputing '95*, 1995.

[2] N. Nieuwejaar, D. Kotz, A. Purakayastha, C. Ellis and M.Best, "File-Access Characteristics of Parallel Scientific Workloads," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, pp. 1075-1089, October 1996.

[3] E. Smirni, R, Aydt, A. Chien and D. Reed, "I/O Requirements of Scientific Applications: An Evolutionary View," in *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, 1996, pp. 49-59.

[4] J. Lofstead, M. Polte, G. Gibson, S.A. Klasky, K. Schwan, R. Oldfield, M. Wolf and Q. Liu, "Six Degrees of Scientific Data: Reading Patterns for Extreme Scale Science IO," in *Proceedings of the 20th international ACM symposium on High-Performance Parallel and Distributed Computing*, San Jose, CA, June, 2011.

[5] Y. Cui, K.B. Olsen, T.H. Jordan, K. Lee, J. Zhou, P. Small, D. Roten, G. Ely, D.K. Panda, A. Chourasia, J. Levesque, S.M. Day and P. Maechling, "Scalable Earthquake Simulation on Petascale Supercomputers," in *Proceedings of SC '10*, New Orleans, Louisiana, USA, 2010.

[6] R. Sankaran, E. Hawkes, J. Chen, T. Lu, and C. Law, "Direct Numerical Simulations of Turbulent Lean Premixed Combustion," *Journal of Physics: conference series*, vol. 46, pp. 38–42, 2006.

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

[8] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham and R. Ross, "Understanding and improving computational science storage access through continuous characterization," *Mass Storage Systems and Technologies, IEEE / NASA Goddard Conference on*, vol. 0, pp. 1-14, 2011.

[9] M. Zingale. FLASH I/O Benchmark Routine -- Parallel HDF 5. [Online]. http://www.ucolick.org/~zingale/flash_benchmark_io/

[10] W. Loewe, R. Klundt. IOR HPC Benchmark. [Online]. http://sourceforge.net/projects/ior-sio/

[11] The Los Alamos National Lab MPI-IO Test. [Online]. http://public.lanl.gov/jnunez/benchmarks/mpiiotest.htm

[12] ROMIO: A High-Performance, Portable MPI-IO Implementation. [Online]. http://www.mcs.anl.gov/research/projects/romio/

[13] G. Memik, M. T. Kandemir, W. Liao, and A. Choudhary, "Multicollective I/O: A technique for exploiting inter-file access patterns," *Trans. Storage*, pp. 349-369, Aug. 2006.

[14] V. Venkatesan, M. Chaarawi, E. Gabriel, and T. Hoefler, "Design and Evaluation of Nonblocking Collective I/O Operations," in *Recent Advances in the Message Passing Inerface (EuroMPI'10)*, vol. 6960, Santorini, Greece, 2011, pp. 90-98.

[15] W. Yu, J. Vetter, "ParColl: Partitioned Collective I/O on the Cray XT," in *ICPP '08*, Portland, OR, 2008, pp. 562-569.

[16] J. Blas, F. Isail, D. Singh, and J. Carretero, "View-based collective I/O for MPI-IO," in *Cluster Computing and the Grid, 2008. CCGRID '08*, 2008, pp. 409-416.

[17] Y. Chen, X. Sun, R. Thakur, P. Roth, W. Gropp, "LACIO: A New Collective I/O strategy for Parallel I/O Systems," in *the 25th IEEE Int'l Parallel and Distributed Processing Sympsium(IPDPS2011)*, 2011.

[18] X. Zhang, S. Jiang, and D. Kei, "Making Resonance a Common Case: A High-Performance Implementation of Collective I/O on Parallel FIle Systems," in *Parallel & Distributed Processing IPDPS '09*, 2009, pp. 1-12.

[19] W. Liao, and A. Choudhary, "Dynamically Adapting File Domain Partitioning Methods for Collective I/O Based on Underlying Parallel File System Locking Protocolssed ," in *High Performance Computing, Networking, Storage and Analysis, SC '08*, Austin, TX, 2008.

[20] CRAY. [Online]. https://fs.hlrs.de/projects/craydoc/docs/books/S-2490-40/html-S-2490-40/chapter-g1s9a5n5-oswald-benchmarkresults.html

[21] NERSC. [Online]. http://www.nersc.gov/users/data-and-networking/optimizing-io-performance-for-lustre/

[22] K. Coloma, A, Ching, A. Choudhary, W.Liao, R. Ross, R. Thakur and L.Ward, "A New Flexible MPI Collective I/O Implemenation," in *n Proceedings of the IEEE Conference on Cluster Computing (Cluster 2006)* , Barcelona, Spain.

[23] Extreme Science and Engineering Discovery Environment. [Online]. https://www.xsede.org/

[24] Oracle. Lustre File System. [Online]. http://wiki.lustre.org

[25] A. V. Kravtsov, A. A. Klypin and A. M. Khokhlov, "Adaptive Refinement Tree: A new High-resolution N-body code for Cosmological Simulations," in *Astrophyc. J. Suppl*, 1997, pp. 73-94.

[26] A. M. Khokhlov, "Fully Threaded Tree Algorithms for Adaptive Refinement Fluid Dynamics Simulations," *Journal of Computational Physics*, 1998.

[27] Y. Yu, D.H. Rudd, Z. lan, N.Y. Gnedin, A. Kravtsov and J. Wu, , "Improving Parallel IO Performance of Cell-based AMR Cosmology Applications ," in *IPDPS '12*, 2012.

.