ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/228577340

Tuning HDF5 for Lustre File Systems

Article · January 2012

CITATIONS
50

READS
666

5 authors, including:

Mark Howison
Brown University
58 PUBLICATIONS **1,264** CITATIONS
SEE PROFILE

Quincey Koziol
The HDF Group
65 PUBLICATIONS **741** CITATIONS
SEE PROFILE

David Knaak
7 PUBLICATIONS **93** CITATIONS
SEE PROFILE

John Shalf
Lawrence Berkeley National Laboratory
285 PUBLICATIONS **10,629** CITATIONS
SEE PROFILE

Some of the authors of this publication are also working on these related projects:

Scientific Visualization View project

Beyond Moore's Law View project

All content following this page was uploaded by John Shalf on 16 May 2014.

The user has requested enhancement of the downloaded file.

Tuning HDF5 for Lustre File Systems

Mark Howison*, Quincey Koziol†, David Knaak‡, John Mainzer†, John Shalf*

*Computational Research Division
Lawrence Berkeley National Laboratory
One Cyclotron Road, Berkeley, CA 94720
Email: {mhowison,jshalf}@lbl.gov

†The HDF Group
1901 S. First Street, Suite C-2
Champaign, IL 61820
Email: {koziol,mainzer}@hdfgroup.org

‡Cray Inc.
901 Fifth Avenue, Suite 1000
Seattle, WA 98164
Email: knaak@cray.com

Abstract—**HDF5 is a cross-platform parallel I/O library that is used by a wide variety of HPC applications for the flexibility of its hierarchical object-database representation of scientific data. We describe our recent work to optimize the performance of the HDF5 and MPI-IO libraries for the Lustre parallel file system. We selected three different HPC applications to represent the diverse range of I/O requirements, and measured their performance on three different systems to demonstrate the robustness of our optimizations across different file system configurations and to validate our optimization strategy. We demonstrate that the combined optimizations improve HDF5 parallel I/O performance by up to 33 times in some cases – running close to the achievable peak performance of the underlying file system – and demonstrate scalable performance up to 40,960-way concurrency.**

I. Introduction

Centralized parallel I/O subsystems are the norm for large supercomputing resources that serve the HPC community. However, the exponentially increasing parallelism of leading-edge HPC systems is pushing existing storage solutions to the limit, and the performance of shared-file I/O implementations often falls orders of magnitude below the maximum theoretical bandwidth of the hardware. The fallback position of writing one-file-per-processor can recoup lost performance, but as we enter the petascale era, and 100,000-way concurrency becomes commonplace, this approach results in untenable data management issues and overwhelms parallel file systems with difficult-to-parallelize metadata operations. Increasingly, the burden is falling on application developers to navigate the intricacies of effective parallel I/O, for instance by implementing their own I/O aggregation algorithms or experimenting with file system striping parameters.

Middleware layers like MPI-IO and parallel HDF5 address these problems by hiding the complexity of performing coordinated I/O to single, shared files, and by encapsulating general purpose optimizations such as collective buffering. However, the optimization strategy is dependent on the file system software and the system hardware implementation. Maintaining I/O performance requires constant updates to the optimization strategy for these middleware layers. Ultimately, our goal is to ensure that application developers have a performance-portable I/O solution that minimize changes to their use of the middleware APIs, regardless of the underlying file system – through different generations of GPFS, Lustre, or any other parallel file system that emerges.

In particular, file systems such as Lustre [1] require reconsideration of file layouts and I/O strategies from top-to-bottom rather than mere parameter tuning. Lustre is a scalable, POSIX-compliant parallel file system designed for large, distributed-memory systems, and can therefore support any POSIX-compliant I/O pattern. It also features separated metadata and object storage and uses a client-server model with a server-side, distributed lock manager that maintains coherency across all clients. Yet, in practice large, contiguous I/O operations that are aligned to avoid the lock manager perform best, and certain metadata operations are best avoided.

In this paper, we will describe the optimizations that were developed to overcome the performance pitfalls of the Lustre file system and specifically how they were appied to enable scalable performance for parallel applications creating HDF5 files. The optimizations required changes that spanned HDF5 and the lower-level MPI-IO layer, but ultimately result in performance that runs close to the maximum practical performance of the file system implementation.

To motivate our optimization strategy and validate the performance improvements delivered by our optimizations strategy, we selected three HPC applications (GCRM, VORPAL, and Chombo) to represent three common I/O patterns found in the DOE Office of Science computing workload. The key patterns represented by these applications are: rectilinear 3D grids with balanced, cubic partitioning; rectilinear 3D grids with unbalanced, rectangular partitioning; and contiguous but size-varying arrays, such as those found in adaptive mesh refinement. For these three patterns on production systems with a range of concurrencies up to 40,960-way, we observe write bandwidths that are $1.4\times$ to $33\times$ those of the original approaches used by these applications.

II. Background

A. HDF5

The Hierarchical Data Format v5 (HDF5) [2] I/O library stores data in binary files organized for high-performance access, using a machine-independent, self-describing format. HDF5's "object database" data model enables users to focus on high-level concepts of relationships between data objects rather than descending into the details of the specific layout of every byte in the data file.

The HDF5 library is designed to operate on large HPC systems, relying on an implementation of the MPI standard for communication and synchronization operations and optionally also for collective I/O operations. The HDF5 library's design includes a modular "virtual file layer" for performing I/O to files using software drivers. HDF5 can either use the MPI-IO routines for collective and independent I/O operations (the "MPI-IO virtual file driver"), or can use a combination of MPI communications and POSIX file I/O operations to bypass MPI-IO when an application's I/O pattern does not make good use of MPI-IO (the "MPI-POSIX virtual file driver").

The Network Common Data Form (netCDF) [3] library also offers a flexible data model and machine independence similar to HDF5. The most recent version, netCDF-4, has adopted HDF5 as its intermediate layer. Thus, the optimizations we describe in this paper will seamlessly apply to the same I/O patterns in netCDF-4.

Because netCDF did not introduce parallel support until netCDF-4, the pNetCDF library [4] was designed as a parallel interface to the original netCDF file format and is built directly on top of MPI-IO. pNetCDF does not produce files compatible with netCDF-4, although applications that use the pNetCDF API can be adapted to using the netCDF-4 API in a fairly straightforward manner.

B. Related Work

Many HPC applications perform "append-only" I/O, writing data to a file that is never revisited during execution. Instead, the file may be used later to restart the application, loaded into an analytics or visualization program for post-run analysis, or simply discarded because a restart was not needed. Both the Parallel Log-structured File System (PLFS) [5] and the Adaptable I/O System (ADIOS) [6] can accelerate this use case. PLFS uses file-per-processor writes (or what it terms *N-N* access) to avoid the lock contention problems that arise with parallel access to shared files. ADIOS offloads I/O operations onto designated I/O nodes, allowing a computational code to execute non-blocking I/O routines and continue running while I/O is handled in the background. ADIOS also provides interoperability with existing data formats like HDF5 and netCDF, although a post-processing step is necessary to render such a file from the internal format used by ADIOS.

PLFS and ADIOS can avoid the performance pitfalls of coordinated, shared file access by using independent access. However, if there are a very large number of processes, the number of files created becomes a bottleneck for the Lustre metadata server and can become a file management headache for the user. Also, ADIOS sheds many features of HDF5 in order to achieve performance. ADIOS is able to write to HDF5 files, but its simpler data model does not support key features of HDF5 for implementing complex data models, such as HDF5's hierarchical storage model and full capabilities for attaching attributes to objects. The goal of our work is to maintain the rich data model and support the existing codebase of HDF5 by matching the performance of writing a single, shared-file to that of writing independent files.

One key strategy for bringing single-shared-file performance up to the level of the independent access approach used by PLFS and ADIOS is to employ "collective" optimizations, which have a long history of use in different MPI-IO implementations (see [7] for an overview). In general, collective optimizations use the additional information provided by a complete view of an I/O operation to decrease the number of I/O accesses and reduce latency. One such optimization called two-phase I/O [8], [9] or collective buffering [10] assigns a subset of tasks to act as "aggregators". Aggregators gather smaller, non-contiguous accesses into a larger, contiguous buffer in the first phase, and in the second phase write this buffer to the file system. This optimization has existed for many years in ROMIO, which is an MPI-IO implementation from Argonne National Laboratory (ANL) that contains support for many file systems, including Lustre.

A survey of 50 HPC projects at the National Energy Research Supercomputing Center found that most HPC applications already use contiguous access, but that transfer sizes varied by several orders of magnitude (from several kilobytes to hundreds of megabytes) [11]. On parallel file systems like Lustre that use server-side file extent locks, varying transfer sizes often lead to accesses that are poorly distributed and misaligned relative to the lock boundaries. Thus, one of the main benefits of collective buffering is that the buffer size can be set to a multiple of the lock granularity, which can drastically reduce lock contention. From a software engineering perspective, it is best that this feature exist in a layer of the I/O stack below the application, such as MPI-IO, because modifying an application's I/O operations to align to lock boundaries (e.g., by adding buffering or padding the I/O operations) is a complexity that most application developers wish to avoid.

Within this context, Liao and Choudhary [12] explored three modifications to the ROMIO collective buffering algorithm to respect lock boundaries when partitioning the shared file among aggregators. The first strategy is to take the simple, even partitioning used by ROMIO and move each boundary to the nearest lock boundary. The other two strategies use cyclic assignments of aggregators to I/O servers, either in a one-to-one mapping (called "static-cyclic") or a several-to-one mapping ("group-cyclic"). Liao and Choudhary presented results at up to 1,024-way concurrency and concluded that a group-cyclic approach is best for file systems like Lustre and GPFS that use server-side lock mechanisms.

Dickens and Logan [13] have reported concurring results for the static- and group-cyclic approaches using their Y-Lib library on Lustre systems, again at up to 1,024-way concurrency. The static-cyclic approach was also described earlier by Coloma et al. [14] under the term "persistent file realms," but with results extending to only 64-way concurrency.

We validate these ideas for lock-aligned collective buffering on Lustre using the Cray implementation of MPI-IO, and with a more complicated codepath that passes through the HDF5 layer. We also show results for 40,960-way concurrency, which is 40 times larger than previously published results.

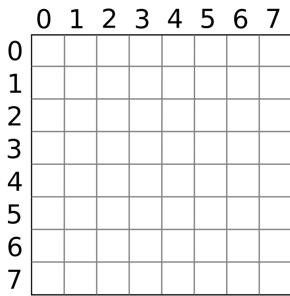

Fig. 1. Contiguous HDF5 dataset storage.

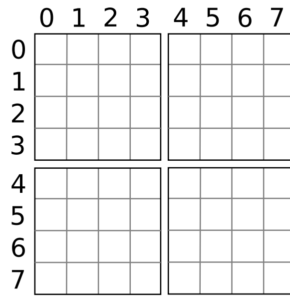

Fig. 2. Chunked HDF5 dataset storage.

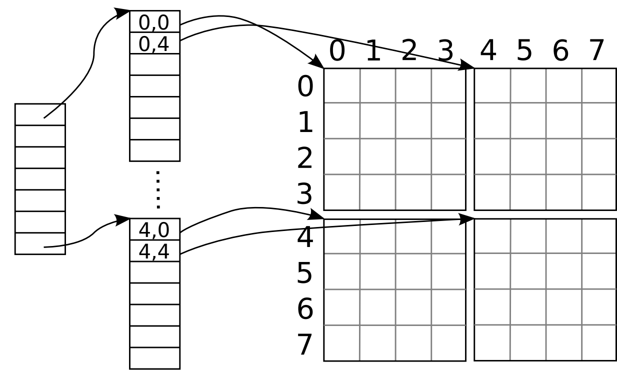

Fig. 3. Chunked HDF5 dataset with untuned chunk index B-tree.

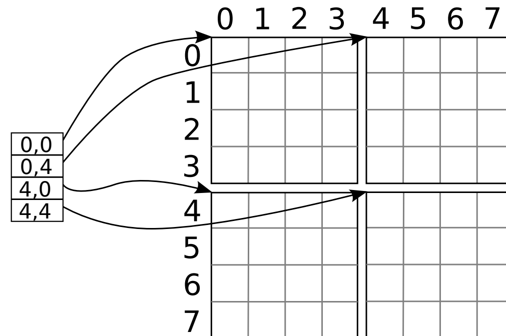

Fig. 4. Chunked HDF5 dataset with tuned chunk index B-tree.

Although we found that our methods scaled well, there are other efforts to prepare collective buffering for scalability to the petascale. For instance, Nisar et al. [15] suggest setting aside a subset of tasks, called "I/O delegates", that are solely responsible for managing collective I/O operations such as aggregation. Although we do not pursue that modification here, it is compatible with the optimizations we present.

III. Software Modifications

The parallel HDF5 implementation is built on top of the MPI-IO library. In many cases, we found that performance optimizations for Lustre required changes that spanned both the HDF5 and MPI-IO libraries. Our work involved close cooperation between developers of these respective libraries, using guidance from detailed performance analysis of the target applications. Our analysis was facilitated by detailed traces of the application performance along with integrated instrumentation of the MPI-IO libraries provided by our collaborators at Cray Inc.

A. HDF5

The HDF5 library stores application data in *datasets* within HDF5 files. HDF5 datasets are multi-dimensional arrays whose elements can be composed of integer, floating-point, or other, more complex, types. A dataset's elements are either stored contiguously in a file (Figure 1), or in a *chunked* form (Figure 2). A chunked dataset's elements are stored in equal-sized chunks within the file, allowing fast access to subsets of dataset elements, as well as the application of compression and other filtering operations.

The HDF5 library normally packs data in HDF5 files together as tightly as possible, instead of aligning data to a particular byte boundary. However, most parallel file systems are tuned to perform best when data accesses fall on a particular chunk boundary. The HDF5 library allows the application to request alignment of all objects in a file over a particular size threshold, with the `H5Pset_alignment` API call. This allows aligning the chunks for chunked datasets to a favored block boundary for the file system.

For optimal performance, the dimensions for each chunk can be chosen so that the subset of the dataset that each parallel process accesses maps exactly to one chunk in the file. For example, if 4 processes were involved in performing I/O, the HDF5 dataset could be divided into 4×4 chunks of elements as shown in Figure 2, with each chunk aligned in the file. This would minimize lock contention in the parallel file system and maximize the file system throughput.

HDF5 files contain *metadata* that is used to store information about the datasets and other objects in the file. In particular, chunked datasets use a B-tree to map requests for accessing array elements from array coordinates to file offsets for each chunk, shown in Figure 3. Normally, the B-tree for a chunked dataset is tuned to maintain a compact form, limiting the depth and breadth of the B-tree produced. However, when the dimensions of a dataset are known in advance, the B-tree node width can be tuned with the `H5Pset_istore_k` API call to exactly match the number of chunks that will be produced, shown in Figure 4. This will minimize the number of I/O operations to bring the B-tree into memory, greatly reducing the cost of mapping from array coordinates to file offsets.

Metadata in HDF5 files is cached by the HDF5 library to improve access times for frequently accessed items. When operating in a sequential application, individual metadata items are flushed to the file (if dirty) and evicted from the metadata cache. However, when operating in a parallel application, these operations are deferred and batched together into *eviction epochs*, to reduce communication and synchronization overhead. At the end of an eviction epoch (measured by the amount of dirty metadata produced), the processes in the application are synchronized and the oldest dirty metadata items are

```c
H5AC_cache_config_t mdc_config;
hid_t file_id;

file_id = H5Fopen("file.h5", H5ACC_RDWR, H5P_DEFAULT);

mdc_config.version = H5AC__CURR_CACHE_CONFIG_VERSION;
H5Pget_mdc_config(file_id, &mdc_config)

mdc_config.evictions_enabled = FALSE;
mdc_config.incr_mode = H5C_incr__off;
mdc_config.decr_mode = H5C_decr__off;

H5Pset_mdc_config(file_id, &mdc_config);
```

Fig. 5. Example `C` code for disabling metadata cache evictions in HDF5.

flushed to the file.

To reduce the frequency of performing small I/O operations, it is possible to put the eviction of items from the HDF5 library's metadata cache entirely under the application's control with the sequence of API calls shown in Figure 5. This sequence of calls disables evictions from the metadata cache, unless `H5Fflush` is called or the file is closed. Suspending automatic eviction of cached metadata items also prevents frequently dirtied items from being written to the file repeatedly. Suspending metadata evictions may not be appropriate for all applications however, because if the application crashes before the cached metadata is written to the file, the HDF5 file will be unusable.

In addition to the application-controlled behavior described above, two improvements to the HDF5 library were made that impacted the results described here. The first improvement was to refactor the HDF5 library's behavior when the `H5Fflush` API routine is called. Previously, this API call would extend the file's size on disk to include all space allocated within the file (but possibly not written to yet). However, when operating in a parallel application, this operation resulted in a call to `MPI_File_set_size`, which currently has very poor performance characteristics on Lustre file systems. [1] Because an HDF5 file's size is not required to be accurately set until the file is closed, this operation was removed from `H5Fflush` and added to the code for closing a file.

The second optimization made to HDF5 again involved revising the behavior of the metadata cache when flushing multiple metadata items, either at the end of an eviction epoch or when closing the file. Previously, the HDF5 library would only use process 0 for writing dirty metadata items to the file, since early parallel file systems performed poorly when multiple processes performed small I/O operations simultaneously. However, modern parallel file systems, such as Lustre, have multiple storage nodes and can easily handle multiple I/O operations from many processes. With this in mind, the HDF5 library was modified to divide up the dirty metadata items into groups, with each process flushing one group of metadata items simultaneously.

Both of the improvements to the HDF5 library described above have been incorporated into the publicly available distribution, beginning with the 1.8.5 release in June, 2010.

B. MPI-IO

The Cray systems integrate MPI-IO capabilities into the Message Passing Toolkit (MPT). Cray added new algorithms to the 3.1 and 3.2 releases of MPT to improve I/O workload distribution through the use of new collective buffering techniques that respect Lustre stripe boundaries. The work described in this paper played a critical role in the early evaluation and tuning these new collective buffering implementations as MPT evolved.

The collective buffering algorithm used in MPT prior to release 3.1, `CB 0`, was based on the original ROMIO algorithm, which divided the I/O workload equally among all aggregators without regard to physical I/O boundaries or Lustre stripes. This method is inefficient when the division of workload results in multiple aggregators referencing the same physical I/O block or Lustre stripe, or when each aggregator has multiple segments of data with large gaps between the segments. The `CB 1` algorithm improved performance over `CB 0` by aggregating data into stripe-sized chunks and aligning I/O with stripe boundaries but did not maintain a one-to-one mapping between aggregators and I/O servers over multiple I/O calls. The `CB 2` algorithm quickly superseded `CB 1` with a much more effective approach to data aggregation, beginning with MPT release 3.2. The `CB 2` algorithm implements a Lustre-optimized scheme that uses static-cyclic and group cyclic Lustre stripe-aligned methods described by Liao and Choudhary [12]. The Cray implementation of these methods merged the Lustre "abstract device-driver for I/O" (ADIO) code from Sun Microsystems (available with the

[1] We have contacted the Lustre developers about the `MPI_File_set_size` performance issue, and their initial guess was that all processes were participating in the truncate call. However, our I/O traces show that only process 0 is calling truncate. Another possibility is that a large file could be fragmented on disk to the extent that the truncate call itself is causing the delay, but this seems unlikely to cause the catastrophic slowdowns we have recorded (on the order of tens of seconds).

ANL MPICH2 1.1.1p1 release) with its own code to provide additional tunable parameters to optimize performance.

To control these new features, Cray MPT 3.2 introduces two new MPI-IO hints: `striping_factor` sets the Lustre stripe count at file creation (or defaults to the stripe count of an existing file) and `striping_unit` does the same for the Lustre stripe size. Setting the `cb_nodes` hint to the stripe count creates a static-cyclic, one-to-one mapping between aggregators and OSTs. Alternatively, setting `cb_nodes` to a multiple of the stripe count creates a group-cyclic, several-to-one mapping. The value of the `cb_buffer_size` hint is ignored, and the buffer size is instead replaced with `striping_unit`.

`CB 2` is generally the optimal collective buffering alignment algorithm to use with MPT on Lustre file systems, because it minimizes lock contention and reduces I/O time. However, the communication overhead associated with this algorithm can exceed the savings in I/O time from accommodating the lock mechanism under some circumstances. This is the case if each process is writing small (relative to the stripe size) segments of data and the offsets for all processes' data are spread far apart relative to stripe size. Write performance for this I/O pattern will be poor whether collective buffering is used or not, but may be better by setting the MPI-IO hint `romio_cb_write=disable`. However, this I/O pattern is seldom observed in practice.

IV. Application Benchmarks

We have selected three HPC applications that represent common I/O patterns found in HPC computing. For each of these, we implemented a stand-alone benchmark to model the I/O pattern of the application.

A. Global Cloud Resolving Model

The Global Cloud Resolving Model (GCRM), a climate simulation developed at Colorado State University and led by David Randall [16], runs at resolutions fine enough to accurately simulate cloud formation and dynamics. In particular, it resolves cirrus clouds, which strongly affect weather patterns, at finer than 4km resolutions. Underlying the GCRM simulation is a geodesic-grid data structure containing nearly 10 billion total grid cells at the 4km resolution and scaling from tens of thousands to hundreds of thousands of processors – an unprecedented scale for atmospheric climate simulation codes that challenges existing I/O strategies.

Researchers at Pacific Northwest National Lab and LBNL have developed a data model, I/O library, and visualization pipeline for these geodesic grids [17]. The I/O library uses Morton ordering to linearize the indexing of individual cells in the grid and handles cell-, edge-, and corner-centered values. Currently, the library implements modules for pNetCDF and netCDF-4 and writes regular 2D grids (cell index $\times$ height) for cell-centered data and 3D grids (cell index $\times$ height $\times$ edge/corner) for edge-centered and corner-centered data. For our test purposes, we implemented a stand-alone I/O benchmark that recreates these 2D and 3D array patterns. Our benchmark uses H5Part, which is a simple veneer library that sits on top of HDF5. [18]. Our optimizations to HDF5 will benefit the netCDF-4 module in the GCRM I/O library, since netCDF-4 uses HDF5 as its lower layer.

The climate simulation code is intended to scale to 1km resolution at 40,960-way concurrency, and we perform a strong scaling study with our I/O benchmark up to this scale. Prior to our optimizations, the GRCM I/O library was able to achieve only 1 GB/s write bandwidth in tests at 4 km resolution. In order to keep the I/O time consumption to less than 5% of the total runtime the GCRM I/O library must sustain at least 2GB/s, and proportionally better performance when scaled to finer resolutions.

B. VORPAL

VORPAL [19] is a versatile plasma simulation code developed by Tech-X that enables researchers to simulate complex physical phenomena in less time and at a much lower cost than empirically testing process changes for plasma and vapor deposition processes. The kinetic plasma model incorporated in it is based on the particle-in-cell algorithm both in the electromagnetic and electrostatic limit. The I/O activities are dominated by periodically check pointing the variables, including 3D grids for storing scalar or vector field values. Because of load balancing, the grid decomposition can be uneven across tasks. We have implemented a stand-alone I/O benchmark in H5Part that writes 3D vector fields where adjacent tasks' subfields differ by ± 1 in each dimension.

C. Chombo

Chombo [20] is an adaptive mesh refinement (AMR) package used by applications to implement finite difference methods for solving partial differential equations on block structured grids. Chombo provides an framework for rapidly assembling portable, high-performance AMR applications for a broad variety of scientific disciplines, including combustion and astrophysics. To investigate Chombo's I/O behavior, we used a standalone benchmark that creates Chombo data structures, fills them with arbitrary data, and writes to a 1D dataset in HDF5 using the same I/O interface available in the production Chombo library. The benchmark synthesizes the same I/O patterns used by the full-scale Chombo simulations without the complexity and overhead of performing the computation. We are not targeting a specific application, but rather a generic use case of Chombo's I/O functionality.

V. Experimental Testbed

All of our test systems are Cray XTs with large scratch file systems organized roughly according to the block diagram shown in Figure 6, although the details of the interconnect between I/O servers and compute nodes differ among the systems.

A. JaguarPF

JaguarPF is a 18,688 node Cray XT5 system located at Oak Ridge National Laboratory (ORNL) with dual six-core

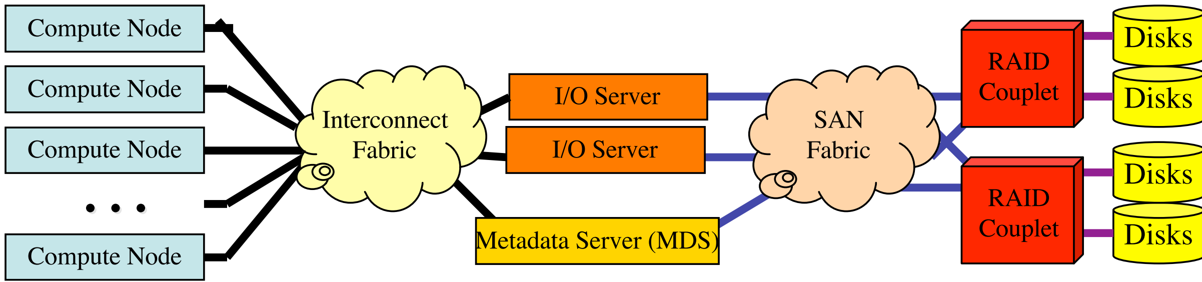

Fig. 6. The general architecture of cluster file systems such as Lustre that have separate object and metadata stores. In Lustre, the I/O servers are called Object Storage Servers (OSSs) and each can implement multiple Object Storage Targets (OSTs) that are visible to the client running on the compute nodes.

AMD Opteron processors and 16GB of memory per node (for 224,256 cores). JaguarPF's scratch space is served by a center-wide Lustre file system [21] with 96 Object Storage Servers (OSSs) serving 672 Object Storage Targets (OSTs), although Lustre limits the number of OSTs assigned to a single shared file to 160. The OSSs are backed by 48 Data Direct Network (DDN) S2A9900 controller couplets configured with 28 tiers of $8+2$ RAID6 arrays. A high-performance InfiniBand interconnect called the "Scalable I/O Network" (SION) connects all of the systems at Oak Ridge Leadership Computing Facility to the OSSs. Compute nodes on JaguarPF are connected through its SeaStar2+ interconnect to Lustre router nodes that forward I/O traffic to the JaguarPF segment of SION.

B. Franklin

Franklin is a 9,660 node Cray XT4 system located at the National Energy Research Scientific Computing Center (NERSC) with a quad-core AMD Opteron processor and 8GB of memory per node. Each XT4 node contains a quad-core 2.6 GHz AMD Opteron processor (for 38,640 cores total), tightly integrated to the XT4 interconnect via a Cray SeaStar2 ASIC and 6.4 GB/s bidirectional HyperTransport interface. All the SeaStar routing chips are interconnected in a 3D torus topology, where each node has a direct link to its six nearest neighbors. The XT4 runs Cray's Compute Node Linux (CNL) on compute nodes and SuSE SLES 9.0 on login nodes.

Franklin has two scratch spaces served by separate, dedicated 209 TB Lustre file systems. Each has 24 Object Storage Servers (OSSs) and one Metadata Server (MDS) connected directly via the SeaStar2 interconnect in a toroidal configuration to the compute nodes. The 24 OSSs implement a total of 48 OSTs and connect to six DDN 9550 RAID couplets, each with 16 tiers of $8+2$ RAID6 arrays.

C. Hopper (Phase 1)

Hopper is a Cray XT5 system that is being delivered to NERSC in two phases. Hopper Phase 1, which is currently delivered and operational, has 664 compute nodes each containing two 2.4 GHz AMD Opteron quad-core processors (for 5,312 cores total). When Phase 2 arrives, Hopper will have over 150,000 compute cores.

Like Franklin, Hopper also has two scratch spaces served by separate, dedicated 1 PB Lustre file systems. Each has 24 Object Storage Servers (OSSs) that implement 144 OSTs, and one Metadata Server (MDS). Similar to JaguarPF and unlike Franklin, Hopper uses Lustre router nodes in the main SeaStar2 interconnect to forward I/O traffic to the OSSs.

VI. Methodology

Our results are highly variable because the file systems we tested are shared resources and therefore prone to contention from other users. Even in cases of minimal contention for disk access, there can be contention for the interconnect between the OSTs and compute nodes from non-I/O MPI communication (e.g., see [22]). Finally, the complexity and depth of the I/O software stack and the server-client model used by Lustre also add to the variability.

We used three techniques to help us attain peak bandwidth values. First, for each benchmark run, we used three repetitions separated by 5 or 10 seconds. Second, we reran these groups of three repetitions at different times over the course of several weeks. Finally, on Franklin, we were able to monitor the status of the Lustre file system using an in-house web interface to the Lustre Monitoring Tool which allowed us to see conspicuous cases of contention (e.g., another user ran an application with regular checkpoints that overlapped with our run). We report maximum observed bandwidths in all of our plots to give the best approximation of peak bandwidths.

Throughout the process of optimizing HDF5 and collecting performance data, we used a modified version of the Integrated Performance Monitor to perform I/O tracing. A detailed discussion of these methods for the GCRM benchmark can be found in [23].

For each of our benchmarks, we also obtained an equivalent file-per-processor bandwidth by simulating the I/O pattern with a synthetic IOR test [11]. As stated earlier, file-per-process writes can usually achieve higher bandwidth than with single-shared-file writes but have some significant drawbacks. By running IOR tests in file-per-process mode for the various I/O patterns, we see what the upper limit is for single-shared-file and therefore what the goal is for optimizations. Because the Lustre file system sets a hard limit of 160 OSTs over which a shared file can be striped, we also restricted IOR to use only 160 OSTs on JaguarPF (out of the 672 available). Although this may lead to lower bandwidths when compared to other publications, we believe it would be an unfair comparison if

TABLE I
GCRM PARAMETERS

Cell-centered Data

Cores	640	2,560	10,240	40,960
Time steps	240	60	15	4
File size (GB)	243.8	243.8	243.8	260.0

Edge-centered Data

Cores	640	2,560	10,240	40,960
Time steps	40	10	3	1
File size (GB)	235.8	235.8	283.0	377.3

TABLE II
CHOMBO PARAMETERS

JaguarPF

Cores	2,560	10,240	40,960
Files	16	4	1
File size (GB)	45.6	182.4	729.6
Total size (GB)	729.6	729.6	729.6

Franklin & Hopper

Cores	640	2,560	10,240
Files	32	8	2
File size (GB)	11.4	45.6	182.4
Total size (GB)	364.8	364.8	364.8

the IOR tests had access to additional hardware resources. Also, we made sure to choose a subset of 160 OSTs that spanned all 96 available OSSs.

We used IOR in POSIX mode, which means it can write a large amount of data into the OS write buffer, then return a bandwidth that is actually a measure of the memory bandwidth and not the I/O bandwidth. To mitigate this effect, we modified IOR to allocate and touch a dummy array that filled 75% of available system memory. In previous experiments, we have found that this a good heuristic for defeating the OS write cache during benchmarking. This step is necessary to accurately simulate HPC applications that use a significant portion of memory for their data and therefore would not leave memory available for OS write buffers. We did not use the `O_DIRECT` flag to disable write buffering completely, because this would also have less accurately simulated HPC applications, which typically have some (albeit limited) memory available for write buffering. We also modified IOR to include a barrier between "segments" so that we could more accurately simulate the time-varying nature of our I/O patterns. [2]

VII. EXPERIMENTAL RESULTS

Our optimizations outperformed the baseline for all three benchmarks, on all three test systems, and across a range of concurrencies. In most cases, we were able to achieve at least half of the bandwidth of the file-per-processor approach while writing to a single, shared file. We now describe how we staged the optimizations and show their progressive improvements in bandwidth.

A. GCRM

The GCRM benchmark writes a fixed amount of data per processor per time step (1.6 MB for data-centered data and 9.6 MB for edge-centered), thus the file size scales weakly for a fixed number of time steps. To achieve strong scaling of file size (234 GB to achieve roughly 30 seconds of continuous writing), we weakly scaled down the number of time steps written per file (see Table I).

Our baseline configuration for the GCRM benchmark was to write from all processors using the MPI-POSIX virtual file driver with the default Lustre stripe size of 1 MB and striping over all available OSTs. We tested two progressive optimizations (see Figure 7):

1) In "chunking," we enabled HDF5's chunking mechanism together with 1 MB alignment, effectively mapping each processor's write call to a single, contiguous HDF5 dataset chunk. We also increased the stripe width to 2 MB for cell-centered data and 10 MB for edge-centered to map each chunk to a single OST.
2) In "metadata," we enlarged the size of the chunk index B-tree nodes and deferred all metadata writes until file close.

B. Chombo

The Chombo benchmark writes a single time step into a file, with each processor writing a different amount of data according to the layout of the adaptive mesh. Axillary 1D datasets describe which "boxes" belong to which processor and store the offsets for the boxes into the adaptive mesh data structure; these writes occur from processor 0 only and are included in our benchmark timings. It is possible for a processor to own no data, in which case it makes an empty hyperslab selection in HDF5. Because IOR does not support transfer sizes that vary by processor, we estimated the file-per-processor case by dividing the total file size by the number of processors and using that uniform transfer size (18,680 KB) in IOR.

Overall, the file size scales weakly. To achieve strong scaling of file size, we weakly scaled down the number of files written. To accommodate 40,960-way concurrency on JaguarPF, we had to write more data on that system (see Table II).

Our baseline configuration for the Chombo benchmark was to use the MPI-IO virtual file driver in HDF5 in collective mode with the `CB 0` algorithm. We used the default Lustre stripe size of 1 MB, striped over all available OSTs, and set `cb_nodes` to the number of stripes while leaving `cb_buffer_size` at the default value of 16 MB. Again, we tested two progressive optimizations (see Figure 8):

1) In "CB 2 / truncate," we enabled the `CB 2` algorithm, and in the HDF5 layer removed the `MPI_File_set_size` call and enabled the round-robin metadata cache flushing routine.
2) In "stripe size," we increased the stripe size, testing values in the range $\{4, 8, 16, 32, 64\}$ MB. Because the

[2] Despite these precautions, we did measure three outliers on JaguarPF that had bandwidths $2\times$ to $3\times$ higher than all other measurements. We suspect these may have been from caching effects, but were unable to verify this in the absence of rigorous memory or kernel profiling.

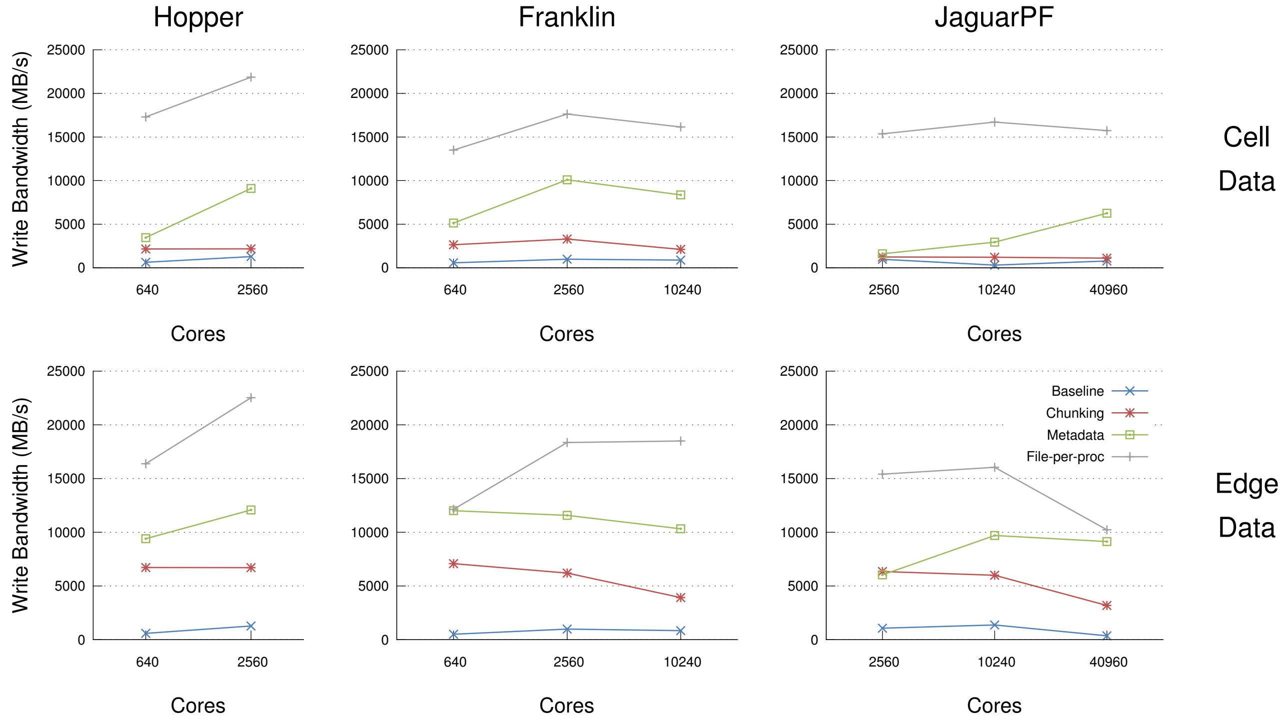

Fig. 7. Performance data for two configurations of the GCRM benchmark showing progressive optimizations to the HDF5 layer. File-per-processor tests from IOR show the ceiling we are trying to reach with our shared-filed approach.

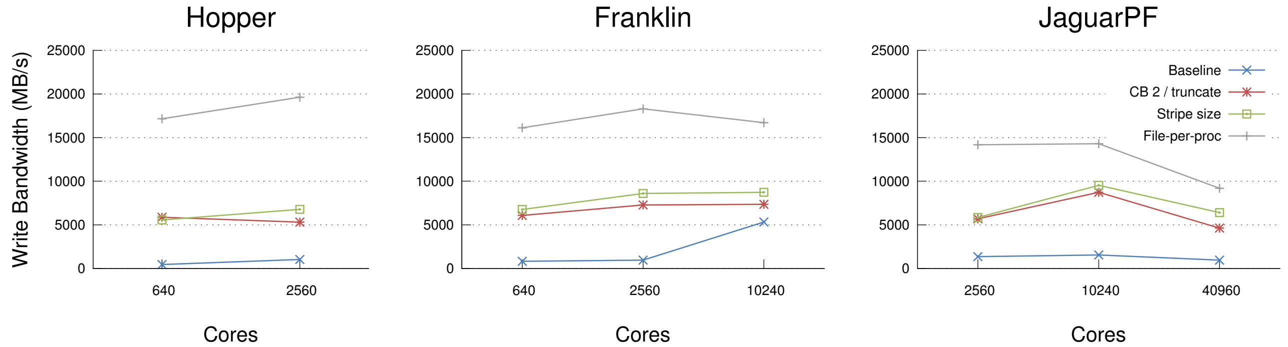

Fig. 8. Performance data for one configuration of the Chombo benchmark showing progressive optimizations to the MPI-IO and HDF5 layers. File-per-processor tests from IOR show the ceiling we are trying to reach with our shared-filed approach.

`CB 2` algorithm derives the `cb_buffer_size` value from the stripe size, this had the effect of increasing the amount of buffering relative to the amount of synchronization among CB aggregator nodes. In almost all cases this led to better performance, except on Hopper at 640-way concurrency where the 1 MB buffer size was optimal.

C. VORPAL

The VORPAL benchmark writes a slightly varying amount of 3D field data per processor per time step. We used two different grid sizes, 40^3 and 80^3, with variations of ±1, leading to 1,390 to 1,615 KB writes at 40^3 and 11,556 to 12,456 KB writes at 80^3. Again, we had to pick a uniform transfer size for IOR, and used the transfer sizes 1,500 KB and 12,000 KB that correspond to the uniform case where all processors have the same field dimensions (see Table III).

Our baseline and optimizations for the VORPAL benchmark (see Figure 9) were the same as for the Chombo benchmark. Like with the Chombo benchmark, increasing the stripe size did not always improve performance, as was the case on JaguarPF.

D. Analysis

For the GCRM pattern, a relatively small amout of metadata (only a few MBs) can cause catastrophic slowdowns when the metadata writes are pooly aligned and fragmented. This effect, on Franklin specifically, is described in detail and with supporting I/O traces in [23]. Our optimizations to consolidate and align metadata writes contribute large performance gains,

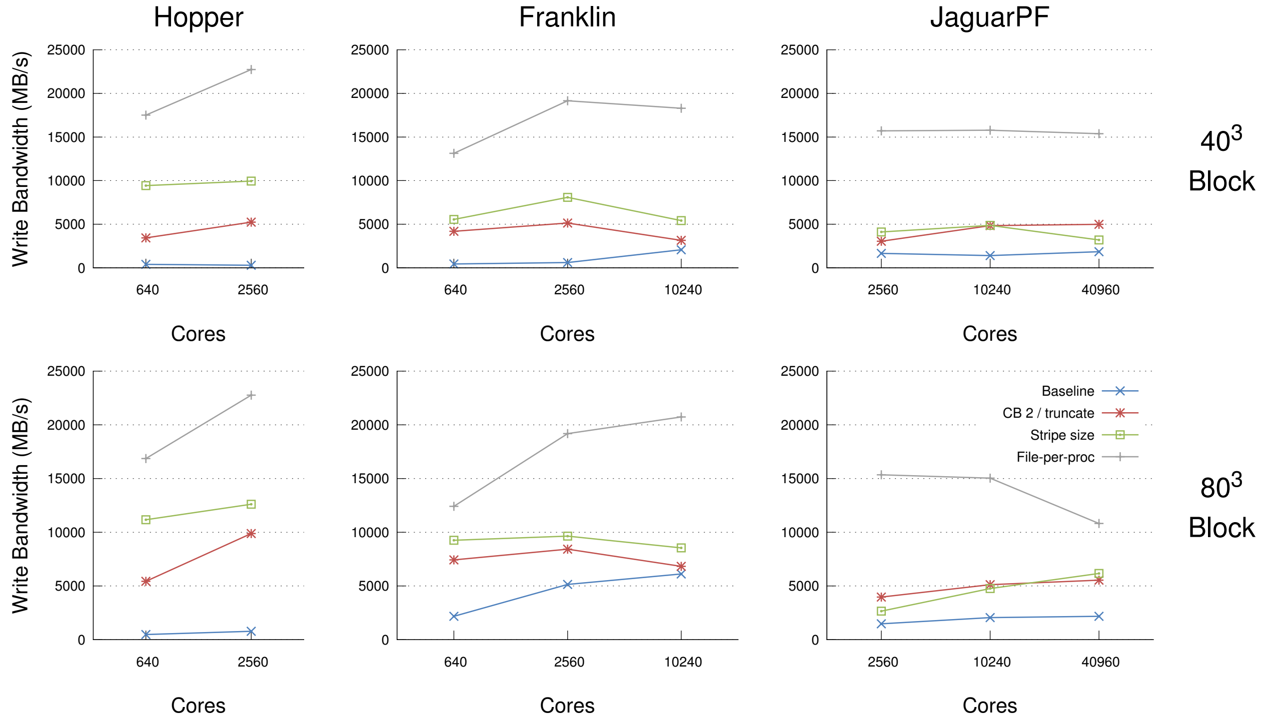

Fig. 9. Performance data for two configurations of the VORPAL benchmark showing progressive optimizations to the MPI-IO and HDF5 layers. File-per-processor tests from IOR show the ceiling we are trying to reach with our shared-filed approach.

TABLE III
VORPAL PARAMETERS

JaguarPF

Cores	2,560	10,240	40,960
40^3 time steps	128	32	8
40^3 file size (GB)	468.5	468.5	468.5
80^3 time steps	16	4	1
80^3 file size (GB)	468.7	468.7	468.7

Franklin & Hopper

Cores	640	2,560	10,240
40^3 time steps	256	64	8
40^3 file size (GB)	234.2	234.2	234.2
80^3 time steps	32	8	2
80^3 file size (GB)	234.3	234.3	234.3

most notably on Franklin, but also on Hopper and JaguarPF at higher concurrency. At lower concurrency, there are many fewer write calls per time step and per OST (since both Hopper and JaguarPF have more OSTs available then Franklin), which causes worse performance through a "Law of Large Numbers" effect that is also detailed in [23].

For the VORPAL and Chombo patterns, increasing the stripe size for the `CB 2` algorithm leads to performance gains on Hopper and Franklin because the larger buffer size during the aggregation phase reduces the frequency of synchronization points. Surprisingly, the increased buffer size seems to have little effect on performance on JaguarPF, which may be related to the larger scale and complexity of its interconnect. Yu and Vetter's [24] modifications to introduce "partitioned collective I/O" to Cray's `CB 0` algorithm on the older Jaguar XT4 system demonstrated impressive performance gains, and a similar approach to tuning the communication in the aggregation phase of the `CB 2` algorithm is a promising direction for future research. Tailoring the communication phase of collective buffering to the underlying network topology was also a successful strategy for Yu et al. [25] in their I/O tuning of the Blue Gene/L architecture.

File-per-processor performance on JaguarPF seems to drop at 40,960-way concurrency for all I/O patterns. We believe this is caused by the increasing metadata pressure of creating and writing to so many individual files. This is the same conjecture that Fahey et al. [26] made to explain a similar decline in performance at scale for the Lustre file system on Jaguar XT4.

VIII. CONCLUSION AND FUTURE WORK

We have demonstrated a collection of optimizations for MPI-IO and HDF5 that enable high-performance, shared-file I/O on Lustre file systems. Our results cover three commonly-used HPC I/O patterns and were validated across three different Lustre configurations at a range of concurrencies up to 40,960-way parallel. Overall, we are able to demonstrate performance that is much more competitive with file-per-processor performance than our unoptimized baseline.

Although some of the optimizations have dramatic effects on data layout, they require no changes to the user's data model, which points to the advantages of HDF5's separation of data model from data layout. Moreover, all of the optimizations are compliant with the existing HDF5 file format, which preserves backward compatibility. The optimizations

will also be transparently usable by the netCDF-4 I/O library used by the Community Climate System Model (CCSM4), which will be important for accommodating the huge data volumes produced by that codebase as is anticipated over the next five years.

We expect that future improvements to MPI-IO and HDF5 will continue to focus on optimizing aggregation and metadata operations, and could include the following:

Reducing communication overhead in MPI-IO. As described earlier, information is exchanged among all processes as a step in the collective buffering process. The overhead is significant in some cases and also creates a synchronization point that may be avoidable. Yu and Vetter [27] have studied this phenomenon on the Cray XT and proposed a partitioned approach to collective communication that can yield up to $4\times$ improvements in collective I/O performance.

Topologically aware assignment of aggregators in MPI-IO. Because aggregators must send data through the interconnect to get to the file system, placing aggregators closer to the I/O nodes could reduce network traffic.

Dedicating processes to serve as aggregators in MPI-IO. As mentioned in Section II-B, setting aside a set of processes solely responsible for managing collective I/O operations would allow more concurrency of computation and I/O.

Aggregating HDF5 metadata operations. Using collective I/O operations when flushing metadata items will give the MPI-IO implementation the maximum amount of information about the I/O operation to perform, allowing it to aggregate the I/O accesses and optimize them for the underlying parallel file system. HDF5's caching mechanism could also be modified to aggregate multiple metadata accesses into fewer, larger ones.

Acknowledgment

We thank Noel Keen (LBNL) for sharing his source code for the standalone Chombo benchmark and for initially discovering the performance issue with truncates on Lustre.

This research used resources of the National Center for Computational Sciences at Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725; and resources of the National Energy Research Scientific Computing Center (NERSC), which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

[1] "Lustre," http://www.lustre.org/.

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

[3] Unidata, "netCDF (network Common Data Form)," http://www.unidata.ucar.edu/software/netcdf.

[4] 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 *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, 2003.

[5] 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 *SC '09: Proceedings of the 2009 ACM/IEEE Conference on Supercomputing*, Portland, Oregon, 2009.

[6] J. Lofstead, F. Zheng, S. Klasky, and K. Schwan, "Adaptable, metadata rich IO methods for portable high performance IO," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Rome, Italy, 2009.

[7] R. Thakur, W. Gropp, and E. Lusk, "Optimizing noncontiguous accesses in MPI-IO," *Parallel Computing*, vol. 28, no. 1, pp. 83–105, 2002.

[8] J. M. del Rosario, R. Bordawekar, and A. Choudhary, "Improved parallel I/O via a two-phase run-time access strategy," *SIGARCH Computer Architecture News*, vol. 21, no. 5, pp. 31–38, 1993.

[9] R. Thakur and A. Choudhary, "An extended two-phase method for accessing sections of out-of-core arrays," *Scientific Programming*, vol. 5, no. 4, pp. 301–317, 1996.

[10] B. Nitzberg and V. Lo, "Collective buffering: Improving parallel I/O performance," in *HPDC '97: Proceedings of the 6th IEEE International Symposium on High Performance Distributed Computing*, 1997.

[11] H. Shan, K. Antypas, and J. Shalf, "Characterizing and predicting the I/O performance of HPC applications using a parameterized synthetic benchmark," in *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, Austin, Texas, 2008.

[12] 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 *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, Austin, Texas, 2008.

[13] P. M. Dickens and J. Logan, "A high performance implementation of MPI-IO for a Lustre file system environment," *Concurrency and Computation: Practice and Experience*, 2009.

[14] K. Coloma, A. Ching, A. Choudhary, W. Liao, R. Ross, R. Thakur, and L. Ward, "A new flexible MPI collective I/O implementation," in *IEEE Conference on Cluster Computing*, 2006.

[15] A. Nisar, W.-k. Liao, and A. Choudhary, "Scaling parallel I/O performance through I/O delegate and caching system," in *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, Austin, Texas, 2008.

[16] D. A. Randall, T. D. Ringler, R. P. Heikes, P. Jones, and J. Baumgardner, "Climate modeling with spherical geodesic grids," *Computing in Science and Engineering*, vol. 4, pp. 32–41, Sep/Oct 2002.

[17] K. Schuchardt, "Community access to Global Cloud Resolving Model and data," http://climate.pnl.gov/.

[18] Andreas Adelmann, Achim Gsell, B. Oswald, T. Schietinger, E. Wes Bethel, John Shalf, Cristina Siegerist, Kurt Stockinger, Prabhat and Mark Howison, "H5part software," 2006–2010, https://codeforge.lbl.gov/projects/h5part/.

[19] "VORPAL, Versatile Plasma Simulation Code," http://www.txcorp.com/products/VORPAL/.

[20] "Chombo, Infrastructure for Adaptive Mesh Refinement," http://seesar.lbl.gov/ANAG/chombo/.

[21] G. Shipman, D. Dillow, S. Oral, and F. Wang, "The Spider center wide file system: From concept to reality," in *Proceedings of the Cray User Group (CUG) Conference*, Atlanta, GA, May 2009.

[22] J. Mache, V. Lo, and S. Garg, "The impact of spatial layout of jobs on I/O hotspots in mesh networks," *Journal of Parallel and Distributed Computing*, vol. 65, no. 10, pp. 1190–1203, 2005.

[23] A. Uselton, M. Howison, N. Wright, D. Skinner, N. Keen, J. Shalf, K. Karavanic, and L. Oliker, "Parallel I/O performance: From events to ensembles," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Atlanta, Georgia, 2010.

[24] W. Yu, J. Vetter, R. S. Canon, and S. Jiang, "Exploiting Lustre file joining for effective collective I/O," in *Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid 07)*, 2007.

[25] H. Yu, R. K. Sahoo, C. Howson, J. G. C. G. Almási, M. Gupta, J. E. Moreira, J. J. Parker, T. E. Engelsiepen, R. B. Ross, R. Thakur, R. Latham, and W. D. Gropp, "High performance file I/O for rhe Blue Gene/L supercomputer," in *International Symposium on Performance Computer Architecture*, Austin, Texas, 2006.

[26] M. Fahey, J. Larkin, and J. Adams, "I/O performance on a massively parallel Cray XT3/XT4," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2008.

[27] W. Yu and J. S. Vetter, "ParColl: Partitioned collective I/O on the Cray XT," in *ICPP '08: Proceedings of the 2008 37th International Conference on Parallel Processing*, 2008, pp. 562–569.

View publication stats