J Supercomput (2017) 73:2069–2097
DOI 10.1007/s11227-016-1904-7

Reducing I/O variability using dynamic I/O path characterization in petascale storage systems

Seung Woo Son[1] · **Saba Sehrish**[2] · **Wei-keng Liao**[3] · **Ron Oldfield**[4] · **Alok Choudhary**[3]

Published online: 1 November 2016

Abstract In petascale systems with a million CPU cores, scalable and consistent I/O performance is becoming increasingly difficult to sustain mainly because of I/O variability. The I/O variability is caused by concurrently running processes/jobs competing for I/O or a RAID rebuild when a disk drive fails. We present a mechanism that stripes across a selected subset of I/O nodes with the lightest workload at runtime to achieve the highest I/O bandwidth available in the system. In this paper, we propose a probing mechanism to enable application-level dynamic file striping to mitigate I/O variability. We implement the proposed mechanism in the high-level I/O library that enables memory-to-file data layout transformation and allows transparent file partitioning using subfiling. Subfiling is a technique that partitions data into a set of files of smaller size and manages file access to them, making data to be treated as a single, normal file to users. We demonstrate that our bandwidth probing mechanism can successfully identify temporally slower I/O nodes without noticeable runtime overhead. Experimental results on NERSC's systems also show that our approach isolates I/O variability effectively on shared systems and improves overall collective I/O performance with less variation.

Keywords Parallel I/O · I/O variability · Subfile · PnetCDF

✉ Seung Woo Son
seungwoo_son@uml.edu

1 University of Massachusetts Lowell, Lowell, MA, USA

2 Fermi National Accelerator Laboratory, Batavia, IL, USA

3 Northwestern University, Evanston, IL, USA

4 Sandia National Laboratory, Albuquerque, NM, USA

1 Introduction

Scientists and engineers are increasingly utilizing leadership-level high performance computing (HPC) systems to run their extreme-scale data-intensive applications, such as thermonuclear reactions, combustion, climate modeling, and so on [11,37,39,41]. Scalable parallel I/O libraries and runtime systems are one of the key components to sustain scaling of those applications [5,21]. The I/O requirements of such applications, however, can be staggering, ranging from terabytes to petabytes, and managing such massive data sets imposes a significant performance bottleneck [7,22].

Current parallel I/O libraries already use state-of-the-art techniques and optimizations; significant challenges still exist in accomplishing scalable, consistent I/O performance. The reason is that the file servers are shared resources among various applications; therefore, they often experience unbalanced I/O load at runtime. This results in fluctuating file system performance from a perspective of individual application. [4,6,15,24,30,55]. In petascale systems, the amount of I/O bandwidth available to any particular job or application can fluctuate to a large extent depending on the degree of other concurrent jobs accessing the shared file system. Shared burst buffer systems have been introduced to mitigate contention on the file servers [28,40,54], but that interference effects are still a problem and an effective scheduling is required to address that I/O interference [17,36,46,50,53,56,61]. Another common cause of I/O fluctuation is an RAID rebuild from a disk drive failure. Since the collective I/O performance is limited by the slowest process (also called stragglers [4,6]), ensuring no process remarkably lags behind is critical for scalable collective I/O performance.

Many parallel applications express their problem using a global data structure in a multidimensional array format and partitions the array among all processes, such that each process computes on sub-domains in parallel. There have been numerous techniques to coordinate application processes' I/O requests, and the collective I/O in MPI-IO [34] has been a widely used optimization to allow coordination among participating processes and rearrange their I/O requests to achieve better I/O performance. Optimizations to improve collective I/O performance include two-phase I/O [38,47], disk-directed I/O [18], server-directed I/O [42], persistent file domain [26], active buffering [33], collaborative caching [27], and adaptive file domain [25]. While these improvements are effective, collective I/O operations at scale face new challenges on modern HPC systems. That is, as system scales and application complexity grow, various contentions, such as contention on communication network owing to the high ratio of application processes to file servers and contention on file locking among processes accessing shared files, can be a significant barrier to achieve scalable I/O performance at scale.

1.1 Contribution

In this paper, we make the following four contributions:

– We demonstrate that various contention on shared I/O resources significantly limits the performance of collective I/O.

- We demonstrate that there is a higher correlation between the bandwidth measurements and latency measurement at runtimes.
- We propose a probing technique to calculate file servers' available bandwidth at runtime and isolate the impact of accessing relatively slower I/O servers by excluding them from being used for file striping.
- We propose and implement a transparent subfiling through data layout transformation mechanism that divides the data among files in the context of higher I/O library, called PnetCDF.

The study presented in this paper supports the view of conventional collective I/O where N processes write to 1 shared file or an N-1 pattern, and still offers scalable I/O performance in the presence of fluctuating performance among file servers. We extend our prior work on the design and evaluation of file partitioning [44] and have implemented the proposed scheme into a high-level I/O library, parallel netCDF [23]. Our experimental evaluations of several production-level applications running up to 40,960 processes on NERSC's Hopper [35] have shown significant I/O performance improvements. We show that our approach effectively isolates the impact from temporarily slower I/O nodes, thereby reducing write I/O time significantly with less deviation. Since the mechanism is implemented at high-level I/O library layer (PnetCDF), all files including subfiles are in a self-describing portable data format. Maintaining portable data representation is beneficial to applications because it preserves layouts across all I/O software layers and provides transparent access to data structures. Also, the metadata available at high-level I/O library made much flexible partitioning like per-array partitioning or use of different dimension for partitioning. Lastly, our evaluations using a set of micro benchmarks and real applications demonstrate that our approach improves I/O performance significantly while limiting the number of (sub)files generated, which is proportional to the number of I/O nodes available.

The remainder of this paper is organized as follows. Section 2 extends the discussion of our motivation. The design of our approach is described in Sect. 3. Our modification to PnetCDF to implement our idea is provided in Sect. 4. Section 5 presents our experimental evaluation results. We discuss related work in Sect. 6. Finally, Sect. 7 summarizes the major findings of this paper and discusses the list of possible future work topics.

2 Background

To establish the theoretical depth of the ideas in this paper, we first describe distinctive features of the I/O architecture and software layers adapted by many HPC systems at scale, followed by the implications of such architecture to I/O performance.

2.1 I/O architecture and software components

The system architecture illustrated in Fig. 1a, a representative block diagram for many modern HPC systems such as Cray XT6 or IBM BG/P systems, has thousands of compute nodes, each with several multi-core CPUs, and tens of GB memory per node,

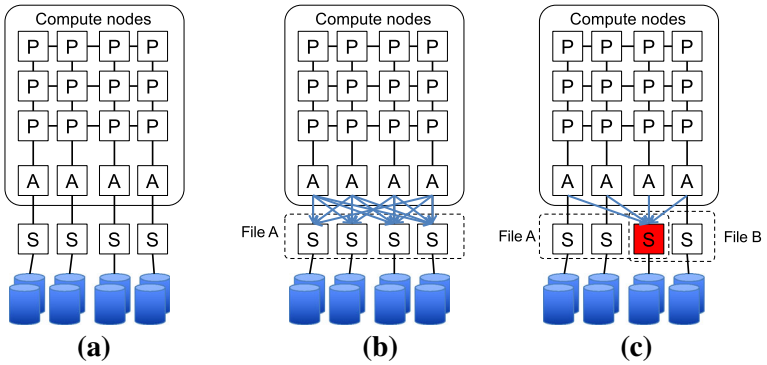

Fig. 1 I/O architecture and I/O contention. *P* Compute nodes, *A* I/O aggregators, *S*: I/O nodes. **a** A typical system configuration. **b** Contention among processes (I/O aggregators). All aggregator processes are competing to access File A stripped across all I/O nodes. **c** Contention among jobs. Aggregator processes from different applications are competing to write to File A and B whereby one I/O node (in red color) is shared between two files

offering peak performance of tens of petaflops for the entire machine. The I/O and inter-node communication on the compute nodes travels on several internal networks. The compute nodes communicate using a high-bandwidth, low-latency network, e.g., a 3D torus in Intrepid [1] and Hopper [35]. Each compute node is connected to other neighbor nodes through a network topology. Each network node handles both inbound and outbound data traffic among other nodes. Tens to hundreds of storage servers, each is attached to storage devices, are connected at the other end of this interconnection network. The storage system typically provides some form of redundancy, e.g., RAID or replication, to provide high availability.

Because of the layered system architecture as described above, there are multiple layers of software involved in the I/O path. At the application layer, high-level I/O libraries, such as HDF5 [51] or PnetCDF [23], are typically used, but MPI-IO or POSIX I/O calls also be used directly. MPI-IO optimizations such as two-phase I/O are achieved through communications over the interconnection network among compute nodes, whether MPI-IO APIs are called directly from applications or indirectly through higher-level libraries.

Collective I/O is an optimization employed in many MPI-IO implementations for improving the I/O performance to shared files. The motivation is that file systems prefer large contiguous requests to small noncontiguous ones. ROMIO, an implementation of MPI I/O functions adapted by many MPI implementations, first calculates the aggregate access region, a contiguous file region denoted by the start and end offsets among all requesting processes. This aggregate access region is then partitioned into non-overlapping, contiguous subregions denoted as file domains, each of which is assigned to an individual process. These set of processes, called aggregators, can make I/O calls [such as open(), read(), write(), and close()] on behalf of all requesting processes located in its file domain. There are two parameters in ROMIO, which are tunable by users through MPI hint mechanisms: the aggregators (or `cb_nodes`) and the collective buffer size (or `cb_buffer_size`) [47,48]. The aggregators are

a subset of processes acting as I/O delegates for the remainder of the processes. In ROMIO, the choice of aggregators depends on the file systems. For most file systems, one MPI process per compute node is picked to serve as an aggregator. In the systems containing multi-core CPUs in each node, this strategy avoids the intra-node resource contention that could be caused by two or more processors making I/O calls concurrently. For the Lustre file system, the current implementation of ROMIO picks the number of aggregators equal to the file striping count (or `striping_factor`). This design produces an one-to-one mapping between the aggregators and the file servers to eliminate the possible lock conflicts on the servers [25,57]. The collective buffer size indicates the amount of temporary buffers that can be used during data redistribution. If the file domain is bigger than the collective buffer size, multiple collective I/O operations need to be performed, each of which handles a file sub-domain of size no larger than the collective buffer size. The striping count of a file is the number of I/O servers, or object storage targets (OSTs) in Lustre, where a file is stored. Like all parallel file systems (PFS), files are striped using fixed block lengths, which are stored in the OSTs in a round-robin fashion.

2.2 Contention on I/O path

While collective I/O is an effective technique to improving I/O performance on shared files, it continues to face significant scalability challenges [4,30,55,58] for several reasons. First, as demonstrated by several prior studies as well as illustrated in Fig. 1b, the synchronization cost among aggregators acquiring lock on the shared file within the assigned file domain pose a limit to the scalable I/O performance. Recent studies [25, 58] also observed similar findings, and this problem would merely aggravate as the number of processes increases. More importantly, in accessing shared storage systems, there are higher degrees of I/O variability. This variability is hard to remove completely because of the various ways an application can access "shared" file systems. For example, as illustrated in Fig. 1c, two applications (or jobs) running concurrently on the system can access the file system simultaneously at a given time. Another example of such a case can occur when one application is trying to read the data stored in the shared storage while the other application is writing checkpoint data. This I/O variability is a big barrier to achieve scalable collective I/O operations because the I/O performance is coupled with the slowest storage nodes. In other words, even if most storage nodes are relatively fast, the overall collective I/O time is determined by the slowest nodes.

To test our hypothesis, we wrote a micro benchmark code where each process opens a file with a striping factor of 1 and writes 1GB of data on it. We have collected the write I/O time observed at each I/O server. Details of our experimental setup are given in Sect. 5. Figure 2 shows that although each I/O node writes the same 1GB of data, there are a couple of I/O nodes showing an excessively high write I/O time than others. In fact, the slowest I/O node showed almost 9 times higher I/O time than most other I/O nodes. If a file is striped on the I/O nodes including that particularly slower one, which is quite common in production runs, it would be of a significant barrier to

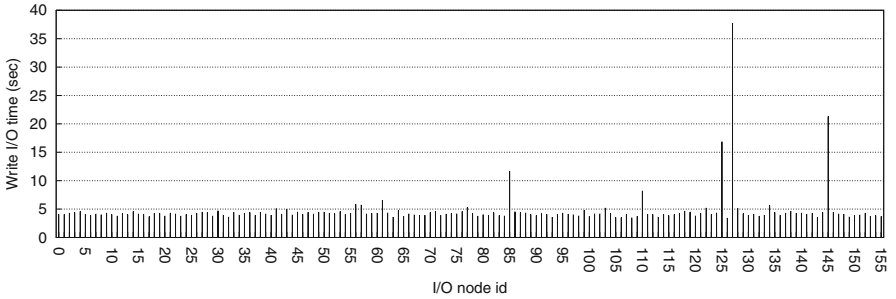

Fig. 2 The write I/O time distribution measured for all 156 I/O nodes when 156 processes write 156 files with the same size exclusively

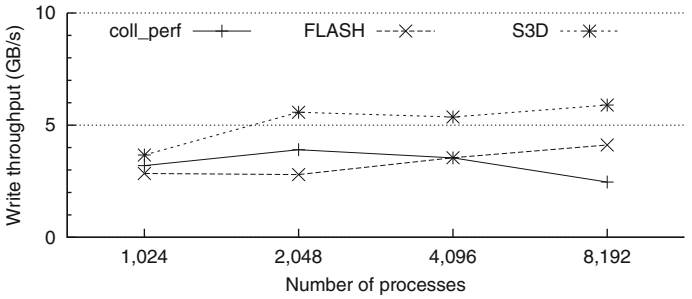

Fig. 3 We observed that all benchmarks we evaluated did not scale well when there is an excessive imbalance in I/O node performance. Details about our experimental setup are explained in Sect. 5. Note that, while we observe the same impact on GCRM, we do not present the results with GCRM in this figure because it runs on different number of processes

achieve scalable I/O performance. As shown in Fig. 3, this imbalance severely limits the scalability of the benchmarks we tested.

3 Design of transparent subfiling layer

This section describes our subfiling mechanism in the context of higher-level I/O libraries. We discuss how our mechanism determines the number of subfiles with existence of imbalanced performance in I/O servers and how datasets are divided into subfiles.

3.1 Dynamic probing for selecting storage nodes

Our mechanism to isolate the impact of accessing relatively slower I/O nodes is to probe each I/O server's load at runtime and estimate available bandwidth on each server before the file is created for striping. The goal of this step is twofold. First, we would like to monitor each OST's current bandwidth availability at runtime. Because the I/O access pattern in HPC systems is typically bursty, we measure each server's I/O bandwidth by writing a small dummy dataset just before writing actual data to the

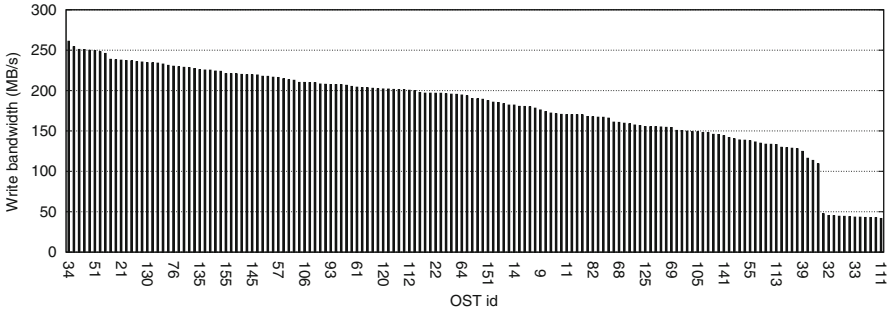

Fig. 4 Distribution of write I/O bandwidth observed for all 156 OSTs when a dummy file of 16MB is written to each OST. Each OST's observed bandwidth is sorted in descending order

file. Second, we would like to select the list of OSTs that can be used for storing each subfile by taking account of those available I/O bandwidth.

We consider two criteria in designing the probing module. First, the overhead incurred by probing should be minimized as it will not be the part of actual I/O. Second, the probed bandwidth should reflect the temporal behavior of I/O node as much as possible. Combining these two, we determine each I/O node's bandwidth by writing several tens of MB of randomly generated data to each I/O server. To make the actual I/O happen for this relatively small-sized files, we explicitly bypass buffer cache using POSIX I/O with the O_DIRECT flag; otherwise, they could sit on the buffer cache on compute nodes. For the same reason, we also have to make sure that the dummy file is large enough to fill the RPC buffer size.

Figure 4 shows the distribution of measured write I/O bandwidth for all 156 I/O nodes (OSTs) available on NERSC's Hopper. This graph confirms that certain OSTs show relatively slower bandwidth than the others. We again attribute this to an imbalance when accessing shared storage, as extensively discussed in recent studies [4,30,55]. Given these observed I/O bandwidths, we use following algorithm to select the a set of I/O nodes being used for file striping. Assuming B_i to be the sorted bandwidth observed in each I/O server, i, we denote the aggregate I/O bandwidth, $\mathbb{B}_i$, using i OSTs by $\mathbb{B}_i = i \times B_i$, where $1 \leq i \leq 156$. To illustrate how to calculate $\mathbb{B}_i$, let us assume that there are five OSTs and the observed bandwidths are 100, 110, 120, 90, and 70, respectively. Then, we first sort the measured bandwidth, which results in 120, 110, 100, 90, and 70. Therefore, if we add the node with 120, $\mathbb{B}_1$ is 120, which is 120×1. If we add the node with the next highest measured bandwidth, $\mathbb{B}_2$ is 220, which is 110×2. The entire values for $\mathbb{B}_i$, where $i = 1, 2, 3, 4, 5$, are 120, 220, 300, 360, and 280. As we can see, $\mathbb{B}$ drops after adding the node with 70 from 360 to 280 as $\mathbb{B}$ is bound the bandwidth of the slowest OSTs. Therefore, we exclude that node from being used for striping and use only the first four OSTs for file striping.

Figure 5 shows the estimated maximum aggregate I/O bandwidth based on our algorithm. As shown in the figure, the aggregate bandwidth increases as more I/O nodes are added, but gradually saturates and then eventually declines because the aggregate bandwidth has limited the performance of the slowest node. To select the maximum number of I/O nodes that provide us the highest aggregate I/O bandwidth,

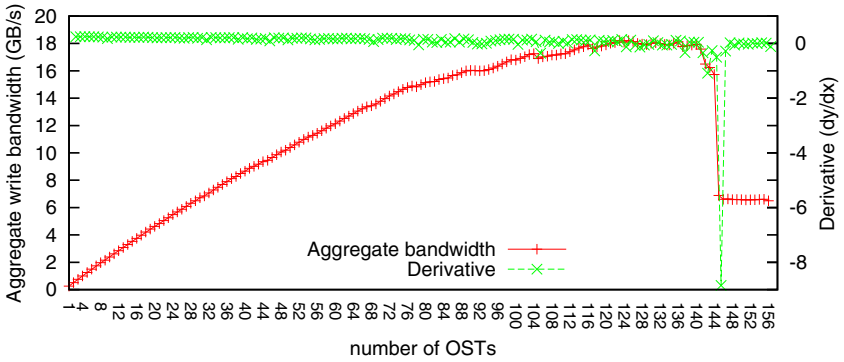

Fig. 5 Aggregate I/O bandwidth graph and its derivative (i.e., the slope) at each point. Our algorithm determines the subset of I/O nodes that would offer the best available aggregate bandwidth, as it sorts the observed the bandwidth of I/O nodes and selects relatively faster (or less busy) I/O nodes among them

we calculate the derivative of $\mathbb{B}_i$, which represents the slope of $\mathbb{B}_i$ at each value of i. Since our goal here is to maximize the number of I/O nodes, we select i when $\mathbb{B}'_i$ is negative and is less than a certain threshold, δ. The threshold value is basically meant for capturing the degree of slowness in the aggregate bandwidth when a node is added. The mechanism for runtime probing and I/O node selection described so far is given in Algorithm 1. Using the results given in Fig. 5, our algorithm excludes 12 OSTs that show less than 50MB/s for striping (sub)files. The aggregate bandwidth was estimated to peak when the first 128 OSTs were added, but the significant bandwidth drop occurs when 145th OST is added. We note that if all probed bandwidth values are similar to each other, our algorithm will select most of the available OSTs. We also note that this behavior is temporal, so depending on actual I/O load and the time of writing, our algorithm selects a different set of I/O nodes, and we observe that typically less than 5% of I/O nodes (chosen randomly) is excluded from our algorithm.

Since I/O nodes are shared resources, it is possible that multiple applications can probe the bandwidth simultaneously. However, such chances are extremely rare because probing happens only once when a file is created. If two probes did occur at the same time, the bandwidths obtained should be halved. However, this also means that two jobs are most likely competing for the shared file systems. Therefore, both applications are expected to observe slower I/O performance. An ideal solution could be conveying individual application's probing information to system-level scheduler, but in this study, we focus on user-level solution that makes best use of available information to maximize I/O performance.

3.2 Mapping multidimensional arrays to subfiles

When the subset of I/O nodes that are relatively faster than others has been isolated, the ideal solution would be to stripe files across those I/O nodes. However, on current PFSs like Lustre, users have no way to stripe files across a set of specific I/O nodes, or OSTs in Lustre. Because of this, we cannot provide the performance on normal files on

Algorithm 1: Algorithm for determining the storage nodes that would potentially give the maximum achievable aggregate bandwidth at a given time. The obtained I/O node lists are broadcasted to all processes.

Input: N: number of I/O nodes;
Output: N': number of selected I/O nodes;
$S[N']$: I/O node list of N';

B_i: each OST's bandwidth;
lb: lower bound of bandwidth;
for *each sampling process,* P_i , $1 \leq i \leq N$ **do**
obtain B_i by writing a dummy data using POSIX I/O to storage node i;
`/* gather all OST's write bandwidth */`
`MPI_Allgather(&`B_i, ...);
sort the gathered B_i in descending order;
for *each* i **do**
calculate aggregate bandwidth, $\mathbb{B}_i = B_i * i$;
calculate $\mathbb{B}'_i$, the derivative of $\mathbb{B}_i$;
if $\mathbb{B}'_i < \delta$ **then**
lb = B_i;
break;
while $i \leq lb$ **do**
$S[i] = i$;
$i++$;
`/* broadcast number of selected I/O nodes as well as corresponding node IDs */`
`MPI_Bcast(&`N'`, 1, MPI_INT, 0, MPI_COMM_WORLD);`
`MPI_Bcast(&`$S[N']$`,` N'`, MPI_INT, 0, MPI_COMM_WORLD);`

selected OST in the experiments. The same reason of Lustre not allowing users to stripe files across selected OSTs prevents the use of our subfiling in ROMIO. We note that MPI-IO processes one file at a time. This leads to our "subfiling" mechanism where each subfile has a striping factor of 1 (or stripe_count==1). Lustre does allow users to pick the starting OST for a file. Therefore, after the set of I/O nodes is determined, we partition arrays among them. Figure 6 gives an overview of our subfiling scheme. From an application's perspective, the partitioning is transparent, meaning that all processes open and access a single file throughout program execution. The subfiling mechanism then internally splits application processes into set of subprocesses, each creates its own subfile collectively. The subfile created by each subprocess group is accessed *exclusively* by that group. The goal of this mechanism is to reduce the contention as illustrated in Fig. 1b.

In higher I/O libraries like PnetCDF or HDF, there are sequences of steps to follow in order to perform I/O, and a typical example of such steps is as follows:

```
1. file open/creation
2. dimension definition
3. array definition
-------------------------------------
4. I/O operation (write/read)
5. file close.
```

There might be other steps like adding attributes to a file or array, but the above steps are the key steps that most application writers need to include in their I/O routine. Based

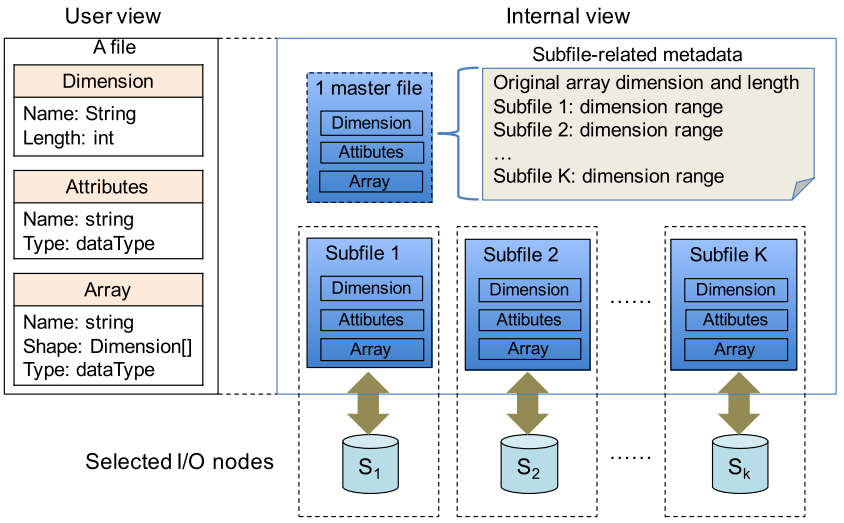

Fig. 6 Overview of our subfiling mechanism. At the higher-level I/O library, a file is represented as multidimensional arrays. Arrays also have attributes and may share dimension. In our approach, each array is internally divided into K subfiles; each is stored in a single I/O node. All files (both master and subfiles) are in self-describing file format

on this API usage sequence, we decide to include the partitioning mechanism when array definition is finished and the data in memory are ready for write/read; in other words, just before step 4 above. We choose this time because number of dimensions, length of each dimension, and datatype of each element for each array are finalized at this point. The header information is also written at the end of partitioning. To enable subfiling, users need to express their intention through the MPI hint mechanism.

The default partitioning policy is along the most significant dimension. For example, an array of Z–Y–X dimension, each with the same dimension size will be partitioned along the dimension Z. There are, however, certain applications that prevent applying the default policy. For instance, in the S3D application, the dataset called `u` is a four-dimensional array with the most significant dimension length 3. Such a small dimension length limits the number of subfiles, preventing the application from exploiting potential benefits of partitioning in larger subfile counts.

When partitioning along a dimension other than the most significant dimension is needed, application writers can specify the dimension name using a hint, called `par_dim_id`. When this hint is given to our subfiing module, it converts the ID into the index of the dimension defined in the array. We use the dimension ID because it can be reused by any array definition with a different order of the dimension list, making per-variable partitioning feasible.

Once all user's file partitioning strategies are delivered through the hint mechanism, our proposed module creates subfiles. The details of the file creation are as follows. First, it obtains all relevant hints using `MPI_Info_get()`. We convert those acquired information into metadata and store them in both master and the subfile's header information. If no hints were provided regarding file partitions, the normal procedure

Algorithm 2: Algorithm illustrating our subfiling mechanism. A file is partitioned only when partitioning is enabled through the MPI hint. Otherwise, normal files will be created.

```
get user's hints about subfiling;
if subfiling is enabled then
    obtain N' by executing Algorithm 1;
    determine MPI_Comm split parameters;
    /* split the original communicator into subcommunicators      */
    MPI_Comm_split (..., color, ..., &subcomm);
    /* create a subfile                                           */
    MPI_Info_set(info, "romio_lustre_start_iodevice", offset);
    MPI_Info_set(info, "striping_factor", "1");
    create a subfile associated with it;
    for each array, A_i do
        get par_dim_id;
        for dim_id, d[i][j] in A_i do
            dim_sz = d[i][j]→size;
            if j == par_dim_id then
                dim_sz = d[i][j]→size / N';
                define a new dimension using dim_sz and d[i][j];
                for each subfile, k do
                    create metadata for partition range, R_j;
                    store R_j to the master file;
        /* master file: replace the original var with scalar value */
        A_i →ndims_org = A_i →ndims;
        A_i →ndims = 0;
        A_i →dimids = NULL;
        define an array with newly-defined dimension;
else
    execute the normal array definition procedure;
```

will be executed; it creates a normal single file without partitions. Otherwise, it splits the original MPI communicator because each process is divided into a subprocess group. The split processes then collectively create their own subfile using a dataset function available in the high-level I/O library, for instance, `ncmpi_create` in PnetCDF. After a subfile is created, our algorithm traverses each array definition and determines which dimension ID it needs to use for partitioning. Unless the user gives a hint for the dimension ID for partitioning, the default is the first dimension ID for an array. It calculates a new dimension size for each partition. Only the partitioning dimension will be affected; all the remaining dimensions will have the same size as the original. Once a new dimension size is determined, we define a new dimension for the subfile and create an array with the new dimension lists. If the array is partitioned, we update the original array definition in the master file using a scalar value. This is because the master file does not have a physical space allocated for the partitioned array as the actual data will be stored in the subfiles. We repeat these steps until all arrays in the original file are processed. The algorithm described so far is given in Algorithm 2.

In default, we partition all arrays defined in a file. There are, however, several situations where this may not be an ideal case. First, in real applications, there are

certain arrays merely for a bookkeeping purpose, typically through associated attribute fields, instead of storing actual data. Those arrays are ones that application writers typically use to store the information needed to either restart the simulation or visualize plot files later. While our subfiling mechanism provides transparent access to those datasets, it is better to store this information in a single file. Second, arrays with fewer number of dimensions often do not contribute a significant portion to the total dataset. For instance, if there are two arrays, one 3D and the other 2D, the 2D array is less than 10% of total dataset size assuming that the length of all dimensions is the same and at least 10. In this case, partitioning the 2D array may not be a good idea because it will result in more, but smaller I/O requests. To handle these cases, our module accepts another hint, called `min_ndimes`, to allow selective array partitioning. This hint limits applying subfiling for arrays with a dimension length equal to or less than `min_ndims`.

Our main mechanism to convey the user's partitioning strategy is through MPI hints. We, however, also need to minimize the number of MPI hints, especially when applications run with larger process counts. This is because most high-level I/O libraries internally include all-to-all communication in order to make sure that the hint information conveyed to each process is the same across all processes. Therefore, we also use an environment variable so that the overhead incurred by our approach is minimized with larger process runs.

3.3 Memory-to-file layout transformation

The created subfiles need to be accessed transparently as, from an application's viewpoint, all I/O accesses go through the original file (i.e., master file). The metadata in both master and subfiles have sufficient information such as the number of subfiles, how each array is partitioned among subfiles, etc. Therefore, no modification is needed in the application codes. We also note that reading data from subfiles can be also done transparently by retrieving the same metadata internally.

Figure 7 shows our transformation mechanism from memory to file layout. This transformation has two steps: (1) calculating each process's requests to subfiles and

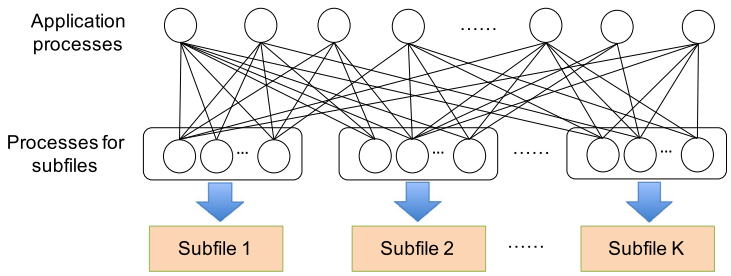

Fig. 7 Any I/O requests to a file partition not owned by processes in the same group need to be communicated before actual I/O requests are made. All these data exchange would require sufficiently complicated communication among processes. This approach is similar to application-level two-phase I/O mechanisms [12,13] while ours is transparent to applications

Algorithm 3: Algorithm for handling I/O requests to subfiles transparently.

```
Input: A_i, start[], count[], stride[], buf, bufcount
initialize my_req[] and others_req[];
if A_i is partitioned then
    for each partition, F_i do
        for each dim, D_j ∈ A_i do
            retrieve partition_index and par_dim_id from stored metadata;
            retrieve partition range from stored metadata;
            /* determine my_req[].start[] and my_req[].count[]        */
            if j == par_dim_id then
                my_req[i].start = start[j] ∩ range[D_j];
                my_req[i].count = count[j] ∩ range[D_j];
            else
                my_req[i].start = start[j];
                my_req[i].count = count[j];
    /* communicate my_req among all processes                         */
    MPI_Alltoall (my_req, ..., others_req, ... );
    /* exchange buf                                                   */
    for each process, i do
        if others_req[i].count != -1 && i != myrank) then
            MPI_Irecv (xbuf[i], ...);
    for each process, i do
        if others_req[i].count != -1 && i != myrank then
            MPI_Isend (buf, ...);
    MPI_Waitall (...);        /* wait until all buffers are exchanged */
    /* issue all I/O requests belonging to my rank                    */
    for each process, i do
        if my_req[i].count != -1 then
            call nonblocking I/O for buf belonging to local process;
        if others_req[i].count != -1 && i != myrank then
            call nonblocking I/O for xbuf[i] on behalf of remote processes;
    wait until all I/O requests are finished;
else
    proceed to the normal I/O routine;
```

exchanging them among all processes; (2) exchanging requests among processes in each split communicator and making I/O requests using I/O calls. The I/O calls can be either synchronous or asynchronous. We represent all I/O requests as start, count, and stride offset list for each dimension because our partitioning mechanism is implemented at the higher-level I/O library layer.

In Step 1, each process first calculates the list of start and count offsets to each subfile, dividing the data in memory among the processes who own the partitions. This is done by logically dividing the start and count offset, denoted as `my_req[]`, into file partitions, each can be directly accessed by the processes within a sub-communicator. In our implementation, we do not restrict the number of such delegate processes in each subprocess group. Any process in fact can be a delegate such that it will not create load imbalance at an application layer by selecting limited number of delegates because non-delegate processes do not read/write files directly. This phase requires one call of `MPI_Allreduce()` among all processes.

Step 2 starts with everyone's `my_req`, and calculates the portion of requests by other processes that belong to this process's file partitions. `others_req[i].{start,count}` indicates how many noncontiguous requests from process *i* accessing this process's file partition. All these calculations require one `MPI_Alltoall` and multiples of `isend`, `irecv`, and `wait_all`, but is required to ensure that delegates collect the request information from all other processes.

After that, each process sends requests to the delegate(s) in other communicators. Only delegates may have multiple I/O requests because non-delegate processes will not participate in this case. Non-delegate processes instead will call data exchange routines if they have requests to delegates. Delegate processes iterate until they receive requests from all other processes, and issue a non-blocking I/O. Each iteration goes through all `others_req[*]` and continues until all requests are processed. We ensure that they are all processed by calling `wait_all()` at the I/O library layer. The procedure described so far is given in Algorithm 3.

4 Extending the PnetCDF library

We describe our modifications to the PnetCDF library to implement the mechanisms described in Sects. 3.2 and 3.3. The probing mechanism described in Sect. 3.1 is not dependent on specific I/O libraries and, thus, it can be implemented separately.

4.1 PnetCDF subfiling

Figure 8 shows an example of PnetCDF code that includes the sequence of dimension and array (variable) definition followed by the write calls. In PnetCDF, all processes in the communicator must make an explicit call (`ncmpi_enddef`) at the end of the define mode to ensure that the values passed in by all processes match. From our design perspective, this is the time when all dimensions of arrays are known; therefore, our array partitioning is performed at the end of this call.

The NetCDF header generated by this example code is given in Fig. 9. After partitioning, both master and subfiles have additional attributes than the original file. For instance, the master file (Fig. 9b) has global attributes that indicate the file name for each subfile, the number of partitions, and the original dimension size for an array, "cube". The header for subfiles, on the other hand, has attributes for describing the range of partitioned dimension as well as the subfile index. All these additional metadata will be used in directing I/O requests to proper files, either master or subfiles.

We use a per-variable attribute to specify which dimension a variable needs to be partitioned along. As an example, if a user wanted to use the second most significant dimension than the default in the "cube" array, this option can be transferred to our subfiling module by adding the `ncmpi_put_att_int` API call with "par_dim_id" set to cube_dim[1]. This mechanism also can be used for handling record variables. Since record variables use unlimited dimensions as the most significant dimension (defined as UNLIMITED), we cannot determine the partition size. Therefore, we have to choose the second most-significant dimension as the partitioning one for record variables. For example, let us assume that there is a variable, xytime, with

```
MPI_Info_set (info, "nc_subfiling_enabled", "true");
ncmpi_create(comm, ..., info, &ncid);
...
/* dimension definition */
ncmpi_def_dim(ncid, "z", 100L, &cube_dim[0]);
ncmpi_def_dim(ncid, "y", 100L, &cube_dim[1]);
ncmpi_def_dim(ncid, "x", 100L, &cube_dim[2]);
...
/* variable (array) definition */
ncmpi_def_var(ncid, "cube", NC_INT, 3, cube_dim, &cube_id);
...
ncmpi_enddef();
...
/* perform I/O */
ncmpi_put_vara_all(ncid, cube_id, start[], count[], buf, bufcount, MPI_INT);
...
```

Fig. 8 A PnetCDF example code that creates a file with subfiling enabled. The number of subfiles is determined through the probing mechanism explained in Sect. 3.1. This example creates a variable (array) named "cube" of Z–Y–X dimension, each with 100 length. From an application writer's viewpoint, it only requires to add an MPI hint, `nc_subfiling_enabled`, to enable subfiling

```
netcdf test {
dimensions:
  z = 100;
  y = 100;
  x = 100;
variables:
  double cube(z, y, x);

// global attributes:

data:
  cube = ..... ;
}
```

(a)

```
netcdf test {
dimensions:
  z = 100;
  y = 100;
  x = 100;
variables:
  double cube;
    cube: num_subfiles=2;
    cube: ndims_org=3;

// global attributes:
  :subfile 0: "test.0";
  :subfile 1: "test.1";

data:
   cube = 0;
}
```

(b)

```
netcdf test.0 {
dimensions:
  z.cube = 50;
  y.cube = 100;
  x.cube = 100;
variables:
  double cube(z.cube,
              y.cube,
              x.cube);
    cube: range(z)=0,49;

// global attributes:
  :subfile_index = 0;

data:
   cube =  ..... ;
}
```

(c)

```
netcdf test.1 {
dimensions:
  z.cube = 50;
  y.cube = 100;
  x.cube = 100;
variables:
  double cube(z.cube,
              y.cube,
              x.cube);
    cube: range(z)=50,99;

// global attributes:
  :subfile_index = 1;

data:
  cube = ..... ;
}
```

(d)

Fig. 9 NetCDF file header information dumped by `ncmpidump` when the file is divided into 2. **a** Original NetCDF file (i.e., a normal file without subfiling). **b** The master NetCDF file after subfiling. The data section is 0, meaning empty. **c** First subfile. **d** Second subfile

three dimensions: time, x, and y. Let us assume further that we define the last two dimensions, x and y, as 100 and time as UNLIMITED. In this case, the variable `xytime` can be partitioned along either the x or y dimension.

4.2 Coordinating I/O among subfiles

While there are several data mode functions available in PnetCDF, we focus on the collective versions of those functions in our implementation. An example of such functions that write a variable "cube" is `ncmpi_put_vara_all` shown in Fig. 8. In this function, the varid (i.e., cube_id), start, count, and stride values (not used in above example) refer to the data in the file whereas buf, bufcount, and datatype (`MPI_INT`) refer to data in memory. When this API gets called, our subfiling module intercepts it transparently and coordinates the data movement between memory to

subfiles using the algorithm described in Algorithm 3. When each process issues I/Os to its own subfile, we use a non-blocking API available in PnetCDF. They will allow PnetCDF to aggregate multiple smaller, noncontiguous requests to generate larger ones for better I/O performance. These routines follow the MPI model of posting operations, then waiting for completion of those operations.

We illustrate how I/O requests to the subfiles are processed using the example code in Fig. 8. Let us assume that 4 processes access this shared array and the number of subfiles is set to 2. Each I/O request can be expressed as start offset, count and stride for each dimension. Since the array is in 3 dimensional, we have start[3], count[3], and stride[3]. For illustrative purposes, let us assume that stride count is 1, meaning that all array elements are accessed contiguously. In this case, one subfile (50 by 100 by 100) is owned by P_0 and P_1 whereas the other subfile (50 by 100 by 100) is owned by P_2 and P_3. Assuming a block–block access pattern and user's file partition, we calculate each process's request to each subfile. For instance, P_0's original request, denoted as start{0, 0, 0} and count{100, 50, 50}, is now divided into two portions: a portion belonging to its own subfile (denoted as start{0, 0 ,0} and count{50, 50, 50}) and the other (denoted as start{50, 0, 0} and count{50, 50, 50}) to be sent to the remote process that owns that subfile. Once all this information is obtained, all processes now exchange information (using all-to-all communication) and determine which process has a portion of the data not belonging to its own subfile. Afterwards, all processes have knowledge of which sub-I/Os they need to handle by themselves. The code then communicates the corresponding buffers and issues all those received I/O requests using PnetCDF's nonblocking I/O calls. The I/O to subfiles returns when all the issued nonblocking I/O calls are completed.

Our discussion so far assumes that the buffer in memory is contiguous. Real applications, however, often take advantage of MPI-derived datatype as a method to define arbitrary collections of noncontiguous data in memory and to transfer it to the file in a single MPI-IO call. To handle user buffers in derived datatypes, our memory-to-file data layout transformation first packs the noncontiguous buffer to a contiguous one before determining memory regions belonging to each subfile. Subsequent buffer exchange and issuing of nonblocking calls are all based on this contiguous buffer. We note that no conversion and byte swap are performed at this layer because they are done in PnetCDF layer underneath.

5 Evaluations

5.1 Experimental setup

All our experiments are performed on the Cray XE6 machine, Hopper, at NERSC. Hopper has a peak performance of 1.28 Petaflops/sec, 153,216 processors cores for running scientific applications, 212 TB of memory, and 2 Petabytes of online disk storage. The Hopper system has two locally attached high-performance scratch disk spaces, `/scratch` and `/scratch2`, each of 1 PB capacity. They both have the same configuration: 26 OSSs (Object Storage Servers), each of which hosts 6 OSTs (Object Storage Target), making a total of 156 OSTs. The PFS deployed in Hopper is

Lustre [32] mounted as both scratch disk spaces. When a file is created in /scratch, it is striped across two OSTs by default. Lustre provides users with a tunable striping configuration for a directory and files; both directory and files have the same striping configuration. In our experiment, we use all available OSTs for striping and 1 MB as default stripe sizes. We use the default number of aggregator nodes, which is automatically set to the stripe count of files. Therefore, without any tuning on the striping configuration, a file is striped on two randomly selected OSTs, which is accessed by two aggregator processes.

We implemented our proposed approach into the parallel netCDF 1.3.1. We added approximately 1,500 lines of new code to implement the subfiling feature in PnetCDF. Our implementation is configured to link with Cray's xt-mpich2 version 5.6.0. We used a separate ROMIO module described in [25] as a standalone library, which is then linked with the native MPI library. Our previous experience tells that this optimized ROMIO is about 30% faster than the system's default one. In other words, our base collective I/O performance is already optimized for our evaluation platform. All applications including benchmarks and our modified PnetCDF are compiled using PGI compiler version 12.9.0 with the "-fast" compilation flag.

For all experiments, we measure the I/O throughput as the number of bytes read or written by the benchmarks and applications during the time it took to complete. We observe that probing is one time cost, executed during file creation. Therefore, the impact of probing is negligible, less than 1% of the total I/O time in our observation. Since we measure the collective I/O time, this overhead is already included in the timings reported in this paper. We ran all experiments at least five times, and present the average of those runs with a standard deviation. We note that the proposed subfiling is designed to work with I/O systems showing variation due to multiple jobs or aggregators competing for shared I/O resources. If we ran sufficient larger jobs that occupy most of the available compute nodes, the opportunity of seeing such I/O contention shall decrease. For large-scale runs, we anticipate all available I/O nodes be selected to serve the I/O unless certain nodes are offline for an RAID rebuild or maintenance. While our goal was to measure the overall performance of the system for applications running during production, large variations in performance from week to week and even day to day were seen, most often due to storage component failures and firmware problems.

We evaluate our dynamic subfiling mechanism as compared with the base case, where all arrays are stored in a normal file (without subfiling) striped using all available OSTs. To show the effectiveness of our dynamic bandwidth probing, we ran two schemes of our subfiling cases: striped over all OSTs and striped over selected OSTs. While there are several other techniques whose goals are similar to ours like PLFS [3] and ADIOS [31], we do not compare our approach against them because fair comparison is hard to make; they do not preserve canonical order of the original dataset whereas all subfiles are also stored in portable NetCDF format. Unlike ours, PLFS is a system-level approach and does not select I/O nodes based on the workload to stripe files. In our partitioning mechanism, the number of generated subfiles is proportional to the number of aggregator processes, whereas the number of files generated by PLFS is proportional to the number of processes.

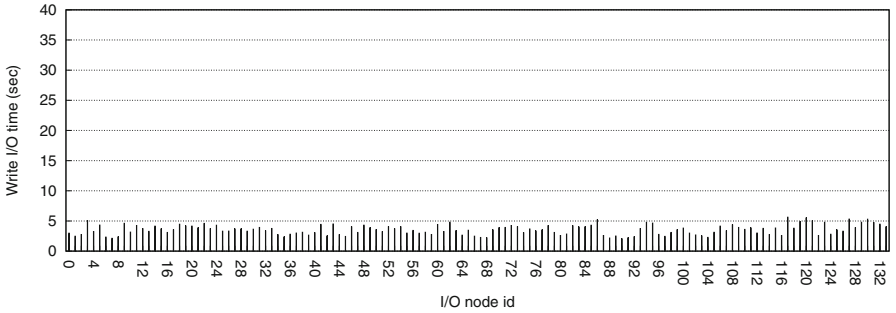

Fig. 10 Balanced write I/O time observed when only subset of I/O nodes that were detected through our dynamic bandwidth probing

Lastly, our proposed approach relies on a mechanism to change the stripe width (or factor) when a file is created. The hints for such mechanism are available at the high-level I/O libraries, e.g., PnetCDF or even MPI-IO, but the actual implementation is dependent on underlying PFSs. While lusture implements this mechanism, but GPFS does not. Therefore, we evaluate our mechanism on Lustre. This might be a limitation to our proposed system, but most HPC systems deploy Lustre, so our contribution is still meaningful for the majority of HPC community.

5.2 Collective I/O benchmark

Before presenting our evaluation with the collective I/O performance benchmark, we first show how effectively our approach can isolate slower I/O nodes. In order to do this, we wrote a small test code that writes 1GB of dummy data to each I/O node selected by our dynamic probing module. Figure 10 shows the write I/O time, collected using the TAU profiling tool [43], observed at each I/O node, which was selected by our probing module. In this result, 134 out of 156 OSTs were selected for writing. As compared with Fig. 2, all selected OSTs showed similar, balanced write I/O time.

To understand the performance of our approach against the base case, we ran a collective I/O test program, coll_perf, originally from ROMIO test suite, that writes and reads the three-dimensional arrays in a block-partitioned manner. We made it write/read four 3D variables. The application-level data partitioning is done by assigning a number of processes to each Cartesian dimension. In our experiments, we set the subarray size in each process to $128 \times 128 \times 128$ of 4-byte integers, corresponding to 8MB. All data are written to a single file for the base case (normal file). For our subfiling cases, all four variables are partitioned along the most significant dimension.

Figure 11 shows the write throughput of coll_perf with and without our subfiling mechanism. We scale both schemes by increasing the number of processes from 1,024 up to 8,192. The results indicate that writing data into a normal file does not scale well with larger number of processes; the write throughput actually went up and down when the number of processes are increased. On the other hand, our subfiling schemes improve the write throughput significantly by 12–94% when used with all 156 OSTs and 36–137% when used with selected OSTs, respectively.

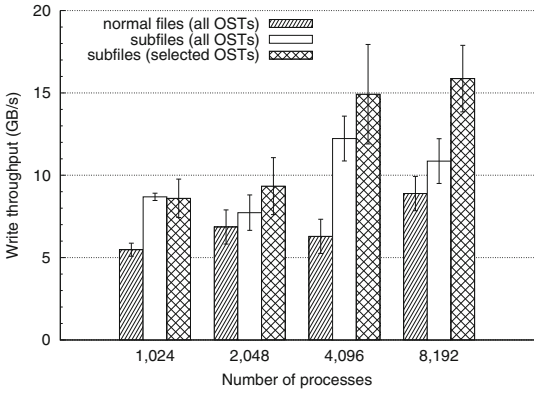

Fig. 11 Write throughput results for coll_perf

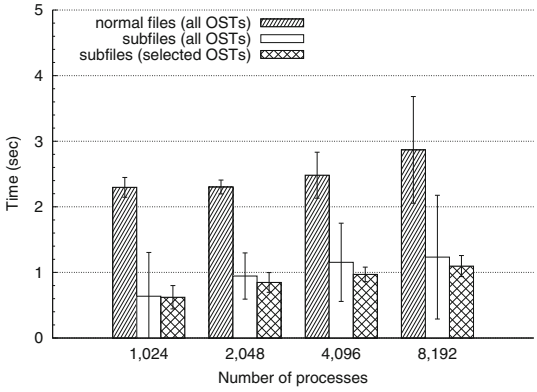

Fig. 12 The average write I/O time for coll_perf with *error bars*. Using all OSTs shows much higher deviation, whether files are partitioned or not

5.2.1 Performance breakdown

To understand the performance improvement by our approach in detail, we analyzed the performance of coll_perf using the TAU profiling tool [43]. Figure 12 illustrates that each aggregator's POSIX write() time increases slowly as the number of processes increases. This is because coll_perf performs the weak scaling test, so I/O time also increases with larger number of processes. The most remarkable observation in this result is that writing to subfiles reduces the write I/O time significantly regardless of using all OSTs or selected OSTs. The average write time improvement is about 70%, which clearly indicates that enabling subfiles minimizes the impact of the contention on the file server. Another notable insight from this result is that the write I/O time has high deviations when all OSTs are used, and the variations increased with larger process counts. The subfiles with selected OSTs show low deviations mainly because relatively slower OSTs were effectively eliminated before the time of writing.

Read performance In our next experiments, we would like to understand how the read from subfiles performs. To do this, we run the same weak scalability tests (1024 to 8192 processes) on the read case; each case collectively reads the entire files in a

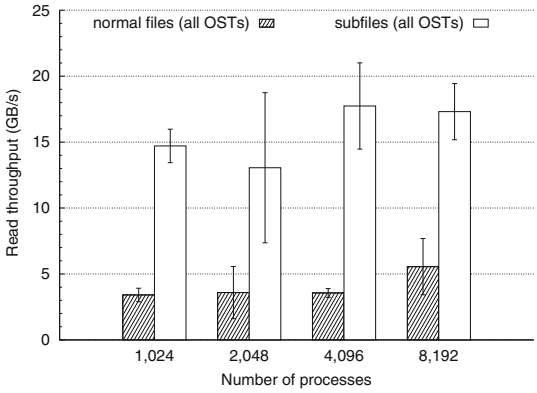

Fig. 13 Read throughput results for coll_perf

block–block partitioned manner. Since partitioning on selected OSTs does not have fixed the number of OSTs per run, we evaluate only reading from all OSTs for a fair comparison. In other words, we simply would like to compare our subfiling approach with the traditional approach, which stripes file across all OSTs. For the same reason, we do not perform the bandwidth probing to detect individual I/O server's performance as we have to read the data regardless of the existence of I/O variation unless the files are replicated. To ensure data are read from the storage nodes, all caches are flushed before each run.

Figure 13 shows that the normal (i.e., without subfiling) file case is not scalable, while our subfiling scheme shows much higher performance improvement than the write case. Also, the observed read throughput is about 30% lower than that of the write throughput. Our TAU profiling result indicates a notable increase in read I/O time; reading from the normal file is about 6× slower than reading from subfiles. We attribute this to the pretty aggressive read-ahead mechanism used in Lustre file system. In the case of reading from non-subfiled files on all OSTs and given the default stripe size of 1MB, the majority of prefetched data by an aggregator is irrelevant parts of the data, thus slowing down the overall performance. In our subfile case, the read-ahead mechanism is entirely reading from a single OST, so the benefit of read-ahead is maximized. We note that the read-ahead is per-file basis. Because of our 1-to-1 mapping between subfiles and I/O nodes, there are no data prefetched from a different OST; in other words, there are no unused read-ahead data.

Overhead Our subfiling approach introduces additional communication during memory-to-file layout transformation time: `MPI_Isend`, `MPI_Irecv()`, `MPI_Alltoall()`, and `MPI_wait()`. To quantify this overhead, we have measured time spent on those additional communication costs using TAU. The results indicate that the coordination overhead incurred by the additional communication is negligible; the extra communication overhead accounts for less 1% of the collective I/O operations. The time spent on the all-to-all communication is small because, during that phase, we only exchange each process's requests to each subfile. The buffer exchange phase also does not incur much overhead because only partic-

ipating process pairs exchange small amount of buffer. Since our algorithm selects the delegation process in other subprocess groups in a balanced manner, the pairwise communication is also mostly balanced.

5.3 Application I/O performance

5.3.1 FLASH

FLASH [22,62] is 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 [11]. The problem domain is divided into blocks distributed among a number of processes. A block is a three-dimensional array with 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 process is used to generate a slightly unbalanced I/O load. Because of the fixed number of blocks for each process, an increase in the number of processes linearly increases the aggregate I/O amount as well. The main I/O routine in FLASH is to write checkpoint files and plot files for visualization, which contain centered and corner data. Checkpoint files are the largest of the three output data sets, the I/O time of which dominates the entire I/O routines. We set the block size to be $16 \times 16 \times 16$, which is about 64 MB of data per process. All 24 variables are written using collective write calls. During write, every process writes a contiguous chunk of a variable, appended to the data written by the previous ranked process. In our experiments, we have evaluated the performance of writhing both checkpoint and plot files.

To understand how we partitioned the data, we next describe the FLASH I/O format. By default, all mesh variables (including density, pressure and temperature) are written to the same dataset (variable) in the output file. This file format is used for both checkpointing and plot files. In case of checkpoint files, only 10 variables (out of entire 24) are those mesh variables, each of which is a four-dimensional (4D) array of double-precision. All unknown variables are defined as a 5D array, the first dimension being the number of unknown variables. Since this dimension length is only 10, we partition these unknown variables along the second most significant dimension. We partition only unknown variables in our experiments; therefore, other variables are stored in the master file without subfiling. The plot files have three mesh variables and we again applied subfiling for the unknown variables only.

Figure 14 shows the aggregate I/O bandwidth of FLASH for the non-subfiled case and our two approaches. As expected, using a non-subfiled file did not scale well with increased process counts. The maximum aggregate I/O bandwidth we observed is about 8 GB/s when it is run with 8,192 processes. This is significantly below the maximum I/O bandwidth on Hopper. The subfiles with all OSTs slightly outperform the non-subfiled case, by 28% on average, but there is higher deviation with larger process counts. Overall, the subfiles with selected OSTs can achieve about 70% I/O bandwidth improvement than the non-subfiled case.

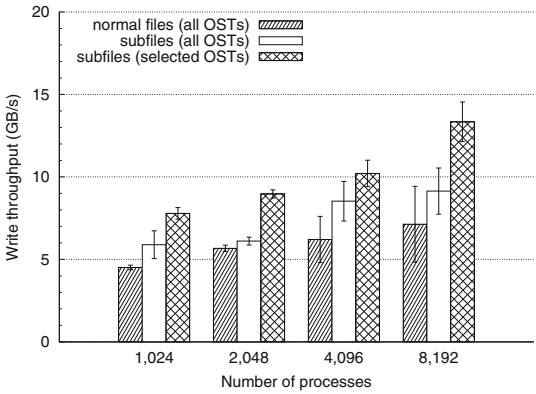

Fig. 14 FLASH I/O write throughput

5.3.2 S3D

S3D [39] simulates turbulent combustion using direct numerical simulation of a comprehensive Navier–Stokes flow. The domain is decomposed among processes in 3D. All processes periodically participate in writing out a restart file. This file can be used both as a mechanism to resume computation and as an input for visualization and post-simulation analysis. We used $50 \times 50 \times 50$ fixed subarrays.

The checkpoint files consist of four global arrays: two 3-dimensional, temp (z, y, x) and pressure (z, y, x) in double precision, and two 4-dimensional arrays [double yspecies (nsc, z, y, x) and double u (three, z, y, x)]. Since the sizes of the most significant dimension in 4D variables are relatively small, 3 and 11 for `three` and `nsc`, respectively, we partition these variables along the second most significant dimension, z.

Figure 15 shows the aggregate I/O bandwidth of S3D for all three cases we evaluated. We have observed that the non-subfiled file case scales to only a limited extent. We note that all MPI-IO optimizations such as collective buffering including aggregation are already applied to the base case as enabling those MPI-IO features is known

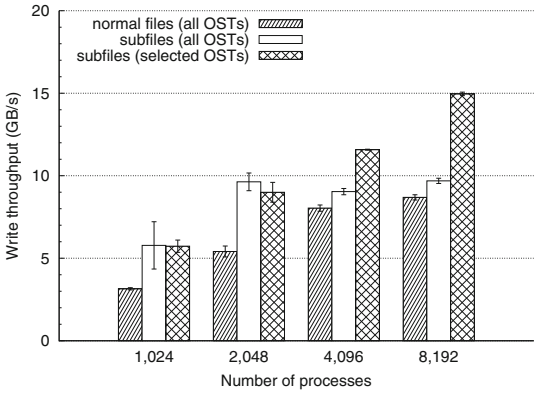

Fig. 15 S3D I/O write throughput

to give the best possible performance when testing with PnetCDF output to a single file using collective I/O [23,26,47]. Therefore, we attribute this marginal scalability to I/O interference at the file servers. The subfiles using all OSTs can achieve higher performance improvement than the non-subfiled file case up to 2048 processes, but only marginal improvement beyond that point. The subfiles case with selected OSTs, on the other hand, consistently outperforms than the non-subfiled file case, by 60% on average. We also observed that the subfiles with all OSTs performed better than the subfiles with selected OSTs in case of 1,024 and 2,048 processes. This is because when the smaller sized jobs were run, there were less chances of interfering with other applications' I/O because I/O time is relatively short. The reason why the subfile with selected OSTs did not perform in 1,024 and 2,048 process cases is that there were fairly higher I/O variation between the time the probing module is executed and the time actual I/O occurred. The subfile with selected OSTs, however, still showed less variations on the observed write bandwidth although the peak bandwidth suffered from unexpected slower I/O performance on certain I/O nodes.

5.3.3 GCRM

The GCRM is a climate application framework designed to simulate the circulations associated with large convective clouds [37]. It uses geodesic parallel I/O (GIO) library, which interfaces PnetCDF. In our experiments, we exclude non-grid variables, as they are written individually, which generates uneven file access degrees. GCRM-IO partitions a semi-structured geodesic mesh between processes, each of which writes a sub-block of the mesh in its partition. There are 38 grid variables, each of which is approximately evenly partitioned among all processes. The grid and sub-domain resolution are controlled by the user and we evaluate 4 cases: 640 processed with resolution level 8, 2560 processes with level 9, 10,240 processes with level 10, and 40,960 processes with resolution level 11. The resolution levels correspond to the geodesic grid refinement at about 31.27, 15.64, 7.819, and 3.909 km, respectively. The GCRM is run with the direct I/O mode and all grid variables are written to a single file for both base and our approach. When our partitioning is applied, all variables are partitioned along with the second most significant dimension because all variables in GCRM are record variables.

Figure 16 shows the I/O throughout results for GCRM with and without our approach. Again, we have observed that I/O throughput without subfiles suffers from I/O variability significantly while both subfile cases scale with increased number of processes. We also observed that, while the impact of subfiles is similar to S3D, GCRM shows more consistent results among different schemes. We note that GCRM is already optimized for high I/O rates while still writing to shared files. That is, the aggregator processes consolidate writes into large chunks of data, making additional improvements in the underlying MPI-IO libraries and PFS easier. Therefore, in Fig. 16, the performance differences are mainly from the impact of I/O variability. Specifically, the normal files with all OSTs exhibit a significant bandwidth drop in case of 40,960 processes because the file is striped across all OSTs. While the subfiles with all OSTs are more scalable than without subfiles, it shows a huge variation on the observed bandwidth. The subfile with selected OSTs showed not only scalable write bandwidth

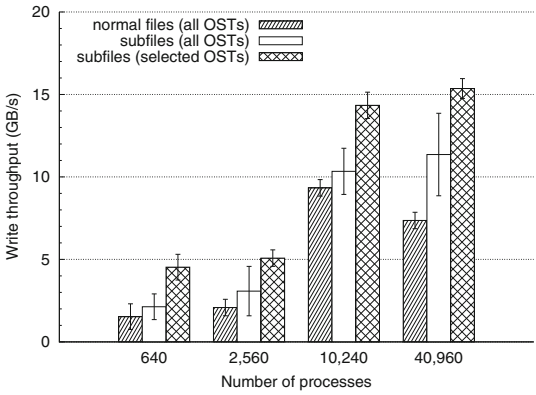

Fig. 16 GCRM I/O bandwidth

but also very less variation on the observed write bandwidth. These results clearly demonstrate that our approach is less susceptible to I/O variability.

6 Related work

PLFS [3] introduced a virtual layer that remaps an application's preferred data layout into one optimized for the underlying PFS. Like PLFS, split writing and hierarchical striping [59] also use a library approach to reduce contention from concurrent access at runtime. However, the split files are merged at close time, preventing later accesses from leveraging the benefits of subfiles. It also requires application modification. Yu and Vetter proposed an augmented collective I/O, called ParColl, with file area partitioning and I/O aggregator distribution [58]. PIDX [19,20] is a parallelization of IDX data format, and uses a novel aggregation technique to improve its scalability. Dickens and Logan [7] demonstrated that the collective I/O operations on Lustre perform poorly because of high communication overhead to make and write large, contiguous blocks of data. They then proposed a new approach, called Y-Lib, to collective I/O in Lustre, which improves performance by reducing contention among processes participating in collective operations.

Our earlier study by Gao et al. [14] is similar to our approach, but it requires user intervention of how each subfile is partitioned using a set of new APIs. Also, it only allows partitioning along the most significant dimensions of an array, and does not support record variables. In our new design and implementation, we remove these restrictions to enable any further layout transformation between memory and subfiles. All these data transformations would require sufficiently complicated communication among processes, which does not occur in the subfiling. Further, unlike the subfiling, our approach gives more flexibility by allowing application writers to specify per-variable partitioning. ADIOS's BP file format [31] is similar to our subfiling, but ADIOS is restricted in selecting how the data are stored across subfiles. Fu et al. [12,13] proposed an application-level two-phase I/O, called reduced-blocking I/O (rbIO), and demonstrated that rbIO performs better than the n to n approach. rbIO is similar to our approach in that it reduces conflicts using the subfiles and application 2-phase I/O.

However, the partition in rbIO is done by the application writers, and the coordination does not cross the partitioned process group. Kendall et al. also used an application-level 2-phase I/O in order to organize I/O requests to multiple-file dataset [16]. Their optimization, however, is targeted mainly for visualization workloads, and application writers manually provide the list of starts and sizes of a block that each process needs to read or write.

Many recent studies have identified that staggering file servers are one of the main reasons of inconsistent I/O performance in large petascale and beyond systems [4,30,55]. Xie et al. [55] characterizes the I/O bottlenecks in supercomputers, and it demonstrates that slower I/O servers limit the aggregate and striping bandwidth and reduce the parallelism. Also, due to locking protocols, lower bandwidths are observed while writing to a shared file. In [30], it is shown that the I/O load variation on I/O servers leads to performance degradation, and adaptive I/O methods are proposed using a grouping approach to balance the workload; i.e., for a group of writer processes, assign a sub-coordinator to each group, and assign a coordinator for all the sub-coordinators. In a recent study on Hopper [4], it is shown that once the I/O stragglers are isolated from the I/O, and using one file for all processes, the performance can be significantly improved. But, this approach is a system-level solution and requires admin privilege to remove stragglers from the file system. On the other hand, our solution is completely at user-level. Automatic storage contention alleviation and reduction (ASCAR) system [24] is a storage traffic management system for improving the bandwidth utilization and fairness of resource allocation. ASCAR controls I/O traffic from the clients using a rule-based algorithm that manages the congestion window and rate limit. Dai et al. proposed a two-choice randomized dynamic I/O scheduler that schedules the concurrent I/O operations in a balanced way to avoid slower I/O servers, thereby achieving high throughput [6]. Our approach does take the slower I/O servers into account and dynamically isolates these servers from the collective I/O operation. Using one partition per file server can potentially achieve better performance by minimizing file system locking contention.

In recent HPC system designs, a new storage layer of faster storage medium such as non-volatile memory, called a burst buffer (BB), has been introduced to reduce the contention on the PFSs [2,28,29,40,54]. While BBs are certainly effective layers to bridge the gap between applications and file systems, it can still suffer from the interference. To address this problem, recent studies proposed several HPC I/O scheduling mechanisms [10,50]. Wachs et al. [52] proposed the Argon storage server that explicitly manages I/O time slice among sharing jobs. Song et al. [45] applied similar time slice coordination across PFS servers, thereby reducing interference in PFS and achieving QoS of I/O-intensive applications [60]. A high-level scheduler (such as system-wide schedule [49], or direct coordination between applications [8]) can also manage inter-application interference by coordinating per-application I/O behaviors, such as write I/O during checkpoint. A system wide I/O scheduler can also effectively manage I/O workloads using additional information, such as runtime profiling, that can be extracted from the I/O applications. Thapaliya et al. [50] showed that adapting the scheduler to the BB usage model and workloads is crucial to reduce contention on the BB nodes.

To the best of our knowledge, no prior studies discussed how transparently store canonically ordered multidimensional dataset into a set of partitioned files, which themselves are also in self-describing formats. Our partitioning mechanism requires almost no modifications to existing applications except providing an MPI hint conveyed to our proposed software module within PnetCDF. Further, our application-level file partition and data exchange mechanism automatically coordinate collective I/O requests across partitioned files, reducing the overhead otherwise incurred to create large, contiguous I/O requests.

7 Conclusion and future work

We propose a file partitioning approach to accomplishing scalable collective I/O performance while keeping a conventional way of accessing large multi-dimensional arrays to a user. We employ a dynamic probing technique to estimate relatively slower I/O nodes and isolate the impact of those nodes, which are the limiting factors to achieve scalable collective I/O performance at scale. We implement the proposed technique in PnetCDF and evaluate its effectiveness using several benchmarks and real I/O-intensive applications on NERSC's Hopper. Our experimental results demonstrate that the proposed technique consistently improves the collective I/O performance significantly by reducing write I/O time with less variation.

We will continue to evaluate our approach on other platforms like Mira, IBM Blue Gene/Q, at Argonne National Laboratory [1], and other high-level I/O libraries. We also plan to compare our approach with similar ones implemented at application layers or another interface layers such as PIO (parallel I/O) library [9] included in Community Earth System Model (CESM). Future research will focus on investigating how the data exchange mechanism we proposed in this paper can be applied to more general layout transformation techniques like transposing array dimensions.

Acknowledgements This work is supported in part by the following grants: NSF awards CCF-0833131, CNS-0830927, IIS-0905205, CCF-0938000, CCF-1029166, and OCI-1144061; DOE awards DE-FG02-08ER25848, DE-SC0001283, DE-SC0005309, DESC0005340, and DESC0007456; AFOSR award FA9550-12-1-0458. 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.

References

1. Argonne Leadership Computing Facility. http://www.alcf.anl.gov/intrepid/
2. Bent J, Faibish S, Ahrens J, Grider G, Patchett J, Tzelnic P, Woodring J (2012) Jitter-free co-processing on a prototype exascale storage stack. In: IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST), pp 1–5
3. Bent J, Gibson G, Grider G, McClelland B, Nowoczynski P, Nunez J, Polte M, Wingate M (2009) PLFS: A checkpoint filesystem for parallel applications. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis
4. Byna S, Uselton A, Praphat Knaaky D, He YH (2013) Trillion particles, 120,000 cores, and 350 TBs: lessons learned from a hero I/O run on hopper. In: Cray user group meeting

5. Carns P, Latham R, Ross R, Iskra K, Lang S, Riley K (2009) 24/7 characterization of petascale I/O workloads. In: Proceedings of the First Workshop on Interfaces and Abstractions for Scientific Data Storage
6. Dai D, Chen Y, Kimpe D, Ross R (2014) Two-choice randomized dynamic I/O scheduler for object storage systems. International Conference for High Performance Computing, Networking, Storage and Analysis, pp 635–646
7. Dickens PM, Logan J (2009) Y-lib: a user level library to increase the performance of MPI-IO in a Lustre file system environment. In: Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing, pp 31–38
8. Dorier M, Antoniu G, Ross R, Kimpe D, Ibrahim S (2014) CALCioM: mitigating I/O interference in HPC systems through cross-application coordination. In: Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium, pp 155–164
9. Edwards J A high-level parallel I/O library for structured grid applications. https://github.com/NCAR/ParallelIO
10. Fang A, Chien AA (2015) How much ssd is useful for resilience in supercomputers. In: Proceedings of the 5th Workshop on Fault Tolerance for HPC at eXtreme Scale, pp 47–54
11. Fryxell B, Olson K, Ricker P, Timmes FX, Zingale M, Lamb DQ, MacNeice P, Rosner R, Truran JW, Tufo H (2000) FLASH: an adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes. Astrophys J Suppl Ser 131(1):273
12. Fu J, Liu N, Sahni O, Jansen KE, Shephard MS, Carothers CD (2010) Scalable parallel I/O alternatives for massively parallel partitioned solver systems. In: Proceedings of Workshop on Large-Scale Parallel Processing
13. Fu J, Min M, Latham R, Carothers CD (2011) Parallel I/O performance for application-level checkpointing on the blue gene/P system. In: Proceedings on Workshop on Interfaces and Architectures for Scientific Data Storage, pp 465–473
14. Gao K, Liao Wk, Nisar A, Choudhary A, Ross R, Latham R (2009) Using Subfiling to improve programming flexibility and performance of parallel shared-file I/O. In: Proceedings of the International Conference on Parallel Processing, pp 470–477
15. Gunasekaran R, Kim Y (2014) Feedback computing in leadership compute systems. In: 9th International Workshop on Feedback Computing (Feedback Computing 14). USENIX Association, Philadelphia, PA (2014). https://www.usenix.org/conference/feedbackcomputing14/workshop-program/presentation/gunasekaran
16. Kendall W, Huang J, Peterka T, Latham R, Ross R (2011) Visualization viewpoint: towards a general I/O layer for parallel visualization applications. IEEE Comput Graph Appl 31(6):6–10
17. Kim Y, Atchley S, Vallée GR, Shipman GM (2015) LADS: optimizing data transfers using layout-aware data scheduling. In: Proceedings of the 13th USENIX Conference on File and Storage Technologies, FAST'15, pp 67–80
18. Kotz D (1997) Disk-directed I/O for MIMD multiprocessors. ACM Trans Comput Syst 15(1):41–74
19. Kumar S, Vishwanath V, Carns P, Levine JA, Latham R, Scorzelli G, Kolla H, Grout R, Chen J, Ross R, Papka ME, Pascucci V (2012) Efficient data restructuring and aggregation for IO acceleration in PIDX. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis
20. Kumar S, Vishwanath V, Carns P, Summa B, Scorzelli G, Pascucci V, Ross R, Chen J, Kolla H, Grout R (2011) PIDX: Efficient parallel I/O for multi-resolution multi-dimensional scientific datasets. In: Proceedings of the 2011 IEEE International Conference on Cluster Computing, pp 103–111
21. Lang S, Carns P, Latham R, Ross R, Harms K, Allcock W (2009) I/O performance challenges at leadership scale. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, pp 40:1–40:12
22. Latham R, Daley C, Keng Liao W, Gao K, Ross R, Dubey A, Choudhary A (2012) A case study for scientific I/O: improving the FLASH astrophysics code. Comput Sci Discov 5(1):015, 001
23. Li J, Liao Wk, Choudhary A, Ross R, Thakur R, Gropp W, Latham R, Siegel A, Gallagher B, Zingale M (2003) Parallel netCDF: a high-performance scientific I/O interface. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis
24. Li Y, Lu X, Miller EL, Long DDE (2015) ASCAR: automating contention management for high-performance storage systems. In: IEEE 31st Symposium on Mass Storage Systems and Technologies, MSST, pp 1–16

25. Liao Wk, Choudhary A (2008) Dynamically adapting file domain partitioning methods for collective I/O based on underlying parallel file system locking protocols. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis
26. Liao WK, Coloma K, Choudhary A, Ward L, Russell E, Pundit N (2006) Scalable design and implementations for mpi parallel overlapping I/O. IEEE Trans Parallel Distrib Syst 17(11):1264–1276
27. Liao WK, Coloma K, Choudhary A, Ward L, Russell E, Tideman S (2005) Collective caching: application-aware client-side file caching. In: Proceedings of 14th IEEE International Symposium on High Performance Distributed Computing, pp 81–90
28. Liu N, Cope J, Carns PH, Carothers CD, Ross RB, Grider G, Crume A, Maltzahn C (2012) On the role of burst buffers in leadership-class storage systems. In: Proceedings of the IEEE Conference on Mass Storage Systems, pp 1–11
29. Lofstead J, Ross R (2013) Insights for exascale IO APIs from building a petascale IO API. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, pp 87:1–87:12
30. Lofstead J, Zheng F, Liu Q, Klasky S, Oldfield R, Kordenbrock T, Schwan K, Wolf M (2010) Managing variability in the IO performance of petascale storage systems. In: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC '10, pp 1–12
31. Lofstead JF, Klasky S, Schwan K, Podhorszki N, Jin C (2008) Flexible IO and Integration for scientific codes through the adaptable IO system (ADIOS). In: Proceedings of the 6th International Workshop on Challenges of Large Applications in Distributed Environments, pp 15–24
32. Lustre File System. http://www.lustre.org
33. Ma X, Winslett M, Lee J, Yu S (2003) Improving MPI-IO output performance with active buffering plus threads. In: Proceedings of the 17th International Symposium on Parallel and Distributed Processing
34. Message Passing Interface Forum. MPI-2: extensions to the message passing interface. http://www.mpi-forum.org/docs/docs.html
35. National Energy Research Scientific Computing Center. http://www.nersc.gov/users/computational-systems/hopper/
36. Park S, Shen K (2012) FIOS: a fair, efficient flash I/O scheduler. In: Proceedings of the 10th USENIX Conference on File and Storage Technologies, pp 13–13
37. Randall D, Khairoutdinov M, Arakawa A, Grabowski W (2003) Breaking the cloud parameterization deadlock. Bull Am Meteor Soc 84:1547–1564
38. del Rosario JM, Bordawekar R, Choudhary A (1993) Improved parallel I/O via a two-phase run-time access strategy. In: Proceedings of Workshop on Input/Output in Parallel Computer Systems, pp 56–70
39. Sankaran R, Hawkes ER, Chen JH, Lu T, Law CK (2006) Direct numerical simulations of turbulent lean premixed combustion. J Phys Conf Ser 46(1):38
40. Sato K, Mohror K, Moody A, Gamblin T, d. Supinski BR, Maruyama N, Matsuoka S (2014) A user-level infiniband-based file system and checkpoint strategy for burst buffers. In: 2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), pp 21–30
41. Schuchardt K, Palmer B, Daily J, Elsethagen T, Koontz A (2007) IO strategies and data services for petascale data sets from a global cloud resolving model. J Phys Conf Ser 78:012089
42. Seamons KE, Chen Y, Jones P, Jozwiak J, Winslett M (1995) Server-directed collective I/O in Panda. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis
43. Shende SS, Malony AD (2006) The TAU parallel performance system. Int J High Perform Comput Appl 20(2):287–311
44. Son SW, Sehrish S, k. Liao W, Oldfield R, Choudhary A (2013) Dynamic file striping and data layout transformation on parallel system with fluctuating I/O workload. In: 2013 IEEE International Conference on Cluster Computing (CLUSTER), pp 1–8
45. Song H, Yin Y, Sun XH, Thakur R, Lang S (2011) Server-side I/O coordination for parallel file systems. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (2011)
46. Tavakoli N, Dai D, Chen Y (2016) Log-assisted straggler-aware I/O scheduler for high-end computing. In: 2016 45th International Conference on Parallel Processing Workshops (ICPPW), pp 181–189. doi:10.1109/ICPPW.2016.38
47. Thakur R, Choudhary A (1996) An extended two-phase method for accessing sections of out-of-core arrays. Sci Progr 5(4):301–317

48. Thakur R, Gropp W, Lusk E (1999) Data sieving and collective I/O in ROMIO. In: Proceedings of the 7th Symposium on the Frontiers of Massively Parallel Computation
49. Thapaliya S, Bangalore P, Lofstead J, Mohror K, Moody A (2014) IO-Cop: managing concurrent accesses to shared parallel file system. In: 43rd International Conference on Parallel Processing Workshops, pp 52–60
50. Thapaliya S, Bangalore P, Lofstead J, Mohror K, Moody A (2016) Managing I/O interference in a shared burst buffer system. In: 2016 45th International Conference on Parallel Processing (ICPP), pp. 416–425
51. The HDF Group (2000–2010) Hierarchical data format version 5. http://www.hdfgroup.org/HDF5
52. Wachs M, Abd-El-Malek M, Thereska E, Ganger GR (2007) Argon: performance insulation for shared storage servers. In: Proceedings of the 5th USENIX Conference on File and Storage Technologies
53. Wang T, Oral S, Pritchard M, Wang B, Yu W (2015) TRIO: burst buffer based I/O orchestration. In: 2015 IEEE International Conference on Cluster Computing, pp 194–203
54. Wang T, Oral S, Wang Y, Settlemyer B, Atchley S, Yu W (2014) BurstMem: a high-performance burst buffer system for scientific applications. In: 2014 IEEE International Conference on Big Data (Big Data), pp 71–79
55. Xie B, Chase J, Dillow D, Drokin O, Klasky S, Oral S, Podhorszki N (2012) Characterizing output bottlenecks in a supercomputer. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, pp 8:1–8:11
56. Yildiz O, Dorier M, Ibrahim S, Ross R, Antoniu G (2016) On the root causes of cross-application I/O interference in HPC storage systems. In: 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp 750–759
57. Ying L (2008) Lustre ADIO collective write driver—white paper. Tech. rep, Sun and ORNL
58. Yu W, Vetter J (2008) ParColl: partitioned collective I/O on the cray XT. In: Proceedings of the 37th International Conference on Parallel Processing, pp 562–569
59. Yu W, Vetter J, Canon RS, Jiang S (2007) Exploiting lustre file joining for effective collective IO. In: Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid, pp 267–274
60. Zhang X, Davis K, Jiang S (2011) QoS support for end users of I/O-intensive applications using shared storage systems. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, pp 18:1–18:12
61. Zhou Z, Yang X, Zhao D, Rich P, Tang W, Wang J, Lan Z (2016) I/O-aware bandwidth allocation for petascale computing systems. Parallel Comput 58:107–116
62. Zingale M (2001) FLASH I/O benchmark routine—parallel HDF 5. http://www.ucolick.org/~zingale/flash_benchmark_io/