24/7 Characterization of Petascale I/O Workloads

Philip Carns, Robert Latham, Robert Ross, Kamil Iskra, Samuel Lang
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439
{carns,robl,rross,iskra,slang}@mcs.anl.gov

Katherine Riley
Argonne Leadership Computing Facility
Argonne National Laboratory
Argonne, IL 60439
riley@mcs.anl.gov

***Abstract*—Developing and tuning computational science applications to run on extreme scale systems are increasingly complicated processes. Challenges such as managing memory access and tuning message-passing behavior are made easier by tools designed specifically to aid in these processes. Tools that can help users better understand the behavior of their application with respect to I/O have not yet reached the level of utility necessary to play a central role in application development and tuning. This deficiency in the tool set means that we have a poor understanding of how specific applications interact with storage. Worse, the community has little knowledge of what sorts of access patterns are common in today's applications, leading to confusion in the storage research community as to the pressing needs of the computational science community.**

This paper describes the Darshan I/O characterization tool. Darshan is designed to capture an accurate picture of application I/O behavior, including properties such as patterns of access within files, with the minimum possible overhead. This characterization can shed important light on the I/O behavior of applications at extreme scale. Darshan also can enable researchers to gain greater insight into the overall patterns of access exhibited by such applications, helping the storage community to understand how to best serve current computational science applications and better predict the needs of future applications. In this work we demonstrate Darshan's ability to characterize the I/O behavior of four scientific applications and show that it induces negligible overhead for I/O intensive jobs with as many as 65,536 processes.

I. Introduction

Efficient use of extreme-scale computing resources often requires extensive application tuning. To tune applications most effectively, application developers need to be able to observe the behavior of applications before and after changes are made, so that they can assess the impact of tuning efforts. In the areas of memory and communication subsystem behavior, many tools are available that provide insight into how the application is interacting with the subsystem [1], [2], [3], [4]. These utilities play an important role in the performance tuning of applications at extreme scale.

Unfortunately, similar tools are not available for I/O. Existing I/O tools typically fall into two categories. The first category relies on tracing and logging each individual I/O operation, a task that becomes increasingly expensive at larger scale. These tools often lack postprocessing capabilities necessary to identify salient characteristics. The second category relies on profiling and sampling in order to reduce overhead, but in doing so these tools sacrifice detail about the access patterns being generated. Just as successful tools for memory and communication characterization have been tailored to high-perfromance computing (HPC) demands and patterns, tools for extreme scale I/O characterization must be crafted to capture an appropriate level of detail with minimal impact on behavior.

Additionally, there exists an overall lack of understanding of how today's computational science applications interact with the storage system. This lack of understanding has created confusion in the storage research community as to how to best focus effort. If analysis tools for application I/O incurred little overhead, these tools could be enabled for all application runs. Doing so would allow us to identify trends in applications, help us understand successful I/O strategies, and inform the storage research community as to the needs of computational science.

In order to fit these two roles, a parallel I/O workload characterization tool must meet the following goals:

- Reflection of application-level behavior
- Transparency to users
- Leadership-class scalability

Reflection of application-level behavior. A 24/7 characterization tool should capture application-level behavior in order to distinguish between jobs and identify how each one interacts with the storage system. This is a straightforward goal with subtle implications.

Most HPC job workloads include a mixture of MPI-IO and traditional POSIX interface usage. Both should be captured in order to accurately represent all applications. File-system-level instrumentation is insufficient because MPI-IO or I/O forwarding may have already transformed the access pattern expressed by the application before it reaches the file system. Tying characterization to a specific file system or storage device limits portability as well.

Transparency to users. A characterization tool must be transparent to end users in order to be suitable for long-term deployment for all applications. Any burden on users or administrators acts as a barrier to participation, particularly if the goal is to characterize the entire system. Even more important, a characterization tool must minimize resource usage to the point that it has negligible impact on application performance. Performance is the foremost priority of a leadership computing platform and cannot be compromised by full-time characterization. Further, the characterization tool

itself should be robust enough that it does not add additional points of failure to the system.

Leadership-class scalability. Today's largest HPC deployments consists of hundreds of thousands of cores. A characterization tool must continue to work efficiently even at this scale in order to be practical for full-time use. Properties of scalable characterization tools include bounded data set sizes, elimination of redundant information, a "shared-nothing" architecture for data collection, and scalable intercommunication algorithms. The data set size can be bounded through the use of statistical metrics and space efficient data structures. Redundant data can be eliminated by general-purpose compression algorithms or data-specific reduction operators. In this context, a shared-nothing architecture means that each process operates independently such that more processes can be added without increasing contention. This can be implemented by avoiding the use of shared resources at run time. Scalable intercommunication can be achieved by leveraging collective operations in existing high-quality MPI implementations.

In this paper we present Darshan, a parallel I/O characterization tool that provides insight into application behavior and is suitable for 24/7 deployment on petascale systems. In Section II we discuss prior work in I/O characterization for computational science and existing tools for analysis of I/O in parallel applications. In Section III we describe the Blue Gene/P system on which we ran our experiments. In Section IV we describe the Darshan implementation, including techniques for minimizing measurement overhead, and show how it addresses all three of the goals set forth for a petascale I/O characterization tool. In Section V we show examples of the use of Darshan and give preliminary insight into the behavior of four relevant scientific application case studies. In Section VI we demonstrate the scalability of Darshan on an I/O-intensive 65,536-process job. In Section VII we conclude and point to areas of future development.

II. Related Work

Nieuwejaar et al. initiated the Charisma project in 1993 to study multiprocessor I/O workloads [5]. This culminated in an analysis of three weeks of data from two dissimilar HPC systems with up to 512 processes in order to identify common trends in parallel I/O. Their study identified several access pattern characteristics and established common terminology to describe them. The Charisma data-capture methodology consisted of each process recording trace information for every I/O operation, buffering it locally, and then sending it to a centralized trace server. The data was then postprocessed to identify workload characteristics.

Vetter and McCracken investigated MPI application scalability using statistical analysis with the mpiP tool [6]. Rather than capture a verbatim trace of MPI activity, it summarizes statistics at a per process level at run time and merges the statistics at the completion of the job. This technique scales well and has been tested on jobs with as many as 4,096 processes [7]. While mpiP does include MPI-IO functions in its analysis, it focuses primarily on MPI messaging and does not attempt to characterize file access patterns. It also does not capture POSIX API activity.

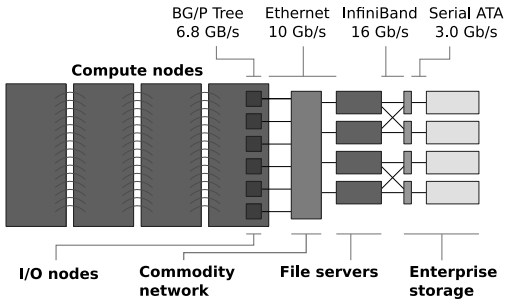


Fig. 1. IBM Blue Gene/P I/O system

Byna et al. have utilized tracing and characterization of I/O patterns as a means to improve MPI-IO prefetching [8]. Their method consists of running an application job once to generate a complete MPI-IO trace, postanalyzing the trace to create an *I/O signature*, and then using the signature to guide prefetching on subsequent jobs. The I/O signature is a compact, parameterized representation of the stride, repetition, timing, size, and other features. This signature technique holds promise for I/O workload studies as well if it can be adapted to operate at run time.

Noeth et al. have explored both intra- and internode runtime compression techniques for MPI communication traces in order to to reduce memory usage and log file size [9]. Their work does not explore the execution time overhead, nor does it capture file access patterns.

The HPC community has produced a wide variety of modern tools for generating traces of individual I/O operations in large-scale parallel applications, including HPCT-IO [10], LANL-Trace [11], and IOT [12]. Various tools are also available for general-purpose instrumentation, profiling, and tracing of general MPI and CPU activity, including Jumpshot [1], [2], FPMPI [13], TAU [3], and STAT [4]. However, these tools focus primarily on in-depth analysis of individual application runs rather than long-running workload characterization.

III. Background

The experiments in this paper were conducted on two IBM Blue Gene/P (BG/P) systems at the Argonne Leadership Computing Facility (ALCF) at Argonne National Laboratory. The first is a 4,096-core research and development system named Surveyor. Surveyor's storage subsystem consists of four file servers running PVFS and a DataDirect Networks S2A9550 SAN. The second BG/P system is a 163,840-core production system named Intrepid. Intrepid provides 80 TBytes of RAM and a peak performance of 556 teraflops. Its storage system consists of 128 file servers running both PVFS and GPFS and 16 DataDirect Networks S2A9900 SANs. The Intrepid parallel file systems have a total capacity of 5.2 PBytes and a peak I/O rate of approximately 78 GBytes/s.

Figure 1 illustrates the I/O architecture of the ALCF BG/P systems. I/O forwarding is used to minimize the number of compute processes visible to the file system. Each set of 64 quad-core compute nodes (CNs) forwards system calls to an

intermediate I/O node (ION) via a custom tree network. A daemon on the ION known as the CIOD (Control and I/O Daemon) is responsible for accepting system call requests and invoking them on behalf of the compute node kernel. This is the same approach as used in the previous Blue Gene/L systems [14]. Each ION is identical to a CN except that it runs a Linux kernel and possesses an additional network connection. A commodity-switched 10 Gb/s Myrinet network connects all IONs to all file servers, while a point-to-point InfiniBand network connects file servers to the SAN devices.

Unless otherwise noted, all experiments in this paper were carried out using the production Parallel Virtual File System volume on Intrepid. The PVFS project is a multi-institution collaborative effort to design and implement an open source, production parallel file system for HPC applications at extreme scale [15], [16].

The MPI implementation on the BG/P systems is based on MPICH [17], including MPI-IO support through ROMIO [18]. The MPI-IO abstract device implementation used on BG/P systems converts all I/O into POSIX operations for I/O forwarding purposes. It does not use the native PVFS library interface.

IV. Darshan

Darshan is a parallel I/O characterization tool designed to meet the goals set forth in Section I. It is implemented as a set of user space libraries. These libraries require no source code modification and can be added transparently during the link phase of MPI compiler scripts such as `mpicc` or `mpif90`. This approach is a compromise to the transparency goal in that existing binaries must be recompiled (or relinked) in order to use Darshan. However, it can be automatically applied to newly compiled applications or introduced as part of an MPI upgrade. In exchange for this compromise, Darshan can utilize portable, low-overhead mechanisms for intercepting I/O routines.

Darshan captures MPI-IO routines using the profiling (PMPI) interface to MPI. POSIX routines are captured by inserting wrapper functions via the GNU linker's `--wrap` argument. These mechanisms have been tested with the MPICH MPI implementation for both GNU and IBM C, C++, and Fortran compilers. It also works correctly for both static and dynamic compilation, requires no additional supporting infrastructure for instrumentation, and is compatible with other MPI implementations and compilers.

Rather than capture a complete trace of all I/O operations, Darshan characterizes the application by using statistics and cumulative timing information. The advantage of this approach is that the data can be stored compactly using a bounded amount of memory. The data is recorded independently on each process at run time and then merged and stored as the job is shutting down. Darshan invokes no communication or storage routines until the end of the job. It therefore reduces the scope of the scalability challenge to a single shutdown routine. The following subsections discuss the function wrapping overhead, statistical metrics, memory overhead, and storage techniques in greater detail.

TABLE I
BG/P function wrapping latency

Function	Time (ns)	Est. Overhead (%)
read (1 byte) /dev/zero	65937	0.52
read (1 byte) PVFS	814500	0.042

A. Function Wrapping Overhead

The PMPI interface and link-time wrappers are used to intercept I/O function calls, while the `MPI_Wtime()` function is used to collect timing information. `MPI_Wtime()` reports elapsed time in seconds as a floating-point value. While these methods are highly portable, we must take care to ensure that they are light weight enough to meet the Darshan goal of performance transparency. We found that each `MPI_Wtime()` call introduces 165 ns of latency, while the function wrapping accounts for 14 ns of latency. While this wrapping and timing method would be considered expensive for fine-grained computation, it is more than adequate for comparatively coarse grained I/O operations. Table I shows the projected percentage overhead of this approach for two different read operations. The read from `/dev/zero` represents the lowest-latency I/O operation possible on the Blue Gene, as it is forwarded only as far as the I/O node and does not interact with the storage system. The read from PVFS represents the latency of an I/O operation that must retrieve data from a parallel file system. The overhead in these cases is projected to be 0.52% and 0.042%, respectively. This relative cost would be even lower for I/O operations that transfer more than one byte of data at a time.

Darshan also provides a mechanism to disable the timing of function calls both at run time (via environment variables) and at compile time. Significant characterization is still possible even if it is deployed on a system where the cost of `MPI_Wtime()` is too high relative to I/O latencies.

B. Statistical Metrics

Darshan characterizes applications by recording compact statistical metrics. It creates a *file record* in memory for each opened file and tracks the rank of the process and the last 11 characters of the file name. Full paths are avoided for space reasons, but the last characters of the file name are often sufficient to classify types of files opened by the application. Darshan also records a 64-bit hash of the full file path name. This hash serves as a compact way to uniquely identify common files across processes. The hash algorithm is Jenkins' 64-bit lookup8 routine, which is optimized to use $41 + 5 \times n$ instructions that can be executed in parallel on superscalar CPUs [19]. We expect file names to often differ by only a single byte in HPC workloads (for example, if the process rank is used in the name). The choice of hash algorithm is therefore critical in order to minimize the probability of collision.

Each file record also contains a fixed-size array of counters, with each counter consisting of either a 64-bit integer or a double-precision floating-point variable. Each counter is set

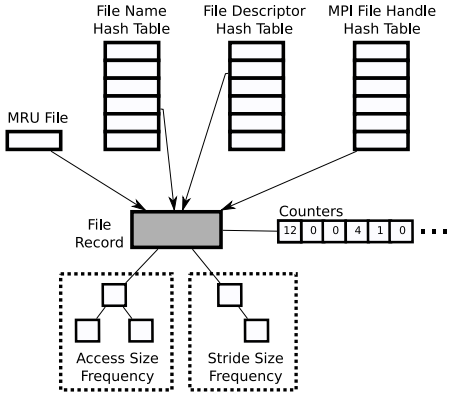


Fig. 2. Darshan file record

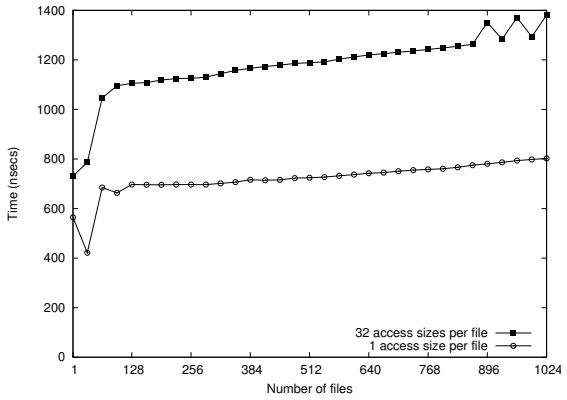


Fig. 3. Darshan operation characterization latency on BG/P

or incremented by an MPI-IO or POSIX function wrapper or, in some cases, deduced from a combination of a wrapper and the previous state of the counters. Following are examples of the metrics recorded:

- Counters for POSIX operations: read, write, open, seek, stat, mmap, fopen, and the like
- Counters for MPI-IO operations: collective, independent, split, or nonblocking read and write
- Counters for MPI-IO datatypes and hints
- Counters for unaligned, sequential, consecutive, and strided access
- Timestamps for open, close, first I/O, and last I/O
- Cumulative bytes read and written
- Cumulative time spent within POSIX and MPI-IO I/O operations
- Histograms of access, stride, datatype, and extent sizes

Darshan also tracks a small amount of job-level information, including the user id, start and end times, number of processes, and command line arguments. Where possible, we have used terminology that is consistent with that of Charisma [5]. In the Charisma terminology, sequential accesses begin at any offset higher than the end of the previous access, while consecutive accesses begin precisely at the end of the previous access. Strides are recurring patterns of gaps between accesses.

In a longer-running system workload study Darshan could also yield aggregate statistics, such as the average size of files, the users accounting for the most I/O activity, the adoption rate of MPI-IO, the number of files opened per job, the percentage of system time that is spent performing I/O, and the ratio of read to write activity. These statistics provide insight into HPC storage systems that are normally treated as a black box.

C. Memory Overhead

Darshan maintains a record in local process memory for each file accessed over the lifetime of the job (up to a limit that will be discussed shortly). Figure 2 illustrates how Darshan utilizes file records. A file record is indexed by three open hash tables, each with 2^8 indices, that correspond to file name, POSIX file descriptor, and MPI-IO file handle. The indexing ensures that the record uniquely identifies the file regardless of access mechanism. The file descriptor and MPI-IO file handle indexes are updated to refer to the same record if the file is closed and reopened later. The file name is already hashed for identification purposes as described in Section IV-B. The file descriptor value itself is bitmasked to serve as an index into the second hash table, since we expect it to be naturally well distributed. The file handle is an opaque type that we hash in the same manner as file names. In addition, each process maintains a most recently used (MRU) pointer. We have observed that a given file is often accessed repeatedly by an application before switching to another file, in which cases the MRU avoids the need to search hash tables at all.

Once the file record is identified, Darshan records metrics in an array of counters that are stored persistently when the job completes. In addition, Darshan maintains a frequency count of the most common access and stride sizes. These frequency counts are kept in independent red-black binary trees for each file record. The trees are limited to 32 elements each and are implemented by using the GNU C Library `tsearch()` routine. Once the job is complete, Darshan iterates through these trees and records the four most popular strides and access sizes in the persistent counters. Each file record consumes a total of between 1,168 and 2,064 bytes of memory at run time depending on the complexity of the access patterns for that file. When the job completes, all of this is discarded except for the 1,088 byte counter array.

Darshan imposes a tunable limit of 1,024 files that will be tracked per process. This limits the total file record memory consumption to approximately two megabytes per process in the worst case. Once the file count limit has been reached, Darshan condenses all file records into a single record for all files and continues gathering statistics. Thus, applications that open more than 1,024 files will still be characterized but at an aggregate level rather than differentiating per file. As an example, an application that writes a single checkpoint file every 30 minutes can run continuously for 21 days before Darshan falls back to aggregate characterization.

Figure 3 shows the time needed for Darshan to locate a file record and store statistics for I/O operations on a BG/P compute node. This experiment measures a worst-case scenario in which every operation is performed on a different file descriptor in round-robin fashion. This is an unlikely

TABLE II
DARSHAN BINARY OUTPUT FILE SIZE EXAMPLES, 32,768 PROCESSES

Files	Without Reduction	Uncompressed	Final Size
1	34.0 MBytes	2.2 KBytes	203 Bytes
262,144	272.0 MBytes	272.0 MBytes	13.3 MBytes

access pattern in practice, but it helps illustrate an upper bound on the search time because the MRU pointer is not a factor. The experiment was performed for two cases: one with the same access size in all cases and one with 32 random access sizes. The single-access size case measures the base cost of locating a file and characterizing the access. The 32-access size case adds the extra overhead of searching a fully populated binary tree. This search and characterization require 1.38 microseconds in the worst case, which is likely to be indistinguishable from normal variance in an 814.50 microsecond single-byte file system read as measured in Section IV-A.

When we combine these results with the function wrapping and timing overhead measured in Section IV-A, the total overhead to characterize a 1-byte PVFS read on Intrepid comes to between 0.09% and 0.2%, depending on the complexity of the access patterns.

D. Characterization Output Techniques

Each MPI process using Darshan tracks data independently until `MPI_Finalize()` time. At that point Darshan combines the data and stores it persistently in a single output file for the job. This approach is the key to achieving scalability because it is the only time that Darshan performs communication or storage activity. Although the approach outlined in Section IV-C maintains a small counter footprint in memory, petascale jobs would still generate an unwieldy amount of aggregate data if the counters were written naively to disk. Darshan takes three steps to mitigate this problem. First, it identifies shared files and reduces them to a single record. Second, it compresses all data in parallel at each process. Third, it uses MPI-IO collective writes to leverage machine-specific optimizations for concurrent I/O.

In order to identify shared files, the rank 0 process broadcasts its list of file name hashes to all other process for comparison. An `MPI_Allreduce()` is then used to construct a mask of shared records. `MPI_Allreduce()` is a collective operation that combines and distributes data from all processes. The shared records are then reduced to rank 0 by using a custom operator. All records are finally compressed in parallel with zlib and then written collectively to an output file. The resulting file is fully gzip compatible and can be processed by using serial or parallel utilities. Note that this algorithm currently identifies only globally shared files, but it could be extended to identify shared files on subsets of processes as well.

Table II shows examples of output file sizes generated by Darshan. The first row is from an application that opens a single shared file across 32,768 processes. Naively writing characterization data from each process would result in 34 MBytes of data, but reduction and compression bring the output size down to 203 bytes. The second row is from an application in which each process opened 8 unique files. In this case, reduction was not applicable, but compression was still able to limit the output file to 13.3 MBytes. The data produced by Darshan lends itself well to compression. Most applications exercise only a subset of POSIX and MPI-IO functionality, which leads to many counters with a default value of zero. In the examples of Table II, zlib was able to achieve 90.6% compression on a single record and 95.1% on 8 records per process. Note that zlib consumes a memory footprint of up to 256 KBytes per process for compression, but this can be tuned at run time.

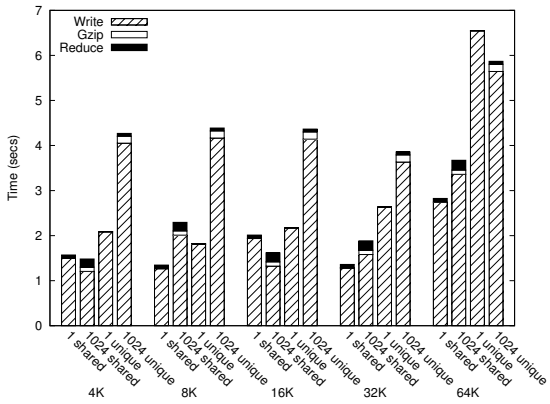


Fig. 4. Darshan output time for varying BG/P job sizes

Figure 4 shows the cost of the reduction, compression, and collective write phases of Darshan at job shutdown time on Intrepid. Each data point represents the best time out of three runs in order to eliminate the impact of transient shared file-system load. All output files were written to PVFS. Five job sizes were tested, ranging from 4,096 to 65,336 MPI processes and doubling the scale each time. All jobs were executed using four processes per compute node. Four scenarios were tested at each data point using a synthetic benchmark: a single shared file, 1,024 shared files, 1 unique file per process, and 1,024 unique files per process. Note that the 1,024 unique files per process case, at the largest size shown here, emulates an application that accessed *67 million* total files in a single run. This is unlikely behavior in practice but is used here as an example of scalability.

The cost of all scenarios in the experiment is dominated by the time needed to write output to the file system, which includes creating, writing, closing, and renaming the output file. The most expensive reduction steps occur when 1,024 shared files must be reduced. The largest example with 1,024 shared files across 65,536 MPI processes took 220 milliseconds to reduce. The most expensive gzip steps occur when 1,024 unique file records must be compressed at each process. The largest example with 1,024 unique files across 65,536 MPI processes took 161 milliseconds to compress in parallel.

In all cases, Darshan reduced, compressed, and wrote its output file in less than 7 seconds. In fact, there is little change in the cost of writing output for each job size until the 65,556 process example, when the cost increases slightly because of the overhead of accessing a single small file at that scale. This

TABLE III
MADBENCH2 I/O CHARACTERISTICS

Characteristic	Count	Percentage (%)
POSIX reads and writes	16384	
MPI-IO reads and writes	16384	
18.66 MByte access size	16384	100
1920 Byte stride	12288	75
Aligned in file	32	0.002
Named datatype	16384	100
Consecutive access	0	0
Sequential access	14344	88
Read/write alternation	7168	44

extra shutdown time is not likely to be noticeable, because jobs at this scale on Intrepid typically take several minutes to boot and shut down.

E. Limitations

Darshan is designed to identify salient I/O characteristics while introducing negligible overhead in production environments. However, there are limitations to the information that Darshan can provide. For example, it cannot provide the level of detail, cross-process correlation, or temporal information of a traditional tracing tool. It also cannot map access to specific lines of application code as in traditional instrumentation or sample-based profiling tools. This kind of information is best gathered through focused analysis with more invasive developer tools.

At this time, Darshan is also limited to MPI applications, even though it records both POSIX and MPI-IO activity. It is feasible to adapt it for other environments, however, minus the file record reduction capability.

V. CASE STUDIES

In this section we present four case studies of scientific application benchmarks that have been characterized by using Darshan, highlighting relevant characteristics in each case. Each of these benchmarks was executed at relatively small scale on the Surveyor BG/P system. Although this is far from a complete workload survey, it does give an example of the kind of information that Darshan can provide to an application developer or administrator.

A. MADBench2

MADBench2 is a benchmark derived from a cosmology application that analyzes Cosmic Microwave Background data sets [20]. It operates primarily on floating-point matrices that are too large to maintain simultaneously in main memory. MADBench2 therefore leverages an out-of-core algorithm in which matrices are written to disk when calculated and then read back later as needed. We executed MADBench2 on Surveyor with 1,024 processes in order to capture a Darshan characterization. MADBench2 was configured as an I/O-only benchmark, using MPI-IO in shared file mode, with four matrices of size $50,048^2$ each.

Table III summarizes a subset of interesting Darshan characteristics for MADBench2. The same file was opened exactly once by all processes. Each process performed 8 MPI write operations and 8 MPI read operations. These corresponded to the same number of underlying POSIX I/O operations. Although the file was opened collectively, all I/O was performed independently. The MPI-IO model appears to be a direct translation of POSIX `read()` and `write()` calls to `MPI_File_read()` and `MPI_File_write()` calls. The same access size (18.66 MBytes) was used for all reads and writes. These were not consecutive, however, and instead exhibited a stride of 1,920 bytes between accesses. Further investigation indicated that this was a result of the file block size argument to MADBench2. The MADBench2 file block size was set to 4,096 for this job, which caused MADBench2 to seek 1,920 bytes in order to align reads and writes on a block boundary. However, the PVFS file system was configured with a stripe unit (and corresponding reported block size) of 4 MBytes. Darshan correctly detected that only 32 of the accesses were aligned in file, rather than all 16,384 as one might expect. The out-of-core algorithm for MADBench2 resulted in each process alternating between read and write several times within the same file (7 per process). It also resulted in 12% of access being nonsequential, as processes seeked backwards to read previously written data.

Darshan also indicated that MADBench2 spent approximately 81% of its total run time within MPI-IO read or write calls. The file size grew to a maximum of 74.65 Tbytes, but almost twice that amount was both written to and read from the file over time. Overall, MADBench2 is a well-behaved application with respect to I/O. It reads and writes data using large buffers and makes an explicit effort to align I/O to block boundaries. Its most unusual characteristic is that it spends a significant portion of its time overwriting data.

MADBench2 offers little room for MPI-IO optimization or tuning, since all data is accessed contiguously by using named datatypes and independent access. This apparent limitation, however, presents an opportunity to measure the basic overhead of using MPI-IO rather than POSIX for rudimentary access patterns. Over the life of the job, MPI-IO added less than 0.02% overhead for reads and 0.01% overhead for writes.

B. Chombo I/O Benchmark

Chombo is a framework for adaptive mesh refinement scientific applications [21]. The Chombo I/O benchmark is derived from this framework [22]. It creates simulated Chombo data structures and writes them to a single file using the HDF5 high-level I/O library [23]. Chombo accesses a variety of small auxiliary files at run time, but for the purpose of this case study we will focus on characterization of the large data file only. Darshan simplifies this task by reporting seperate characteristics for each file opened by the application.

The Chombo I/O benchmark (Nov. 27, 2007, version) was executed on Surveyor with 512 processes and the `in.r444` example control file provided with the benchmark. This scenario writes an output file that is approximately 18.24 GBytes in size. HDF5 uses MPI-IO internally for all I/O operations, and Darshan indicates that the benchmark spent 75% of its aggregate run time in MPI-IO write function calls.

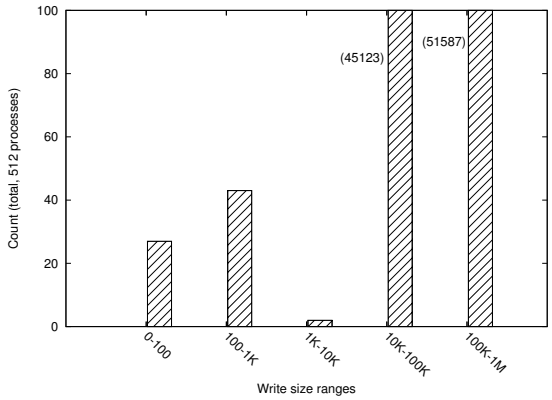


Fig. 5. Chombo write size histogram

TABLE IV
CHOMBO CHARACTERISTICS

(a) most common accesses

Size (bytes)	Count
28800	15622
16000	15109
480896	13818
254592	11428

(b) most common strides

Size (bytes)	Count
2885376	2725
2530944	1654
1752192	1567
86400	1481

(c) I/O patterns

Characteristic	Count	Percentage (%)
POSIX writes	96783	–
MPI-IO writes	96783	–
Writes preceded by seek	49123	50.76
Consecutive	47661	49.25
Sequential	96762	99.98
Unaligned in file	96780	99.99

Figure 5 illustrates a histogram of write sizes generated by the application. The overwhelming number of writes were between 10 KByte and 1 MByte in size, with none in the larger size ranges. Tables IV(a) and IV(b) show the specific values of the four most common access sizes and stride sizes observed by Darshan. Chombo generates a very large number of small writes by parallel file system standards, and it frequently skips through the file to different positions.

As in the MADBench2 example from Section V-A, Chombo issues exactly the same number of independent MPI-IO writes as POSIX writes. This one-to-one mapping indicates that each MPI-IO write is describing only one contiguous datatype. Table IV(c) highlights several other I/O pattern details. Approximately half the writes followed a stride, but nearly all were sequential. Although Chombo accesses many disjoint regions, it never seeks backwards on a given process. Nearly every access was unaligned in the file system.

This characterization of Chombo indicates that its I/O performance may be constrained primarily by the number of small, strided writes expressed using independent operations for each contiguous region. Without use of noncontiguous datatypes or collective I/O operations, the MPI-IO implementation is unable to aggregate access using data sieving or two-phase I/O. These characteristics indicate that Chombo will be a challenging benchmark for the file system, especially at large scale. The Chombo benchmark does include a compile time option to use collective operations, but we were unsuccessful in using it at the time of this writing.

TABLE V
BG/P FORWARDING BANDWIDTH BY MEMORY ALIGNMENT

Alignment	Read (MB/s)	Write (MB/s)
8 bytes	503.2	188.7
16 bytes	506.8	666.1

Darshan also recorded 70 writes that were not aligned on a 16-byte boundary in memory. This is only a small fraction of the total number of writes and thus does not have a significant impact on this code, but it may be a significant factor for other applications. Table V shows the result of an additional experiment to measure the impact of memory alignment for transfers between a BG/P compute node and I/O node. This experiment used 4 MByte buffers that were aligned 16 byte or 8 byte boundaries. Each buffer was written to `/dev/zero` to exercise the forwarding infrastructure without interference from an external file system. The tree network requires 16-byte alignment and must perform an extra copy for unaligned data. A performance degradation is therefore expected for unaligned access, but the 60% drop-off observed here is surprising. Note that the Darshan memory alignment threshold used by Darshan is a compile time choice that can be altered for different architectures.

C. S3D-IO

The S3D application [24] simulates turbulent combustion using direct numerical simulation of a compressible Navier-Stokes flow. The domain is decomposed among MPI processes in 3D. Periodically, all processes participate in writing out a restart file. This restart file can be used both as a means to resume computation and as input for visualization and analysis tools. S3D-IO extracts just the portion of S3D concerning restart dumps, allowing us to focus on strictly I/O characteristics. It supports multiple output methods, but we tested using Parallel-netCDF (pnetcdf) [25].

S3D-IO uses the pnetcdf collective interface, but through the use of MPI-IO hints we were able to evaluate both collective and independent I/O. The collective version spent 10% of its run time in MPI-IO write calls. Initial independent I/O performance was quite poor, spending 99% of runtime in I/O. Once we identified an issue that limited performance, we were able to reduce the runtime spent performing I/O in independent mode to 44%.

Table VI and Figure 6 show some characteristics of S3D at small scale in three different modes: collective I/O, independent without data sieving, and independent with data sieving. In truth, we only inadvertently benchmarked independent I/O without data sieving, but it is illustrative to discuss how and why that happened. Our trace of S3D-IO with collective I/O performed as expected: Darshan reported that the MPI-IO library optimized writes such that the restart data could be written out in a few multimegabyte calls. When we disabled

TABLE VI
S3D/PNETCDF CHARACTERISTICS, 16 PROCESSES

	Coll.	Indep.	Indep. Sieving
POSIX writes	5	102401	81
POSIX reads	0	0	80
MPI-IO writes	64	64	64
Unaligned in file	4	102399	80
Total written (MByte)	6.25	6.25	87.11
Run time (s)	6	1443	11
MPI-IO time/proc (s)	0.60	1426.47	4.82

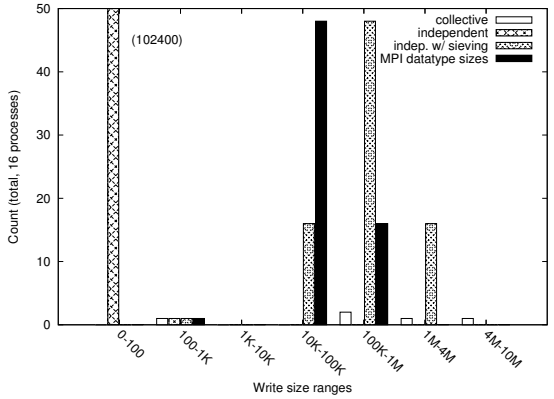


Fig. 6. S3D/PnetCDF write size histogram, 16 processes

collective I/O, however, independent I/O performed astonishingly poorly. Darshan showed the application issuing one million 8-byte writes. On Blue Gene, this workload is a recipe for disaster because of I/O forwarding latency and the lack of write caching at the compute node. The MPI-IO library has fewer opportunities for optimization in the independent I/O case, but Darshan clearly showed that the MPI-IO library was making *zero* optimizations. A quick check of the source code showed why: S3D-IO disabled a key independent I/O optimization.

S3D-IO was initially developed on a Cray XT-3. Because of a poor interaction between the MPI-IO library on that system and the Lustre file system, performance is actually better in some cases if certain optimizations are disabled. An MPI-IO hint can disable the data sieving optimization, which will result in many smaller writes but does not require acquiring an explicit user-level lock. While this approach gives good performance on the Cray system, it is perhaps the worst possible approach one could take on Blue Gene.

With data-sieving re-enabled, Darshan clearly demonstrated the benefits and the costs of that optimization. In data sieving, the MPI-IO library services a noncontiguous I/O request by operating on a single, large contiguous block. The library will end up transferring unnecessary data in this case. It makes up for this, however, by replacing many latency-bound small operations with a few large operations that are much more efficient. Thus, while the S3D restart file in this case was only 6 MBytes, the MPI-IO library wrote out 91 MBytes of data. Seeing this information makes it clear why S3D performs so much better in the collective I/O mode.

Collective I/O on Blue Gene is almost always a win; but if for some reason an application cannot use collective I/O routines, Darshan gives us some insight on how to tune some of the MPI-IO parameters. In Figure 6, the histogram suggests the independent case with data sieving might perform better if we increased the intermediate buffer to something larger like 6 MBytes.

TABLE VII
HOMME'S MOST COMMON ACCESS SIZES

(a) Collective

Size (bytes)	Count
6078464	1
16777216	128
8388608	59
4194304	4

(b) Independent

Size (bytes)	Count
30848	864000
–	–
–	–
–	–

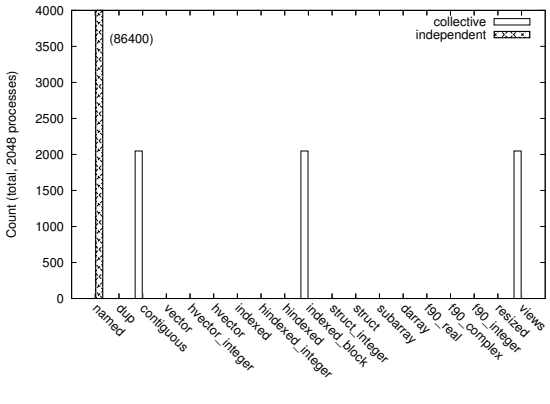
D. HOMME

The HOMME (High-order Multiscale Modeling Environment) [26] application models atmosphere physics using spectral element techniques. It periodically writes out restart files directly through MPI-IO. We can compile HOMME to write out these restart files in either independent or collective mode, providing an illustrative example of the differences from an application perspective. We ran a 2048-processor "aquaplanet" simulation with HOMME and traced the output of the restart file in both independent and collective mode. Both cases create several small files and a 2.5 GB restart file.

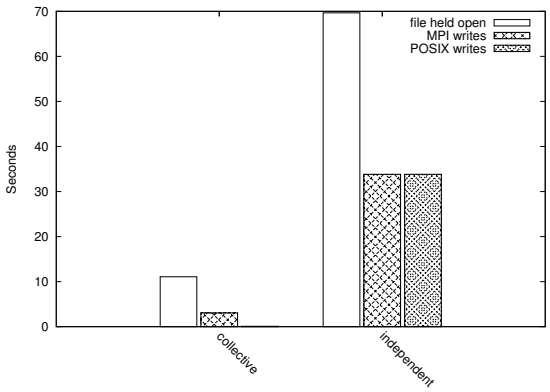
The collective versus independent comparison in HOMME is slightly different from that of S3D-IO. In S3D we had to compare independent versus collective by adjusting MPI-IO hints. In HOMME's collective case, it constructs an MPI file view using an indexed-block type to describe the file layout of a process's data and then writes all the records in a single collective call. In independent mode, HOMME processes just iterate over the records, writing them out one at a time. Those familiar with I/O on large systems will recognize the benefits of the collective I/O call, yet on several systems the HOMME developers found buggy MPI-IO implementations. The independent approach proved easier for these buggy implementations to handle.

Using Darshan, we can demonstrate and quantify just how much better suited the collective I/O approach is to the storage system on Blue Gene. In Table VII we see the most common access sizes for the two approaches. The 864,000 small (30k) writes are too tiny for the Blue Gene storage system to handle efficiently. Far better is the collective story, where there are far fewer operations and the vast majority are large (16 MBytes). This 16 MByte value is the default buffer size for a key MPI-IO collective optimization. The Darshan output suggests that we might be able to extract somewhat better performance by increasing this buffer to 24 MBytes or 32 MBytes.

The datatype histogram in Figure 7(a) shows how often each type showed up as inspected with `MPI_Type_get_envelope()` (both on I/O calls and set views). The final column shows the number of times set view was called. It confirms our understanding of the HOMME code: the collective I/O case constructs a type to



(a) datatype histogram



(b) average time per process operating on output file

Fig. 7. HOMME characteristics with 2048 processes

describe the I/O while the independent case uses only a simple predefined (built-in) MPI datatype.

Figure 7(b) compares the average time per process spent with the file open, in MPI write functions, and in POSIX write functions. Collective mode is 10 times faster actually writing, demonstrating again the importance of coalescing small accesses into larger requests on this system. Collective mode also spends more time organizing within the MPI write routine than actually accessing the file system. It is curious that independent I/O spends a lot of time with the file open but not actually writing anything. The collective I/O run took 120 seconds, while the independent run took 176 seconds. This graph accounts for the the entire 56-second difference in execution time between the two runs.

Darshan provides one additional bit of insight for which we do not have a graph, but which shows up in the trace output and has significant implications with regards to performance. Collective I/O, because of optimizations in the MPI-IO layer, wrote out data with perfect alignment in file. The independent case, on the other hand, wrote out unaligned blocks nearly every time. On PVFS, unaligned writes do not incur too large a penalty, but on GPFS, unaligned writes may incur more significant overhead as the lock management must acquire exclusive access to file system blocks to complete the operation.

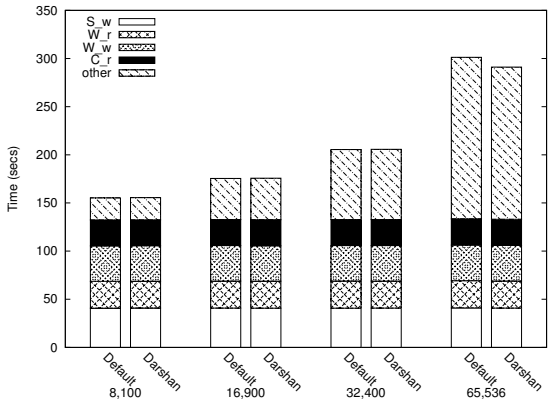


Fig. 8. MADBench2 /dev/zero execution time for varying BG/P job sizes

VI. Darshan scalability analysis

Thus far we have outlined the design of Darshan and demonstrated examples of characterization of four different scientific applications. In this section, we choose one of those applications, MADBench2, to measure the overhead of Darshan at larger scale. This type of measurement is difficult to perform on Intrepid because of performance variance from competing jobs and various caching effects. To mitigate the variance, we have modified MADBench2 to both read from and write to `/dev/zero` rather than a normal file on PVFS. Access to `/dev/zero` still utilizes the CIOD I/O forwarding system, but it is not impacted by any shared resources or storage devices. MADBench2 is a good candidate for this modification because it opens only one file and does not rely on the presence of valid data within the file at any point. As indicated in Section IV-A, performing I/O to special device files places an upper bound on the performance of I/O operations. This configuration is therefore more likely to illustrate overhead than access to a standard file system.

Figure 8 shows the execution time of each phase of MADBench2 at various scales on Intrepid. MADBench2 reports I/O time for four different phases. S_w and W_w are write phases, while W_r and C_r are read phases. In addition, we analyzed time stamps from scheduler logs to determine the complete execution time of each job. The difference between the I/O phases and the overall job time, including time needed to write the Darshan output to PVFS when Darshan is enabled, is shown as “other” in this graph. The parameters to MADBench2 were scaled at each job size in order to fully utilize the compute node memory and move approximately the same amount of data per compute node.

In Figure 8 we see that the overhead from Darshan is negligible for MADBench2. The I/O costs and total run time were nearly identical for a standard build and a Darshan-enabled build. The largest example, with 65,536 processes, issued 1,048,576 total I/O operations and exceeded 175 GBytes/s during writes to `/dev/zero`. The resulting Darshan output file for each job ranged from 273 to 283 Bytes and yielded similar characteristics to those shown in Section V-A. Based on these results we believe that Darshan will have negligible impact on the performance of most storage systems.

VII. Conclusions and Future Directions

Understanding and tuning I/O behavior on extreme-scale systems present an increasing challenge and yet are more important than ever in order to make efficient use of such systems. In this work we have presented Darshan as a tool to enable continuous characterization of I/O workloads with negligible impact on users or administrators. We evaluated the overhead of Darshan and then used it to characterize four relevant scientific application benchmarks. We also demonstrated that Darshan scales effectively to at least 65,536 processes without noticeable impact on application performance. Our experiments confirm that Darshan is a viable mechanism for 24/7 characterization of petascale I/O workloads.

The next step in our work will be to deploy Darshan for production use on the BG/P systems at Argonne National Laboratory. Our intent is to perform a long-running study of workload characteristics for leadership-class systems, as well as to help guide developers and administrators in tuning efforts. These tasks will require the development of additional analysis tools to extract information from a large collection of jobs. We also hope to encourage the use of Darshan as a common part of the application developer's tool set. We have taken steps toward this goal by developing a utility that graphically summarizes high-level application I/O characteristics. We intend to release Darshan publicly under an open source license for community use.

In terms of extending Darshan itself, we would like to add support for additional levels of the I/O software stack, such as high-level libraries and I/O forwarding software. We will also explore the possibility of adding more sophisticated runtime analysis of access patterns. Our current experiments indicate that we have room for additional computation without impacting application performance. If we discover a need to track a greater number of files, we will also explore techniques such as runtime compression to reduce Darshan's memory footprint.

Acknowledgments

We thank Ray Grout, Jacki Chen, and Wei-keng Liao for providing the S3D-IO benchmark, Julian Borrill for providing the MADBench2 benchmark, Mark Taylor for providing the HOMME benchmark, and the Applied Numerical Algorithms Group at NERSC for providing the Chombo I/O benchmark.

This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.

References

[1] O. Zaki, E. Lusk, W. Gropp, and D. Swider, "Toward scalable performance visualization with Jumpshot," *High Performance Computing Applications*, vol. 13, no. 2, pp. 277–288, Fall 1999.

[2] A. Chan, W. Gropp, and E. Lusk, "An efficient format for nearly constant-time access to arbitrary time intervals in large trace files," *Scientific Programming*, vol. 16, no. 2-3, pp. 155–165, 2008.

[3] S. Shende and A. D. Malony, "The TAU parallel performance system," *International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–331, Summer 2006.

[4] D. Arnold, D. Ahn, B. de Supinski, G. Lee, B. Miller, and M. Schulz, "Stack trace analysis for large scale debugging," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium, 2007*, March 2007, pp. 1–10.

[5] N. Nieuwejaar, D. Kotz, A. Purakayastha, C. S. Ellis, and M. Best, "File-access characteristics of parallel scientific workloads," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, no. 10, pp. 1075–1089, October 1996. [Online]. Available: http://www.computer.org/tpds/td1996/l1075abs.htm.

[6] J. S. Vetter and M. O. McCracken, "Statistical scalability analysis of communication operations in distributed applications," *SIGPLAN Notices*, vol. 36, no. 7, pp. 123–132, 2001.

[7] "mpiP: Lightweight, scalable MPI profiling," http://mpip.sourceforge.net/.

[8] S. Byna, Y. Chen, X.-H. Sun, R. Thakur, and W. Gropp, "Parallel I/O prefetching using MPI file caching and I/O signatures," in *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–12.

[9] M. Noeth, J. Marathe, F. Mueller, M. Schulz, and B. de Supinski, "Scalable compression and replay of communication traces in massively parallel environments," in *SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. New York, NY, USA: ACM, 2006.

[10] S. Seelam, I.-H. Chung, D.-Y. Hong, H.-F. Wen, and H. Yu, "Early experiences in application level I/O tracing on Blue Gene systems," in *Proceedings of the 2008 IEEE International Parallel and Distributed Processing Symposium*, 2008.

[11] "HPC-5 open source software projects: LANL-Trace," http://institute.lanl.gov/data/software/#lanl-trace.

[12] P. C. Roth, "Characterizing the I/O behavior of scientific applications on the Cray XT," in *PDSW '07: Proceedings of the 2nd International Workshop on Petascale Data Storage*. New York, NY, USA: ACM, 2007, pp. 50–55.

[13] "FPMPI-2 fast profiling library for MPI," http://www.mcs.anl.gov/research/projects/fpmpi/WWW/.

[14] J. J. Ritsko, I. Ames, S. I. Raider, and J. H. Robinson, "Blue Gene," *IBM Journal of Research and Development*, vol. 49, March/May 2005.

[15] "The Parallel Virtual File System," http://www.pvfs.org/.

[16] R. Latham, N. Miller, R. B. Ross, and P. H. Carns, "A next-generation parallel file system for Linux clusters," *LinuxWorld Magazine*, January 2004. [Online]. Available: http://www.pvfs.org/documentation/papers/linuxworld-JAN2004-PVFS2.pdf.

[17] E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the MPI message passing interface standard," *Parallel Computing*, vol. 22, pp. 789–828, 1996.

[18] R. Thakur and E. Lusk, "On implementing MPI-IO portably and with high performance," in *Proceedings of the 6th Workshop on I/O in Parallel and Distributed Systems*. ACM Press, 1999, pp. 23–32.

[19] B. Jenkins, "Lookup8 hash algorithm," http://burtleburtle.net/bob/c/lookup8.c.

[20] J. Carter, J. Borrill, and L. Oliker, "Performance characteristics of a cosmology package on leading HPC architectures," in *HiPC:International Conference on High Performance Computing*. Springer, 2004, pp. 176–188.

[21] "Chombo - infrastructure for adaptive mesh refinement," https://seesar.lbl.gov/ANAG/chombo/.

[22] "Chombo I/O benchmark," http://www.nersc.gov/~ndk/ChomboBenchmarks/chomboIOBenchmark.html.

[23] "HDF5 home page," http://www.hdfgroup.org/HDF5/.

[24] 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*, vol. 46, pp. 38–42, 2006. [Online]. Available: http://stacks.iop.org/1742-6596/46/38

[25] J. Li, W. keng Liao, A. Choudhary, R. Ross, R. Thakur, R. Latham, A. Siegel, B. Gallagher, and M. Zingale, "Parallel netCDF: A high-performance scientific I/O interface," in *Proceedings of Supercomputing*, 2003.

[26] R. D. Nair and H. M. Tufo, "Petascale atmospheric general circulation models," *Journal of Physics: Conference Series*, vol. 78, p. 012078 (5pp), 2007. [Online]. Available: http://stacks.iop.org/1742-6596/78/012078.