...And eat it too: High read performance in write-optimized HPC I/O middleware file formats

Milo Polte*, Jay Lofstead†, John Bent‡, Garth Gibson*, Scott A. Klasky§, Qing Liu§ Manish Parashar¶,
Karsten Schwan†, Matthew Wolf†,
*Carnegie Mellon University †Georgia Institute of Technology ‡Los Alamos National Lab
§Oak Ridge National Lab ¶Rutgers University

I. Introduction

As HPC applications work on increasingly larger datasets, both the frequency of checkpoints needed for fault tolerance [1] and the resolution and size of Data Analysis Dumps are expected to increase proportionally. For example, a common practice amongst codes such as GTC[2] running at scale on Jaguar, the petascale computer at ORNL, is to write restart files once per hour. This ensures that the most time lost in a simulation running on p-processors is p hours. Application scientists are always forced to balance how much time is spent writing all of the output data with the total runtime cost of the simulation run, and very often try to write the minimum amount of data due to inefficient output mechanisms.

As applications run at larger scales, in order to maintain an acceptable ratio of time spent performing useful computation work to time spent performing I/O, write bandwidth to the underlying storage system must increase proportionally to the increase in the computation size. Unfortunately, popular scientific self-describing file formats such as netCDF [3] and HDF5 [4] are designed with a focus on portability and flexibility. Extra care and careful crafting of the output structure and API calls is required to optimize for write performance using these APIs. Without this extra effort, this often results in poor performance from the underlying storage system [5].

To provide sufficient write bandwidth to continue to support the demands of scientific applications, the HPC community has developed a number of I/O middleware layers. A few examples include the Adaptable IO System (ADIOS) [6], [7], a library developed at Oak Ridge National Laboratory providing a high-level IO API that can be used in place of pnetcdf or HDF5 to do much more aggressive write-behind and efficient reordering of data locations within the file; and the Parallel Log-structured Filesystem (PLFS) [8], a stackable FUSE [9] filesystem developed at Los Alamos National Laboratory that decouples concurrent writes to improve the speed of checkpoints. Since ADIOS is an I/O componentization that affords selection of different I/O methods at or during runtime, through a single API, users can have access to MPI-IO, Posix-IO, parallel HDF5, parallel netCDF, and staging methods [10]. The ADIOS BP file format [11] is a new log-file format that has a superset of the features of both HDF5 and netCDF, but is designed to be portable and flexible while being optimized for writing. PLFS takes a different approach and is mounted as a stackable filesystem on top of an existing parallel filesystem. Reads or writes to the PLFS filesystem are transparently translated into operations on per-process log files stored in the underlying parallel filesystem. Since PLFS performs this translation without application modification, users can write in HDF5, netCDF, or app-specific file formats and PLFS will store the writes in a set of efficiently written log-formatted files, while presenting the user with a logical 'flat' file on reads. Despite their different approaches, the commonality behind both of these middleware systems is that they both write to a log file format. As shown in [12] and [8], writes are fully optimized in both systems, sometimes resulting in 100x improvements over writing data in popular file formats.

While these techniques dramatically improve write performance, the obvious concern with any write optimized file format would be a corresponding penalty on reads. In the log-structured filesystem [13], for example, a file generated by random writes could be written efficiently, but reading the file back sequentially later would result in very poor performance. Simulation results require efficient read-back for visualization and analytics and while most checkpoint files are never used, the efficiency of restart is still important in the face of inevitable failures. The utility of write speed improving middleware would be greatly diminished without acceptable read performance.

In this paper we examine the read performance on large parallel machines and compare these to reading data either natively or to other popular file formats. We compare the reading performance in two different scenarios: 1) Reading back restarts from the same number of processors which wrote the data and 2) Reading back restart data from a different number of processors which wrote the data.

We observe that not only can write-optimized I/O middleware be built to not greatly penalize read speeds, but for important workloads techniques that improve write performance can, perhaps counterintuitively, improve read speeds over reading a contiguously organized file format. In the remainder of this paper, we investigate this further through case studies of PLFS and ADIOS on simulation checkpoint restart.

II. PLFS

As described above, the Parallel Log-Structured Filesystem (PLFS) is a FUSE-based stackable filesystem that accelerates the write performance of N-1 checkpoints by decoupling the

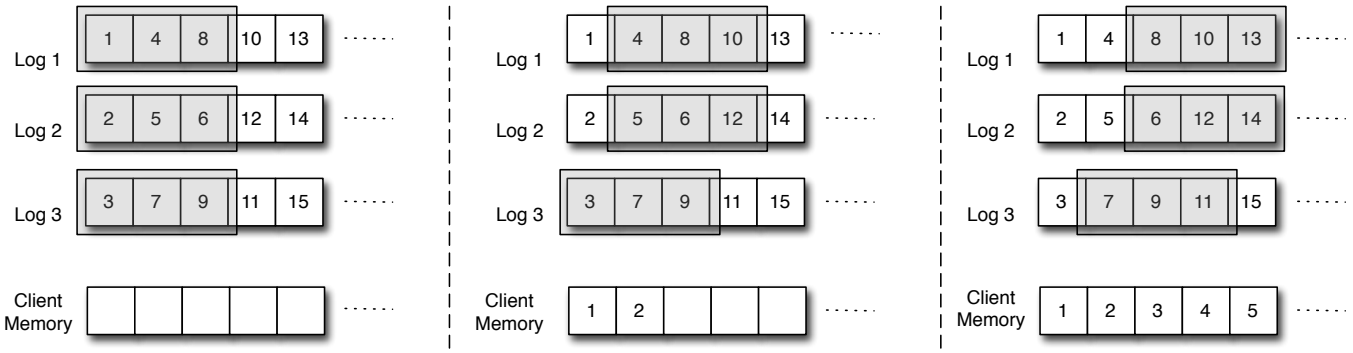

Fig. 1. Progress of sequential reading of a checkpoint log file

concurrent access of multiple processes writing to a single, shared file into parallel writes appending to per-process log files.

Filesystems perform well on sequential access to single-writer files compared to strided, concurrent writes to a shared file. This is true even of parallel filesystems; although they spread I/O across multiple servers and spindles, parallel filesystems still must deal with contention within in a single file object or stripe. So it is unsurprising that PLFS's log-structured writing has achieved checkpoint high write performance improvements on real applications and checkpoint benchmarks [8]. However, for pathological combinations of write and read patterns (for example, a random write pattern read back sequentially), a log file format is expected to exhibit poor read performance due to the additional seeks.

To understand why PLFS performs so well on its read path, it helps to understand that while in reading sequentially from a log file format consisting of random writes can be slow, the checkpointing and scientific applications that write to PLFS do so in a structured manner. An analysis of the trace repository of checkpoints run through PLFS available at [14] shows that all of the checkpoints' processes write increasing offsets. The result is that every data log file is written as a series of entries, monotonically increasing in the logical offset.

Figure 1 is a graphical representation of three time steps of a single client reading sequentially from a PLFS checkpoint file created by three writing processes and stored as log structured files on an underlying parallel filesystem. Squares represent segments of client memory or entries in a log file, and the numbers inside correspond to their logical offset. Grey boxes represent the parallel filesystem's read-ahead buffers for each log. Note that the log files are monotonically increasing in logical offset, and the next request from the client is always at the front of one of the log files. Due to this property, as the client continues reading from the checkpoint file, rather than the log format resulting in expensive seeks, the read ahead buffers will slide forward through the log files, reading them sequentially in parallel. ADIOS's BP format is somewhat different, but by storing variables together in log formatted files, it too allows for efficient read back in an analogous manner.

Below we discuss how this property allows for efficient read-back of PLFS checkpoint files on uniform and non-uniform restart, by examining one checkpoint benchmark: The `mpi_io_test` benchmark from LANL [15]. This benchmark is designed to represent a simple checkpoint I/O workload. In the examples below, it is configured to write a single 20 GB file in 47KB strided writes from a varying number of processes on the LANL Roadrunner system. We write both to PLFS and to the underlying parallel filesystem directly and compare the results.

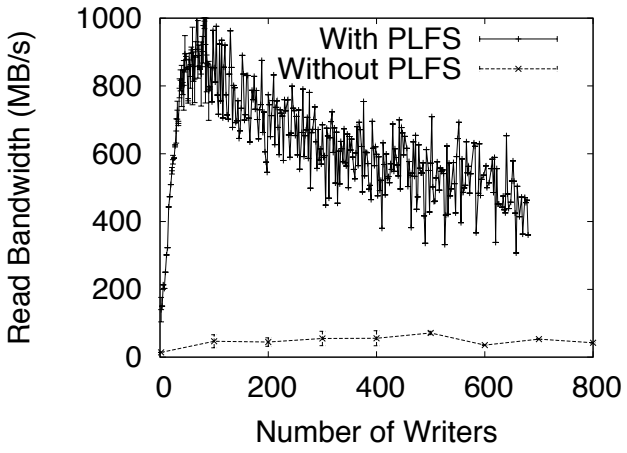

Fig. 2. Uniform Reads for PLFS

A. Uniform Restart in PLFS

Reading back a checkpoint file with the same number of processes is highly efficient in PLFS. During a checkpoint restart, each reading process will read back one writing process's data from the checkpoint in the same pattern in which it was written. In the case of PLFS, since the checkpoint was written in per-process monotonically increasing offsets, the readers will each process individual log structured files in sequential order. The scenario is similar to that shown in Figure 1, except with multiple clients each reading individual log files allowing the underlying parallel filesystem to make efficient use of read ahead buffers and disks. Results are shown

in show in Figure 2. Performance quickly ramps up and then gradually falls off as the overhead of many readers begins to dominate the transfer time of 20 GBs.

By contrast, all processes reading directly from a single file stored on the parallel filesystem elicits poor performance as multiple readers are at any given time all issuing a series of small reads to a relatively narrow window that gradually moves through of the file. The result is possible contention within the file, more seeks, and poor utilization of spindles and servers (since a narrow region of the file is generally not spread as widely across disks and servers as multiple log structured files). The result is that the read back speed of readers without PLFS does not scale noticeably with more processes.

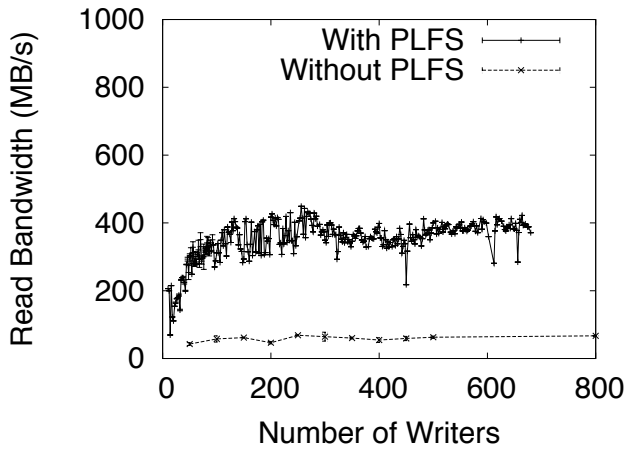

Fig. 3. Non-Uniform Reads for PLFS

B. Non-Uniform Restart in PLFS

Reading back a checkpoint file with fewer processes is somewhat less efficient in PLFS, as shown in Figure 3. Here a file is written by a varying number of processes and read back by one fewer reader processes. The situation is similar to the previous section, however, now each log structured file is processed by two processes instead of one. The result is some additional read contention within each PLFS log file.

Reading directly from a single, flat file stored on the underlying parallel filesystem is analogous to the uniform case, however, and once again we see it exhibit a non-scaling read performance.

III. ADIOS

The ADIOS architecture is designed for portable, scalable performance in two key ways. First, the componentization of the IO implementation affords selecting the highest performance output mechanism for a particular platform without requiring any source code changes. Second, the default BP file format achieves excellent write performance by allowing delaying consistency checks [11]. This enables aggressive buffering and decouples processes during the output process. The output format is organized around the concept of a *Process Group*, a collection of data typically written by a single process. Each process is assigned a section of the file to write the local data with sufficient annotations to both identify the items written and place the portions of global items properly in the global space. For example, global arrays are written as a local array in the assigned process group with annotation of the global array it is part of, the global dimensions, the local dimensions, and the offsets into the global space this piece represents. Scalars, other local values, and data attributes are stored similarly. This format decouples processes and data reorganization when writing by avoiding constructing a contiguous global array on disk when writing and selecting a slice appropriate for the local process on read, assuming the same number of processes and decomposition are used when reading. The perceived potential penalty for the gain in write performance for this decision is the inability to read the data back in efficiently. For restarts of a p process output read back in on p processes, this is an optimal format. Each process reads the data from exactly one process group independently and requires no data reorganization. For other process configurations, the performance story was unknown. This evaluation resolves the restart performance question.

A. Evaluation

Four different experiments are performed on one of the most difficult data decompositions, a 3-D domain decomposition. For this application, the simulation domain is a 3-D space divided into boxes along the X, Y, and Z dimensions with each process responsible for one rectangular area. For netCDF and HDF5 formatted data, this output is reorganized into a contiguous format for the entire 3-D domain with selective reading on restarts. The BP format stores each rectangular area as a single block in a process group with each area in a non-contiguous space in the file. In order to read the entire 3-D domain from a single process, in order, the system would need to skip around the file or read in chunks reorganizing in memory. An IO kernel for the Pixie3D MHD code [16] is tested.

Pixie3D has three modes of operation tested for this study. The Small configuration consists of 256 KB of data per process, Medium has 16 MB per process, and 128 MB per process for Large. The data is divided up evenly among 8 variables and also contains many small scalars values. To test the restart read performance, various process counts from 128 to 2048 are employed to write the data with the same or one-half the number of process used to write to read the data back in. NetCDF-5 formatted data is compared.

For all tests, the best results for a series of four runs for each data point is selected for the BP and netCDF performance. The horizontal axis represents the number of writers employed to generate the restart data no matter the number of processes used to read the data back in.

The experiments are performed on Jaguar, the Cray XT4 at ORNL, striping across all 144 Lustre storage targets and using a stripe size of 1 MB.

B. Pixie3D Uniform Restarts

To establish a baseline, a uniform restart is tested. This is using the same number of processes to read the restart as wrote it originally. Figure 4 shows the performance results. For large data, the performance approaches the IOR benchmark performance [17] for the machine.

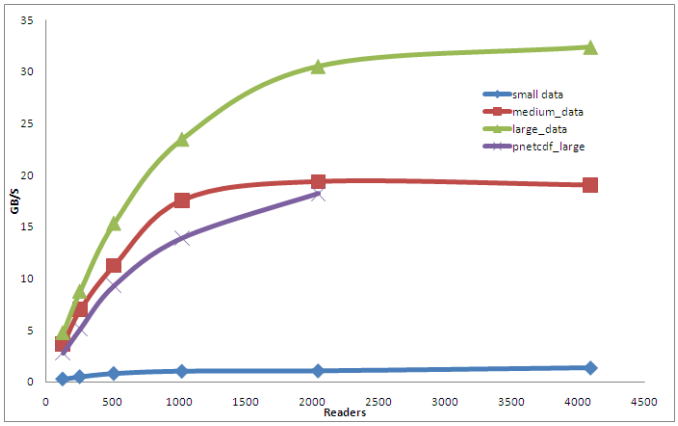

Fig. 4. Uniform Reads for the Pixie3D data

C. Pixie3D Small

For the small data, good read performance just cannot be attained no matter the number of processes used to read or write the data. The overall data size is too small to overcome the inherent overhead of the parallel file system. Figure 5 shows the performance for reading the restart output on half as many processes as wrote the restart. The horizontal axis represents the number of processes that wrote the data originally.

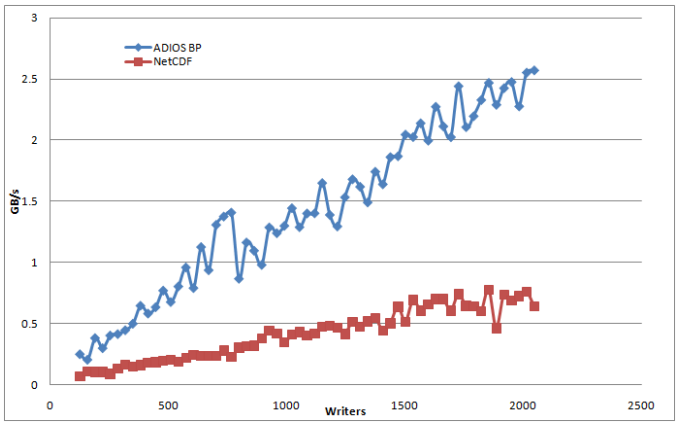

Fig. 5. Small Model, Half Process Count

The BP formatted data was able to be read faster than the netCDF formatted data. The trendlines for the performance clearly show the performance gap should continue to widen as the process count increases.

D. Pixie3D Medium

Restarts using the medium data model can achieve much better performance, but still cannot achieve more than a fraction of the theoretical maximum performance for the system. Figure 6 shows the performance for restarting using half as many processes.

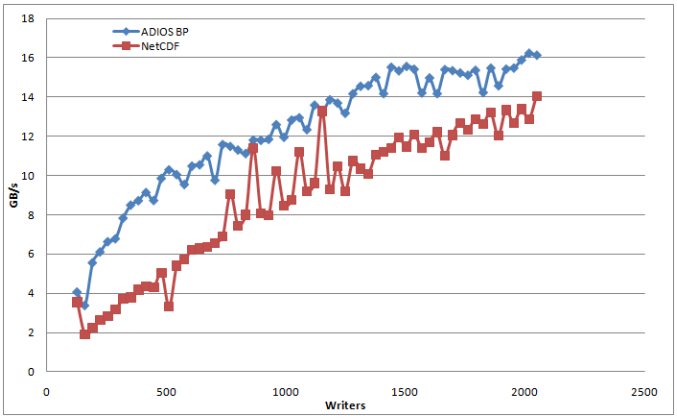

Fig. 6. Medium Model, Half Process Count

For the 'half' case, the BP performance consistently outperforms netCDF with the performance gap narrowing slightly as the process count increases.

E. Pixie3D Large

The large data cases finally reach the maximum general performance seen for applications in production use. Figure 7 shows the performace for using half as many processes to restart.

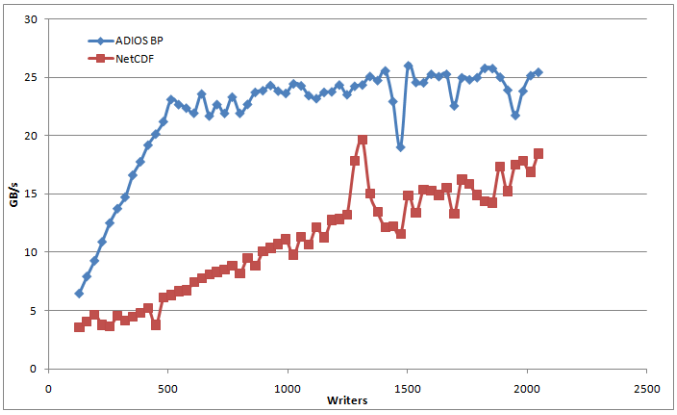

Fig. 7. Large Model, Half Process Count

F. Discussion

Overall, the performance for all configurations of BP data is either absolutely better or about the same as a contiguous format like netCDF. Comparing the half-processes restarts with the uniform restarts, the performance for large and medium data is about 80% of the uniform read rate. Small data is about the same performance.

IV. Conclusion and Future Work

Previously, both ADIOS's BP format and PLFS's log-based format have shown excellent performance for writing data due to the lack of reorganization of the data when writing. This generated two main open questions. First, what is the performance for reading all of the data, such as for a restart. Second, for reading a subset of the data, such as for a typical analysis task, how much does the log-based format penalize the user. To answer the restarts question, the 'worst case' scenario, reputed to be a 3-D domain decomposition, was tested. As has been shown, both of these log-based formats do not suffer from the expected penalties for storing the data in the log-based formats when reading restarts. This holds not just for reading on the same number of processes as wrote the data, but also when fewer read the data back in.

When reading on the same number of processes as generated the data, no reorganization effort is required for any process to read the desired data and the data is stored in larger, single chunks on the various storage targets. Each process can read large, contiguous blocks of data from the file reducing the likelihood of interfering with other processes reading other data. This result was expected. The more difficult question of the performance when restarting on a different number of processes requires a deeper analysis.

The belief that a canonical storage format is most efficient for reading is based on the notion that most often, contiguous chunks of data will be read. The closer that data is stored together, the better the overall performance. While this may have been true on a single spindle with a single reader, with parallel processes and file systems, a different consideration must be made. Parallel file systems rely on large, contiguous reads or writes to each storage target for optimal performance. Read caching on the disks and other hardware layer components yield optimal sizes generally no less than 1 MB per operation. Reading a smaller chunk will cause the entire 1 MB chunk to be read expecting that the rest of the block may be requested shortly. For large, global arrays, the 1 MB increment is too small when the data is spread across many storage targets. Instead of reading small bits from each storage target to assemble the proper data pieces, reading large, contiguous chunks is much more efficient. When the data is stored in a canonical format, the data is spread too thin to gain the performance advantages of large, contiguous reads, whereas the log-based formats yield better performance due to the locality. Internal to the I/O library, the system can read a few large chunks of data and reorganize in memory. Canonical format data must be read from a larger number of physical storage targets in smaller chunks and reorganized in memory. This yields contention on the storage targets (e.g. jitter) and in the machine network due to the large number of smaller reads. Although neither BP nor the PLFS format is necessarily directly knowledgeable about the underlying storage organization, the formats are designed understanding that the file system employed will use the strategy of striping the data across many storage targets and performing fewer, larger operations to attain high performance on these systems. The typical canonical formats do not take this current file system design into account largely due to required backwards compatibility and desire to maintain compatibility with the large number of useful analysis tools. This compatibility is commendable, but it does not mean that these formats are necessarily good choices for storing restarts on modern parallel file systems.

The second question of the performance for data analysis tasks is still open. These results demonstrate that a log-based format has viable performance for restarts of various data sizes and process counts but not necessarily superior for read workloads overall. Particularly missing is evidence that the log-based format works well for analysis patterns. Work is ongoing to evaluate how the restart performance relates to the analysis task performance.

References

[1] B. Schroeder and G. A. Gibson, "Understanding failures in petascale computers," *Journal of Physics: Conference Series*, vol. 78, p. 012022 (11pp), 2007. [Online]. Available: http://stacks.iop.org/1742-6596/78/012022

[2] S. Klasky, S. Ethier, Z. Lin, K. Martins, D. McCune, and R. Samtaney, "Grid -based parallel data streaming implemented for the gyrokinetic toroidal code," in *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*. Washington, DC, USA: IEEE Computer Society, 2003, p. 24.

[3] H. L. Jenter and R. P. Signell, "NetCDF: A Public-Domain-Software Solution to Data-Access Problems for Numerical Modelers," 1992.

[4] "The HDF Group," http://www.hdfgroup.org/.

[5] National Energy Research Scientific Computing Center, "I/O Patterns from NERSC Applications," https://outreach.scidac.gov/hdf/NERSC_User_IOcases.pdf.

[6] 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 *CLADE '08: Proceedings of the 6th international workshop on Challenges of large applications in distributed environments*. New York, NY, USA: ACM, 2008, pp. 15–24.

[7] O. R. N. Laboratory, "http://adiosapi.org/."

[8] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate, "Plfs: A checkpoint filesystem for parallel applications," *SC Conference*, vol. 0, 2009.

[9] "FUSE: Filesystem in Userspace," http://fuse.sourceforge.net/.

[10] H. Abbasi, J. Lofstead, F. Zheng, S. Klasky, K. Schwan, and M. Wolf, "Extending i/o through high performance data services," in *Cluster Computing*. Austin, TX: IEEE International, September 2007.

[11] J. Lofstead, F. Zheng, S. Klasky, and K. Schwan, "Input/output apis and data organization for high performance scientific computing," in *In Proceedings of Petascale Data Storage Workshop 2008 at Supercomputing 2008*, 2008.

[12] J. Lofstead, F. Zheng, S. Klasky, and K. Schwan, "Adaptable, metadata rich io methods for portable high performance io," in *In Proceedings of IPDPS'09, May 25-29, Rome, Italy*, 2009.

[13] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Transactions on Computer Systems*, vol. 10, pp. 1–15, 1991.

[14] "PLFS I/O Map Repository," http://institute.lanl.gov/plfs/maps/.

[15] Gary Grider and James Nunez and John Bent, "LANL MPI-IO Test," http://institutes.lanl.gov/data/software/, jul 2008.

[16] L. Chacón, "A non-staggered, conservative, $\nabla \dot{s} B \rightarrow = 0$, finite-volume scheme for 3D implicit extended magnetohydrodynamics in curvilinear geometries," *Computer Physics Communications*, vol. 163, pp. 143–171, Nov. 2004.

[17] H. Shan and J. Shalf, "Using ior to analyze the i/o performance for hpc platforms." Cray User's Group, 2007.

[18] J. Li, W. keng 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," *SC Conference*, vol. 0, p. 39, 2003.