SANDIA REPORT

SAND2009-7763
Unlimited Release
Printed January 2010

Advanced I/O for Large-Scale Scientific Applications

Final Report for LDRD 120479

Gerald F. Lofstead, Karsten Schwan, Scott Klasky, Ron A. Oldfield

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94-AL85000.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2009-7763
Unlimited Release
Printed January 2010

Advanced I/O for Large-Scale Scientific Applications

Final Report for LDRD 120479

Gerald F. Lofstead II
Ph.D. Candidate
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280
lofstead@cc.gatech.edu

Karsten Schwan
College of Computing
George Institute of Technology
Atlanta, GA 30332-0280
schwan@cc.gatech.edu

Scott Klasky
Oak Ridge National Laboratory
P.O. Box 2008 MS6008
Oak Ridge, TN 37831-6008
klasky@ornl.gov

Ron A. Oldfield
Scalable System Software (1423)
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-1319
raoldfi@sandia.gov

Abstract

As scientific simulations scale to use petascale machines and beyond, the data volumes generated pose a dual problem. First, with increasing machine sizes, the careful tuning of IO routines becomes more and more important to keep the time spent in IO acceptable. It is not uncommon, for instance, to have 20% of an application's runtime spent performing IO in a 'tuned' system. Careful management of the IO routines can move that to 5% or even less in some cases. Second, the data volumes are so large, on the order of 10s to 100s of TB, that trying to discover the scientifically valid contributions requires assistance at runtime to both organize and annotate the data. Waiting for offline processing is not feasible due both to the impact on the IO system and the time required. To reduce this load and improve the ability of scientists to use the large amounts of data being produced, new techniques for data management are required. First, there is a need for techniques for efficient movement of data from the

compute space to storage. These techniques should understand the underlying system infrastructure and adapt to changing system conditions. Technologies include aggregation networks, data staging nodes for a closer parity to the IO subsystem, and autonomic IO routines that can detect system bottlenecks and choose different approaches, such as splitting the output into multiple targets, staggering output processes. Such methods must be end-to-end, meaning that even with properly managed asynchronous techniques, it is still essential to properly manage the later synchronous interaction with the storage system to maintain acceptable performance. Second, for the data being generated, annotations and other metadata must be incorporated to help the scientist understand output data for the simulation run as a whole, to select data and data features without concern for what files or other storage technologies were employed. All of these features should be attained while maintaining a simple deployment for the science code and eliminating the need for allocation of additional computational resources.

Acknowledgment

The authors would like to thank the following people for their help and insights that have affected the development and outcomes of this work. First, Matthew Wolf at Georgia Tech has consistently provided an alternative scientist perspective that has been key in developing approaches so that they apply more broadly than the few production codes tested. Steven Hodson provided some initial insights into the idiosyncracies of parallel file systems, in particular Lustre, that has help develop a thought pattern to better understand how to manage the overall IO process rather than blindling assuming the file system will just 'do the right thing'. Rickey Kendall's sponsorship of the compute time on the various Oak Ridge resources has been essential for the development and testing of the techniques at production scale. Extensive testing and measurement assistance has been provided by Fang Zheng. Asynchronous approaches and the different concerns and ideas that generated changes to broadly support both synchronous and asynchronous techniques were contributed by Hasan Abbasi.

Contents

Figures

Tables

Introduction

As compute partitions on supercomputers continue to scale, IO capacity has not, as evidenced, for instance, by the growth of the ratio of compute to IO nodes on petascale machines. At the same time, large scale science codes generate substantial data volumes. An example is the GTC plasma fusion code running on 29,000 cores and generating 50+ TB of data over a 24 hour runtime. Scaling a GTC run to 120,000 cores demands that 200+ TB of data will be moved in similar timeframes. To keep large scale science productive, therefore, highly optimized IO approaches are critical. Ensuring these techniques use as few additional compute resources as possible is also critical. While the number of cores per node increases, memory sizes are not always keeping pace. For example, in the latest upgrade for the XT5 Jaguar machine at ORNL, it moved from quad-core to six-core processors without a memory upgrade. This trend along with machines like the BlueGene/P with only 512 MB per core indicate working in a memory constrained environment is important.

The feasibility of optimizing IO for large-scale codes is again demonstrated by the GTC code, which spends only 4% of its runtime performing IO when running at these scales. This was attained by carefully managing the IO process at a low level, using the ADIOS IO componentization system, which provides a simple API while offering interchangable, highly optimized IO routines that can be changed for each data grouping in the code by simply editing an entry in an external XML configuration file. This enables the custom approaches necessary for achieving peak performance for different data sizes, system architecture, and underlying parallel file system configurations. Specifically, to achieve high performance IO, four techniques are essential. First, each parallel file system has a particular mode of operation that works best. Lustre, with its single metadata server, for instance, performs well when file open calls are serialized external to the metadata server. Second, depending on the metadata server interactions and other storage related idiosynchracies, it may be advantageous to split output into multiple files rather than a single, large file. This can lead to orders of magnitude improvements in performance. Third, proper selection and management of storage targets to avoid concurrent use by different files and processes can have a significant impact. Fourth, as dynamic system conditions change, such as the load level on particular storage targets, the storage mechanism must adapt to maintain performance. A single overloaded storage target could extend overall IO duration by as much as a factor of 100. To attain high IO performance, then, we must consider all of these issues or stated more generally, we must make it easy to dynamically vary how IO is performed to accommodate these factors at a consistently high level.

To complicate matters further, a daunting task for end users is how to select 'interesting' data from large output sets. Moving the data across even a high-speed network can take days. To minimize data movement, automatically generated annotations called *data characteristics* are essential for productive data use. For example, something as simple as storing the array minimum and maximum values can eliminate the need to scan entire data files just to find where a value reaches a threshold. For example, in the Chimera supernova simulation, a scientist may be interested only in data items where a temperature value exceeds 10^6 degrees for the first time.

Combining the management of IO to achieve excellent performance with the productivity needs of scientists yields an interesting and challenging problem. Techniques employed for best IO per-

formance, for instance, may not result in highest scientific productivity. For example, generating 600 files per output may be the best IO performance, but forcing the scientist to sort through these 600 files for each output looking for the particular data characteristic of interest is only marginally better than scanning complete data sets. Therefore, an essential part of IO is the organization and annotation of the data to better manage complexities in the downstream use of the output data. For example, by maintaining a metadata layer collecting knowledge about the entire simulation run, it becomes possible for a scientist to use such metadata to identify which data is interesting and then proceed directly to the proper output file(s) containing the relevant data. Another example would be to output multiple different organizations of the same data based on different downstream usage needs. For such tasks, the scientist should not be burdened with selecting the proper file(s) and searching manually through the generated data.

To address these issues, this LDRD, in cooperation with researchers at Oak Ridge National Laboratory and Sandia National Laboratory, supported research and development of a number of new capabilities for scientific applications. First, we designed an adaptable IO layer, called ADIOS. The ADIOS [20] API provides an IO abstraction nearly as simple as standard POSIX IO while affording selection of which IO method to employ at runtime via a change to an external XML configuration file. Intial performance evaluations [22] demonstrate the variability of the performance for different IO methods at different scales and different data volumes. Second, some intial work adapting the IO configuration to the file system [19] have demonstrated the ability to achieve as much as 75% of peak IO performance by carefully managing interactions with the storage system. The challenge this work introduces is an intermediate count of files based on the size of the storage system rather than solely the process count or logical structure of the science code. Third, the BP (Binary Packed) file format [21] enhances IO performance by offering data annotation and automaic characterization with a minimal amount of coordination among the collective processes performing IO at the same time.

This report is a compilation of the research and development activities supported by this LDRD and various researchers at Georgia Institute of Technology and Oak Ridge National Laboratory. Some of this text is copied verbatim from previous publications cited throughout the document.

Motivation

Existing petascale applications and recent related work have all contributed to the need for this work.

Applications

Consider the following two applications *GTS* and *XGC*. GTS is a particle in cell plasma fusion physics code designed to scale from 64 to 100k cores using first principle techniques. XGC is another plasma fusion physics code. These applications are selected both for their scalability and the extreme output data volumes generated. They both also have rich offline workflow systems attempting to use the generated data.

GTS

As a particle in cell code, GTS provides an opportunity to study the IO patterns and generalize to other leadership class applications such as *XGC* and *S3D*. One particular challenge with GTS on even the largest systems that is a generally more common problem on low memory systems like the BlueGene/P, is the inability to allocate any memory for asynchronous data transfers. These conditions require high performance, synchronous IO routines. GTS currently generates on the order of 200 MB of data per process leaving no room for a buffer to stage asynchronous data movement.

XGC

XGC has a different set of challenges. The data generated per process is smaller that GTS, but the number of variables is large. With around 100 variables that need to be output, managing the metadata for efficient access to those variables over the various timesteps they are output is nearly 2 orders of magnitude larger data.

Solution Approach

To provide a high performance IO system with annotation and data organization to aid scientist productivity that can meet the requirements of the above two example applications, as well as providing both the scalability and flexibility required for future application deployments, we present the concept of **Extreme Scale Data Management**.

Unlike traditional IO approaches, extreme scale data management will manage both the movement of data and aid access through metadata creation and management. This can be conceptually

partitioned into the following:

1. Generate Data
2. Annotate Data
3. Managed Data Movement
4. Managed Data Access

Generate Data

Scientific simulations generate massive quantities of data. While all of the data is not necessary for offline analysis, more frequent output can yield unexpected scientific discoveries due to the smaller set of changes between each output. By carefully managing the output based on the underlying system characteristics, more consistent, higher performance IO will enable scientists to better gauge how much data can be written how often.

Annotate Data

By incorporating meaningful annotations for the massive quantities of data generated, scientist productivity can be assisted greatly. Rather than scanning all of the data to find interesting portions, it is possible to evaluate the annotations to determine which portions deserve closer inspection. We have demonstrated that simple data annotation can be performed 'for free' in the compute partition with the cost of the operation being lost in the variability of the IO operations themselves.

Managed Data Movement

We use adaptable, autonomic techniques to move the data from the compute partition to storage that can detect system problems and work around them ensuring continued high performance IO. If sufficient time is available for IO, additional data organizations can be output to aid scientist productivity. For example, proper decomposition into stripes based on the number of storage targets and properly distributing the data based on how it is used can greatly increase the parallel access performance for data.

Managed Data Access

With the output data set from the simulation run, we provide simple APIs and user interfaces for seeing an overview of the simulation run and aiding in the selection of interesting portions of the data. These APIs must be able to select data using abstract specifiers like a timestep and locations

within a global array while the internal implementation is capable of navigating the metadata to efficiently locate and retrieve the selected data no matter how the storage was originally performed.

The Adaptable IO System (ADIOS)

The Adaptable IO System (ADIOS) [20] provides an API nearly as simple as standard Fortran POSIX IO. An external XML configuration file describes the data and how to process it using provided, highly tuned IO routines. More importantly, output can simultaneously use multiple IO implementations, using the concept of 'data grouping' embedded into ADIOS. The idea is to facilitate changing IO methods based on the IO patterns of different IO operations and to make it possible to create "dummy" methods that can be trigger events for other systems like workflows. Once the code has been changed to use ADIOS for IO, any of the various IO routines can be selected just by changing the XML file. No source code changes are ever required.

The basic design of the ADIOS API was inspired by GTC fusion code [18] and the Chimera supernova code CHIMERA, both developed at Oak Ridge National Laboratory. Over the life of the GTC fusion code, the IO approach has changed eight times, each motivated by a change of platform or a need for more data annotation. Specialized routines have had to be added for each in situ visualization system employed. Each time one of these changes occurred, the base code had to be reevaluated to ensure that it was both operating properly and generating the proper data in the output. These evaluations cost days to weeks of time for the developers and thousands of hours of compute time with no science output. Through a system like ADIOS, the user can quickly test the various IO method available and select one that gives the best combination of performance and required features. The GTC code required the ADIOS API to support a variety of different outputs with varying frequency and sizes while Chimera required support for writing a large number of variables with variable reporting/formatting requirements. We have further demonstrated the generality of ADIOS by integrating successfully with XGC0, XGC1, FLASH, GTS, and S3D with at most, only minor tweaks of the system required.

Based on GTC and Chimera, we extracted these five main requirements.

- *Multiple, independently controlled IO settings* - Each gross IO operation needs to be independently configurable from others within the same code. For example, the output strategy for diagnostic messages should be different from restarts.
- *Data items must be optional* - Variables that are members of a data grouping need to potentially be strictly optional to account for different output behavior for different processes within a group. For example, if the main process in a group writes header information and the other participating processes do not, the system should be able to handle it properly.
- *Array sizes are dynamic* - The sizes of arrays need to be specified at runtime, particularly at the moment the IO is performed. The key insight here is not just that the values need to be provided at runtime, but we need a way to do this that is both consistent with the standard IO API as well as not impacting the actual data written.
- *Reused buffers must be handled properly* - It is important to support constructed output values in shared buffers. This means that we need to accommodate both copying stack temporary values when they are given to us as well as being able to handle the source code reusing a buffer to construct a second output artifact in an effort to save memory.

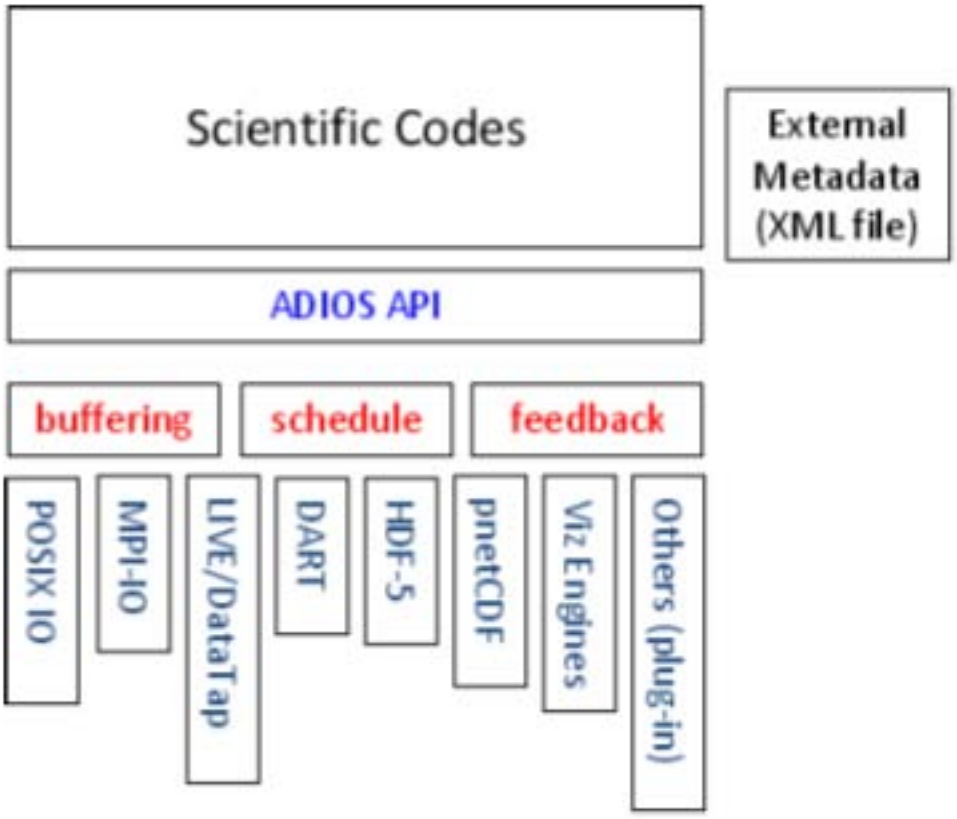

Figure 1. ADIOS Architecture.

- *Buffer space for IO is strictly limited* - The scientific codes have strict limits on how much memory they are willing to allow IO to use for buffering. For example, it might be stated that IO can use 100 MB or just 90% of free memory at a particular point in the code once all of the arrays have been allocated. Respecting this memory statement like a contract is critical to acceptance by the community.

The ADIOS API addresses these five requirements while providing an API nearly as simple as POSIX IO, fast IO, and transparent low-impact integration of auxiliary tools like workflow and in situ visualization. At a high level, ADIOS structurally looks like Figure 1.

The four parts each provide key benefits.

1. *ADIOS API* - The core ADIOS API calls that are used in the scientific codes.

2. *Common services* - Internal services for encoding/decoding, buffering and other shared operations are available for use by any of the transport methods.

3. *Transport methods* - Perform the data operations and auxillary integrations. For example, MPI-IO, POSIX IO, and Kepler or Visit integration.

4. *External metadata XML file* - Controls how these layers interact

The key success of ADIOS is the simple API and supporting XML configuration file. This separation affords us the opportunity to keep the API in the source code simple and consistent while placing annotation and configuration information in the associated XML file. To deliver on the fast IO, we provide transport methods implemented and tested by experts so that high performance IO is easily achieved. Finally, to achieve the transparent selection of IO methods, we have a standard transport method interface for cleanly integrating new IO methods and manipulations as part of the library. Through these features, we achieve our three goals: 1) An API almost as simple as POSIX IO, 2) Fast IO, and 3) Changes in data use require no source code changes. Highlights of the supporting XML configuration file, simple API, and the transparent IO selection will be addressed below in more detail.

XML Format

Since the XML controls how everything else works, we will discuss it first. The XML document provides a key break between the simulation source code and the IO mechanisms and downstream processing being employed. By defining the data types externally, we have an additional documentation source as well as a way to easily validate the write calls compared with the read calls without having to decipher the data reorganization or selection code that may be interspersed with the write calls.

One nice feature of the XML name attributes is that they are just strings. The only restrictions for their content are that if the item is to be used in a dataset dimension, it must not contain a comma and must contain at least one non-numeric character. This is useful for putting expressions as various array dimensions elements.

The main elements of the XML file format are of the format `<element-name attr1 attr2 ...>`. For example, the description below is structured like the XML document in Figure 2:

Elements:

- *adios-group* - a container for a group of variables that should be treated as a single IO operation (such as a restart or diagnostics data set).

- *global-bounds* - [optional] specifies the global space and offsets within that space for the enclosed var elements. Also specifies $M \times N$-style operations through the coordination-communicator and/or coordination-var (more details below).

- *var* - a variable that is either an array or a primitive data type, depending on the attributes provided.

```
<adios-config>
<adios-group name>
<global-bounds dimensions offset
     coordination-communicator
     coordination-var>
  <var name path type
    dimensions write copy-on-write/>
</global-bounds>
<mesh type time-varying>
...
</mesh>
<attribute name path value/>
</adios-group>
<transport group method base-path
    priority iterations>
    parameters</transport>
<buffer size-MB free-memory-percentage
     allocate-time/>
</adios-config>
```

Figure 2. ADIOS configuration document.

- *mesh* - [optional] a mesh description for the data compatible with the VTK data requirements.
- *attribute* - attributes attached to a var or var path.
- *transport* - mapping a writing method to a data type including any initialization parameters.
- *buffer* - internal buffer sizing and creation time. Used only once.

Attributes:

- *path* - HDF-5-style path for the element or path to the HDF-5 group or data item to which this attribute is attached.
- *dimensions* - a comma separated list of numbers and/or names that correspond to integer var elements to determine the size of this item
- *write* - [optional] if it is set to "no", then this is an informational element not to be written intended for either grouping or dataset dimension usage
- *method* - a string indicating a transport method to use with the associated adios-group.

- *group* - corresponds to an adios-group specified earlier in the file.

$M \times N$ communication is implicit in the XML file through the use of the global-bounds. If the global-bounds element is specified, then we have the ability to coordinate either on the compute nodes using the coordination-communicator or downstream using the coordination-var. Which communication mechanism (e.g., MPI or OpenMP) is used to coordinate is left up to the transport method implementer and potentially selected by the parameters provided in the 'transport' element in the XML file. For example, if the MPI synchronous IO method is employed for a particular IO group, it uses MPI to coordinate a group write or even an MPI collective write. Alternatively, a different transport method could use OpenMP. We define that the communicator 'passed in' must make sense to the transport method selected and that the ordering of processes is assumed to be in rank order for that communicator. Similarly, if the coordination-var is provided as well, an asynchronous IO method may choose to send the data downstream annotated with these attributes so that an another process can reassemble the data according to these parameters.

Changing IO Without Changing Source

The transport element provides the hook between the adios-group and the transport methods. Simply by changing the method attribute of this element, a different transport method will be employed. If more than one transport element is provided for a given group, they will be invoked in the order specified. This neatly gives triggering opportunities for workflows. To trigger a workflow once the analysis data set has been written to disk, make two element entries for the analysis adios-group. The first indicates how to write to disk and the second will perform the trigger for the workflow system. No recompilation, relinking, or any other code changes are required for any of these changes to the XML file.

ADIOS API

Since scientific codes are written in both Fortran and C-style languages, we developed and tested ADIOS from the beginning to have both a Fortran and a C interface. The calls look nearly identical between the two APIs and only differ in the use of pointers in C. The details of these calls will be discussed in more details in Appendix **??**. The API itself has two groups of operations. First are the setup/cleanup/main loop calls and second are those for performing actual IO operations.

Setup/Cleanup/Main Loop

This portion of the API focuses on calls used in generally a single location within the code. These are also calls with global considerations.

```
adios_init ("config.xml")
```

```
...
// do main loop
adios_begin_calculation ()
// do non-communication work
adios_end_calculation ()
...
// perform restart write
...
// do communication work
adios_end_iteration ()
! end loop
...
adios_finalize (myproc_id)
```

Adios_init and adios_finalize perform the expected sorts of one-time initialization and cleanup operations. The myproc_id parameter to adios_finalize affords the opportunity to customize what should happen when shutting down each transport method based on which process is ending. For example, if an external server needs to be shutdown, only process 0 should send the kill command.

Adios_begin_calculation and adios_end_calculation provide an optional mechanism by which the scientific code can indicate when asynchronous methods should focus their communication efforts since the network should be nearly silent. Outside of these times, the code is deemed to be likely communicating heavily. Any attempt to write during those times will likely negatively impacting both the asynchronous IO performance and the interprocess messaging. Adios_end_iteration provides a pacing indicator. Based on the entry in the XML file, this will tell the transport method how much 'time' has elapsed so far in this transfer.

IO Operation

Each IO operation is based around some data collection, referred to as a data 'group', opening a storage name using that group, writing or reading, and then calling close. Since our system focuses on supporting both asynchronous and synchronous operations, the semantics for these calls assume stricter requirements. For example, a supplied buffer is expected to be valid until after the associated adios_close call returns.

```
adios_open (&group_handle, "restart.01",
           ,``w'', MPI_COMM_WORLD)
adios_group_size (&group_handle, size, &total_size)
...
adios_write (group_handle, "zion", zion)
...
adios_write (group_handle, "mzeta", mzeta)
...
```

```
adios_close (group_handle)
```

Adios_group_size provides a mechanism by which the size of a process's output is specified so that a proactive spacing of process output in the file can take place. Ultimately, this should be optional defaulting to buffering locally and then deciding on offsets based on exactly what is written by the processes. Adios_open, adios_write, and adios_close all work as expected. The third parameter is a mode (r/w/a) with the last being an optional communicator the transport method(s) should be able to understand. The string second parameter to adios_write specifies which var in the XML the provided data represents. One special note is that adios_close is considered a 'commit' operation. Once it returns, all provided buffers are considered reusable, or in the case of reading, populated.

Common Services

In an effort to make writing a transport method as simple as possible, we have created a few shared services. As the first two services we have full encoding and decoding support for our binary format and rudimentary buffer management. One of the future research goals of ADIOS is to extend support for more common services including feedback mechanisms that can change the what, how, and how often IO is performed in a running code. For example, if an analysis routine discovers some important features in one area of the data, it could indicate to write only that portion of the data and to write it more often.

Transport Methods

We have partnered with experts on each IO method we have at this time. For example, we have a synchronous MPI-IO method based on work done by and verified by Steve Hodson at ORNL, a collective MPI-IO method developed by Wei-keng Liao at Northwestern, a POSIX IO method developed by Jay Lofstead with recommendations for performance enhancements by Wei-keng Liao, DataTap [10] asynchronous IO by Hasan Abbasi at Georgia Tech, and a NULL method for no output, which is useful for benchmarking the performance of the code without any or selectively less IO. A basic parallel HDF-5 and serial NetCDF methods are currently available. For visualization transport methods, we have an initial pass at a VTK interface into Visit and a custom sockets connection into an OpenGL based renderer. We have under development an asynchronous MPI-IO method by Steve Hodson and have planned a parallel netCDF method based on existing tools we have written to convert our default data encoding into this format.

The BP File Format

With current and next generation high performance machines, the probability is high that one of many nodes performing data output fails to complete that action. While the loss of that node's data may be acceptable to the scientific application, the failure of all nodes to complete their output due to a single node's problems or worse, the corruption of past output when writing to the same file, is not. Instead, data output should be implemented to be robust to failures, adopting principles from other areas of Computer Science (e.g., consider the Google file system [9]). In response to these needs, we have implemented the BP file format[1] as part of ADIOS, the adaptable IO system [20]. Analogous to proxy processes in other systems, the idea of the BP 'proxy' data format is to act as an *intermediate data format* used during data output and initial processing (e.g., the aforementioned lightweight data characterization). We call BP an intermediate format because typically, converters will be applied to BP files so that subsequent data analysis and storage can leverage the extensive tool chains existing for popular formats like HDF-5 and NetCDF.

The BP file format implementation is designed to maximize parallel output performance while also containing sufficient information to (later) validate data consistency. Our *delayed consistency* model implies synchronization only at the start and end of each IO operation. As a result, per node variations in process performance and/or in messaging delays experienced by consistency operations are avoided. The actual delay incurred corresponds to the longest total IO time for any single process rather than the sum of the longest times for any process for each step in the entire IO action. Further, by not exposing the actual calls made to the science code, end users can freely choose between using a fully consistent parallel API (e.g., during code development) vs. our delayed consistency methods (e.g., during high end production runs). Similarly, users can first employ MPI-IO or POSIX IO and then convert from the BP format to either HDF-5 or NetCDF, as needed. All such choices simply entail changing one entry in the ADIOS XML configuration file.

Our principal motivation for associating lightweight data characterization with output actions is to improve scientists' productivity, which is being reduced by the need to move across wide area links the ever larger datasets produced by petascale codes. Data characterization affords them with the opportunity to quickly inspect output data, to assess whether such movements are important before performing them. We term such actions data characterization because they can be user-specified and need not be complete, thereby avoiding the known high overheads of data indexing like that used by FastBit and PyTables. Further, entries for data characterization are made default parts of the BP intermediate file format, much like 'attributes' in other systems [6], so that end users need not be concerned with format changes or updates when data characterizations are changed.

As our primary research examples have all been write intensive science codes, we have focused our initial work at delivering the best possible write performance while still affording good read performance. We collect sufficient information during the write operations to enable good performing read operations, but we have not spent any time optimizing this portion of the code. Initial tests for restart-style reading of the BP format has shown that performance for data per process of

[1] The BP name derives from the "binary packed" format, a purely log-based format that was originally used by ADIOS.

greater than a few 10s of KB yields the same or better performance, even for non-integer multiples of processes reading the data. Additional testing for analysis-style read operations is ongoing.

The BP file format was specifically designed to support delayed consistency, lightweight data characterization, and resilience. The basic file layout is shown in Figure 3.

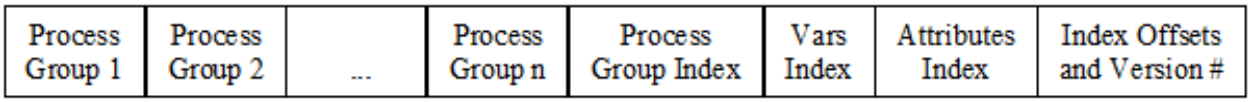

Figure 3. BP File Layout

Each process writes its own output into a *process group* slot. These slots are variably sized based on the amount of data required by each process. Included in each process output are the data characteristics for the variables. For performance, we are investigating the advantages of padding these slots to file system whole stripe sizes. These and other size adjustments are possible and centrally managed during the file open process. This flexibility will be required to get the best possible performance from an underlying transport and file system.

The three index sections are stored at the end for ease of expansion during append operations. Their manipulation is currently managed by the root process of the group performing IO. The overhead of these indices is acceptably small even for a large number of processes. For example, for 100,000 processes and a large number variables and attributes in all process groups, such as 1000, the total index size will be on the order of 10 MB. Given the total size of the data from an output operation of this size, 10 MB constitutes little more than a rounding error. Since these are at the end of the file, we reserve the last 28 bytes of the file for offset locations and for version and endian-ness flags.

Delayed consistency is achieved by having each process write independently with sufficient information to later validate that the consistency was not violated. While the replication of this data may not seem desirable, consider the ramifications of a three orders of magnitude performance penalty for instead, maintaining a single copy or consider the potential that the single copy being corrupted renders the entire output useless. We have measured the overhead per process to be on the order of a few hundred bytes for a few dozen variables. This cost, we believe, is well worth the time savings and greater resilience to failure.

Data characteristics are replicated into the indices stored at the end of the file. As mentioned above, the location of the index is stored at a known offset from the end of the file, thereby making it easy to seek to the index. Since the index is internally complete and consistent, it can be separated out and queried to determine if the associated data contains the desired features.

The BP format addresses resilience in two ways. First, once the initial coordination to determine file offsets is complete, each process can output its entire data independently and close the local connection to the file. This will commit the local data to storage, which constitutes some level of safety. Afterwards, ADIOS gathers all of the index data for each single output to the root

process of the output operation, merges it together, and writes it to the end of the file as a footer. This merging operation is strictly appending various linked lists together making it efficient to perform. Second, the replicated metadata from each process in the footer gives a list of offsets to where each process group was written. Should this fail, it is possible to linearly search the file and determine where each process group begins and ends.

Evaluation

To evaluate, we need to examine each of our three goals: 1) an API almost as simple as POSIX IO, 2) fast IO, and 3) changing IO without changing source.

Simple API

Standard POSIX IO calls consist of open, write, and close. ADIOS nearly achieves the same simplicity with the sole addition of the adios_group_size call. This one addition provides a predictive maximum size a process will write making it possible to offset each process to a non-overlapping portion of the file without requiring later coordination. The write calls are slightly more complex in that they require a var name as well as a buffer. Note that since we have described the types fully in the XML, we need not specify a buffer size directly. If we need to specify the bounds, we will make additional write calls to add the sizing information so that ADIOS can properly figure out how large the buffer should be. We found no way to simplify this API further except at the cost of functionality or complexity. All efforts have focused on keeping this API as simple as possible with descriptive, clear annotation in the XML as the preferred method for altering the behavior of the write calls.

To simplify things further, it is possible to use code generated by preprocessing he XML to replace all of the adios_group_size, adios_write, and adios_read calls automatically. This further insulates the end user from having to deal with the complexities of their code by solely working within the XML file for all of their data description and output needs. In order to update what data is part of a group, change the XML and recompile and the code will be updated automatically. We realize that this cannot handle all of the ways that data is written, but we believe it handles a sufficiently large percentage that most of the exception cases will be restructured to fit the new model rather than having to write the calls manually.

Fast IO

The main performance evaluations were performed using the GTC fusion code. We also have some preliminary results with Chimera as well as GTS. For us, the time that matters is how long the code runs for a given amount of work. We judge our IO performance by running the code without IO and with IO comparing the total runtime difference. We use that and the data volume generated to determine our IO performance.

GTC Performance

To evaluate the system, we have run regular tests with GTC at ORNL just before the machine was partially shutdown for an upgrade. The system is a Cray XT4, dual core AMD x64 chips with

2 GB of RAM per core and around 40-45 GB/sec peak IO bandwidth to a dedicated Lustre parallel file system. Our tests showed a consistent average aggregate write performance of 20 GB/sec for a 128 node job [28]. See Figure 4.

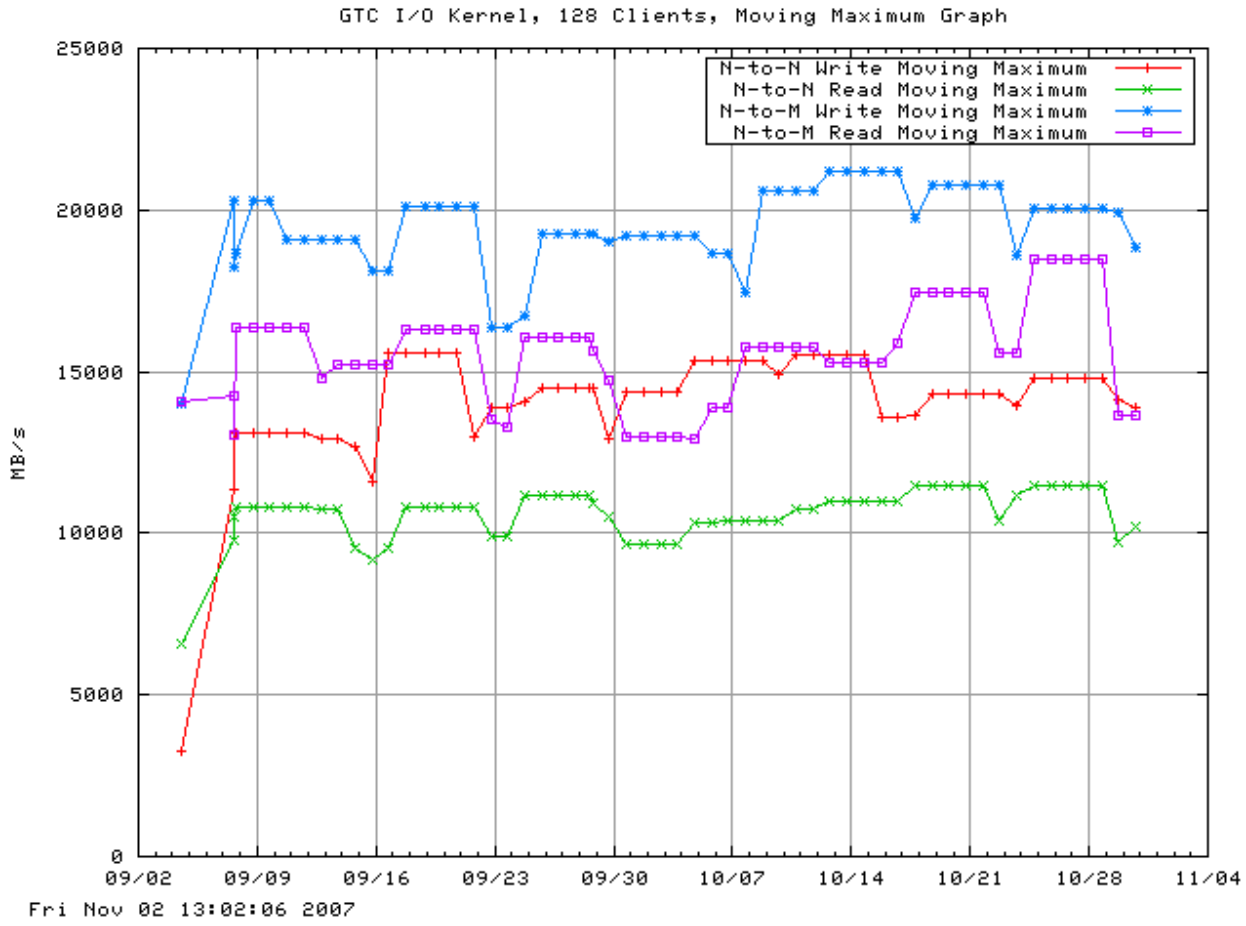

Figure 4. GTC on Jaguar with ADIOS

A second evaluation was performed on the ewok system at ORNL. This is an Infiniband cluster of 81 nodes of dual core AMD x64 chips, 2 GB of RAM per core, and about 3 GB/sec peak IO bandwidth to a Lustre file system shared with several other clusters. Two sets of 5 runs for GTC on 128 cores were performed. Each run generated 23 restart outputs for a total of 74.5 GB. The first set was configured to generate output using the MPI synchronous transport method while the second set was configured to generate no output using the NULL method. We were able to demonstrate an average 0.46 GB/sec performance. Given the ability to login to various nodes on the machine directly and the shared storage system, there is a large variability in the performance. Two of our runs with IO were faster than one without IO.

Chimera Performance

The Chimera integration demonstrated that ADIOS can operate well with a much larger number of elements, the various array dimension variants, and it provided us with an impetus to develop a clean reading API. This reading API differs from the writing one in that instead of calling adios_write () for each data item, adios_read () is called. On adios_close (), the provided buffers are populated normally. While the reading API is implemented and tested, not all of the ADIOS features have been optimized to work well with reading yet. Initial performance results for Chimera have been favorable. Without any tuning of parameters and just using stock ADIOS IO creating the same number of files with all of the annotation, we were able to reduce the wall clock runtime by 6.5%. More extensive testing is ongoing.

GTS Performance

We did complete one set of test runs with the GTS fusion code on the Jaguar system at Oak Ridge National Laboratory. We used 1024 cores writing 64 files per restart to the attached Lustre system and yielded 13 GB/s aggregate performance across all of the restarts written.

Changing IO Without Changing Source

By editing the method entry of the XML file, the IO routine selected when the code runs will be changed. An important concept of this worth repeating is that multiple method entries can be set for each adios-group within the XML file specifying multiple outputs for a single data group. These will be performed in the order specified transparent to each other and to the scientific code. For example, if the analysis data should be written using MPI-IO to disk and then be picked up for processing by a workflow system, adding a transport method that triggers the workflow system as a second method entry for the analysis data group will cause the data to be written to disk and then the workflow system will be notified. Note that the success of this approach would depend on the data being written using a synchronous IO routine. To this end, we have created two visualization transport methods. The first was tested with Visit through a VTK API interface and the second to a custom OpenGL renderer using a socket connection.

Delayed Consistency and Abstracted API

We evaluate the Chimera supernova code with weak scaling using both the native HDF-5 calls used in that code vs. our ADIOS calls for POSIX, for independent MPI-IO, and for collective MPI-IO. Results appear in Figure 5. The two most important items to notice in the graph are (1) that the parallel HDF-5 calls no longer scale linearly beyond 2,048 processes and (2) that the performance difference for 16K processes is three orders of magnitudes when compared with POSIX IO. We note that these runs use parallel HDF-5 tuned to use independent MPI-IO as the transport and all

recommended performance options are enabled, based on the recommendations by IO experts at ORNL.

Figure 5. Chimera Weak Scaling

A detailed profile of the HDF-5 vs. ADIOS calls is attained for a 512 process run. The tables below are the total time spent in all processes on each operation. Three key differences are apparent in Tables 1 and 2. First, ADIOS does a cascaded MPI_Open call. The real cost of the open operation is the sum of the MPI_File_open and the MPI_File_recv. Even combined, they are less than the parallel open performed by HDF-5. The second and more striking is the number of writes and the time spent performing them. The third and most important for our delayed consistency argument is the MPI_Bcast calls to enforce consistency. ADIOS performance is improved by its ability to alter how the open operation is performed, buffering the write commands to perform larger, less frequent calls, and not requiring the broadcasts for consistency checking.

Briefly, the conversion time for a BP file generated from an 8192 process run into an HDF-5 format is only 117 seconds.

The performance impact of collecting the data characteristics during the IO process depends on the number of arrays and the number of elements. The calculation of the characteristics is embarassingly parallel and takes only a few seconds at most for as much as 512M elements.

Table 1. Parallel HDF-5

Parallel HDF-5		
Function	*# of calls*	*Total Time (sec)*
write	144065	33109.67
MPI_Bcast	314800	12259.30
MPI_File_open	2560	325.17
H5P, H5D, etc.	–	8.71
other	–	60

Table 2. ADIOS Independent MPI-IO

ADIOS Independent MPI-IO		
Function	*# of calls*	*Total Time (sec)*
write	2560	2218.28
MPI_File_open	2560	95.80
MPI_Recv	2555	24.68
other	–	65

Related Work

There has been significant prior research into studying improvements to the data movement and annotation. The work can be divided into five areas. First are the transport level efforts. Before the 100s of terraflops era, the performance of IO was successfully managed either through POSIX style 1-file-per-process approaches, MPI-IO [13] independent or collective access. At extreme scale, the overhead of coordinating the collective IO calls degrades rather than increases performance [33]. The number of files generated with the 1-file-per-process approach becomes unmanageable. Even using MPI-IO independent access to a single file can be limited based on the file system [5]. To manage the synchronous time impact of writing, systems like DataTap [2] shift to using asynchronous IO, but face scheduling issues to avoid impacting the existing communication phases of the host application. This approach and other data staging efforts [25] require additional resources and break the direct connection between the end of an IO phase in the host application and the actual writing of data to disk. The communication interference issues have been successfully addressed [1] through several scheduling techniques affording the introduction of services into this output stream or in a staging area. While this approach is attractive, it is not appropriate for all applications. The introduction of risk of losing the output between the time when the application "completes" the IO and the data is flushed to disk may be unacceptable or the additional resources required to perform the staging work may not be available. In these cases, a synchronous approach is required. Furthermore, the asychronous writing and staging approaches do not address reading data.

Delayed consistency has previously been studied for file systems. For example, the Lightweight File Systems [26] project at Sandia Labs has stripped down POSIX semantics to a core of authentication and authorization affording layering of other semantics, like consistency, on an as-needed basis. Other file systems like the Serverless File System [3] have distributed metadata weakening the immediate consistency across the entire network of machines. NFS [24] relies on write-back local caches limiting the globally consistent view of the file system to the last synchronization operation. Our work considers the use of delayed consistency within single, large-scale files, the intent being to 'fix them up' whenever possible without inhibiting the performance of the petascale application.

The two most successful file-formatting efforts are NetCDF [23] and HDF-5 [11]. NetCDF is attractive for writing large, contiguous arrays and related scalar values in a single-level heirarchy. HDF-5 extends this notion by incorporating a block-based approach for storing data, a multi-level heirarchy, and the ability to attach attributes to any data item or hierarchy level. Likely, these efforts are successful because they provide a standard API to handle the IO and the ability to read the data back on various different platforms. To extend the ability of these APIs to perform well at larger scale, parallel versions of these APIs were developed [11, 17]. Additionally, they can be configured to take advantage of different transports as a compile-time option. The BP file format [21] was developed to scale beyond the levels capable by HDF-5 and NetCDF by reducing the coordiation required during the output. While this sacrifices consistency at runtime, it can still be validated offline. Once code has been validated to generate consistent output, paying the cost of validating the consistency during every output is excessive overhead that should be avoided, if

possible.

The third area of related work is related to the file systems. The standard POSIX semantics offered by systems like NFS [24], Lustre [5], and gpfs [29] works well for maintaining a level of consistency and safety for files. While these approaches work well for typical interactive use, it does not necessarily handle parallel access well. For example, metadata operations can be a bottleneck [12]. Managing metadata effectively can be done while maintaining POSIX semantics as demonstrated in Ceph [31]. However, those semantics are not always required for storage operations. By removing all but the most essential services, a custom file system can give excellent performance since it does not pay the performance penalties of any services that are not required [26]. While this works well, it would still require customizations similar to the proposed work so that it can adapt to different application requirements. In that sense, this work builds on the insights gained from the custom file system research while moving the changes into the application layer to achieve greater customization to the application characteristics on a variety of systems.

A different approach for the file system performance has been investigated in relation to Quality of Service (QoS) on shared systems. AQuA [32] provides adaptive control of storage bandwidth in a shared environment. While it can provide guarantees about performance, it requires the ability to proportionally allocate bandwidth to various users. We seek to provide an approach that does not throttle the performance of any particular application when available bandwidth is available. Instead, we shift the workload to less busy areas of the storage system.

The fourth area of related work is related to data management. The Storage Resource Management (SRM) [30] project manages storage allocation, streaming data between sites, and enforcing secure interfaces to the storage systems. This nicely manages to use of a generated data set from where it was generated to where it is needed, but it does not annotate the data in any way that assists the user with data selection or analysis avoiding unnecessary data movement. The Oak Ridge National Laboratory Dashboard [8] inspects data files and generates structural views of the data sets and even integrates SRM data management. It does not, however, take advantage of or introduce any data annotation to aid in data selection. At best, the user can see at a glance which timesteps are available for each variable generated. Some early work has been done on scientific databases [27], but it has not been addressed with the complexities the extreme levels of parallelism in use today and in the near future.

The last area deals primarily with data streaming systems. There has been a lot of work with managing the data transmission to work around bottlenecks and to annotate and select meaningful data from the stream. Many of these [15, 7, 4] use a centralized site for applying selection or annotation operations. More scalable solutions, for example [16, 14], use in-network aggregation and distributed deployment of analysis routines. Others push the data selection operations to the source [6] to reduce the data volumes. While some aspects of these systems are very relevant, they have not been applied to HPC environments. The DataTap system mentioned above is one system seeking to directly bring these innovations into the HPC realm.

Conclusions

This LDRD contributed to the development of key technologies that have potential to impact a broad set of HPC applications across the DOE complex. In particular, the ADIOS API has proven to be both simple to use and effective at improving overall IO performance, and reducing performance variability for a number of HPC applications. Performance evaluations demonstrate the ability to achieve as much as 75% of peak IO performance by carefully managing interactions with the storage system. The BP file format [21] also enhances IO performance by offering data annotation and automaic characterization with a minimal amount of coordination among the collective processes performing IO at the same time.

Ongoing and future work will look at how to isolate and address the synchronous IO bottlenecks in API and file system characteristics for extreme scale machines by introducing autonomic IO routines that adapt to changing system conditions, providing annotation and technical mechanisms through which data can be organized during output to aid analysis performance, and through the use of new tools and techniques for reading and managing the adapted output file(s) enabling a focus on simulation run (or runs) rather than file(s).

The broader impact of this work is to reduce the time spent creating IO routines for science codes while still achieving a high percentage of peak IO performance. By managing the entire output as a single entity, it will make scientists more productive when using these extreme scale machines.

References

[1] Hasan Abbasi, Matthew Wolf, Greg Eisenhauer, Scott Klasky, Karsten Schwan, and Fang Zheng. Datastager: scalable data staging services for petascale applications. In *HPDC '09: Proceedings of the 18th ACM international symposium on High performance distributed computing*, pages 39–48, New York, NY, USA, 2009. ACM.

[2] Hasan Abbasi, Matthew Wolf, and Karsten Schwan. Live data workspace: A flexible, dynamic and extensible platform for petascale applications. In *CLUSTER '07: Proceedings of the 2007 IEEE International Conference on Cluster Computing*, pages 341–348, Washington, DC, USA, 2007. IEEE Computer Society.

[3] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. Serverless network file systems. *ACM Trans. Comput. Syst.*, 14(1):41–79, 1996.

[4] Shivnath Babu and Jennifer Widom. Continuous queries over data streams. *SIGMOD Rec.*, 30(3):109–120, 2001.

[5] Peter J. Braam. Lustre: a scalable high-performance file system, Nov. 2002.

[6] Fabian Bustamante, Greg Eisenhauer, Karsten Schwan, and Patrick Widener. Efficient wire formats for high performance computing. In *In Proceedings of Supercomputing 2000*, 2000.

[7] Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Monitoring streams: a new class of data management applications. In *VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases*, pages 215–226. VLDB Endowment, 2002.

[8] J. Cummings, S. Klasky, R. Barreto, N. Podhorszki, G. Park, C. S. Chang, L. Sugiyama, and P. Snyder. Edge kinetic-MHD code coupling and monitoring with Kepler workflow. *APS Meeting Abstracts*, pages 3009–+, November 2007.

[9] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, 2003.

[10] Karsten Schwan Hasan Abbasi, Matthew Wolf. Live data workspace: A flexible, dynamic and extensible platform for petascale applications. In *Cluster Computing*, Austin, TX, September 2007. IEEE International.

[11] HDF-5. http://hdf.ncsa.uiuc.edu/products/hdf5/.

[12] Steve Hodson. Using lustre effectively. In *NCCS Scaling Workshop*, Oak Ridge, TN, August 2007.

[13] Message Passing Interface. Mpi-2: Extensions to the message-passing interface, 1996.

[14] Navendu Jain, Lisa Amini, Henrique Andrade, Richard King, Yoonho Park, Philippe Selo, and Chitra Venkatramani. Design, implementation, and evaluation of the linear road bnchmark on the stream processing core. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 431–442, New York, NY, USA, 2006. ACM.

[15] Rainer Koster, Andrew P. Black, Jie Huang, Jonathan Walpole, and Calton Pu. Infopipes for composing distributed information flows. In *M3W: Proceedings of the 2001 international workshop on Multimedia middleware*, pages 44–47, New York, NY, USA, 2001. ACM.

[16] Vibhore Kumar, Brian F. Cooper, Zhongtang Cai, Greg Eisenhauer, and Karsten Schwan. Resource-aware distributed stream management using dynamic overlays. In *In Proc. of 25th IEEE International Conference on Distributed Computing Systems (ICDCS-2005*, pages 783–792, 2005.

[17] Jianwei Li, Wei keng Liao, Alok Choudhary, Robert Ross, Rajeev Thakur, William Gropp, Rob Latham, Andrew Siegel, Brad Gallagher, and Michael Zingale. Parallel netCDF: A high-performance scientific I/O interface. In *Proceedings of SC2003: High Performance Networking and Computing*, Phoenix, AZ, November 2003. IEEE Computer Society Press.

[18] Z. Lin, T. S. Hahm, W. W. Lee, W. M. Tang, and R. B. White. Gyrokinetic simulations in general geometry and applications to collisional damping of zonal flows. *Physics of Plasmas*, 7(5), May 2000.

[19] J. Lofstead, S. Klasky, M. Booth, H. Abbasi, F. Zheng, M. Wolf, and K. Schwan. Petascale io using the adaptable io system. In *In Proceedings of the 2009 Cray User Group Meeting*. Cray User's Group, 2009.

[20] Jay Lofstead, Scott Klasky, Karsten Schwan, Norbert Podhorszki, and Chen Jin. Flexible io and integration for scientific codes through the adaptable io system (adios). In *CLADE 2008 at HPDC*, Boston, Massachusetts, June 2008. ACM.

[21] Jay Lofstead, Fang Zheng, Scott Klasky, and Karsten 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.

[22] Jay Lofstead, Fang Zheng, Scott Klasky, and Karsten Schwan. Adaptable, metadata rich io methods for portable high performance io. In *In Proceedings of IPDPS'09, May 25-29, Rome, Italy*, 2009.

[23] netCDF. http://www.unidata.ucar.edu/software/netcdf/.

[24] NFS. http://www.ietf.org/rfc/rfc3010.txt.

[25] Arifa Nisar, Wei keng Liao, and Alok N. Choudhary. Scaling parallel i/o performance through i/o delegate and caching system. In *SC*, page 9, 2008.

[26] R.A. Oldfield, L. Ward, R. Riesen, A.B. Maccabe, P. Widener, and T. Kordenbrock. Lightweight i/o for scientific applications. *Cluster Computing, 2006 IEEE International Conference on*, pages 1–11, 25-28 Sept. 2006.

[27] John L. Pfaltz, Russell F. Haddleton, and James C. French. Scalable, parallel, scientific databases. In *In 10th International Conf. on Scientific and Statistical Database Management*, pages 4–11, 1998.

[28] Sarp Oral GTC Test Results. http://users.nccs.gov/ oral/jagregtests/gtc128.html.

[29] Frank Schmuck and Roger Haskin. Gpfs: A shared-disk file system for large computing clusters. In *In Proceedings of the 2002 Conference on File and Storage Technologies (FAST*, pages 231–244, 2002.

[30] Arie Shoshani, Alex Sim, and Junmin Gu. Storage resource managers: Middleware components for grid storage, 2002.

[31] Sage Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Conference on Operating Systems Design and Implementation (OSDI '06)*, November 2006.

[32] Joel Wu and Scott A. Brandt. The design and implementation of aqua: an adaptive quality of service aware object-based storage device. In *Proceedings of the 23rd IEEE / 14th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 209–218, May 2006.

[33] Weikuan Yu and Jeffrey Vetter. Parcoll: Partitioned collective i/o on the cray xt. *Parallel Processing, International Conference on*, 0:562–569, 2008.

DISTRIBUTION:

1 Gerald Lofstead
Georgia Institute of Technology
College of Computing, Georgia Tech 266 Ferst Dr.
Atlanta GA 30332-0765

1 Karsten Schwan
Georgia Institute of Technology
College of Computing, Georgia Tech 266 Ferst Dr.
Atlanta GA 30332-0765

1 Scott Klasky
Oak Ridge National Laboratory
P.O. Box 2008, MS6008 Oak Ridge, TN 37831-6008

1 MS 0123 Donna Chavez, 1011

1 MS 1319 Ron Oldfield, 01423

1 MS 1319 Ron Brightwell, 01423

1 MS 0899 Technical Library, 9536 (electronic)

Sandia National Laboratories