SAND2013-8853C

Dynamic File Striping and Data Layout Transformation on Parallel System with Fluctuating I/O Workload

Seung Woo Son
Northwestern University

Saba Sehrish
Northwestern University

Wei-keng Liao
Northwestern University

Ron Oldfield
Sandia National Laboratories

Alok Choudhary
Northwestern University

ABSTRACT

As the number of compute cores on modern parallel machines increases to more than hundreds of thousands, scalable and consistent I/O performance is becoming hard to obtain due to fluctuating file system performance. This fluctuation is often caused by rebuilding RAID disk from hardware failures or concurrent jobs competing for I/O. We present a mechanism that stripes across a dynamically-selected subset of file servers with the lightest workload to achieve the best I/O bandwidth available from the system. We implement this mechanism into an I/O software layer that enables memory-to-file data layout transformation and allows transparent file partitioning. File partitioning is a technique that divides data among a set of files and manages file access, making data appear as a single file to users. Experimental results on NERSC's Hopper indicate that our approach effectively isolates I/O variation on shared systems and improves overall I/O performance significantly.

Categories and Subject Descriptors

B.4.3 [**Input/Output and Data Communications**]: Interconnections (Subsystems)—*Parallel I/O*

General Terms

Algorithm, Experimentation, Performance

Keywords

Parallel I/O, Collective I/O, PnetCDF, File Partitioning

1. INTRODUCTION

Scientists and engineers are increasingly using modern parallel machines in order to run their large, often data-intensive applications, such as thermonuclear reactions, combustion, climate modeling, and so on [9, 27, 28, 29]. Scalable parallel I/O libraries are one of the key components to scaling those applications [6, 17]. The I/O requirements of such applications can be staggering, ranging from terabytes to petabytes, and managing such massive data sets presents a significant bottleneck [8, 18].

In many parallel applications, the problem domain is expressed by a global data structure in a multidimensional array form and partitioned among all processes, thereby making each process run in parallel on sub-domains. There are many approaches proposed to coordinate I/O requests from multiple processes, and collective I/O in MPI-IO [26] has been widely used to allow collaboration among participating processes and rearrange their I/O requests to achieve high performance. There have been many optimizations to improve collective I/O performance, including two-phase I/O [7, 32], disk-directed I/O [14], server-directed I/O [30], persistent file domain [21], active buffering [25], collaborative caching [22], and adaptive file domain [20]. Even with these improvements, however, collective I/O operations in large-scale are facing new challenges on modern parallel machines. As the scale of parallel machines grows, various access contentions can significantly degrade the I/O performance, such as communication network contention because of the high ratio of application processes to file servers, and file locking contention among processes in a single job because of the shared-file access.

Furthermore, despite the use of state-of-the-art techniques described above, significant challenges still exist in achieving scalable yet consistent I/O performance. The file servers often exhibit unbalanced I/O load from various applications sharing the storage resources, resulting in fluctuating file system performance [5, 23, 34]. In petascale systems at scale, the amount of I/O throughput available to any particular job can fluctuate to a large extent based on the behaviors of other running jobs accessing the shared file system. Another source for this kind of fluctuation is a RAID rebuild from a hardware failure. Since the performance of collective I/O is determined by the slowest participating process, it is important to ensure no process remarkably lags behind.

The study presented in this paper supports the view of conventional collective I/O, yet provides more scalable I/O performance in the presence of fluctuating file server performance. We make the following main contributions:

- We demonstrate that collective I/O performance could suffer from fluctuating file system behavior because of contention on shared I/O resources.

- We propose a dynamic bandwidth monitoring to probe the file servers and isolate the impact of accessing slower I/O servers by excluding them from being used for file striping.
- We propose a transparent file partitioning and data layout transformation mechanism that divides the data among files.

We have implemented the proposed scheme into a high-level I/O library, parallel netCDF [19], as a prototype. Our experimental evaluations on NERSC's Hopper [3] using several benchmarks running up to 8,192 processes have shown significant I/O performance improvements. We show that our approach effectively isolates the impact of accessing slower I/O nodes and reduces write I/O time significantly with less variation. Since the partition is done at high-level I/O library layer (PnetCDF), each file partition is also a self-describing file. Maintaining portable data representation is important because it provides seamless access to data structures, and layouts across all I/O software layers. Also, the richer information 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 with real I/O applications demonstrate that our transparent file partitioning brings significant I/O performance improvement while maintaining comparative number of partitioned files.

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

2. BACKGROUND

To establish the theoretical depth of the ideas in our study, we first describe unique characteristics and features of the I/O architecture and software layers adapted by many leadership-class computing systems. We then elaborate on the implications of such architecture to I/O performance.

2.1 I/O architecture and software components

The system architecture, typically seen in modern HPC systems such as Cray XT6 or IBM BG/P systems, often have thousands of compute nodes, each equipped with several multi-core processors, and tens of GB memory per node, offering peak performance of a couple of Peta-flops 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 custom high-bandwidth, low-latency network, for instance a 3-dimensional torus in Intrepid [1] and Hopper [3]. Each compute node is connected to other nearby nodes through a network topology. Each network node handles not only data bound to itself, but also data to be transferred to other nodes. At the other end of this interconnection network are tens to hundreds of storage servers, which are attached to storage devices. The entire storage system typically provides some form of redundancy to provide high availability.

Multiple layers of software are involved in the I/O path in the system architecture described above. Applications use I/O libraries, such as HDF5 [33] or PnetCDF [19], or may use MPI-IO or POSIX I/O calls directly. When MPI-IO is used either by a higher-level library or the application directly, MPI-IO optimizations such as two-phase I/O are achieved through communications over the interconnection network among compute nodes.

Collective I/O is an optimization in many MPI-IO implementations that improves the I/O performance to shared files. In ROMIO, an implementation of MPI I/O functions adapted by many MPI implementations, 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 in order to eliminate the possible lock conflicts on the servers [20, 35]. The striping count of a file is the number of I/O servers, or Object Storage Targets (OSTs) for Lustre, where a file is stored. Like all parallel file systems, files are striped into fixed-length blocks, and they are stored in the OSTs in a round-robin fashion.

2.2 Contention on I/O Path

While collective I/O often offers huge improvements for I/O performance on shared files, recent studies revealed that it continues to face significant challenges at scale [36, 23, 34, 5] for several reasons. First, as demonstrated by several prior studies, global synchronization cost and lock contention among aggregators accessing the shared file within the assigned file domain during collective I/O operations pose a limit to the I/O performance. Similar observations have been made in recent studies [36, 20], but the problem will only exacerbate as the number of processes increases to thousands and more. More importantly (especially in accessing shared storage systems), there are higher levels of variability in I/O performance in petascale machines. This variability is hard to avoid because of the different ways applications access "shared" file systems. For example, multiple applications running simultaneously on the petascale machine use the file system at the same time. Another example of such a case occurs when analysis code is trying to read the data stored in the shared storage while simulation code is writing their output data. This I/O variability is a big barrier to achieve scalable collective I/O operations because I/O performance is tied to the slowest storage nodes. In other words, even if most storage nodes perform relatively fast, the overall collective I/O time is determined by the slowest nodes.

To quantify our hypothesis, we wrote a small program where each process opens a file striped on a single I/O node 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 is given in Section 5. Figure 1 shows that, although the amounts of bytes written to each I/O node

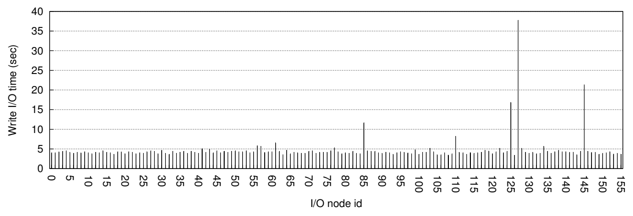

Figure 1: The write I/O time distribution among all I/O nodes while the amount of bytes written to each I/O node remain the same.

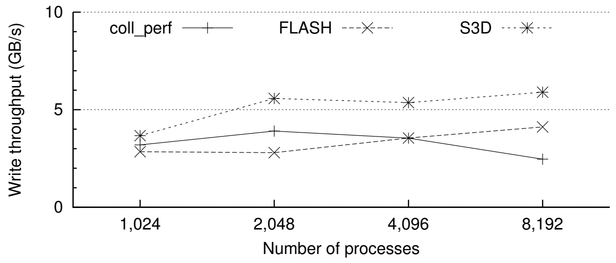

Figure 2: This poor scalability observed for all three benchmarks we evaluated when there is an excessive imbalance in I/O node performance. Details about our experimental setup is explained in Section 5.

are equal, a couple of I/O nodes exhibit an excessively high write I/O time relative to others. The slowest I/O node is in fact almost 9x slower than most other I/O nodes. Such an imbalance is a significant barrier to achieve scalable I/O performance in a production run. As shown in Figure 2, this excessive imbalance severely limits the scalability of all three benchmarks we tested.

3. DESIGN OF FILE PARTITIONING LAYER

In this section, we describe our file partitioning mechanism in the high-level I/O library context. We discuss how our mechanism determines the number of file partitions considering temporal behaviors of underlying I/O servers and how datasets are mapped into partitioned files.

3.1 Runtime Storage Nodes Selection

Our mechanism to isolate the impact of accessing imbalanced I/O nodes is to use a runtime bandwidth probing to identify each I/O server's load before the file striping layout is determined. The goal of this step is twofold. First, we would like to monitor each OST's current bandwidth availability. Because the I/O pattern in HPC systems is typically bursty, we probe each I/O server's bandwidth by writing a small dummy dataset just before writing actual file. Second, once the behaviors are identified, we would like to select the list of OSTs that can be used for storing each partitioned file.

We consider two criteria when designing our runtime probing module. First, the impact of probing should be minimized as it will not be the part of actual I/O. Second, the sampled bandwidth should reflect the temporal behavior of each I/O node. Combining these two, we determine each I/O node's

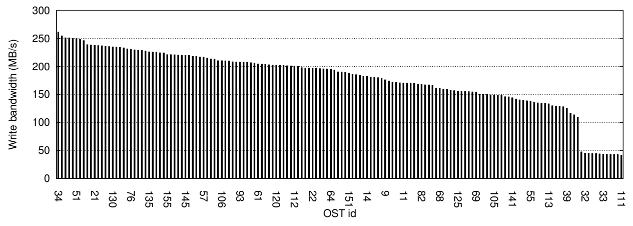

Figure 3: Distribution of write bandwidth observed for all 156 OSTs when 16MB of dummy data is written to each OST. Each OST's measured bandwidth is sorted in descending order.

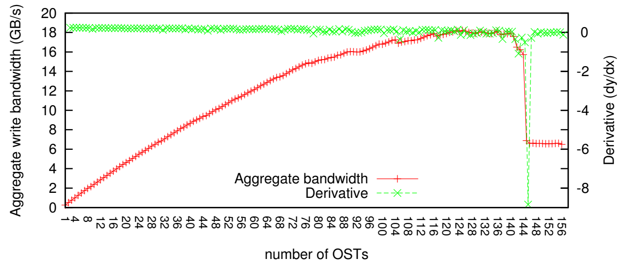

Figure 4: Maximum achievable aggregate bandwidth and its derivative (i.e., the slope) at each point.

bandwidth by writing several tens of MB of dummy data to each I/O server. We use POSIX I/O with the O_DIRECT flag because our probing module writes relatively small files, so they could sit on clients' buffer cache unless we explicitly bypass them. We also have to make sure the sampling data size we use is large enough to fill the RPC buffer size; otherwise the data will sit on the client and will not be transferred to the I/O server.

Figure 3 shows the distribution of measured write bandwidth across all 156 I/O nodes (OSTs) available on NERSC's Hopper, sorted in descending order. This graph indicates that certain OSTs exhibit relatively slower bandwidth than the others. We again attribute this to an inherent imbalance when accessing shared storage, as extensively discussed in recent studies [23, 34, 5]. Given these observed I/O bandwidths, we use following algorithm to select the I/O nodes for file striping. Assuming B_i to be the sorted bandwidth observed for 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 $1 \leq i \leq 156$. Figure 4 shows the estimated maximum achievable aggregate I/O bandwidth based on this formula. As we can see, the aggregate bandwidth gradually increases as more I/O nodes are added, but eventually saturates and then declines because the aggregate bandwidth is confined to the slowest node. To select the maximum number of I/O nodes that provide us the best achievable bandwidth, 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 certain probed bandwidth is added. Our dynamic probing and I/O node selection mechanism described so far is given in Algorithm 1. Using the

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, $P_i$ , $1 \le i \le N$ 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 $i \le lb$ do
        $S[i] = i$;
        $i++$;
    /* broadcast number of selected I/O nodes and
       corresponding node IDs                              */
    MPI_Bcast(&$N'$, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&$S[N']$, $N'$, MPI_INT, 0, MPI_COMM_WORLD);
```

results shown in Figure 4, our algorithm excludes 12 OSTs with less than 50MB/s for striping partitioned 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 end up selecting most of the available OSTs.

3.2 Mapping Arrays to File Partitions

Once the I/O nodes are selected, we then partition arrays among them. Figure 5 gives an overview of our file partitioning scheme. The basic concept of our scheme is that, from an application's perspective, partitioning is transparent; that is, all processes open and access a single file throughout program execution. Then, our partitioning mechanism internally splits application processes into set of subprocesses, each of which creates its own file partition collectively. The file partition created by each subprocess group is accessed *solely* by that group.

In higher I/O libraries like PnetCDF and 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 additional 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. Given this use pattern, we perform partitioning when array definition is finished and the data in memory is ready

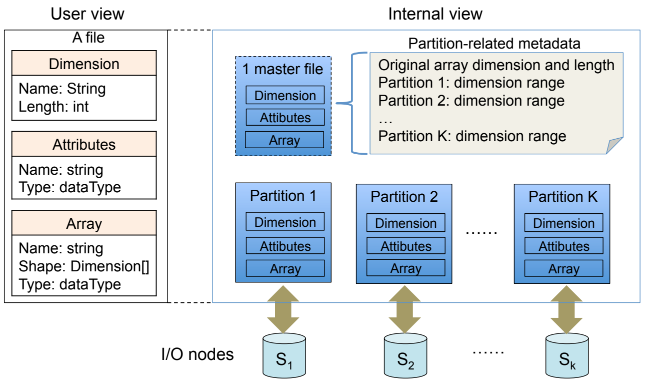

Figure 5: Overview of our file partitioning mechanism. From user's viewpoint at higher-level I/O library layer, a file has named arrays, dimensions, and attributes. Arrays also have attributes and may share dimension. In our approach, each array is internally divided into K file partitions, each of which is stored in a single I/O node. All files (both master and partitioned files) are in self-describing file format.

for write/read; in other words, just before step 4 above. We choose this time because each array's shape (number of dimensions, length of each dimension, and datatype of each element) is finalized at this point. The header information is also written at the end of file partition. In order to convey the user's intention of their file partitioning policy, we use 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 length will be partitioned along the dimension Z. There are however certain applications that prevent applying the default policy. For instance, in the S3D I/O application, the dataset called `u` is a four-dimensional array with the most significant dimension has length 3. Such a small dimension length limits the number of file partitions, preventing the application from exploiting potential benefits of partitioning in larger partition counts.

If partitioning along a dimension other than the most significant dimension is required, application writers simply specify the dimension name using a hint, called `par_dim_id`. Once this hint is given to our partitioning module, it internally 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.

Once all user's file partitioning strategies are specified through the hint mechanism, our proposed module creates partitioned files. The details of the file creation are as follows. It first obtains relevant hints using `MPI_Info_get()`. We store acquired information as a metadata in both master and the partitioned file's header information. If no hints were provided regarding file partitions, the normal procedure will be executed; it creates a single file without partitions. Otherwise, it splits the communicator because each process is divided into a subprocess group. The split processes then collectively create their own file partition using a dataset function provided in the high-level I/O library, for instance,

Algorithm 2: Algorithm illustrating our file partitioning mechanism. A file is partitioned only when partitioning is enabled through the MPI hint.

```
get user's hints about partitioning;
if partitioning 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 file partition                             */
    MPI_Info_set(info, "romio_lustre_start_iodevice", offset);
    MPI_Info_set(info, "striping_factor", "1");
    create a partitioned file 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 = $\frac{d[i][j]\to\text{size}}{N'}$;
                define a new dimension using dim_sz and $d[i][j]$;
                for each partition, $k$ 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;
```

`ncmpi_create` in PnetCDF. After creating a partitioned file, our algorithm traverses each defined array in the original 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 length for each partition. Note that only the partitioning dimension will be affected; all the remaining dimensions will have the same length as the original. Once a new dimension length is determined, we define a new dimension for the partitioned file and create an array with the new dimension lists. If the array is partitioned, we update the original array definition in the master file with a scalar value. In other words, the master file does not have a physical space allocated for the partitioned array as the actual data will be stored in the partitioned files. We repeat these procedures until all arrays in the original file are processed. The algorithm described so far is given in Algorithm 2.

Our default partitioning policy is to divide all arrays defined in a file. There are however 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. While our partitioning mechanism provide transparent access to those datasets, it is better to store this information in a single file. Second, arrays defined with fewer number of dimensions often do not represent a significant portion of 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 the length of all dimensions are 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 deal with the above cases, we provide another hint, called `min_ndimes`, to allow selective array partitioning. This hint limits applying file partitioning for arrays with a dimension length equal or less than `min_ndims`.

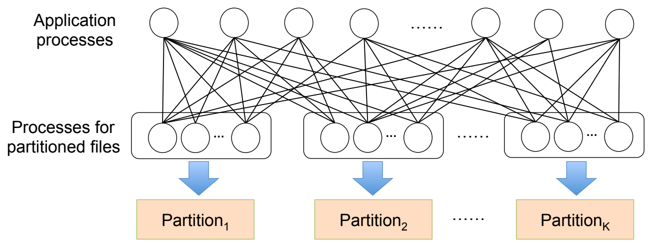

Figure 6: Any I/O requests from applications belonging to a file partition owned by other processes need to communicate among processes before going to I/O servers. All these data exchange would require sufficiently complicated communication among processes.

Our main mechanism to convey the user's partitioning strategy is through MPI hints. We however 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 the hint information conveyed to each process is the same across all processes. Therefore, we use mix of an MPI hint and an environment variable so that the overhead incurred by our approach is minimized with larger process runs.

3.3 Memory-to-File Layout Transformation

Once a file is partitioned, we need to provide a transparent way to access those partitioned files as well. Note that, from an application's viewpoint, all I/O accesses still go through the master file as it has sufficient information about how each array is partitioned in each file. In other words, there is no change in user's I/O routines. We also note that reading datasets already stored in partitioned files can be performed transparently using the same metadata retrieval process.

Figure 6 shows an overview of the memory-to-file layout transformation mechanism. The transformation mechanism to partitioned files is mainly composed of two steps: i) calculating each process's requests to partitioned files and exchanging it among all processes; ii) exchanging requests among processes in each split communicator and issues I/O requests using I/O calls (either synchronous our asynchronous ones). We note that, since our partitioning is done at the higher-level I/O library layer, all user's array partitioning is represented as start, count, and stride offset list for each dimension.

In the first step, each process calculates the list of start and count offsets to each file partition, dividing the data in memory among the processes who own the partitions. This is done by (logically) dividing the start and count offset, de-

Algorithm 3: Algorithm for handling I/O requests to partitioned files 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, $D_j \in A_i$ 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[$j$] $\cap$ range[$D_j$];
                my_req[$i$].count = count[$j$] $\cap$ 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;
```

noted as my_req[], into file partitions, each of which 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 groups. In fact, any process can be a delegate so that we do not make 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 `MPI_Allreduce()` among all processes.

The second step is based on everyone's my_req, and calculates what requests of other processes lie in this process's file partitions. others_req[i].{start,count} indicates how many noncontiguous requests of process i accessing this process's file partition. All these incur an `MPI_Alltoall` and many isend/irecv/wait_all. This step ensures delegates collect the request information from all other processes.

Then each process sends requests to the appropriate remote delegate. Only delegates may have multiple I/O requests. Non-delegate processes will not participate in this loop, but will call to the data exchange routine if they have certain 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 they are all processed by calling wait_all() at the I/O library layer. The procedure described so far is given in Algorithm 3.

```
MPI_Info_set (info, "nc_partitioning_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);
...
```

Figure 7: A PnetCDF example code that creates a file with partitioning enabled set to true. The number of file partitions is determined through the profiling mechanism explained in Section 3.1. This example creates a variable named "cube" of Z-Y-X dimension, each with 100 length. From an application writer's viewpoint, it only requires to add a hint to specify the intention of partitioning to store a variable.

4. EXTENDING THE PNETCDF LIBRARY

In this section, we describe our modifications to the PnetCDF library to implement our file partitioning mechanism described in Section 3.

4.1 PnetCDF File Partitioning

Figure 7 shows a typical example of PnetCDF code that includes the sequence of dimension and array (variable) definition followed by the code to write data on it. In PnetCDF, all processes in the communicator must make an explicit call (`ncmpi_enddef`) at the end of the define mode in order to verify that the values passed in by all processes match. From our design viewpoint, this is the time when all shapes of arrays are known, therefore, our array partitioning is internally executed at the end of this call.

The NetCDF header information for this example of partitioning case is given in Figure 8. Note that, after partitioning, both master and partitioned files have more additional attributes than the original file. For instance, the master file (Figure 8(b)) has global attributes that indicate the file name for each partitioned file, the number of partitions, and the original dimension size for a variable, "cube". The partitioned file header, on the other hand, has attributes for describing the range of partitioned dimension as well as the partition index.

We use a per-variable attribute to convey the user's intent of 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 for the "cube" array, this can be conveyed to our partitioning module by calling the `ncmpi_put_att_int` API with "par_dim_id" set to

```
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_partitions = 2;
    cube: ndims_org = 3;

// global attributes:
  :partition 0: "test.0";
  :partition 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:
  :partition_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:
  :partition_index = 1;

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

(d)

Figure 8: NetCDF file header information by `ncmpidump` when the file is divided into 2. (a) Original NetCDF file (i.e., non-partitioned case). (b) The master NetCDF file after partition. Note that the data section is 0, meaning empty. (c) First partitioned NetCDF file. (d) Second partitioned NetCDF file.

cube_dim[1]. This mechanism also can be used for handling record variables. Record variables are the ones that use unlimited dimensions, so we cannot determine the partition size when the most significant dimension is defined as UNLIMITED. 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 three dimensions: time, x, and y. The last two are assigned fixed length of 100; time is assigned the length UNLIMITED. In this example, the variable `xytime` can be partitioned along either the x or y dimension.

4.2 Coordinating I/O among Partitions

While current PnetCDF provides several data mode functions, we focus on the collective versions of those functions in our implementation. An example of such functions that writes a variable "cube" is `ncmpi_put_vara_all` shown in Figure 7. In this function, the varid (i.e., cube_id), start, count, and stride values (not used in above API) refer to the data in the file whereas buf, bufcount, and datatype (`MPI_INT`) refer to data in memory. When this call gets called, our partitioning module intercepts it and coordinates the data transfer between memory to partitioned files using the algorithm described in Algorithm 3. When each process issues I/Os to its own partition, we use a non-blocking API available in PnetCDF. They can aggregate multiple smaller requests into 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 partitioned files are processed using the example code in Figure 7. Let us assume there are 4 processes to access this array and the number of file partitions is 2. Each I/O request is composed mainly of start offset, count and stride for each dimension. Since our example dataset is 3 dimensional, we have start[3], count[3], and stride[3]. For illustrative purposes, let us assume that stride count is 1, meaning all array elements are accessed contiguously. Given this, one partition (50 by 100 by 100) is owned by P_0 and P_1 whereas the other partition (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 file partition. 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 file partition (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 file partition. Once all this information is obtained, all processes now exchange information (using alltoall) in order to figure out which process has a portion of the data not belonging to its own partition. Afterwards, all processes know 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 partitioned files returns when all the issued nonblocking I/O calls are completed.

Our discussion so far assumes 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. In order 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 partition. 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. EXPERIMENTAL EVALUATIONS

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 parallel file system deployed in Hopper is Lustre [2] 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.

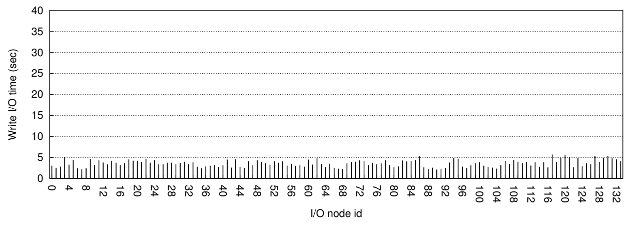

Figure 9: Balanced write I/O time observed when only subset of I/O nodes that were detected through our dynamic bandwidth probing.

We implemented our proposed approach into the parallel netCDF 1.3.1. Our feature added approximately 900 lines of new code to PnetCDF. Our implementation is configured to link with Cray's xt-mpich2 version 5.6.0. We used a separate ROMIO module described in [20] as a standalone library, which is then linked with the native MPI library. Our previous experience tells 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. Unless otherwise stated, all experiments were run five times, and we present the average of those runs with error bars as a standard deviation.

We evaluate our file partitioning scheme against the base scheme, where all arrays are stored in a normal file (non-partitioned) striped across all available OSTs. To show the effectiveness of our dynamic bandwidth probing, we ran two schemes of our partitioning cases: striped over all OSTs and striped over selected OSTs.

5.1 Collective I/O Performance Benchmark

Before presenting our evaluation with the collective I/O performance benchmark, we first show how our approach effectively isolates slower I/O nodes. In order to do this, we wrote a small test case that writes 1GB of data to an individual I/O node selected by our dynamic probing module. Figure 9 shows the write I/O time, collected using the TAU profiling tool [31], observed at each I/O node that was selected by our sampling module. In this example, 22 out of 156 OSTs were selected for writing. As compared with Figure 1, it clearly demonstrates more balanced write I/O time across all selected OSTs.

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, all in a block-partitioned manner. We made it write/read four 3D variables. The 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 128×128×128 of 4-byte integers, corresponding to 8MB. All data is written to a single file

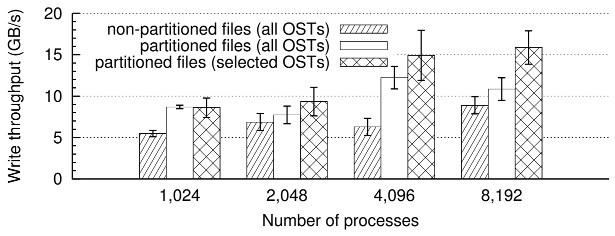

Figure 10: Write throughput results for coll_perf.

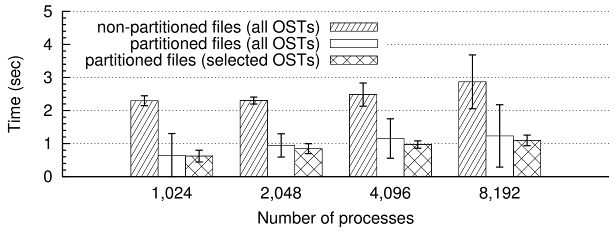

Figure 11: The average write I/O time for coll_perf with error bars. Regardless of file is partitioned or not, using all OSTs shows much "higher" deviation.

for the base case (non-partition). For our partitioning case, all four variables are partitioned along the most significant dimension.

Figure 10 shows the write throughput of coll_perf with and without our file partitioning schemes. 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 single file does not scale 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 partitioning schemes improve the write throughput significantly by 12%–94% when used with all 156 OSTs and 36%–137% when used with selected OSTs, respectively.

To understand the performance improvement obtained by our approach, we have collected the performance breakdown of coll_perf during collective I/O using the TAU profiling tool [31]. Figure 11 shows that the time spent in POSIX write() time by each aggregator process gradually increases as the number of processes increase. This is because the amount of data written increases with larger number of processes. The most important observation we made here is that writing to partitioned files using either all OSTs or selected OSTs reduces the write I/O time significantly, about 70% on average. This indicates that writing to partitioned files clearly lessens the contention on the file server. Another important insight from this graph is the high variations on the write I/O time when all OSTs are used, and the variations increased with larger process counts. The partitioned files with selected OSTs show low deviation from average mainly because relatively slower OSTs were eliminated before the time of writing.

In our next experiments, we would like to understand how the read from partitioned files behaves. To do this, we perform the same weak scalability tests (1,024 to 8,192 pro-

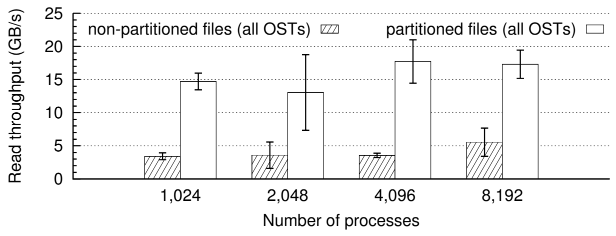

Figure 12: Read throughput results for coll_perf.

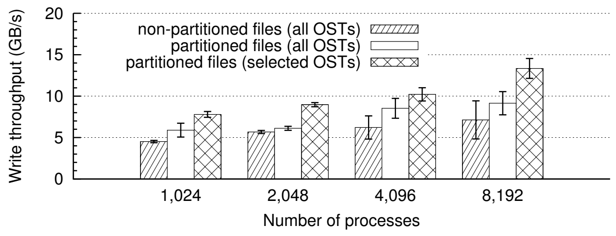

Figure 13: FLASH I/O write throughput.

cesses) on the read case, each case collectively reads the entire files in a 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. To ensure data is read from the storage nodes, all caches are flushed before each run. Figure 12 shows that the non-partitioned file case is not scalable while our partitioning 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 (i.e., non-partitioned) is about 6x slower than reading from partitioned files. We attribute this to the pretty aggressive readahead mechanism used in Lustre file system. In the case of reading from non-partitioned 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 partitioned file case, the readahead mechanism is entirely reading from a single OST, so the benefit of readahead is maximized.

Our partitioning approach introduces additional communication during memory-to-file layout transformation time: `MPI_Isend`, `MPI_Irecv()`, `MPI_Alltoall()`, and `MPI_wait()`. In order 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 file partition. The buffer exchange phase also does not incur much overhead because only participating 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.

5.2 FLASH I/O Benchmark

The FLASH I/O benchmark [38, 18] is the I/O kernel of a block-structured adaptive mesh hydrodynamics code that solves the compressible Euler equations on a block structured adaptive mesh and incorporates the necessary physics to describe the environment, including the equation of state, reaction network, and diffusion [9]. The problem domain is divided into blocks distributed among a number of MPI processes. A block is a three-dimensional array with an additional 4 elements as guard cells in each dimension on both sides to hold information from its neighbors. There are 24 data variables per array element, and about 80 blocks on each MPI process. A variation in block numbers per MPI process is used to generate a slightly unbalanced I/O load. 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 purpose of I/O in FLASH is to write a checkpoint file and two 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 benchmark. We set the block size to be $16\times16\times16$, which corresponds to about 64 MB of data per process.

We use a FLASH I/O format where all mesh variables (including density, pressure and temperature) are written to the same dataset (variable) in the output file. Both checkpointing and plot files are written in this file format. In case of checkpoint files, among 24 variables defined in FLASH, only 10 variables correspond to those mesh variables, each of which is a four-dimensional (4D) array of double-precision typed data. 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 approach; All other variables are stored in the master file without partitioning. The plot files have three mesh variables and we again applied partitioning for the unknown variables.

Figure 13 shows the I/O bandwidth of FLASH for the non-partitioned case and our two approaches. Using a non-partitioned file did not scale well even with increased process counts. The maximum I/O bandwidth observed with 8,192 processes is about 8 GB/s. This is significantly below the maximum I/O bandwidth on Hopper. The partitioned files with all OSTs slightly outperform the non-partitioned file case, by 28% on average, but there is higher variation with larger process counts. Overall, the partitioned files with selected OSTs can achieve about 70% I/O bandwidth improvement than the non-partitioned case.

5.3 S3D I/O Benchmark

The S3D application [28] 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 $50\times50\times50$ fixed subarrays.

The checkpoint files consist of four global arrays: two 3-

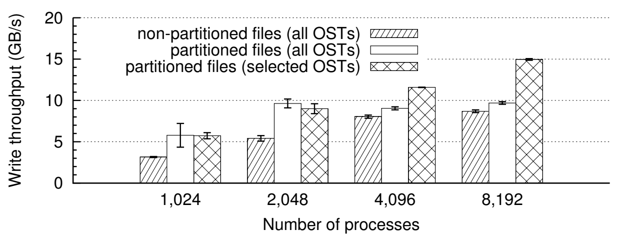

Figure 14: S3D I/O write throughput.

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 length of the most significant dimension in 4D variables are relatively small, 3 and 11 for `three` and `nsc` respectively, we partition these variables along z-dimension, that is, the second most significant dimension.

Figure 14 shows the I/O bandwidth of S3D for all three cases we evaluated. We have observed that the non-partitioned file case is marginally scalable. The partitioned files using all OSTs can achieve higher performance improvement than the non-partitioned file case up to 2,048 processes, but only marginal improvement beyond that point. The partitioned files case with selected OSTs consistently outperforms than the non-partitioned file case, by 60% on average.

6. RELATED WORK

PLFS [4] introduced a virtual layer that remaps an application's preferred data layout into one optimized for the underlying parallel file system. Like PLFS, Split writing and Hierarchical striping [37] 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 partitioned fils. 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 [36]. PIDX [16, 15] is a parallelization of IDX data format, and uses a novel aggregation technique to improve its scalability. Dickens and Logan [8] demonstrated that the collective I/O operations on Lustre performs poorly because of high communication overhead in order 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. [12] 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 partitioned files. 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. A similar idea of subfiling is also provided in the ADIOS BP file format [24]. However, ADIOS has limited flexibility in selecting how the data is stored across subfiles, and also it does not store arrays in canonical order. Fu et al. [11, 10] 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 partitioned files 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 [13]. 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 [23, 34, 5]. [34] 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 [23], 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 [5], 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. 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.

7. CONCLUSION AND FUTURE WORK

This paper has proposed a transparent file partitioning mechanism to provide scalable collective I/O performance while keeping a conventional view of large multi-dimensional arrays to a user. We use a dynamic bandwidth probing to detect slower I/O nodes and isolate the impact of these slower I/O nodes. Our implementation is incorporated into PnetCDF, a high-level I/O library, and we evaluate its performance using a set of I/O benchmarks on NERSC's Hopper. Our experimental results demonstrate that our partitioning scheme consistently improves the performance of collective I/O significantly by reducing write I/O time with less variation.

We will continue to evaluate our approach on other platforms like Intrepid, IBM Blue Gene/P, at Argonne National Laboratory [1], and other high-level I/O libraries. Future research will focus on investigating how the data exchange mechanism we proposed in this paper can be applied on more general layout transformation techniques like transposing array dimensions.

8. REFERENCES

[1] Argonne Leadership Computing Facility. `http://www.alcf.anl.gov/intrepid/`.

[2] Lustre File System. `http://www.lustre.org`.

[3] National Energy Research Scientific Computing Center. `http://www.nersc.gov/users/computational-systems/hopper/`.

[4] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate. PLFS: A Checkpoint Filesystem for Parallel Applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, 2009.

[5] S. Byna, A. Uselton, Praphat, D. Knaaky, and Y. H. He. Trillion Particles, 120,000 cores, and 350 TBs: Lessons Learned from a Hero I/O Run on Hopper, 2013.

[6] P. Carns, R. Latham, R. Ross, K. Iskra, S. Lang, and K. Riley. 24/7 Characterization of Petascale I/O Workloads. In *Proceedings of the First Workshop on Interfaces and Abstractions for Scientific Data Storage*, 2009.

[7] J. M. del Rosario, R. Bordawekar, and A. Choudhary. Improved parallel I/O via a two-phase run-time access strategy. In *Proceedings of Workshop on Input/Output in Parallel Computer Systems*, pages 56–70, 1993.

[8] P. M. Dickens and J. Logan. 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*, pages 31–38, 2009.

[9] B. Fryxell, K. Olson, P. Ricker, F. X. Timmes, M. Zingale, D. Q. Lamb, P. MacNeice, R. Rosner, J. W. Truran, and H. Tufo. FLASH: An Adaptive Mesh Hydrodynamics Code for Modeling Astrophysical Thermonuclear Flashes. *The Astrophysical Journal Supplement Series*, 131(1):273, 2000.

[10] J. Fu, N. Liu, O. Sahni, K. E. Jansen, M. S. Shephard, and C. D. Carothers. Scalable Parallel I/O Alternatives for Massively Parallel Partitioned Solver Systems. In *Proceedings of Workshop on Large-Scale Parallel Processing*, 2010.

[11] J. Fu, M. Min, R. Latham, and C. D. Carothers. 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*, pages 465–473, 2011.

[12] K. Gao, W.-k. Liao, A. Nisar, A. Choudhary, R. Ross, and R. Latham. Using Subfiling to Improve Programming Flexibility and Performance of Parallel Shared-file I/O. In *Proceedings of the International Conference on Parallel Processing*, pages 470–477, 2009.

[13] W. Kendall, J. Huang, T. Peterka, R. Latham, and R. Ross. Visualization Viewpoint: Towards a General I/O Layer for Parallel Visualization Applications. *IEEE Computer Graphics and Applications*, 31(6):6–10, 2011.

[14] D. Kotz. Disk-directed I/O for MIMD multiprocessors. *ACM Trans. Comput. Syst.*, 15(1):41–74, Feb. 1997.

[15] S. Kumar, V. Vishwanath, P. Carns, J. A. Levine, R. Latham, G. Scorzelli, H. Kolla, R. Grout, J. Chen, R. Ross, M. E. Papka, and V. Pascucci. Efficient Data Restructuring and Aggregation for IO Acceleration in PIDX. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, 2012.

[16] S. Kumar, V. Vishwanath, P. Carns, B. Summa, G. Scorzelli, V. Pascucci, R. Ross, J. Chen, H. Kolla, and R. Grout. PIDX: Efficient Parallel I/O for Multi-resolution Multi-dimensional Scientific Datasets. In *Proceedings of the 2011 IEEE International Conference on Cluster Computing*, pages 103–111, 2011.

[17] S. Lang, P. Carns, R. Latham, R. Ross, K. Harms, and W. Allcock. I/O performance challenges at leadership scale. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 40:1–40:12, 2009.

[18] R. Latham, C. Daley, W. keng Liao, K. Gao, R. Ross, A. Dubey, and A. Choudhary. A case study for scientific I/O: improving the FLASH astrophysics code. *Computational Science & Discovery*, 5(1):015001, 2012.

[19] J. Li, W.-k. Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale. Parallel netCDF: A High-Performance Scientific I/O Interface. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, 2003.

[20] W.-k. Liao and A. Choudhary. 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*, 2008.

[21] W.-k. Liao, K. Coloma, A. Choudhary, L. Ward, E. Russell, and N. Pundit. Scalable design and implementations for mpi parallel overlapping i/o. *Parallel and Distributed Systems, IEEE Transactions on*, 17(11):1264–1276, nov. 2006.

[22] W.-k. Liao, K. Coloma, A. Choudhary, L. Ward, E. Russell, and S. Tideman. Collective caching: application-aware client-side file caching. In *Proceedings of 14th IEEE International Symposium on High Performance Distributed Computing*, pages 81–90, 2005.

[23] J. Lofstead, F. Zheng, Q. Liu, S. Klasky, R. Oldfield, T. Kordenbrock, K. Schwan, and M. Wolf. 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, pages 1–12, 2010.

[24] J. F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin. 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*, pages 15–24, 2008.

[25] X. Ma, M. Winslett, J. Lee, and S. Yu. Improving MPI-IO Output Performance with Active Buffering

Plus Threads. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, 2003.

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

[27] D. Randall, M. Khairoutdinov, A. Arakawa, and W. Grabowski. Breaking the Cloud Parameterization Deadlock. *Bull. Amer. Meteor. Soc.*, 84:1547–1564, 2003.

[28] R. Sankaran, E. R. Hawkes, J. H. Chen, T. Lu, and C. K. Law. Direct numerical simulations of turbulent lean premixed combustion. *Journal of Physics: Conference Series*, 46(1):38, 2006.

[29] K. Schuchardt, B. Palmer, J. Daily, T. Elsethagen, and A. Koontz. IO Strategies and Data Services for Petascale Data Sets from a Global Cloud Resolving Model. *Journal of Physics: Conference Series*, 78, 2007.

[30] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, 1995.

[31] S. S. Shende and A. D. Malony. The TAU Parallel Performance System. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, May 2006.

[32] R. Thakur and A. Choudhary. An extended two-phase method for accessing sections of out-of-core arrays. *Sci. Program.*, 5(4):301–317, Dec. 1996.

[33] The HDF Group. Hierarchical data format version 5. `http://www.hdfgroup.org/HDF5`, 2000–2010.

[34] B. Xie, J. Chase, D. Dillow, O. Drokin, S. Klasky, S. Oral, and N. Podhorszki. Characterizing Output Bottlenecks in a Supercomputer. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 8:1–8:11, 2012.

[35] L. Ying. Lustre ADIO collective write driver – white paper. Technical report, Sun and ORNL, 2008.

[36] W. Yu and J. Vetter. ParColl: Partitioned Collective I/O on the Cray XT. In *Proceedings of the 37th International Conference on Parallel Processing*, pages 562–569, 2008.

[37] W. Yu, J. Vetter, R. S. Canon, and S. Jiang. Exploiting Lustre File Joining for Effective Collective IO. In *Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, pages 267–274, 2007.

[38] M. Zingale. FLASH I/O Benchmark Routine - Parallel HDF 5. `http://www.ucolick.org/~zingale/flash_benchmark_io/`, March 2001.