EXTREME SCALE DATA MANAGEMENT IN HIGH PERFORMANCE COMPUTING

A Thesis
Presented to
The Academic Faculty

by

Gerald F. Lofstead II

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
College of Computing, School of Computer Science

Georgia Institute of Technology
December 2010

EXTREME SCALE DATA MANAGEMENT IN HIGH PERFORMANCE COMPUTING

Approved by:

Regents' Professor Karsten Schwan,
Committee Chair
College of Computing, School of
Computer Science
Georgia Institute of Technology

Professor Ling Liu
School of Computer Science
Georgia Institute of Technology

Regents' Professor Karsten Schwan,
Advisor
College of Computing, School of
Computer Science
Georgia Institute of Technology

Scott Klasky
Scientific Computing Group
Oak Ridge National Laboratory

Research Scientist Matthew Wolf
School of Computer Science
Georgia Institute of Technology

Ron Oldfield
Scalable Computing Systems
Sandia National Laboratories

Date Approved: 27 August 2010

To Cheryl, for helping me stay sane as I learned the difference between engineering and research and then worked through this thesis.

ACKNOWLEDGEMENTS

I want to thank my committee for their valuable feedback and questions that helped me reshape my work into this finished product. For all of you, the working relationship we have shared has helped shape me into the researcher I am today. Each of you has contributed a bit of wisdom that I will use throughout my career. I would also like to thank Sandia National Laboratories for the five years of financial support that made my time in grad school far less stressful. I would also like to thank Oak Ridge National Laboratory, particularly Ricky Kendall of the National Center for Computational Sciences, for their generous support in terms of computing time on the big machines that made this work possible.

Finally, I would like to thank my fellow students for their help and friendship throughtout my years at Georgia Tech. In particular Fang Zheng has been a great friend and a tremendous help and Hasan Abbasi has been a constant motivator for completing my degree.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Extreme scale data management in high performance computing requires consideration of the end-to-end scientific workflow process. Of particular importance for runtime performance, the write-read cycle must be addressed as a complete unit. Any optimization made to enhance writing performance must consider the subsequent impact on reading performance. Only by addressing the full write-read cycle can scientific productivity be enhanced.

The ADIOS middleware developed as part of this thesis provides an API nearly as simple as the standard POSIX interface, but with the flexibilty to choose what transport mechanism(s) to employ at or during runtime. The accompanying BP file format is designed for high performance parallel output with limited coordination overheads while incorporating features to accelerate subsequent use of the output for reading operations. This pair of optimizations of the output mechanism and the output format are done such that they either do not negatively impact or greatly improve subsequent reading performance when compared to popular self-describing file formats. This end-to-end advantage of the ADIOS architecture is further enhanced through techniques to better enable asychronous data transports affording the incorporation of 'in flight' data processing operations and pseudo-transport mechanisms that can trigger workflows or other operations.

CHAPTER I

INTRODUCTION

Background and Problem Space Science simulation data management requires consideration of the entire end to end process and all of its stages of moving data from high performance simulations to long term storage. This is particularly important for extreme scale science codes, e.g., XGC-1 fusion, climate, and s3d combustion, that when running on current petascale machines like jaguarpf and roadrunner already generate data volumes at per process data sizes that range from 10s-100+ MiB per process. This means that at 10 MiB/process, 225,000 processes yield 2.25 TiB of data generated at regular intervals. For 128 MiB/proc, this grows to a volume of 28.8 TiBs total, and when writing at 50 GiB/sec, it takes 45 secs. for 10 MiB and 576 secs. for 128 MiB, nearly 10 minutes, assuming, of course, that there are no issues with interference for both storage and/or with potential bottleneck resources like metadata managers. In fact, typical output rates experienced on current machines are frequently less than half of the 50 GiB/sec listed above, thus at least doubling the time spent performing output.

Limited output bandwidths make it difficult for application scientists to save data as often as they would like. This is illustrated with the GTC code for which an output interval of every 15 minutes is common, with output data volumes of up to 128 MiB/process. In fact, with the numbers cited above, the GTC code would experience an IO overhead of at least 40%! Further, in order to analyze generated data for scientific insights being sought, data must be read back in, potentially multiple times, to identify data features of relevance using techniques like statistical analysis or visualization. With reading being at least equally critical to the scientific discovery process, care must be taken to reduce the amount of data movement to and from storage in order for these analysis processes to operate. As a simple illustration, consider if all of the data must be read in twice to generate two different analysis views. Assuming sufficient parallelism is employed to reach the bandwidth capacity of the

file system, it could take nearly 20 minutes just to read the data and then additional time to render the output. If multiple passes must read, manipulate, filter, or otherwise process the data before another phase of processing occurs, each step will require a pair of data movement operations – one to read and one to write. This impediment to realtime exploration of data sets can make the process of using generated data painful at best, or impossible at worst.

The numeric examples shown above demonstrate a continuing need to improve the IO performance of high end applications. While this is one of the problems addressed by this thesis, a further goal of this work is to go beyond high performance data movement to storage, but in addition, to accelerate the process of gaining potential scientific insights from the data produced by simulations. Specifically, we wish to consider the write-read cycle portion of the 'end to end' scientific discovery process. The important detail of this work is that the combination of both the writing and reading performance should be considered so that the 'end to end' process can be accelerated. This is illustrated by several examples. For instance, it may be useful to 'trade' small, additional computation times incurred by applications – resulting in slightly increased output times – to reduce end to end IO time across the pipeline, a specific example being data reorganization when it is written, with the purpose of creating file structures and associated disk layouts that offer improved performance for subsequent 'read' operations. In fact, sometimes, and as shown in this thesis, it is possible to improve both write and read performance in this fashion. Another example is to change the IO behavior of an application to move it from using disk-based methods for code coupling to using in-memory or network-based coupling techniques [26, 1], thereby reducing an application's aggregate disk bandwith requirements. Finally, we can manipulate output data 'in-transit', before it is stored, or for code coupling, to address mismatches in data layouts or organization between coupled codes, thereby reducing total data movement and potentially accelerating the codes being coupled.

The technological basis of this research are the petascale machines and their file systems currently being used at leadership sites like DOE's Oak Ridge National Lab. The high performance machines located at those sites contain 100,000s of cores and 100s of TiB of RAM.

For example, JaguarPF contains 12 cores/node and a total of 300 TiB of RAM. Intrepid contains 4 cores/node and 80 TiB of RAM. In the near future, Blue Waters, Cielo, and Mira are coming online. Blue Waters will have 128 threads/node with 800 TiB of RAM. Cielo will have 16 cores/node and 298 TiB of RAM. The next generation BlueGene, Mira, will have 16 cores/node and 750 TiB of RAM. While this memory capacity is extreme in total, the growth in available RAM per process is matching the growth in core or thread count. Overall, the trend is towards 1 GiB/core or thread. This continuing explosion in core counts and corresponding growth in RAM, even if it just maintains 1 GiB/process, yields staggering data volumes. Managing these data volumes with homogeneous compute architectures follows a more traditional configuration with all compute resources dedicated to either compute or IO tasks. With the introduction of co-processor and compute accelerator architectures such as GPGPUs, the problem becomes worse. For example, currently available is the GE IPN250 blades where each blade has a dual core CPU and 96 GPU cores with 8 GiB/blade. The CPUs are primarily intended for serial tasks and to manage their GPU resources. The tremendous computation acceleration provided by the GPUs demands more frequent data offloading to maintain sufficient detail in the generated data for later analysis. This increased frequency of output will further tax storage architectures beyond what is now being experienced with homogeneous HPC resources. This trend towards co-processors and compute accelerator architectures will only increase as managing the power consumption of extreme scale computing resources becomes a more serious concern.

Technological advances are not likely to lead to 'easy fixes' for the IO problems experienced by future machines. To help manage IO times, one could consider using solid state storage devices, SSDs, or future phase change memory. However, there is a current 48x capacity/price advantage for HDD over SSDs [119], and with HDDs continuing their nearly 30 year historical doubling of space per unit cost every 14 months [36], SSDs will continue to lag the capacity and cost per capacity of HDDs for a long time to come. A reasonable assumption, therefore, is that SSDs will not replace HDDs, but may instead be employed for staging and other temporary use storage areas that are then spooled out to cheaper, slower, higher capacity HDDs or other storage media for end to end use. Later,

the data may be archived on tape or other cheaper, slower storage for long-term storage. The outcome will be additional layers in the memory hierarchy present on future machines. Further complexity is introduced by the continuing desire to manage the cost of obtaining and maintaining new machines. An obvious place to control these costs is to centralize the scratch space storage array and share it across multiple machines, as has already been done for both the NERSC and ORNL leadership computing facilities and soon for Sandia Labs as well. An example of what a data center organization like this would look is illustrated in Figure 1. The boxes 'Super 1' and 'Super 2' represent the high performance computing resources. The 'E2E' resources represent additional, more specialized resources intended for visualization or other 'end to end' tasks off of the main compute resources. This architecture is being adopted because there is considerable cost in building a storage system with sufficient bandwidth. Adding extra capacity to a large scale storage system and sharing that storage across multiple resources is far more cost effective than building equivalent individual systems for each resource. In addition, the shared space enables easier end to end data processing across resources through storage based integration. However, the cost benefits of this approach can lead to additional issues particular to HPC.

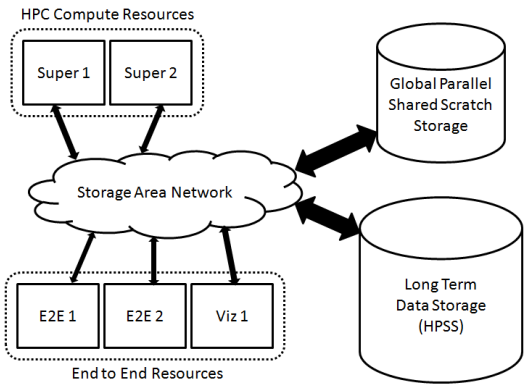


Figure 1: General HPC Data Center System Architecture

Enterprise systems and applications have long used the centralized or global storage facilities present in HPC data centers. Unfortunately, enterprise solutions do not fully address the storage needs of HPC codes, for multiple reasons. First, in contrast to most enterprise

applications, an HPC application can demand instantaneous and sole access to a large fraction of all storage resources. An example is a petascale code that outputs restart data. If IO resources are insufficient, this code will block and waste CPU cycles on compute nodes waiting for output rather than making positive progress for the ongoing scientific simulation. Such latency sensitive behavior is characterized by periodic output patterns, with little or no IO activity for the 15 or 30 minutes of duration of alternating computation and output steps thereby providing distinct deadlines for IO completion. Second, the resource demands imposed by single large-scale codes are magnified by the simultaneous use of the petascale machine by multiple batch-scheduled applications, each desiring a substantial portion of IO system resources and each demanding low latency service. Third, when file systems are shared, like those at ORNL and NERSC, it is not just the petascale codes that demand IO system resources, but there are also additional requests that stem from the analysis or visualization codes running on select petascale machine nodes and/or on attached cluster machines with shared file system access. These are the problems, unique to the HPC domain, that are the focus of our research, as made more precise in the problem statement articulated next.

1.1 Problem Statement

Extreme scale computation in HPC requires careful management of the write-read cycle of the end to end scientific discovery process. A sole focus on optimal write performance can negatively impact the read performance experienced by subsequent analysis codes. Additional time and efforts spent in data generation, including by filtering, sorting, or annotating data, can produce substantial savings by speeding up data selection and retrieval for analysis. In summary, when running petascale simulations, managing both the write and read cycles of data generation is important for improving the end to end processes in which scientific insights are derived from simulation output.

1.1.1 Requirements

Multiple requirements must be met when managing the write/read cycle of petascale simulations.

The first set of requirements is derived from diversity in petascale machines and application needs. Concerning machine diversity, IO techniques should be configurable such that porting an application to a new platform can afford selection of a platform-specific set of IO optimizations without requiring source code changes. Second, to accommodate application needs, the implementation of this selection mechanism must recognize that different kinds of IO operations have different requirements. For example, a restart output should employ as many parallel optimizations as possible to obtain high performance at scale, whereas the use simple POSIX IO suffices for a small-scale, single process diagnostic output. Third, output data should be encoded in ways that are portable to other platforms for analysis or other use and are easily and efficiently convertable to standard formats for rapid integration with third party tools and complex workflows. Fourth, changing the nature of IO data movement should not require source code changes. For example, it should be easy to switch from writing to disk to using an in memory code coupling technique. Fifth, any optimizations employed to improve output performance should not negatively impact subsequent reading performance. Finally, shared resources and bottlenecks must be addressed to avoid scaling issues.

The second set of requirements center around the data generation, or writing, performance at scale. First, the mismatch in scales for the number of compute resources compared with the number of storage components in the system must be considered. Second, any techniques developed must be shown to work at scale to meet the extreme scale environment. Third, the output mechanism should ideally support inclusion of optional mechanisms to apply operations to data prior to writing to disk or use in a coupled code.

The third set of requirements focus on the reading, or data consumption, requirements. First, any output formatting must not skew performance strictly in favor of write time. Ideally, any format employed will also improve read performance. Second, data annotations should be considered during output to leverage the embarassingly parallel data distribution before writing. Characteristics such as local min/max values for variables, for example, can greatly accelerate data selection tasks by avoiding reading the entire data set to find values that exceed some particular threshold.

Thesis statement, solution approach, and contributions. For extreme scale science simulations, it is important to carefully manage the write-read cycle for efficient scientific discovery. Optimizing write performance must consider the application as a whole, in terms of the scale for both IO and for communication, the HPC resource configuration for how to organize and perform operations, manageability and usability of output artifacts, and competing use of shared resources by other machines and applications. At the same time, reads must be considered such that read performance is improved, by annotating, organizing, and potentially, filtering or compressing data. Only by addressing both write and read performance, and without compromising existing tool chains, can we attain consistly high end to end performance for scientific workflows in modern HPC data centers.

Our solution approach is to *manage* the write-read cycle such that the efficiency of data generation *and* of subsequent data analysis is maintained or improved. Specifically, flexibility in selecting options in both the write and read phases affords balancing performance optimizations where it matters most for each science workflow and the platforms employed.

This thesis attains such flexibility by developing middleware methods that insulate codes from the implementation selections driven by platform and end to end data concerns. By using these methods, extreme scale science codes' IO routines can remain static when moving from platform to platform no matter which output format or technique is required for a particular job. Further, middleware methods also have knowledge of the data types, sizes, and distribution of the data. Since these middleware methods operate above the file system, they can take advantage of global system knowledge about transient 'hot spots' in the storage system as part of the optimizations. As a result, data can be dynamically characterized so as to permit 'in transit' data operations that assist in subsequent analysis operations and to enable adaptation of data output structure to the static and dynamic characteristics of the extreme scale system. In addition, by developing an optimized data format that is both portable and quickly convertable to popular formats such as HDF5 and NetCDF for use with existing tools, greater writing and reading performance can be attained. Finally, data characteristics generated during output on the embarassingly parallel compute cores provide statistical measures for rapid data selection during analysis tasks.

1.1.2 Implementation Contributions

The conceptual contributions listed above are implemented in the ADIOS middleware that is used by multiple important petascale applications. ADIOS – the ADaptable IO System – provides an API nearly as simple as POSIX IO calls with the flexibility to alter the actual IO techniques employed through the use of an external configuration file. In addition, a new intermediate file format, BP (Binary Packed), is created that organizes data more naturally for write performance while generally aiding read performance. Additionally, it adds an extensible set of data characteristics for rapid data selection and an organization intended to support easier append operations and recovery from failures during an output operation.

ADIOS has proven itself as a production quality system supporting application runs of more than 200,000 cores for XGC-1 and as an excellent platform for research. The abstraction layer provided by the simple API and the XML file affords transparent integration of experimental IO techniques into production codes without requiring any source code changes. Both the DataTap [3] and DART [25] asynchronous IO techniques have been shown in operation with production codes through the use of the ADIOS API. Further, experimental techniques for including 'in flight' data manipulation operations has also been shown [60]. This production success and extensive use as a research platform has shown the general usefulness of ADIOS for IO both now and for the future.

The remainder of this document delves into detail to support the thesis statement above. Chapter 2 introduces the ADIOS API middleware and the BP intermediate file format. It also demonstrates its write performance advantages. In particular, it explores the need for IO groups, metadata management, delayed consistency, and log-structured formats for scientific data and how ADIOS incorporates those ideas and the performance advantages achieved with these tools. Chapter 3 expands on these concepts by demonstrating the interference effects within extreme scale applications and the transient performance impact of large scale parallel file systems due to simultaneous usage. It then demonstrates an adaptive technique for managing output that manages both the internal and external interference effects to achieve peak performance more consistently and with less variability at scale. Finally, Chapter 4 evaluates the impact on common analysis tasks of the log-structured

format. This is followed by a discussion of previous work related to the thesis as a whole in Chapter 5 and an overall conclusion in Chapter 6.

CHAPTER II

HIGH WRITE PERFORMANCE WITH THE ADIOS MIDDLEWARE

2.1 Introduction

The first step in addressing the write-read cycle of extreme scale scientific computing is the creation of a middleware abstraction layer that affords the flexible optimizations and IO decisions required for good performance. The ADIOS middleware addresses this requirement by providing a simple API capable of driving a variety of different data management techniques and an accompanying file format, BP, designed to facilitate the operational requirements of extreme scale computing. Through this abstraction layer several technical contributions are expressed. First, ADIOS insulates the application from the need to incorporate any file system specific optimizations directly while providing the opportunity to employ such optimized techniques to accelerate the write-read cycle. Second, the techniques presented demonstrate examples of the kinds of file system specific optimizations that can be employed without requiring changing of the application source code. Third, the file organization and additional data characteristics created in the BP file format contribute to the performance as well. The file organization takes into account important considerations for extreme scale computing, such as the cost of global communication, to enable acceleration of the writing portion of the write-read cycle. The data characteristics collected during the embarrassingly parallel portion of the output phase do not noticeably increase the amount of time the output takes, completely disappearing in the variability of the output time, but offer a benefit for later data identification and reading performance acceleration. These contributions are evaluated in the context of petascale science applications and the benefits compared with existing best of breed techniques are shown.

The remainder of this chapter will be organized as follows. First, Section 2.2 gives an overview of the problems in the HPC environment ADIOS and BP are addressing. Section 2.3 describes related work. Section 2.4 describes a representative petascale application,

the GTC fusion modeling code. Section 2.5.1 outlines the ADIOS architecture, with section 2.6 presenting experimental evaluations of ADIOS. Section 2.7 contains conclusion and future work.

2.2 ADIOS and BP Overview

Scientific codes not only take many years to write, debug, and scale properly, but they also have long lifetimes. Once they have been finished, the authors and community are reluctant to redesign or change the code except for extreme cases such as very poor performance when moving to a new platform. Existing IO routines used in scientific codes vary from the simple of standard POSIX IO and raw binary MPI-IO writes to systems like HDF5 and parallel NetCDF with rich data annotation and portable data encoding. The tradeoffs for each of these IO approaches vary depending on the IO patterns, runtime compute system, and IO system being used [91]. For example, the Jaguar Cray XT4 system at Oak Ridge National Laboratory uses Lustre for the parallel file system while the bgl Blue Gene/L system at Argonne National Laboratory uses PVFS for the parallel file system. Therefore, to effectively run something like the GTC fusion code, it may be necessary to replace the IO routines with something that works better on both the Blue Gene/L and with PVFS. Complicating matters further, applications like GTC exhibit multiple different IO patterns in different parts of the code. Each of these IO operations or data 'groups' need to be independently tuned for optimal performance.

2.2.1 The ADIOS API and Transport Methods

The ADaptable IO System (ADIOS) middleware addresses these needs by providing a flexible platform through which different IO approaches can be deployed for each different data group independently.

These different IO approaches encompass both traditional synchronous techniques that block application execution while the IO completes as well as asynchronous techniques. Experimentation with new approaches for performing IO include performing in transit processing, asynchronous methods that give the IO system time to drain the tremendous number of compute nodes while allowing computations to proceed, and varying the degrees of

metadata annotation for improved scalability [28]. The ADIOS API contributes to these efforts by making it easier for developers to experiment with diverse, large-scale codes, in part because the actual IO implementation is separated from host source code.

While IO approaches are being reconsidered, the need for online simulation monitoring to ensure the scientific validity of code executions and to prevent 'useless' or problematic runs can also be examined. Such monitoring typically implies the need to integrate running simulations with analysis workflows, perhaps using workflow systems like Kepler [61], Pegasus [24], or DAGMan [63]. Coupling analysis with online monitoring requires the monitoring system to actively notify analysis or workflow components about the presence of new data. Unfortunately, due to the nature of parallel file systems like Lustre, the common 'notification' approach of looking for file existence to detect the end of a write phase or the presence of some other output can cause contention and slowdowns in the whole IO system thereby impacting the performance of the scientific code. ADIOS can be used to address this problem by supporting alternate methods for integrating monitoring systems with workflow components.

More broadly, the manner in which ADIOS addresses the integration of auxiliary tools with high performance codes meets four key requirements. First, since different IO routines have been optimized for different machine architectures and configurations, no single set of routines can give optimal performance on all different hardware and storage platform combinations. The ADIOS API, therefore, is designed to be able to span multiple IO realizations. Second, while richly annotated data is desirable, the complexity of writing the code to manage the data and the creation of annotations can be daunting. In response, the ADIOS API does not require richly annotated data, but instead, makes it possible and easy for end users to provide the degree of annotation they desire with the ability to add annotations without changing the source code. Third, once the code is stable, no source code changes should be required to support different IO routines for a different platform or IO system. ADIOS meets this requirement by embedding changes in XML files associated with IO rather than in application sources. Fourth, the integration of analysis or in situ visualization routines should be transparent to the source code, where ideally, the scientific

code should run with exactly the same performance whether it is used in conjunction with one of these ancillary tools or not.

The ADIOS API addresses both high-end IO requirements and low-impact auxiliary tool integration under the guidance of the four observations described above. It provides an API nearly as simple standard 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.

2.2.2 The BP File Format

Beyond the ADIOS API is the 'Binary Packed' (BP) file format. It specifically addresses different concerns. File formats like HDF5 and NetCDF are popular in part due to the rich tool chains available for the scientific data stored using these formats. Both HDF5 and NetCDF, however, were initially designed for serial access, limiting scalability when used in massively parallel codes. In response, the broader community has developed parallel versions of their APIs [35, 53], with good results demonstrated in the terascale environment compared to their serial counterparts. However, serious scalability issues remain for petascale machines and beyond. A simple example, explained in this chapter, is the inability of HDF5 to scale to 8192 processes for benchmarks conducted with the Chimera supernova code. Here, with every performance option enabled for parallel HDF5 enabled, we measure 1400 seconds to write a 7 GiB restart file to the Lustre system on the Jaguar machine at Oak Ridge National Laboratory (ORNL) whereas the use of ADIOS and its POSIX IO transport method reduces that time to 1.4 seconds! ADIOS with MPI-IO and collective MPI-IO yields performance of 10 and 14 seconds, respectively. Later improvements to the HDF5 implementation yielded a 10x performance improvement, but that still left a factor of

100x to POSIX IO and 10X to the MPI-IO transport methods within ADIOS. This chapter also evaluates the performance of a BP to HDF5 conversion performed serially on a single login node on Jaguar. For this example, the conversion requires 117 seconds. Even when this is combined with the initial writing time, the performance is improved.

Indirectly, the BP file format addresses other sources of poor IO performance. Overall, IO performance depends on many factors, including the file format used, the implementation and tuning of the associated API, the file system employed, and the architecture of the HPC resource being used for production runs. Our analysis of the parallel HDF5 implementation on Jaguar, for instance, reveals a large number of MPI_Bcast calls that are used to guarantee that all processes writing the collective values are writing consistent data to the file and that all IO processes maintain a coordinated march through the data elements for each collective output. The BP format design affords avoiding these consistency checks during the output operation while making it possible to easily check the consistency later. We still believe consistency checks are important during the development or maintenance of scientific codes, but production should not require the checking overhead. To facilitate this development and deployment structure, ADIOS provides end users easy access to multiple IO methods. On Jaguar, for instance, given the severe performance impact of validating the parallel consistency of data output, end users might use HDF5 and Lustre during testing, but then disable consistency checking during large-scale production runs. The HDF5 and NetCDF file formats require the consistency checks for proper operation. By using BP, we avoid the runtime costs of consistency checking, but can still obtain the data in a format that integrates with the tools currently used in the science workflow. This is done after the simulation has written the data by using a converter to validate consistency again, if desired, and then create the desired HDF5 or NetCDF formats. This notion of delayed consistency is described in more detail below.

ADIOS does not prescribe whether or when the output is converted to HDF5 or NetCDF formats. Such conversions can be done (i) 'in line' as part of the IO operation through an IO Graph [118], to ensure that only one format of the data will ever be stored on disk, (ii) offline using constructs termed metabots [117] that inspect disk-resident data and perform

conversions whenever possible and outside the IO fast path, (iii) as part of a larger, more complex workflow using Kepler [61], Pegasus [24], or DAGMan [63], or (iv) via a stand-alone file format converter. In all of these cases, such conversions can be done efficiently, with our initial results reported with a standalone converter running on a single processor on a login node of the ORNL Jaguar machine resulting in a 117 second conversion time for a 7 GiB output file (i.e., in contrast to the total 1400 sec. IO time for Chimera quoted above for parallel HDF5).

ADIOS and its BP file format not only support the flexible conversion to standard file formats, but they also facilitate the summary inspection of the data to determine if it contains features of interest to end users. One way to provide such functionality is to fully index the data, as done by multiple projects that have developed content indices for HDF5 files [101, 31, 39, 106]. To achieve a similar goal, but with less overhead in space and time, ADIOS supports the notion of *data characteristics* using which one can collect local, simple statistical and/or analytical data during the output operation (or later) for use in identifying desired data sets. Simple characteristics like local process array minimum and maximum values can be collected nearly 'for free' as part of the IO operation. More complex analytical measures like standard deviations or specialized measures particular to the science being performed may require processing that can be done in a variety of ways, including before or after the data has been written to disk. In all such cases, the BP format offers efficient, compact ways of storing these characteristics. When converting BP files to say, HDF5 or NetCDF, attributes can be used to maintain them.

Other features of the BP file format are designed to facilitate appends and rapid data access. A footer index is used to avoid the known limitation of header-based formats like NetCDF [70] where any change to the length of header data will require moving the data to either make room or to remove slack space. Further, by placing version identifier and an offset to the beginning of the indices as the last few bytes of a BP file, it becomes trivial to find the index information and to add new and different data to the file without affecting any data already written. Finally, we incorporate data characteristics into the index, so that we can separate the index for use as a table of contents for the file on a tape storage

system like HPSS [115].

2.2.3 Benefits of ADIOS with BP

To summarize, the ADIOS IO system and its BP file format are designed to help attain scalable, high performance IO while at the same time, maintaining compatibility with the rich tool chains existing for standard file formats like HDF5 and NetCDF. By using either parallel HDF5 or parallel NetCDF for initial code development and testing, the internal file consistency and 'correctness' of data output can be ensured. By switching to the BP format and using POSIX, MPI-IO, or collective MPI-IO methods for large-scale production runs and employing a converter, the IO time experienced by petascale codes can be reduced by up to three orders of magnitude, while still obtaining files with identical format and contents as when directly using native parallel APIs. The key contributions of ADIOS are as follows:

As part of the write-read cycle, existing scientific data formats like HDF5 and NetCDF have given rise to rich tool chains for use by science end users. Performance issues with their direct use by petascale (and beyond) applications, however, demonstrate the need for (1) improvements in scalability with respect to the degree of attainable parallelism, a specific technique used in this chapter being delayed data consistency, (2) the ability to perform rapid data characterization to improve scientific productivity, and (3) resilience to isolated failures through improved data organization.

Delayed consistency is well-known to improve the performance of file systems. This chapter also explores the advantages delayed consistency outside such regular IO paths by applying it to the process of producing output data on the compute nodes of petascale machines. Here, by delaying consistency computations, the total time taken by each compute node to complete its IO is no longer affected by the completion time of other nodes and/or by the control operations like the broadcast calls performed by the current implementation of HDF5. Eliminating control operations also obviates the use of precious machine resources like Bluegene's separate control network for purposes like these, and it removes associated

computations and delays from IO nodes. For instance, the current implementation of parallel HDF5 uses these resources for broadcast calls to ensure data consistency. Finally, the use of and need for such specialized hardware introduces issues with code portability. An example is the exposure of its coordination infrastructure to science codes by the MPI ADIO layer. By instead hiding these behind a simple higher level API, alternative coordination mechanisms and collective operations (e.g., delayed consistency methods) can be implemented without necessitating code changes.

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 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 [30]). In response to these needs, additional considerations were made in designing the BP file format. These considerations are detailed below.

To summarize, this chapter makes the following observations and contributions through the ADIOS middleware:

1. IO groups afford local optimization tailored to each operation rather than only globally;

2. external configuration coupled with the ADIOS architecture successfully demonstrates changability of data transport methods without requiring recompilation of the source code;

3. use of delayed consistency and other techniques for enhanced parallelism, supported by a high level IO API;

4. lightweight methods for data characterization integrated into the process of data output;

5. introduction of an new file format to store raw data and its characterizations in ways

resilient to node failures and designed for high-performance parallel IO and for maintaining compatibility for the file formats necessary for analysis workflows;

6. runtime selection of IO methods to achieve high performance for different platforms and IO patterns; and

7. the use of data characteristics and indexing for rapid data identification and retrieval to enhance scientist productivity.

2.3 Related Work

Many groups have investigated the problems of platform independent IO performance, annotated data, and auxiliary tool integration separately. For example, MPI-IO provides the ADIO Abstract-Device Interface for IO layer for different parallel file systems that is independent from the API layer, but does not address annotation or tool integration. Also, since the MPI-IO API has explicit semantics exposed to the scientific code, certain operations, like collective writes, cannot be changed without impacting the host code. HDF5 provides excellent annotation and data organization APIs, but the virtual file layer relies on MPI-IO, POSIX, or other custom libraries for supporting the actual writing to disk and does not have a concept for integration with auxiliary tools. Silo [99] provides support for VisIt with no additional support for IO performance tuning or extra data annotation beyond what was needed for VisIt. None of these tools can adequately address the need to fully bridge the gap between the science code and the storage system.

Parallel file systems universally separate metadata from storage services, to enhance parallel access. Lustre [14] provides custom APIs for configuring the striping, storage server selection, buffer sizes, and other factors likely to impact performance. However, it is still limited to a single metadata server, causing a known bottleneck. In addition, there are known expensive operations, such as 'ls -l', that cause the metadata server to talk with each storage server to calculate the sizes of the pieces of the files stored on that device. Other file systems [90, 82, 93, 74, 7] have successfully addressed the metadata bottleneck issue with vary degrees of success. One particularly awkward approach addressing the metadata server bottleneck is shown by PVFS. Each client must use multiple steps to create files [49]. The

Lightweight File Systems (LWFS) [79] project at Sandia National Laboratories has taken an extreme position on this topic by eliminating the requirement for online metadata. Offline methods are later used to generate it. In all of these cases, parallel file systems are focused on moving blocks of data with the best performance. They do not address the components of the data itself and not surprisingly, they do not provide for low-impact integration with auxiliary tools due to their specialized nature nor do they addressed the 'internal' file issues raised and solved by ADIOS, such as selective file consistency and rapid data access through data characteristics.

While offering rich tool chains, there have always been scalability challenges for the NetCDF and HDF5 APIs and file formats. For terascale machines, such challenges were successfully addressed by moving from serial to parallel APIs. In fact, in many cases, the performance of parallel HDF5 or NetCDF API has been exemplary [122]. However, additional options must be explored for petascale machines (and beyond), in part because of the high costs of the collective IO operations required for consistency enforcement. Such enforcement is necessary for providing a single, coherent view of the globally distributed file data. We believe that the delayed consistency approach can be used to incur these costs so as to not inhibit parallel program performance.

Delayed consistency has previously been studied for file systems. For example, LWFS [79] 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 [7] have distributed metadata weakening the immediate consistency across the entire network of machines. NFS [73] 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.

Efficient techniques for indexing HDF5 files are offered by FastBit [31] and the Multi-resolution bitmap indexes [101], which uses a bitmap to indicate the presence of different values or value ranges within a given data element. PyTables [106] extends this concept to

use a traditional relational database as a full content index. The use of projection tables [39] takes the approach of listing values and giving the location(s) at which it appears. These all have focused on providing full or a sampling approach data indexing. Unfortunately, these approaches have not been integrated into the base HDF5 system, perhaps because of potential performance penalties and space penalties when adding rich indices to large-scale files. Such penalties are our principal motivation for advocating simple, lightweight data characterization rather than full indexing methods in the BP file format. Our approach is not to give direct access to all data elements but rather to aid in the identification of which data sets are relevant for the desired use. For example, to know which 10 TiB file contains the data where the temperature exceeds 10^6, by looking at the maximum values for the temperature is sufficient. Simple metrics like these can always be collected and will always be available in ADIOS. Additional information must be computed in analysis workflows.

Integration with visualization systems is commonly needed. This is generally handled either through a workflow or through custom calls to the visualization engine. For example, AVS Express [103] can render data files once they have been fixed up in an appropriate format. This can easily be done through a workflow system with the impact of file discovery. In situ visualization systems may require something like VTK [5] or some other custom API calls directly in the scientific code to perform the integration. This nicely addresses the integration, but at the cost of requiring source code changes.

The file formats used by science applications range from the most popular HDF5 and NetCDF to more niche players like SAF [67], PDS [105], GRIB [38], and HDS [104]. Each of these has been optimized for a particular style of data arrangement and annotation. All of these formats share a requirement to use the same consistency validation during output as what is required by the on-disk file, necessitating the use of global consistency checks as part of the IO process (e.g., by using collectives). The use of a format like BP in ADIOS addresses this 'internal' file issue, making it possible to avoid and/or delay consistency validation to improve IO performance.

File conversion is commonly used in computer systems. Simulations frequently use converters as part of code coupling operations to 'fix-up' not just the data format on disk, but

also to perform actions like changing units and data filtering. For NetCDF files, NCO [80], the NetCDF Operators used in the climate community, provides a way to extract values from one or more files into a new file for more convenient use. The climate community uses NCO to avoid some of the issues with the NetCDF format. To provide resilience against file corruption and avoid the performance penalty of resizing the file header when writing each new history output, every output set is written to a new file. NCO is used to construct the view of a single variable over time by pulling the appropriate pieces from the set of NetCDF files into a single, new NetCDF file. The combustion simulation S3D writes each process's output into a separate file for performance and then uses another, independent process to combine all of these outputs into a single new file. This is similar to a workflow related approach using tools like Kepler, Pegasus, or DAGMan. We have also frequently used components to convert our data output into images or just strip out portions relevant to other downstream processing [84] and by using actors in Kepler workflows [62]. Alternative approaches have also been investigated. For example, through the use of an IO Graph [123], we have demonstrated the ability to write data in a different format and/or with different filtering and/or processing without the intermediate data ever hitting disk [118].

None of these examples handle all of the problems. The platform independent IO systems all provide either great performance or annotation. With careful use, systems like HDF5 and parallel NetCDF can achieve both. The parallel file systems all achieve great performance, but none give support for detailed data annotation or integrating auxiliary processing such as triggering a workflow system. Custom API integration with auxiliary tools provides tight integration, but at the cost of source code changes and revalidation when changing platforms. The performance impact of these integrations is also strictly dependent on downstream system. Loose integration with workflows addresses the need for low-impact integration superficially, but can suffer from indirect impact from file system watchers, still require manual annotation and fixup of data before further processing can happen, and are always behind the simulation due to the lag of looking for completed file writes.

2.4 Motivating Examples

The initial development of ADIOS was motivated by the GTC [43] fusion code and the Chimera [65] supernova code. GTC provides a variety of different outputs with varying frequency and sizes while Chimera offers a vastly larger number of variables output per write with some different reporting/formatting requirements. Over the life of the GTC fusion code, it has changed how it performs IO eight times, each motivated by a change of platform or a need for more data annotation. Specialized routines were added for each in situ visualization system employed. Each time a change was required, 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 and add data annotations without changing the source code.

We have further demonstrated the generality of ADIOS by integrating successfully with XGC0, XGC1 [18], FLASH, GTS [114], S3D [19], M3D-OMP, M3D-K [29], Pixie3D [16], and the Chombo [17] Adaptive Mesh Refinement framework. With the exception of the Combo AMR framework, only minor tweaks of the system required. For an AMR system, additional concepts to model the various 'levels' for a variable are required and had to be added to the system.

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

1. *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.

2. *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.

3. *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.

4. *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 MiB 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.

Each of these was motivated by specific examples in GTC and Chimera.

From an IO complexity perspective, GTC has seven different groups of output in three categories with each category being handled differently. The three categories are restarts, analysis data, and diagnostic messages. Each of these categories has different IO requirements based on their output patterns. For example, the large restart data set needs to be written as quickly as possible with a small amount of annotation. To mitigate the runtime performance impact, it is written infrequently. The analysis and particle tracking data, while much smaller, needs to be written more often with good annotation. Finally, the diagnostic messages are written very frequently, but are little more than a few kilobytes per output and always only from a single process. While there is only one output for restarts, there are multiple for the analysis and diagnostic messages. ADIOS provides the flexibility of selecting how each of these seven different data groupings perform IO simply by specifying the selected method for each of these groups in the XML file. It handles the different sizes for the analysis array outputs through the use of var names for array dimensions. Finally, by not requiring all of the vars specified in the XML to be provided by all processes writing, we can handle the optional data elements requirement.

The Chimera supernova code provides very different requirements. It writes three main groups of data. The first is a set of around 400 different key values. Each of these variables

has annotation data associated with it. The second is a set of around 75 model variables. These have fewer annotations associated with them. Third is a diagnostic report output that is a slight superset of the model that has been processed and formatted as a report. All three of these have the same output frequency. Like GTC, some characteristics of the output drove the five requirements above. In particular, many of the array sizes were driven from calculations within the code requiring that all sizing for writing be done at runtime.

For both of these applications as well as any code precisely tweaked to run using the maximum resources on the compute platform, memory is at a premium. To ensure we do not break the trust with the users that the IO system will be well behaved, we instituted a contract in the XML for the maximum amount of memory all IO through ADIOS in the system will use for buffer space. By always managing to this and having a failure mode when no more buffer space is available, we are able to meet the specifications of the user reducing unwanted surprises. We address this in the POSIX IO and MPI-IO transport methods by switching from a buffering mode where we maximize the write block sizes to a direct writing mode. We do output a message indicating that we overflowed the internal buffer allocation, but fail gracefully by using the lower performance option of directly writing items to disk rather than aborting the code. This feedback alerts the user to the problem without causing a loss of the run.

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.

2.5 Software Architecture

This architecture section is divided into three parts: the ADIOS architecture and API layer, the BP file format layer, and the external XML file. Using these in concert, the maximum benefits of this middleware approach are realized.

2.5.1 Architecture of ADIOS Layer

At a high level, ADIOS structurally looks like Figure 2. 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 auxiliary integrations. For example, MPI-IO, POSIX IO, and Kepler or VisIt integration.

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

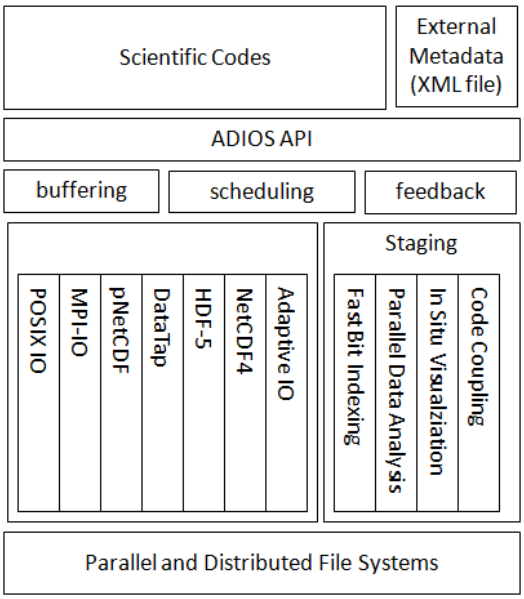


Figure 2: ADIOS Architecture

The ADIOS layer show in Figure 2 has several important characteristics that help us attain portable and scalable implementations of delayed consistency, lightweight data characterization, and output resilience. ADIOS hides all consistency-related functions of lower level APIs by moving them into metadata specifications located in an external XML file. The XML file specifies for each IO group the IO method to use affording flexibly for IO methods to best match the IO patterns of certain code output actions (e.g., restart files vs. intermediate results) and local machine performance characteristics. IO methods also include the aforementioned lightweight data characterization where the simple characteristics of min/max are computed immediately during output. Other work [123] has examined alternative methods that place such computations at different 'locations' in or outside the

IO path. Finally, the BP file format's current implementation is designed to maximize parallel output performance while also containing sufficient information to (later) validate data consistency.

The key technical features of ADIOS are the following:

1. *single API for all IO transport methods* – no matter how IO is actually performed;
2. *external XML file* – for IO description and configuration;
3. *runtime selection of potentially different IO methods* – per grouping of data; and
4. *BP file format* – designed for minimal required coordination, compact metadata storage, and resilience in cases of failures;

2.5.1.1 Single API for all IO Methods

Since all descriptive elements of the output operation have been moved to the XML file, the ADIOS API is nearly as simple as POSIX IO, and in many cases, even simpler. For example, even to have a richly annotated HDF5 file with many attributes and with hierarchical structure, the user need only have an '`open`', '`#include`', and '`close`' statement in the science code. The '`#include`' will insert the generated ADIOS IO calls based on the XML file for use in the science code. For more complex cases where very fine manipulation of the IO operations is required, it is still possible to have an '`open`' followed by a series of '`read`' or '`write`' statements, and finally, a '`close`' statement. The only additional requirement in this case is our '`group_size`' call immediately after the '`open`' call. This provides a way for the underlying API to decide if it should buffer the output for optimal performance, coordinate with other processes participating in the output to determine the local offset in the global file for output, and to initialize any output parameters not set during the '`open`' call.

2.5.1.2 External XML file

The complexity of IO methods is often directly reflected in that of the source code API required for their use. To avoid exposing such complexity to source codes, the ADIOS

XML format describes the structure of output data, any attributes attached to items or groups, and the selection of the particular IO method to employ for this run of the science code. Through the use of this XML file, ADIOS controls what data is written and which method is used for each grouping of data in the code. Since this introduces a consistency requirement between the XML file and the source code, we have developed a method for automatically generating nearly all of the code's IO commands based on the XML file. This eliminates the need to maintain a set of calls in the source code and a set of data descriptions in the external XML file. Using this feature reduces the API calls in the science code to just an '`open`', '`close`', and an include of the generated write or read statements. By properly setting dependencies in the project Makefile, it is easy to automatically generate new include files and properly recompile dependent source files when the XML file changes. This has shown to be effective for nearly all of the IO examples we have encountered in the petascale codes targeted by our work.

2.5.1.3 Runtime Selection of Potentially Different IO Methods

Our hope is that ADIOS can be used to attain high IO performance no matter on which platform a code is deployed. To achieve this, it is necessary to be able to select which IO method to employ for a science code. This is particularly important with limited or costly time allocations on petascale machines, where excessive IO times can substantially reduce scientific productivity. The need for selectivity also extends to individual IO groupings within the code, where for instance, since diagnostic output is likely small but frequently written, it makes sense to write it using one approach (e.g., HDF5), while restarts that are large and infrequent may need to use a different approach (e.g., MPI-IO) to gain the performance benefit of not performing the consistency checks during the large-scale run.

2.5.1.4 The BP File Format

The BP file format assists IO performance, for both write and read activities. By using a structure that minimizes the requirement for runtime coordination and data sharing, requiring each process's output to be self-consistent and complete, and incorporating features for characterizing data and assisting in recovery from failures, the BP format is suitable

for layering high performance, scalable IO methods on top. With only the global index and requiring coordination among output processes, the forced global communication is minimized while maintaining maximal functionality.

2.5.2 The ADIOS API

Since scientific codes are written in both Fortran and C-style languages, the ADIOS API supports both calling structures. 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 A. 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.

2.5.2.1 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", comm)
...
// 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 initialization and cleanup operations. The included `comm` is provided as a mechanism to broadcast the contents of the XML file to all processes that participate in some output. 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 a 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, increasing the overall runtime. `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 so it can ensure the current operation is completed in a timely manner, but without necessarily trying to output everything as quickly as possible.

2.5.2.2 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 closing the operation. Because ADIOS relies heavily on buffered IO, writing and reading are performed during different open/close pairs. For example, to read a value just written, it is necessary to close the file to flush the buffer, open it again, and then read. Otherwise, the value will not necessarily be available. Future enhancements to enable a more dynamic read/write file mode are not prohibited by the BP format nor the ADIOS API implementation itself.

```
adios_open (&handle, "restart", "filename", mode, comm)
adios_group_size (handle, group_size, \&totalsize)
adios_write (handle, "zion", zion)
...
adios_write (handle, "mzeta", mzeta)
```

```
...
adios_close (handle)
```

`Adios_open`, `adios_write`, and `adios_close` all work as expected. The string second parameter to `adios_write` specifies which var in the XML the provided data represents. The `adios_group_size` is a mechanism by which each process tells the ADIOS layer the maximum amount of data it will generate so that ADIOS can coordinate file offsets and determine if adequate buffer space is available. By providing this information, only one coordiation message phase is required no matter the number of variabiles being written. Without this information, it is still possible to gain the delayed consistency advantages, but only if sufficient buffer space is available or sufficiently large gaps are assumed by each process leading to potentially large amounts of 'empty' space in the out file. Due to this added complexity, the `adios_group_size` call will be required for the foreseable future.

2.5.2.3 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.

2.5.2.4 Transport Methods

A variety of transport methods have been developed and work, without modification using the standard ADIOS API. These transport methods include a variety of synchronous MPI-IO methods with different tuning parameters for various file systems and output patterns, two POSIX IO methods with one for a single file output and the other for a one-file-per-process output, DataTap [34] asynchronous IO, Parallel HDF5, NetCDF version 4, and a NULL method for no output, which is useful for benchmarking the performance of the code

without any or selectively less IO. 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.

2.5.3 The BP File Format Architecture

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.

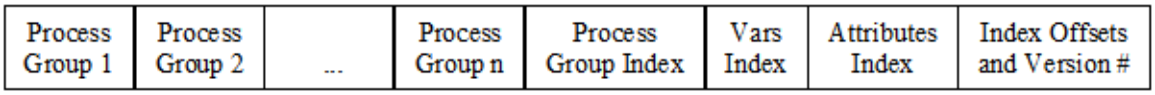


Figure 3: BP File Layout

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, padding these slots to file system whole stripe sizes and other size adjustments are possible because of the abstracted IO interface. This flexibility will be required to get the best possible performance from an underlying 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 MiB. Given the total size of the data from an output operation of this size, 10 MiB 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.

The RICCI properties of BP ADIOS uses the BP format as a default since its design is central to our ability to achieve high performance, compatibility with standard formats, and efficient data annotation and characterization. This section will briefly describes the key features of BP, based on five goals for a high performance file format, termed RICCI:

1. *Resilient* in the presence of a variety of failures;
2. *Independent*, parallel IO with sufficient annotation to validate and enforce consistency later;
3. *Convertible* to both HDF5 and NetCDF;
4. *Characterized* data for easier data analysis and selection; and

5. *Indexed* metadata and characteristics for all of the data for fast, direct access.

2.5.3.1 Resilient

To avoid the loss of previously written data during a later IO operation to the same file(s), BP incorporates three key features. First, there is a local copy of all relevant metadata for each process that writes output. Thus, BP does not rely on a centralized header area for access to local data elements. Second, since each process writes its output into the file independently, its failure does not affect other processes and/or the ability to read other file portions. Third, the BP file index contains replicated metadata to deal with data failures. By storing where each process group resides in the file in the index, the BP format affords proper identification of undamaged sections of the file by indicating where to start parsing. This also holds true for variables as we replicate the location of each in the index as well.

2.5.3.2 Independent

As mentioned in the previous section, BP stores each process's output independently. This achieves two advantages. First, by storing the full output of each process independently with full annotation, the BP format can delay consistency checking outside the IO fast path. This can be done as a stand-alone operation or as part of converting to other formats. Second, the lack of coordination among the processes for each piece of the output eliminates any intermediate synchronization points during the IO operation. The BP format only requires one coordination operation at the start to decide on file offsets to write in parallel and once at the end to collect index information to process 0 to append on the end of the file. This general lack of coordination during the writing process affords each process writing in larger blocks with a slow storage node only affecting the output once at the end by delaying the ultimate completion of the output of the index. The total IO time becomes the longest time overall for any individual process rather than the sum of the longest time for each output element.

2.5.3.3 Convertible

Given our own investment in the use of tools requiring HDF5 and/or NetCDF, BP provides all of the features we have encountered 'in the wild' by users of HDF5 and NetCDF. There are esoteric features of these standards we have chosen to not address in ADIOS because we have not encountered a science code that required the feature. As the codes adopting ADIOS increase, the BP format may evolve to meet these additional needs.

2.5.3.4 Characterized

With simulation sizes growing as the machines grow, data is growing commensurately. File sizes today already are in the 10s of GiB or larger with multi-TiB file sizes becoming more common as use of petascale machines increases. ADIOS characterizes the data as it is written to aid later selection of file(s) for processing. For example, when trying to figure out which set of data output during a materials simulation run is the one containing the interesting feature being investigated, selecting the file where the number of material pieces grows by more than 10% from the previous time step would be sufficient. We collect these characteristics and store them both with the process output and in the index to preserve resilience and to aid in rapid selection and access. While this information is not directly convertible to HDF5 or NetCDF except as attributes, we feel automatically collecting this information increases the value of the output and the ease of selecting the proper potentially multi-TiB file to process with analysis tools.

2.5.3.5 Indexed

The BP format is a collection of independent process outputs rather than a single organization of data that originated from 1 or more processes. This is not to say that there is not a universal view of the file contents available. Through the index, an HDF5 or NetCDF-style header output can be constructed without having to parse each of the process outputs. This affords the advantage of highly parallel, independent IO while maintaining a global view of the file contents. The placement of the index was driven by performance and resilience requirements. If the index were at the beginning of the file, if it expands too much, moving

data would be required. This performance impact along with the potential for file corruption when moving data dictated that the index be placed at the end of the file. This has the added benefit of making it easier to make the process group outputs into file system stripe sized 'chunks' for optimal write performance since it is not necessary to waste potentially 4 MiB of space at the front of the file to store the index information just to keep stripe-sized 'chunks' per process. Since the BP index is small, on the order of 100 bytes per process output and 100 bytes per unique variable or attribute written and generally at most another 50 bytes per instance in the file, filling a stripe width with the header information is difficult. The index itself stores for each process group the process ID, the ADIOS group name written, the offset from the beginning of the file, and the time-index value for this output, if any. The variable and attributes are each indexed separately, but contain essentially the same information. Each index stores a unique set of variables or attributes across all process groups written with a list of characteristics for each. These characteristics include the offset from the beginning for the file for each place it is written in the file, the array dimensions and minimum and maximum values if it is an array, and the scalar value if it is a simple value. This provides direct information of where each portion of any array is written for any timestep as well as characterizing the data for direct evaluation of whether or not the data is likely what the user is interested in analyzing. These indices can easily be separated from the BP file as a separate file for use on a tape storage system or portable file to identify the full contents of the data file it represents. For rapid retrieval, the last few bytes of the file store a version identifier and the offset at which the index is stored.

2.5.4 XML Format

Since the XML controls how everything else works, we will discuss it last. The XML file 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 ...>`. The details of the XML is more fully discussed in Appendix A. The description below is structured like the XML document:

```
<adios-config host-language>
  <adios-group name time-index>
    <global-bounds dimensions offset>
      <var name path type dimensions/>
    </global-bounds>
    <var name path type dimensions/>
    <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>
```

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.

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

- `host-language` - either `Fortran` or `C` to specify the multi-dimensional array element ordering (row-major vs. column-major) for reading properly.
- `time-index` - an implicit variable that increments each time the adios-group is written. This can be used as an array dimension without declaring it elsewhere.
- `path` - HDF5-style path for the element or path to the HDF5 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
- `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.

MxN communication is implicit in the XML file through the use of the `global-bounds`. Which communication mechanism (e.g., MPI, OpenMP, or something else) 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.

2.5.4.1 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 `adios-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 `transport` 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. Each transport will be invoked in order. During the close operation for the writing transport method, all of the data is pushed to disk. Once that completes, the close operation for the next transport, the workflow trigger, is called. At that point, the workflow could be notified that new data is available for processing. This functionality is enabled without recompiling, relinking, or any other code changes.

The impact of these decisions on the reading performance is discussed in chapter 4.

2.6 Experimental Evaluation

Technical evaluations demonstrate the following. First, we discuss our practical experiences with the Chimera supernova code and its IO. We also identify the reasons for the existence of orders of magnitude differences in performance when using alternative methods for IO. Second, we demonstrate the lightweight nature of data characterization, by comparing the times taken to collect base data characteristics against those experienced by external indexing schemes and by a full data scan from a local disk. Finally, we discuss how the BP file format and the ADIOS API jointly achieve resilience to failures.

Evaluations are performed on two different machines: (1) Jaguar, the Cray XT4 system at Oak Ridge National Laboratory, and (2) Ewok, the Infiniband and Linux based end to end cluster at ORNL. Jaguar consists of 7832 compute nodes plus additional login and IO nodes. Each compute node contains a quad core AMD 2.1 GHz Opteron with 8 GiB of memory. The login and IO nodes consist of dual core 2.6 GHz Opteron with 8 GiB of memory. The system is running Compute Node Linux. We used various counts of compute nodes for our

simulations all writing to the 600 TiB Lustre scratch system. For our conversion tests, we ran on a single Jaguar login node against the same Lustre scratch system. Ewok consists of 81 dual core 3.4 GHz Pentium IV with 6 GiB of memory. It is configured with a 20 TiB Lustre scratch space we used for our tests.

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.

2.6.1 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 specifies the largest size the local process will write facilitating pre-calculating offsets in the output file. 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.

An additional option that simplifies the source code even more replaces all of the calls with a single preprocessor string that expands into all of the proper calls. 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 will handle 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. This feat is accomplished through the use of a python 'compiler' for the XML that generates either C or Fortran include files containing all of the necessary calls.

2.6.2 Fast IO

The performance evaluations are performed using the GTC fusion code and the Chimera supernova code. 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.

2.6.3 Chimera Evaluation

The Chimera evaluation has three parts. The first examines the relative performance of the Chimera code with parallel HDF5 compared to various ADIOS-based IO methods. The second evaluates parallel HDF5 performance using independent MPI-IO and compares it against ADIOS using independent MPI-IO. The third uses a sample BP file generated by Chimera to assess the cost of converting it to an HDF5 file that is identical to the one previously generated by Chimera.

Chimera output is not particularly large. In a weak scaling model, each process outputs approximately 920KiB, with the number of processors varying from 512 to 8192. The simulation is configured to run for 400 iterations with an output being performed every 50 iterations. The simulation is run 5 times at each size, collecting the timing for each output, for a total of 40 measurements per size per IO method. Graphs depict the best performance measured for each size for each method with an error bar to show the range of values seen for that size. Best times are shown in order to minimize the impacts other users of the machine have on measurements.

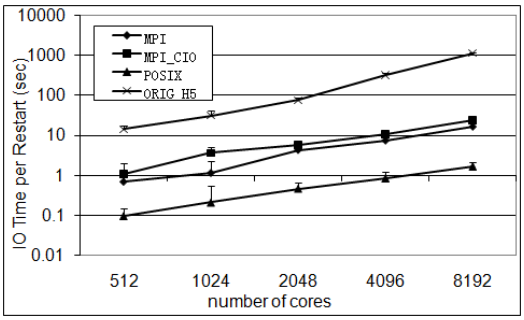


Figure 4: Chimera Weak Scaling

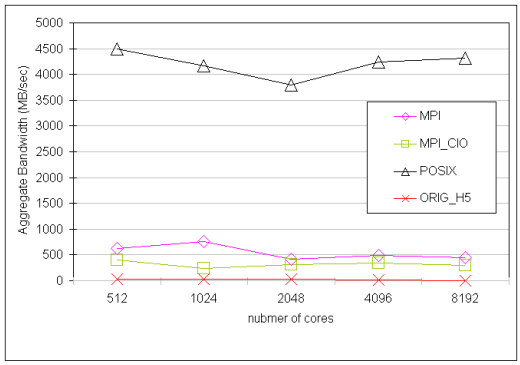


Figure 5: Chimera Aggregate Bandwidth

Important to note about Figure 4 are the facts that the vertical axis is exponential and that the rate of growth in time for parallel HDF5 increases at 2048 processors. Figure 5 notes the bandwidths to the storage system achieved with different IO methods. Clearly, these results demonstrate performance issues with HDF5. We next investigate their principal causes.

2.6.3.1 Principal Causes of Overhead in Parallel HDF5

Using a test case of 512 cores running the Chimera supernova code, 5 sets of restart dumps are used to analyze performance causes, with results appearing in Table 1. Detailed profiling reveals the reasons for inadequate Parallel HDF5 performance:

1. Expensive MPI_Bcast calls are called frequently. In the test case, MPI_Bcast is called 314,800 times with a total wall clock time cost of 12,259 seconds total across all 512 processes (mean of 23.9 seconds for each process overall, max of 54 seconds, min of 4.6 seconds).

2. Too many small, individual writes are performed. Individual write operations are performed 144,065 times with a total wall clock time of 33,109 seconds across all 512 processes (mean of 64.7 seconds for each process overall, max of 96 seconds per write, min of 46 seconds per write).

3. MPI_File_open calls take longer than necessary because it has not been optimized how the MPI_File_open calls are performed, so that the 2560 calls take a total of 325

Table 1: Parallel HDF5

Parallel HDF5		
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

seconds across all 512 processes (mean of 0.63 seconds).

2.6.3.2 Performance Analysis of ADIOS with BP format using Independent IO

For a test case of 512 cores running the Chimera supernova code, this evaluation uses 5 sets of restart dumps, with results appearing in Table 2. In comparison to the parallel HDF5 results shown above, this ADIOS run has a straightforward outcome:

1. Buffered writes take the longest time, since ADIOS, by default if memory is available, buffers all writes to the output file locally and then writes the buffered output in a single write operation to disk. MPI_File_write is called 2560 times with a total wall clock time of 2,218 seconds across all 512 processes (mean of 4.3 seconds, max of 11 seconds per write, min of 0.01 seconds, likely due to cache effects).

2. By coordinating the MPI_File_open calls, the total time to open the file is reduced. Specifically, rather than have all processes call MPI_File_open at the same time, a coordination token is used, in a round robin fashion, to reduce the load on the meta-data server. This reduces the time for the 2560 MPI_File_open calls to 95.80 seconds (mean 0.19 seconds). Including the time for the token passing, the total time for the MPI_File_open and MPI_Recv calls is still only 120.48 seconds across all 512 processes

Table 3: BP to HDF5 File Conversion on 1 processor

Job Size (cores)	*File Size (bytes)*	*Total Conversion Time (sec)*	*Speed (MiB/sec)*
512	467614208	5.361044	83.18
1024	935621632	10.8131	82.51
2048	1872816128	20.725636	86.17
4096	3748777984	48.226855	74.13
8192	7503527936	117.0	61.16

(mean of 0.23 seconds).

The performance impact of collecting the data characteristics is negligible. For all of the tests, the amount of time spent collecting the metrics locally on each compute process is lost in the IO variability.

2.6.3.3 File Conversion Performance

Since the default implementation of the ADIOS MPI-IO, POSIX, and collective MPI-IO methods use the BP format, the time spent converting this format into the HDF5 desired by the Chimera scientists is extremely relevant. Tests are performed on a single Jaguar login node reading from the BP file and writing to an HDF5 file, both stored on the Lustre scratch space.

Five examples are profiled for file conversion performance, described in Table 3. For the 2048 process case, Chimera generates a file of approximately 1.8 GiB. This conversion takes about 20 seconds. For a larger example from a run of 8192 cores, the file generated is about 7 GiB and is converted to HDF5 in 117 seconds. Note that the native Parallel HDF5 calls in Chimera take 1400 seconds to write *each* output while the MPI-IO independent method takes only 10 seconds. Thus, even when combined with the 127 second conversion time, this is a greater than 90% savings in IO time.

Regarding conversion, it is possible to completely 'hide' its costs by automatically performing it either 'in transit' or once data is stored on disks. The evaluation of such approaches is beyond the scope of this thesis.

2.6.4 GTC Evaluation

The GTC evaluation is performed in two phases. The first phase focuses on the raw performance attained over a series of runs over time to demonstrate the consistency of the high performance output. The second phase examines the particle output.

The the first phase tests are run in two sets. The first set of phase one is performed on the Jaguar machine at ORNL. The system is a Cray XT4, dual core AMD x64 chips with 2 GiB of RAM per core and around 40-45 GiB/sec peak IO bandwidth to a dedicated Lustre parallel file system. Our tests showed a consistent average aggregate write performance of 20 GiB/sec for a 128 node job [87]. See Figure 6.

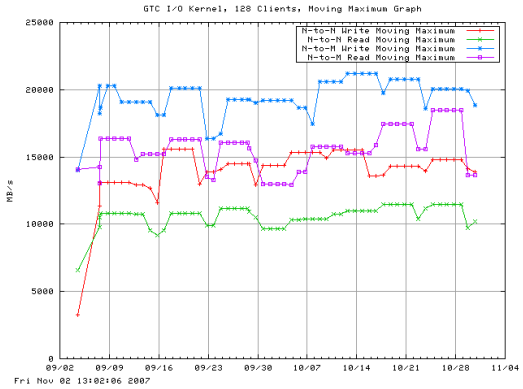


Figure 6: GTC on Jaguar with ADIOS

The second set of tests for phase one are performed on the ewok system at ORNL. This is an Infiniband cluster of 81 nodes of dual core AMD x64 chips, 2 GiB of RAM per core, and about 3 GiB/sec peak IO bandwidth to a Lustre file system shared with several other clusters. Two sets of 5 runs for GTC on 128 cores are performed. Each run generates 23 restart outputs for a total of 74.5 GiB. The first set is configured to generate output using the MPI synchronous transport method while the second set is configured to generate no output using the NULL method. We are able to demonstrate an average 0.46 GiB/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. This variability is addressed in Chapter 3.

The second phase of tests examine the restart and analysis output from GTC. The GTC

code has two large output operations that both occur when restarts are written. The first is the normal state output to enable a restart. The second is a set of particles used for analysis. These particles are tracked as they rotate around the simulation toroid and yield some of the scientific data important as the output of the run. We configure GTC to run for 100 iterations with a restart/particles output every 10 iterations. Tests are run 5 times and again, the 'best' results are shown from the 50 outputs. The evaluation is performed for each of POSIX IO, independent MPI-IO, and collective MPI-IO.

For weak scaling tests, we use the Jaguar machine and run with OpenMP to communicate among the cores within a single node (4 cores per node). Strong scaling tests are run on the Ewok machine, without OpenMP.

This second phase GTC evaluations are divided into two parts. The first evaluates weak scaling with various ADIOS IO routines. The second evaluate the performance of GTC with strong scaling using various ADIOS methods for comparison. We provide these to demonstrate the applicability of ADIOS to a range of HPC applications beyond Chimera.

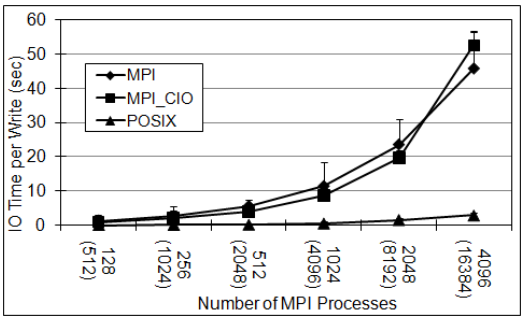


Figure 7: GTC Particles Weak Scaling Time

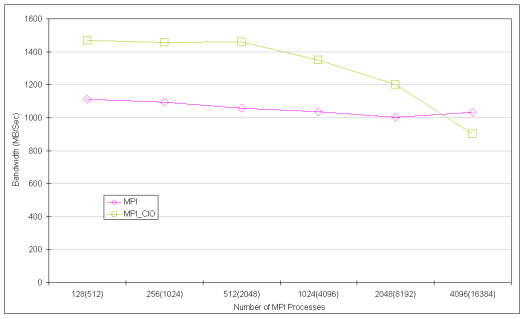


Figure 8: GTC Particles Weak Scaling Aggregate Bandwidth MPI

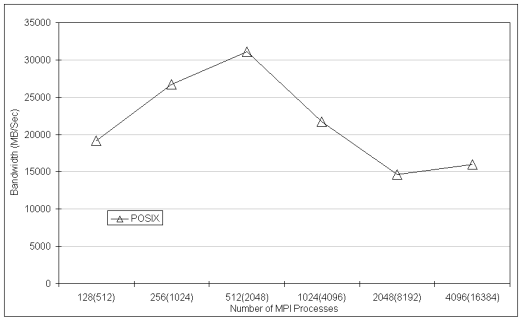


Figure 9: GTC Particles Weak Scaling Aggregate Bandwidth POSIX

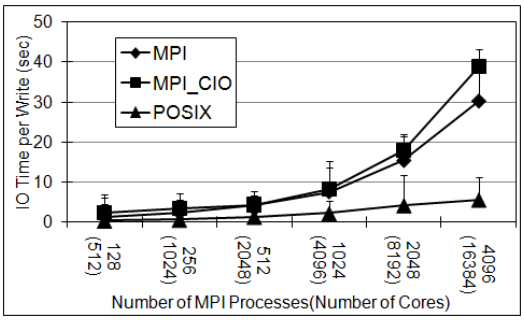


Figure 10: GTC Restarts Weak Scaling Time

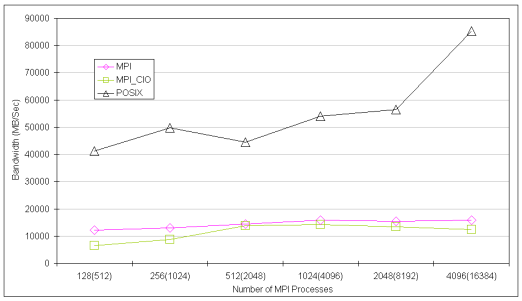


Figure 11: GTC Restarts Weak Scaling Aggregate Bandwidth

2.6.5 GTC Weak Scaling

The GTC configuration outputs particles of the size 11.5 MiB per MPI process. Since OpenMP is used, this is the aggregate for the four cores on that particular node. For the restart output, the output of each MPI process is 116.5 MiB.

For the particle output, shown in Figures 7, 8, and 9, the POSIX output of one file per process is still considerably faster than the other approaches, the 'cost' of that approach being the large number of resulting files on disk. The more interesting result is in the bandwidth measurements. Collective MPI-IO stays about the same margin better than independent MPI-IO until about 1024 MPI processes. At 2048 processes, the margin is reduced considerably. At 4096 processes, the bandwidth is reduced below that of independent MPI-IO. This further emphasizes the observations in the Chimera runs that the coordination required for collective-style IO does not adequately scale.

For the restart output, in Figures 10 and 11, the performance of output methods using collective MPI-IO is initially worse than that of the independent MPI-IO output, and it continues to degrade with increasing simulation sizes. While the particle data is relatively small at 11.5 MiB per process, the restart data, at an order of magnitude larger, clearly demonstrates differences in IO performance. For such output, it would never be appropriate to use the collective MPI-IO method, instead favoring the independent IO method or POSIX methods, if it is possible to cope with the large number of files they create.

As shown above, both the particle and restart data differ in size and show different performance characteristics depending on the size of the run. This demonstrates the need for differentiating IO methods both by run and by output grouping. ADIOS enables such differentiation.

2.6.6 GTC Strong Scaling

Strong scaling results are attained on the Ewok end to end cluster at ORNL. Interesting insights are attained despite the cluster's relatively small size.

Test runs with small data sizes are not meaningful due to caching effects, so discussion is focused on larger data runs on the largest number of processors. The results of these

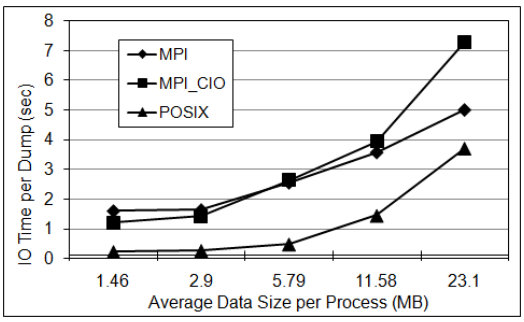


Figure 12: GTC Particles Strong Scaling Time

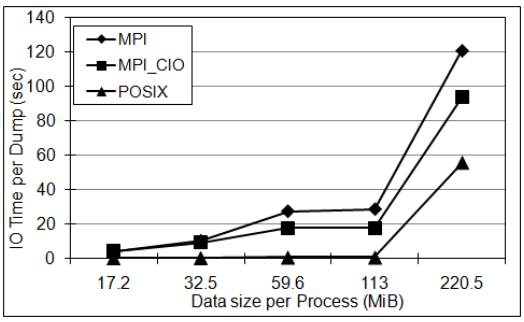


Figure 13: GTC Restarts Strong Scaling Time

evaluations are shown for particles in Figure 12 and for restarts in Figure 13. The most important characteristic of these results is how relative performance differs between Jaguar and Ewok for the independent and collective MPI-IO calls. Although both of these machines are housed at the same location and maintained and configured by the same staff, their different architectures yield the opposite performance results of what we saw for the particles and restarts. When running on Ewok, or by extension probably other Infiniband-based Linux clusters, using collective MPI-IO for restarts and independent MPI-IO for particles is the better configuration, at least at these scales. This further emphasizes the need to have configurable IO as part of scientific simulations in order to achieve best IO performance. ADIOS offers this capability, per IO grouping, by simply changing a single entry in the XML file when the job is submitted.

2.6.7 Evaluation Discussion

The performance advantages of the ADIOS approach are shown by measuring the performance of the native parallel HDF5 output from the Chimera supernova code compared

against various methods integrated with ADIOS. The overheads involved in parallel HDF5 vs. independent MPI-IO are identified and compared with ADIOS performing independent MPI-IO. We show that the coordination and small writes required by HDF5 for file consistency degrade performance by as much as three orders of magnitude compared to other methods. With a total time of 1400 for the real-time consistency output directly to HDF5 vs. a net total time of less than 120 seconds to output data in HDF5 via the BP file format using a delayed consistency method and conversion, it is hard to justify the wall clock expense during a production simulation run to perform real-time consistency validation.

The GTC fusion code is evaluated with both weak and strong scaling for two different outputs performed at the same time. We show that as the size of the simulation run increases and based on data size, it is appropriate to use different IO methods. By also running tests on the small end to end Ewok cluster, performance differences and resulting changes in recommended IO methods are demonstrated.

Both of these results demonstrate the need for using delayed file consistency, with rapid conversion to a consistent HDF5 and/or NetCDF file, and with configurable IO as provided by ADIOS. As platforms change or for different sizes of simulation run, the selection of the IO method for each grouping of data within the simulation is critical for minimizing the time spent in IO.

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.

2.7 Conclusion and Future Work

The middleware approach demonstrated by ADIOS in this chapter achieves the simplicity and flexibility goals for extreme scale data management in HPC. The ability, transparent to the host science code, to change the underlying data transport method requiring neither a source code change nor recompilation demonstrates the flexibility. The performance demonstrated in this chapter for both the small sized, large count of variables case of Chimera and the large sized, small count of variables case of GTC both show the efficacy of this middleware approach. For all of the tests, different transport methods were selected and the code was rerun. Given the new file format, concerns about the cost of conversion to more common file formats were also addressed. The small linear time conversion cost for a single process is considerably less than the savings compared with writing in HDF5 directly from the simulation. Parallel conversion programs would be a nice addition to improve the performance further.

The first major issue revealed as part of this chapter is the existence of variability in the IO time. The magnitude of this impact warrants additional investigation to manage this effect. This is performed and discussed in the next chapter. The second issue is the potential impact on read performance, particularly for various analysis tasks. These evaluations are performed in Chapter 4.

At a more detailed level this chapter has shown the following. ADIOS provides a flexible approach for IO within scientific codes while affording an opportunity to integrate cleanly and transparently with auxiliary tools such as workflow and visualization systems. Revisiting the four goals from the chapter introduction, we can conclude for each as follows. First, different IO routines have been optimized for different machine architectures and configurations. No single set of routines can give optimal performance on all different hardware

and storage platform combinations. The choice of a middleware approach affords the transparent use of optimize IO techniques. Second, while richly annotated data is desired, the complexity of writing the code to manage the data creation can be daunting. The API must be simple enough to be easily written yet provide facilities to generate richly annotated data with little extra effort for the developer. Our choice to have a simple, consistent API for the source code and a richly annotated XML file that describes and annotates the data and controls the IO methods selected achieves the simplicity in programming. Third, once the code is stable, no source code changes should be required to support different IO routines for a different platform or IO system. We have achieved this by supporting simply changing the method entry in the XML file. Forth, adding additional IO integrations such as workflow or in situ visualization routines should be transparent to the source code and facilitated via low-impact system approaches. Simply by adding another method entry to the XML file, a triggering message can be sent to a passive workflow system avoiding the unintentional slowdown caused by "passively" watching for files to appear.

This chapter demonstrates the utility of five basic ideas. First, the idea of a separate grouping for each output operation affords selecting different approaches, each with local optimizations. Since the transport method is selectable at runtime for each IO group, a single API can be used in the code without hurting performace for areas where a particular API may be inefficient. Second, the external configuration affords selection of the various transport methods at runtime without requiring source code changes. Additionally, attributes can be added to the XML file to better annotate the generated data that will be incorporated into the output the next time the science application is executed. Third, it shows that the use of delayed consistency methods applied to the internals of large-scale files can result in up to three orders of magnitude performance improvements in IO on petascale machines. Forth, by providing 'nearly free' data characterization as part of the base API, common questions like when a value or array reaches some threshold value can be answered without analyzing all of the output data. This aids both (1) in selecting which data to analyze from potentially slow storage like a tape library and also (2) in preserving the use of expensive analysis resources for the data most relevant to the scientific question at hand.

Finally, resilience is achieved by replicating metadata to all process outputs. By using a footer index with replicated data from the process groups rather than a header, append operations are facilitated and we avoid relying on centralized metadata for file correctness.

The middleware approach has advantages over other alternatives for abstracting the IO techniques. For example, using conditional compilation directives can give access to alternative IO approaches. By avoiding using a middleware abstraction layer, the user is exposed to the penalities of any change to the IO contents requires changing multiple locations in the code and retesting of each. By using the middleware approach with the external XML configuration file, these sorts of changes become a matter of editing the XML file, recompiling, and running again. The various semantics of the different IO methods are hidden and any related errors in constructing the proper syntax for adding the new variable are reduced. This is especially true for forgetting to add the new variable to a seldom used IO method. The overhead ADIOS introduces is two function calls per ADIOS call. The `adios_write` call first invokes the language specific (Fortran or C-style) interface. This function does any necessary parameter adjustments and invokes a common implementation function. The common function may do a little work that will be shared across all transport methods, but it generally just invokes the transport method implementation for all of the actual work such as encoding and buffering. As a side benefit, when a new transport method becomes available, no changes are necessary to the code to see how it impacts the IO performance of a code. Simply change the XML file to invoke the new method and run some test cases.

In conclusion, ADIOS provides simple APIs for performing IO, proven routines for achieving fast IO, and the flexibility to add workflows, visualization, and other auxiliary tools transparently and with low impact to scientific codes. ADIOS provides a platform for simplifying efficient IO coding for scientists, while affording interesting opportunities to provide value-add features, both without disturbing existing simulation codes. An XML file used for specifying configuration options (and additional information) makes it easy for end users to take advantage of different IO functionalities underlying the ADIOS API, including asynchronous IO options. With asynchronous IO, the IO costs can be reduced

for certain HPC applications, and with ADIOS's configuration options, traditional methods can be used elsewhere. ADIOS also makes it easier to integrate programs' IO actions with other backend systems, such as Kepler [61] and VisIt [112], with low-impact approaches. We have demonstrated the viability of the approach by fully integrating with seven major scientific codes using different IO techniques with direct integration into two visualization systems. Our excellent performance results reinforce the viability of this approach.

For petascale machines, the performance penalties of using full internal file consistency during a production run can be too onerous. Through ADIOS, a developer can debug a code using an underlying API with active consistency checks, like parallel HDF5 or parallel NetCDF, but during a production run, one can switch to whichever IO method gives the best performance, thus yielding the 'most science' with the least IO overhead. In some cases, this will be Parallel HDF5 or Parallel NetCDF. In other cases, through the use of a format like BP, it is possible to deliver excellent performance while still maintaining sufficient metadata for easy conversion to the file format compatible with the science workflow already employed. Further, ADIOS' additional feature of data characteristics can aid in data selection on the large output sets.

We have demonstrated that in some, if not many cases, the use of alternative IO methods can yield dramatically better IO performance during production runs while still maintaining format compatibility via relatively cheap methods for file conversion. We conclude therefore, that to attain high IO performance on petascale machines, it is imperative to be able to configure the IO method employed for each IO grouping within a code differently, at runtime and without any source code changes. ADIOS provides such functionality.

We have demonstrated a simple API and can still generate annotated data with fast IO performance while transparently integrating with workflow or visualization. Without changing the source code, we can then turn off all or any portion of the IO for a baseline run to cleanly collect baseline IO performance metrics.

CHAPTER III

ADAPTIVE TECHNIQUES FOR MANAGING EXTREME SCALE, SHARED FILE SYSTEMS

3.1 Introduction

In the previous chapter, the ADIOS middleware was demonstrated to achieve excellent performance, provide a file format that is easily convertable to other standard formats while offering a scalable data organization, added data characteristics for rapid idenitification of data essentially for free in terms of time and space, and offered the flexibility to change how IO is performed without source code changes making moving extreme scale science codes to new platforms easier by avoiding the necessity of changing the code to perform IO should the new platform demand it.

In this chapter, the writing portion of the write-read cycle is further examined to address one of the glaring issues identified in the previous chapter–the issue of variability in IO performance. It further examines how to work around the limitations of the file system in order to achieve nearly full aggregate bandwidth performance as often as possible, no matter the system conditions.

3.2 Overview

To meet the performance demands of petascale applications and science, HPC file systems continue to grow in both extent and capacity. For example, the new file system at Oak Ridge National Laboratory supporting the petascale Jaguar machine has 672 individual storage targets (OSTs) and over 10 petabytes of storage. Storage targets can be used in parallel, resulting in a theoretical peak of generally around 60 GiB/sec aggregate performance (as much as 90 GiB/sec with optimal network organization) and it is clear that such performance levels are needed when up to 225,000 compute cores can concurrently generate output. Additional performance requirements are due to file system sharing across multiple

machines, as is the case at both ORNL and NERSC, where IO systems are used simultaneously by petascale machine applications that generate output data and by analysis or visualization codes that consume it.

Extensive prior work is focused on the performance of shared file systems used by enterprise applications that generate rich and varying mixes of read/write accesses to large numbers of files. Topics range from driver-level work on efficient algorithms for disk access to system-level strategies for effective buffering to alternative file organizations [89] used in file systems to diverse methods for content distribution across multiple OSTs and/or machines such as file striping, etc. The parallel file systems used at ORNL, NERSC, and other supercomputing sites, in fact, use many of the sophisticated techniques developed in such research. In addition, HPC researchers have developed novel methods in support of high performance IO, which include data staging [3, 76], the use of alternative file formats or organizations [11, 59, 50, 51], and better ways to organize and update file metadata [79, 27, 37, 120, 52].

Despite their use of state of the art approaches like those described above, the large parallel file system installations at sites like ORNL or NERSC continue to face significant challenges when they are used 'at scale'. This is due to several facts. First, in contrast to most enterprise applications, an HPC application can demand instantaneous and sole access to a large fraction of the parallel file system's resources. An example is a petascale code that outputs restart data. If IO resources are insufficient, this code will block and waste CPU cycles on compute nodes waiting for output rather than making positive progress for the ongoing scientific simulation. Such latency sensitive behavior is characterized by periodic output patterns with little or no IO activity for the 15 or 30 minutes of duration of alternating computation and output steps thereby providing distinct deadlines for IO completion. Second, the resource demands imposed by single large-scale codes are magnified by the simultaneous use of the petascale machine by multiple batch-scheduled applications, each desiring a substantial portion of IO system resources and each demanding low latency service. Third, when file systems are shared, like those at ORNL and NERSC, it is not just the petascale codes that demand IO system resources, but there are also additional requests

that stem from the analysis or visualization codes running on select petascale machine nodes and/or on attached cluster machines with shared file system access.

The facts listed above all contribute to an important phenomenon observed in the IO systems used with petascale machines, which is that of high levels of variability in IO performance. Measurable sources of such variability include the following:

- *Internal interference* occurs when too many processes within a single application attempt to write to a single storage target at the same time. This causes write caches to be exceeded leading to the application blocking until buffers clear.

- *External interference* can occur even if an application takes great pains to properly use storage resources, since it is caused by 'shared' access to the file system, an example being analysis codes running on an attached cluster machine that attempt to read data stored in the shared scratch space at the same time as the petascale machine is writing its output data. Another example is simultaneous file system use by multiple applications running simultaneously on the petascale machine.

An additional issue is lack of scalability in metadata operations, which has been considered in extensive past research. The LWFS file system, for example, decouples metadata from data operations and postpones them, when possible [79], and the partial serialization approach described in our own previous work with Jaguar [60, 56] reduces intra-application sources of contention experienced by the metadata server.

Prior work in the enterprise domain does not adequately address the internal or external interference effects observed on petascale machines. This is because in enterprise systems, the principal concern has been to properly sequence and batch read vs. write operations on large numbers of files in ways that leverage processes' sequential read behavior to reduce disk head movement while also effectively using available buffer space [9, 41]. These solutions may help with interference effects on single storage targets, but they do not address the load balancing or uneven usage across the multiple storage targets seen in HPC storage systems.

We have developed a new set of dynamic and proactive methods for managing IO interference. These *adaptive IO* methods improve IO performance by dynamically shifting work from heavily used areas of the storage system (i.e., storage targets – OSTs) to those that are more lightly loaded. Adaptive IO is complemented by additional techniques that stagger file open (i.e., metadata) operations to manage performance impacts on the metadata server. By using adaptive IO, we have been able to substantially improve the IO performance of petascale codes, including that of fusion simulations like GTC [43], XGC1 [18], GTS [114], and Pixie3D [16]. These codes generate restart and analysis data every 15 or 30 minutes, with full scale, production data sizes generally between 64 MiB and 200 MiB per process. For a typical petascale run of around 150,000 processes, 200 MiB per process yields 3 TiB to be written every 30 minutes. Staying within a generally acceptable 5% of wall clock time spent in IO limit, this requires a minimum sustained speed of 35 GiB/sec. With the current Lustre limit of a maximum of 160 storage targets for a single file, and a per storage target theoretical maximum performance of around 180 MiB/sec, a maximum of only 28 GiB/sec can be achieved in theory, assuming perfectly tuned IO routines and an otherwise quiet system. Removing this limit can address internal interference, of course, but it does not help with external interference in a busy system. In response, adaptive IO is designed so as to cope with both internal and external interference effects, the goal being to consistently achieve $> 50\%$ of theoretical peak IO performance.

Experimental results presented in this chapter first assess and diagnose the presence and effects of internal and external interference in petascale storage systems. Based on the insights gained from these evaluations, adaptive IO methods are implemented in the context of the ADIOS IO middleware now widely deployed for petascale codes [48]. The outcome is a substantial improvement in IO performance, ranging from around 2x the average performance for a 16384 process run of XGC1 to more than 4.8x for the 16384 process run of Pixie3D with 16 TiB output per IO, all with less variability in the time spent performing IO.

The remainder of this chapter is structured as follows. Section 3.3 experimentally establishes the existence of both internal and external interference for multiple large-scale parallel

file systems. We then describe the design, software architecture and implementation details of adaptive IO in Section 3.4. Section 3.5 presents experimental evaluations, using both actual petascale applications and synthetic benchmarks, the latter to better characterize certain performance properties and behaviors. Results are discussed in Section 3.5.3 followed by an outline of related work in Section 3.6. Conclusions and future work appear in Section 3.7.

3.3 Problem and Motivation

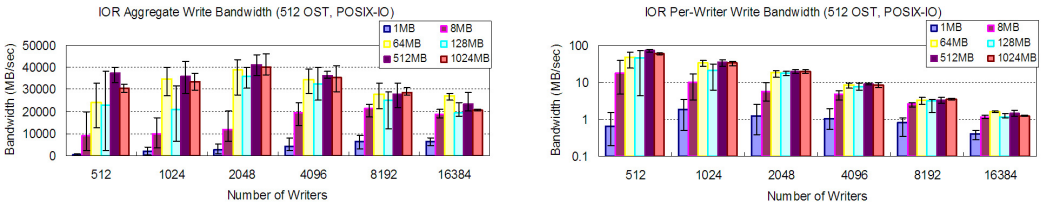
Variability in file system performance due to concurrent use has existed since multi-user operating systems were developed, causing parallel file systems to employ rich caching and other performance management techniques for their internal storage targets. The *internal* and *external* interference effects seen in parallel file systems, however, are not adequately addressed by these techniques, as validated by the performance measurements taken on multiple machines and file systems presented below.

The first set of measurements use the petaflop partition of the Jaguar machine at Oak Ridge National Laboratory. This is a Cray XT5 machine with 18,680 nodes, each with dual, hex-core AMD Opteron processors (224,160 cores) and with 16 GiB of RAM per node. The scratch file system is a 672 storage target Lustre 1.6 system with 10 PiB total storage shared across multiple machines at ORNL. Second are measurements on the XTP machine at Sandia National Laboratories, which is a Cray XT5 with 160 nodes, each with dual, hex core AMD Opteron processors (1,920 cores) with a Panasas file system (PanFS) configured with 40 StorageBlades for a total of 61 TiB of storage. Third are results attained on the Franklin Cray XT4 MPP at NERSC. Franklin has 38,128 Opteron compute cores, and its scratch file system is Lustre with 96 storage targets and 436 TiB storage. Experimental data concerning Jaguar and XTP are collected by the authors of this chapter; performance data on Franklin is obtained from NERSC's online performance monitoring data repository [69].

Measurements reported below first document the existence of internal interference and its impact on aggregate write bandwidth. External interference and its impacts are shown second. The section concludes with a summary of results and insights. To strictly isolate

interference effects, all reported measurements specifically omit file open and close times.

3.3.0.1 Internal Interference



(a) Scaling of Aggregate Write Bandwidth on Jaguar/Lustre.

(b) Scaling of Per-Writer Write Bandwidth on Jaguar/Lustre.

Figure 14: Illustration of Internal Interference Effect

Using Jaguar/Lustre and the IOR benchmark [97], we demonstrate internal interference by writing data of differing sizes via different ratios of processes to storage targets (OSTs). In all such tests, the IOR program is configured to use 512 OSTs, where each process writes data to a separate file and to some fixed OST using POSIX-IO. Writers are split evenly across the 512 OSTs.

Figure 14(a) depicts the scaling of IOR POSIX-IO aggregate write bandwidth on Jaguar with different numbers of writers and different per-writer data sizes. Figure 14(b) shows the corresponding average per-writer bandwidth values at different scales. In both figures, each bar represents the average value among 40 samples with error bars depicting maximum and minimum values. The ratio of processes per storage target ranges from 1 to 32, and the data sizes range from 1 MiB per process to 1024 MiB per process with weak scaling.

Measurements clearly demonstrate the performance effects of internal interference. In Figure 14(b), per-writer write bandwidth consistently decreases with an increasing number of writers, and Figure 14(a) reveals that eventually, the increase in aggregate performance due to an increased total number of writers is dwarfed by the losses in individual writer performance caused by contention. This holds for all cases other than those in which output benefit from the caches associated with storage targets, i.e., with 1 MiB writes. Aggregate bandwidth peaks with a per-writer data size of 8 MiB, then begins to decrease at the scale of 8192 writers (the ratio of writer vs. OST being 16:1); for all other data sizes, aggregate

write bandwidth begins to decrease at the scale of 2048 writers (4 writers per OST). For example, for the largest output size of 1024 MiB per process, the aggregate write bandwidth seen by 16384 writers is 8 GiB/sec less than that of 8192 writers. The effects are amplified at large scales. With per-writer data size equal or larger than 128 MiB, the aggregate write bandwidth degrades by 16%-28% when scaling from 8912 to 16384 writers. Particularly, for the 1024 MiB per writer case, the aggregate write bandwidth seen by 16384 writers is 20.6 GiB/sec, which is only 72% of the bandwidth of 8192 writers. For Sandia's XTP, we did not observe substantial bandwidth degradation except that there is a $< 5\%$ reduction in write bandwidth for the large data sizes (512 MiB or 1024 MiB per writer) when scaling IOR from 512 to 1024 writers. This can be attributed to the XTP machine's relatively small size limiting the contention among concurrent writers and/or the design of PanFS.

3.3.0.2 External Interference

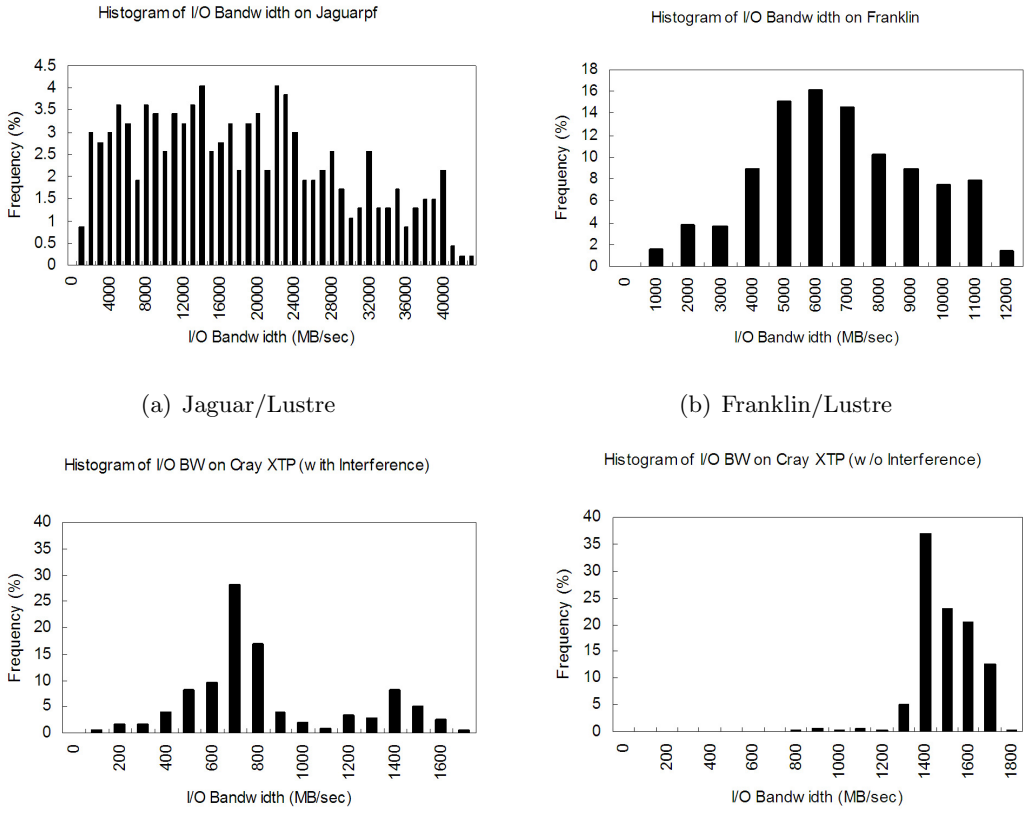
Table 4: IO Performance Variability due to External Interference

Machine	Samples	Avg. IO Bandwidth (MiB/sec)	Std. Deviation	Covariance
Jaguar	469	1.78e+4	1.07e+4	60.09%
Franklin	2581	6.22e+3	2.50e+3	40.22%
XTP(with Int.)	400	7.89e+2	3.44e+2	43.68%
XTP(without Int.)	320	1.44e+3	1.28e+2	8.86%

Tests are run on all three machines to demonstrate the effects of external interference on IO performance. Specifically, hourly IOR tests are launched where each test is configured with 512 writers using POSIX-IO, one file per writer, and one process per storage target. Performance results for these tests have a total of 469 samples of IO actions. Over the past two years, similar experiments have been conducted at NERSC on Franklin using 80 writers, with results from those tests accessible through NERSC's online performance monitoring data repository. The experiments we conduct on Sandia's Cray XTP differ because XTP is not a production machine. Here, tests are run in two controlled ways: the first runs a single IOR program with 512 writers using POSIX-IO and one file per writer (referred to as "XTP(without Int.)"); the second launches two IOR programs at the same time, thereby emulating the presence of multiple simultaneous workloads (referred to as

"XTP(with Int.)").

Table 4 summarizes experimental results, and Figure 3.3.0.2 shows the histograms of IO bandwidth based on the performance data collected. It is clear that in busy production environments like Jaguar and Franklin, IO variability can be substantial, ranging from 40%-60%. On Sandia's Cray XTP, even a moderate degree of sharing (i.e., two simultaneous IOR jobs) can cause IO performance variations of up to 43%.



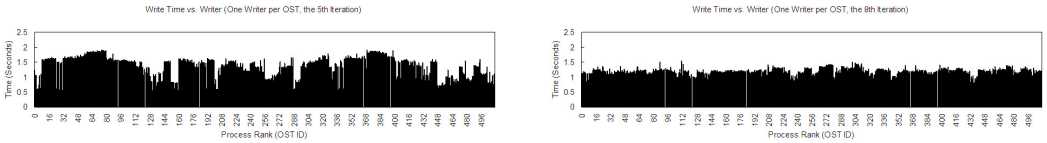
(a) Jaguar/Lustre

(b) Franklin/Lustre

(c) XTP/PanFS (with Int.)

(d) XTP/PanFS (without Int.)

Figure 15: IO Performance Variability due to External Interference



(a) Test 1

(b) Test 2

Figure 16: Illustration of Imbalanced Concurrent Writers

To better characterize the extent of interference, we define the *imbalance factor* of each

IO action to be the ratio of the slowest vs. fastest write times across all writers. Consider two separate samples from the external interference tests for 128 MiB per process on Jaguar. Figures 16(a) and 16(b) show the individual write times for each process for these two tests, respectively. Test 2 took place only 3 minutes later than Test 1. Apparent from these tests is the dynamic and potentially transient nature of external interference, resulting in write times that are much more evenly distributed among all concurrent writers in Test 2 than those in Test 1. In Test 1, an imbalance factor of 3.44 separates the minimum and maximum time spent performing IO. For Test 2, this factor is reduced to 1.86. Interestingly, even for the latter relatively smaller imbalance factor, nearly twice as much data could be written to the faster storage target than to the slower one.

To summarize, we observe a significant imbalance in terms of fastest vs. slowest writes in all IO tests run in our experiments with an overall average imbalance factor of 7.12. Since overall write time is determined by the slowest writer, the purpose of the adaptive IO methods presented in this chapter, then, is to mitigate the performance impact of these 'slow' writers.

3.3.0.3 Alternatives to Adaptive IO

Before describing adaptive IO, we briefly digress to discuss potential alternative solution techniques. One possible way to reduce the effects on applications of IO performance variability is to decouple IO from application actions through the use of asynchronous IO. Unfortunately, given the large volumes of output generated by typical petascale applications, asynchronicity is limited by the total and limited amounts of buffer space available on the machine, which typically extends to only one or at most a few simulation output steps. Such 'near-synchronous' IO, therefore, still causes applications to block on IO when IO performance is consistently too low. Unfortunately and as evident from the experimental evidence presented above, consistently low performance is a natural outcome of internal or external interference.

Data staging [3], a second potential solution to IO performance variability, also has limited applicability. To explain, data staging moves output from a large number of compute

nodes to a smaller number of staging nodes before writing it to disk. However, the total buffer space available in the staging area is limited, thereby limiting the achievable degree of asynchronicity. Further, large staging areas and/or multiple staging areas concurrently used by multiple applications will still lead to internal or external interference. Data staging, therefore, can help with interference issues, but does not directly address them. In fact, our ongoing work is integrating adaptive IO even into the data staging software we are deploying on Jaguar.

Another approach to reducing internal interference is to have the user split output into a collection of files to 'match' the parallel file system being used. In the case of Jaguar and its Lustre FS, for instance, splitting output into 5 parts would enable an application to take full advantage of the entire file system's resources, thereby providing at least a reasonable guarantee of achieving required performance during some normal, productive, busy time. This helps alleviate internal interference, but does not solve it nor does it address external interference. There are two issues with this method: (1) the magic number '5' may work well for Jaguar, but that number will differ for other file systems and machines, and (2) arbitrarily dividing output files in ways motivated by performance rather than file content is not supportive of end users struggling to carry out their scientific tasks.

In summary, the use of asynchronous IO, data staging, and/or target-specific mitigation methods may reduce the effects of IO performance variability on applications, but does not address its root problems. Because of these facts and the substantial performance variability in the storage system, adaptive IO continuously observes the storage system's performance to configure output in a way that transparently 'best' matches its static *and* dynamic characteristics.

3.3.0.4 Summary and Discussion

Experimental results shown in this section demonstrate the existence of internal and external interference on three different machines and with two different file systems. Interference (1) negatively impacts the scaling of IO performance, and perhaps more importantly, (2) introduces substantial IO performance variations that make it difficult to accurately predict and

then properly allocate the amounts of time needed for performing IO. The IO performance variations are shown to be the common rather than the uncommon case, particularly in production environments. This holds both the for POSIX-IO measurements reported above and for tests that use MPI-IO (not reported, for brevity), where for all cases, MPI-IO results show the same trends, but with inferior performance. We conclude, therefore, that internal and external interference are inherent and performance-limiting properties of petascale file systems.

3.4 Software Architecture and Implementation of Adaptive IO

Adaptive IO is implemented in the context of the ADIOS IO middleware [58, 48]. Specifically, adaptive IO is realized as an optional set of techniques bundled into a new IO method.

3.4.1 MPI-IO

The MPI-IO transport method was developed as one of the first options offered by ADIOS. Its common use has resulted in several optimizations, leading to excellent peak IO performance seen on Jaguar and its Lustre file system [60]. For example, the partial serialization of 'open' calls has reduced the time spent by these metadata operations by more than two-thirds [60] for large-scale runs. Substantial performance advantages are derived from limited asynchronicity and by buffering all output data on compute nodes before writing it, if possible. Additional optimizations in certain variants of the base ADIOS transport are tied to the Lustre file system used by many HPC codes. As a result, ADIOS and its MPI-IO base transport constitutes a high performance, well-tuned set of IO abstractions against which adaptive IO can be tested and evaluated.

3.4.2 Adaptive IO

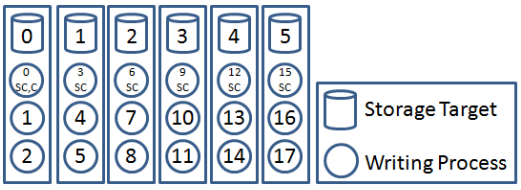


Figure 17: Adaptive IO Organization

Figure 17 depicts a sample configuration. When using the adaptive IO method, middleware enhances the output actions taken by these processes in ways that ascribe to them three different roles, as illustrated in the figure: (1) the numbered circles represent process IDs of writers participating in the output, (2) some of these processes carry out additional actions, acting as sub-coordinators (SC) for a set of writers and a storage target, represented by the vertical boxes in the figure, and (3) one process plays the distinct coordinator (C) role for the entire set of writers. The coordinator and writers only communicate with the sub coordinators, never directly with each other. This isolates the messaging reducing the message load on any particular part of the system.

1. *writers* – all processes in the group write data, an example being process 7 in the figure;

2. *sub-coordinators (SC)* – processes are grouped by the storage target initially assigned, with one process for each such group coordinating access to each target, and in addition, interacting with the coordinator, an example being process 12 in the figure; and

3. *coordinator (C)* – one process is selected to coordinate across all sub-coordinators, but also acts as a sub-coordinator and as a writer, this being process 0 in the figure.

To summarize, all 18 will be writers; processes 0, 3, 6, 9, 12, and 15 will also act as sub-coordinators; process 0 will be the coordinator.

We note that different sub coordinators write to different files, since this is how we can control mappings to certain storage targets. We purposely enhance writers with roles rather than implementing coordinators and sub-coordinators separately from writers. This avoids using additional processes and having to tightly synchronize their coordination actions with the writing actions of separated writer processes. Since process IDs are typically assigned sequentially to cores in a node, grouping them as illustrated reduces the network contention on the node due to simultaneous writing from the same node, but different cores. Finally, by placing the coordination/sub coordination roles into the first process in each group, they can each focus on management after completing their writes instead of possibly being reassigned

adaptively to a different target (file). This choice also avoids any delays in messaging due to the writer role for the process being busy while the coordinator is attempting to start an adaptive writer for this group.

The software architecture chosen for adaptive IO scales to the numbers of writers present in petascale machines like Jaguar and beyond. Specifically, based on the current state of the XT5 partition of Jaguar with approximately 225,000 processing cores and with the current 672 storage targets in the attached Lustre scratch system, this means each sub-coordinator is responsible for, at most, 335 processes. The coordinator is only responsible for the sub-coordinators, giving it 672 processes to manage. Even at the extreme scale of the Jaguar machine, these numbers are manageable and leave room for growth. An additional layer of coordination or distributed or partial coordination would further improve scalability, at the costs of additional messaging and thus, coordination overheads. Another choice would be distributed or partial coordination, e.g., by having small groups of sub-coordinators act independently, with the resulting penalties of reduced flexibility in process-to-storage target mappings and thus, potentially reduced overall performance. Some insights on these tradeoffs are present in prior work on larger-scale management architectures for the enterprise domain [46, 47].

We next explain in more detail the precise actions taken by processes in different roles.

3.4.2.1 Writers

Each writer task simply waits for a start message, writes to the indicated file at the indicated offset, and finally, generates a completion message to the sub coordinator to trigger the next writer. The details of this role are described in Algorithm 1. To ensure a consistent flow of data to storage, file indexing information is transferred separately and after writing is complete, so that this additional metadata transfer can take place concurrently with another process writing to storage.

Algorithm 1 Writer Process

```
1: Wait for message (target, offset)
2: Build local index based on offset
3: Write data
4: Send WRITE_COMPLETE to triggering SC
5: if triggering SC ≠ target SC then
6:    Send WRITE_COMPLETE to target SC
7: end if
8: Send local index to target SC
```

3.4.2.2 Sub-Coordinator (SC)

The sub-coordinator is responsible for scheduling IO to the local storage target and for managing the indexing of the data stored in this file. Communications between the sub-coordinator(s) and coordinator constitute the major elements of the adaptive IO implementation. The details are described in Algorithm 2.

3.4.2.3 Coordinator (C)

The coordinator role is generally idle until the late stages of IO when sub coordinators message their completion. As completions arrive, the coordinator begins to obtain a global view of the relative performance of storage targets. Given this view, it then attempts to shift work from busy (i.e., slower) to less loaded (i.e., faster) storage targets. This continues until all work has been completed, at which point it signals the completion of the composite write operation so that the local indices can be created and a global, master index formed. Adaptive writing requests are spread evenly among the sub coordinators to spread out the accelerated completion of the write rather than pushing sub coordinators to completion one at a time. The sub coordinators are tracked as either *writing*, the initial state for the output operation, *busy*, indicating all processes have been scheduled so no adaptive writes are possible, or *complete* indicating that all writers have completed and this file is available for adaptive writing use. The details are described in Algorithm 3.

This adaptive mechanism scales according to the number of storage targets rather than the number of writers. The coordinator is only involved in the process once the bulk of writers are complete. Then, the largest number of simultaneous adaptive requests is

Algorithm 2 Sub-Coordinator Process (SC)

```
    while not done and missing_indices ≠ 0 do
2:    Signal next waiting writer to write
      Wait for message
4:    if message = WRITE_COMPLETE then
        if source is one of mine, but target is not me then
6:        Send adaptive WRITE_COMPLETE to C
        end if
8:      if source is one of mine and target is me then
          Save index size for index message
10:       missing_indices++
        end if
12:     if all writers completed then
          Send WRITE_COMPLETE to C
14:     end if
      end if
16:   if message = INDEX_BODY then
        Save for index for local file
18:     missing_indices–
      end if
20:   if message = ADAPTIVE_WRITE_START then
        if no waiting writers then
22:       Send WRITERS_BUSY to C
        else
24:       Signal writer with new target and offset
        end if
26:   end if
      if message = OVERALL_WRITE_COMPLETE then
28:     done = true
      end if
30: end while
    Sort and merge the index pieces for file index
32: Write the index
    Send the index to C
```

strictly limited to $SCcount - 1$ as at most one write will be active for any file at one time. A larger pool of writers will only serve to keep the distributed, independent sub coordinators busy longer without affecting the coordinator with any additional simultaneous work. Adaptive IO has been fully implemented and tested, with the exception of the global indexing phase. In the interim, we use a automatic, systematic search of the index in each file for particular data of interest. The inclusion of the data characteristics [60] aid this search by enabling quickly searching for both the content as well as the logical 'location'

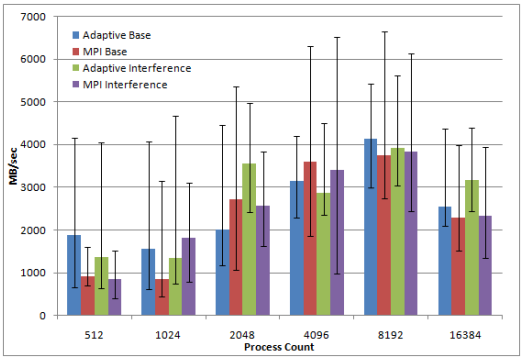
Algorithm 3 Coordinator Process (C)

```
    while any SC state ≠ complete or adaptive write request outstanding do
      Wait for message
 3:   if message = WRITE_COMPLETE then
        if this was an adaptive write then
          Request adaptive write by next writing SC
 6:     end if
        if this is an SC completing then
          Set SC state to complete
 9:       Note final offset
          Request adaptive write by next writing SC
        end if
12:   end if
      if message = WRITERS_BUSY then
        Set SC state to busy
15:     Request adaptive write by next writing SC
      end if
    end while
18: Send OVERALL_WRITE_COMPLETE to all SC
    Gather index pieces
    Merge into global index with local file information
21: Write global index file
```

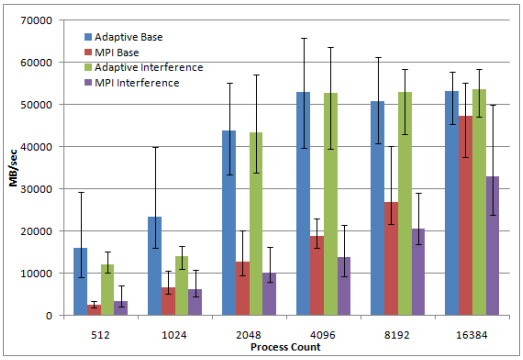
of the data of interest. Also note that the Adaptive IO configuration shown in this section can be generalized, at the consequent cost of additional code complexity. For instance, one might use 2 or 3 simultaneous writers per storage location and/or multiple storage locations per sub coordinator. For example, to allow multiple simultaneous writers instead of having a single queue for each storage target, a list of writing processes for each target would have to be maintained. For multiple storage targets per group, a list of processes and of offsets per target is required, but more important here is the fact that such an approach would introduce potential 'holes' in the local files due to adaptive movement of processes among both group local and other storage targets. We have not experimented with these generalizations.

3.5 Experimental Evaluation

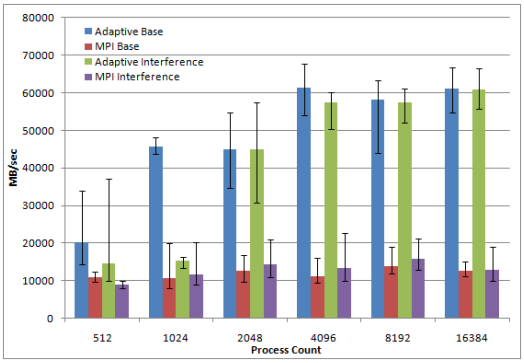
To evaluate the performance of adaptive IO, all tests are performed on the XT5 partition of the Jaguar system at Oak Ridge National Laboratory (see Section 3.3 for detailed machine configuration). Tests aim to understand how different per process sizes of data perform with



(a) Small Data (2 MiB/process)



(b) Large Data (128 MiB/process)



(c) Extra Large Data (1024 MiB/process)

Figure 18: Pixie3D IO Performance

adaptive vs. non-adaptive IO, using two production petascale codes: (1) an IO kernel for the Pixie3D MHD simulation is run at 3 different per process data sizes; (2) the full XGC1 fusion code is run using a single per process data size. The ADIOS [58] layer is used to switch between the MPI-IO and the adaptive transport methods described in Section 3.4.

The four output size sets of tests demonstrate the performance ranging from 2 MiB/process up to 1024 MiB/process. Tests are run with different process counts from 512 to 16384 processes against 160 OSTs for MPI, the maximum allowed for 1 file, or 512 OSTs for adaptive. The 512 OST selection for adaptive is chosen to simplify the discussion of ratios of writers to storage targets. The adaptive approach has been successfully tested with 672 storage targets with no penalties compared with the 512 storage targets measurements presented here. The tests are first run under normal system conditions with whatever other simultaneous jobs happen to be running. A second set of runs are performed with artificial interference introduced in an attempt to show the performance under a more heavily loaded file system. These results are then analyzed to show the performance of the different IO approaches. To ensure accurate measurements, an explicit 'flush' is introduced prior to the file close operation for both the MPI and the Adaptive transport methods. For all cases, at least five samples are generated and included. Where possible, additional samples are included as well to strengthen the numbers. In all cases, the times reported only include the actual write, flush, and file close operations to remove the variability due to the metadata server.

External interference is introduced through a separate program that continuously writes to a file striped across 8 storage targets during the runtime of the interference test cases. A stripe count of 8 is selected to reflect two applications writing using the default stripe count of 4 configured for the file system. Three processes each write 1 GiB continuously to a single storage target, for a total of 24 processes.

3.5.1 Pixie3D

Pixie3D [16] is a 3-Dimensional extended MHD (Magneto Hydro-Dynamics) code that solves the extended MHD equations in 3D arbitrary geometries using fully implicit Newton-Krylov algorithms. Pixie3D employs multigrid methods in computation and adopts a 3D domain

decomposition. The output data of Pixie3D consists of eight double-precision, 3D arrays that represent mass density, linear momentum components, vector potential components, and temperature respectively. The tested configuration consists of three different sized runs, named *small*, *large*, and *extra large*. The small run uses 32-cubes, large uses 128-cubes, while extra large uses 256-cubes. These cubes represent the per process, per variable size of the data. Overall, the small run generates 2 MiB/process, large generates 128 MiB/process, and extra large generates 1 GiB/process. Weak scaling is employed. Please note in the presented graphs that the error bars show the full range of performance achieved. Even though the error bars may appear larger in some cases for the adaptive than the non-adaptive tests, the percentage variation should be considered. The details of the variability is presented in Section 3.5.3.

The first set, shown in Figure 18(a), use the small data model for Pixie3D. With this model, the 2 MiB/process, even at the 16384 process level, never comes close to the 2 GiB cache size for the storage target (32 $\times$ 2 MiB). Given that, in general, the adaptive approach does well. For example, at both 8192 and 16384 processes, the adaptive approach is 10% better on average for base performance. For the interference tests, 8192 processes for adaptive is 3% better on average while the 16384 processes test came to about 35% better. This small data model is maybe 10% of a typical data size for an application like the S3D [19] combustion simulation or the Chimera [65] astrophysics code. Interestingly, although these data volumes are small, as process counts increase, the adaptive approach can still pay off.

The second set, shown in Figure 18(b), use the large data model. This model consists of 128 MiB/process and it quickly overcomes any caching advantage the storage targets may provide. It has consistently better performance both on average and at a maximum. The improvements range from 1% to more than 350% for the base case and 62% to more than 430% for the interference case. This 128 MiB/process data size is comparable to what many of the fusion codes generate on a per process basis, such as GTC [43]. Another way to look at this data is considering a hybrid MPI/OpenMP setup. In this case, we divide the 128 MiB by the number of OpenMP threads to find out what the per process data size would

be. For Jaguar's 12 cores per node, this yields approximately 10 MiB, or about the size of smaller S3D and Chimera runs.

The last set, shown in Figure 18(c), use the extra large data model. Although there are 3.2x more storage targets used for the adaptive approach, it is about 4.8x faster than the non-adaptive one! Once the adaptation can play a role, i.e., there are a few more processes than storage targets, there is a consistently $> 300\%$ performance improvement for both the base and interference tests. We note, however, that this data model is large even by fusion simulation standards, but we use it because of the growth in per node core counts on future platforms, likely resulting in hybrid MPI/OpenMP codes with larger per-node output.

3.5.2 XGC1

The XGC1 [18] code is a fusion gyrokinetic Particle In Cell code that uses realistic geometry to understand the physics on the edge of the plasma in a fusion reactor, such as ITER. These tests are performed using a configuration that generates 38 MiB per process and weak scaling is used. While 38 MiB per process is smaller than the largest runs for XGC1, it is still a representative size for a production run.

The performance of XGC1, shown in Figure 19, falls between that of the Pixie3D small and large data models, as would be expected. In this case, 38 MiB/process is not uncommon for many scientific codes beyond XGC1, such as larger S3D runs. Adaptive IO shows clear advantages. For example for all of the tests, the performance improvement ranges from 30% to greater than 224%.

3.5.3 Additional Insights and Discussion

Adaptive IO benefits from 'locality-awareness', referring to the fact that when outputs are written, there are less vs. more 'busy' areas of the file system, due to external and/or internal interference. Measurements and evaluations appearing in this chapter substantiate that fact, and they also refine our earlier reports in which we note variability in IO performance [56]. We further substantiate these statements by next showing that adaptive IO typically reduces the IO performance variability experienced by applications.

For both Pixie3D and XGC1, once the process count reaches some small multiple of

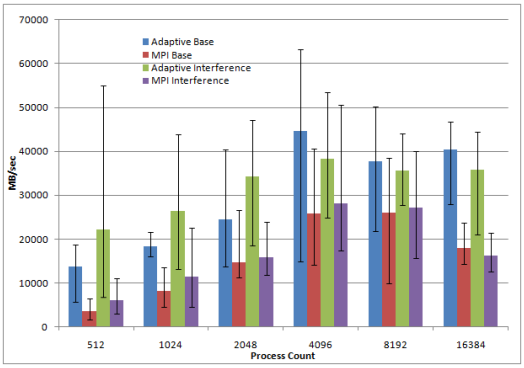
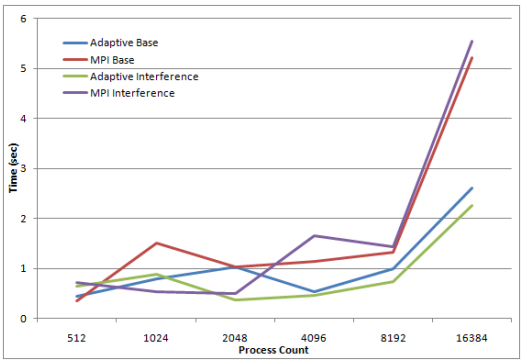
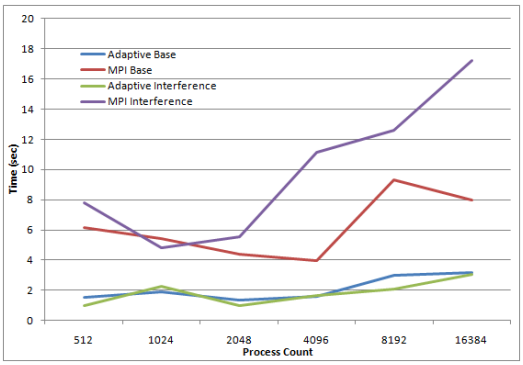


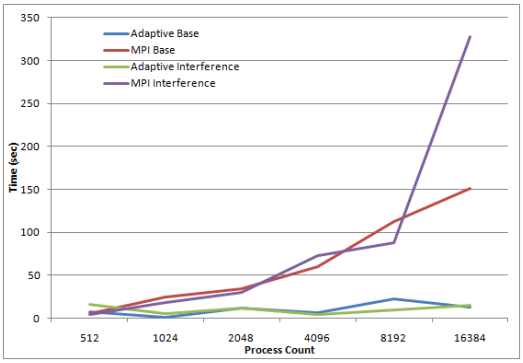
Figure 19: XGC1 IO Performance (38 MiB/process)



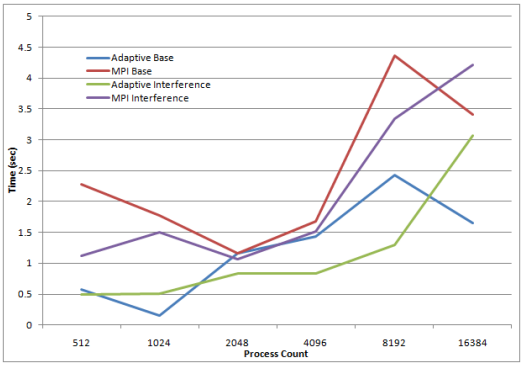
(a) Pixie3D Small



(b) Pixie3D Large



(c) Pixie3D Extra Large



(d) XGC1

Figure 20: Standard Deviation of Write Time

the storage target count, e.g., 4, the adaptive approach offers higher and more consistent performance. The graphs in Figure 20 show the standard deviation of the write times for each of the 4 cases measured. Here, the absolute numbers are less important than the fact that for all cases, once the caches on the storage targets start to be taxed, adaptive IO reduces variability. In some cases, such as in Figure 20(c), the difference is quite large.

Adaptive IO manages these variations by taking advantage of the imbalance factor noted in Section 3.3 to dynamically shift work from slower areas of the file system to faster ones. The remaining variability is due to largely two sources. First, the length of time the last process takes to complete writing cannot be worked around. Second, other use of the file system will always impact the performance of an individual write. Adaptive approaches can minimize the overall impact of this interference, but it will always be present. A potential issue with using adaptive IO, however, is that it requires additional files for output. Specifically, the number of output files is a function of the file system size rather than the process count in the run. By using the global index, access to any data can be performed using a single lookup into the index and then a direct read of the value(s) from the appropriate data file(s), sometimes resulting in improved performance [86] compared to the use of single storage formats. Considering that output sets are generally treated as a unit and that the number of files is a function of the number of storage targets rather than the number of processes, we believe the use of additional files does not strongly impact the ability of the scientist to manage the generated data.

3.6 Related Work

Parallel file systems offer high levels of performance for HPC applications, including Panasas [82], PVFS [90], Lustre [14], and GPFS [93], all of which provide POSIX-compliant interfaces. As stated earlier, there remain certain performance challenges, however. Lustre, for example, has a single metadata server, causing potential bottlenecks by serializing operations requiring metadata updates. More generally, these systems aim to provide general purpose, multi-user file services, which as a goal, is somewhat orthogonal with a single user's desire to receive substantial IO resources and then, to optimize how these resources are used on

behalf of that user. Adaptive IO provides such complementary functionality.

Current work on log-based file systems [85, 11] has improved write performance for checkpoints, but at the potential cost of reduced read performance. PLFS [86] has demonstrated that read performance does not suffer when performing a restart-style read of all of the data, but interference effects have not yet been addressed. Zest [78] skews the IO performance in favor of write performance by strictly acting as a write cache for another read/write file system. For all of these systems, however, it remains important for end users to consider their IO characteristics in order to extract high file system performance. Adaptive IO does not require users to have such knowledge.

Systems like PaScal [15] do an excellent job managing the data bandwidth, but require a different IO hardware architecture. LWFS [79] breaks POSIX requirements in order to better suit client needs, but does not address the internal nor external sources of interference created by the shared file system. The idea is to attain performance gains by eliminating certain consistency and coordination requirements, then permitting them to be specialized to certain applications and their specific needs. Neither of these attempt to manage multiple writer processes within a client to avoid internal interference nor do they explicitly address external interference issues. The Google File System [30] is focused on high aggregate throughput, but is not concerned with maximizing per client performance. Closer to our work is that of Gulati and Varman [33], who provide for scheduling IO operations, but their focus is on using caches to improve read performance, and they do not address the cases where the data is far larger than total cache space.

Using middleware to manage IO to improve performance is not new to HPC. MPI-IO introduced the ADIO layer [40] as a way to install system-specific optimizations of the general MPI-IO implementation. Many optimizations are possible and have been handled at this layer, such as custom drivers for Lustre or PVFS. Collective IO, also handled in this middleware layer, attempts to perform a level of data-size driven adaption by aggregating small writes into single, larger writes to obtain greater performance, but it does not address the issue of large writes from all of the processes, nor does it address interference problems.

At the slightly higher layer of IO APIs, the issue has largely been sidestepped. Both

HDF5 [35], and therefore NetCDF version 4 [110], and PnetCDF [72] have ceded control of this detail to the underlying IO layer, typically using MPI-IO. While they are 'closer' to the application and therefore, have more knowledge about the size and distribution of data, there are many optimizations that could be implemented if they more directly manipulated the lower IO layer. Some work has been done by the PnetCDF team on 'subfiling' [21] to try to address the need to decompose the output to gain greater parallelism. However, this work requires the user to guess the number of files into which to decompose the output and will only decompose along the major dimension of an array rather than based on the overall data sizes. They also did not address the transient performance issues of external interference.

Data staging efforts have primarily focused on reducing data read times in HPC [66], in grid environments [23, 81], and for mobile applications [96]. More recent work has used data staging for enhancing write performance [75], but at a cost of additional compute resources. Demonstrated on a BlueGene/P with dedicated IO forwarding nodes, the IO Forwarding Scalability Layer [6] aggregates requests to reduce contention on the IO system to manage internal interference for writing but does nothing to manage external interference effects. To soften the cost of the additional resources for data staging, DataTap [4] provides data staging-like functionality, but provides much more significant in transit data processing features. DataTap has worked extensively to manage the IO effectively using several scheduling techniques [3].

ADIOS has demonstrated it can successfully achieve high output performance [60], an outcome of its componentization approach being that it can manage different portions of the IO using different techniques, for best overall performance. ADIOS' default BP file format has been shown to help improve IO performance through techniques like delayed consistency checks while maintaining sufficient information for scientists to convert the ADIOS default BP file format into a format like HDF5 or NetCDF for use with current analysis routines [59]. These features make it an excellent choice as a development platform for experimenting with different IO techniques within production science codes. Some results for the ADIOS *stagger* IO approach were reported at the 2009 Cray User's Group [56]. Stagger addressed

internal interference and exposed the magnitude of the transient external interference. Since these results were presented, the XT5 partition and the underlying scratch file system have undergone considerable change, preventing a direct comparison with the results presented in this chapter. In particular, the number of cores was increased by 50% and the connection to the file system was adjusted because it is now being shared as the primary scratch space across most HPC resources at ORNL. These changes in system configuration make managing IO performance vairability even more challenging and motivate us to explore adaptive IO mechanism in this chapter.

Adaptive approaches have been applied to IO systems in the past. These efforts have fallen into a few broad categories. The first group has examined using adaptive techniques to schedule IO for real-time systems [64]. These efforts have looked more at how to have effective IO integrated with a real-time system rather than optimizing for IO performance overall. Second, shared use of enterprise shared storage systems is considered in [111]. The goal is to maintain some level of quality of service for all competing applications, but there is no consideration of balanced IO loads for single applications across multiple storage targets. Third, more recent work investigates pre-fetching and job scheduling [108], integrated with the job scheduler for an HPC resource. The goal is to reduce application load and start up times by pre-staging input data for read-intensive workloads. Streaming systems have also incorporated adaptive techniques [55]. These attempt to predict IO needs for streaming applications and manage by rerouting network paths dynamically and other approaches for removing bottlenecks between the data sources and storage (consumers). These approaches have not addressed the special issues a parallel file system introduces. Also related to our work is the OPAL ADIO library [121] for MPI-IO. It attempts to manage IO based on the disk system itself, but it does not dynamically adjust where data is written. CA-NFS [10] pursues goals similar to ours, but it does not actively manage different storage areas, relies on asynchronous IO, and is limited to the mechanisms of NFS. Most closely related to our work is [100], which dynamically changes disk striping based on data sizes and on information about past usage. Its focus on repeat IO events means that it does not dynamically adjust file system usage across a single large output file, as done in our work.

Observations about the performance variability of shared HPC storage systems appear in [8], where NERSC researchers report that a small number of slow storage targets greatly increased total IO time. System logs and dedicated benchmarks [45] have been used to identify a variety of performance variations in HPC environment. In the enterprise domain, black box approaches [42] have been used to identify sources of performance problems related to storage or the network. Network sources for contention [12] have also been documented. Our observations about IO performance variability comply with these work, and our work explores active management to better handle IO performance variability.

3.7 Conclusions and Future Work

Interference effects cause variable IO performance on both the shared file systems present at NERSC and ORNL, but also on machines with non-shared file systems, like Sandia's XTP. The adaptive IO methods presented in this chapter mitigate such variability by continuously observing the storage system's performance and then balancing the workload being imposed. This substantially improves the IO performance seen by petascale codes, as demonstrated with numerous measurements and on multiple machines.

The reduced variability and high performance can be used to help scientists more accurately predict how long simulations need to run in order to reach particular 'events', reducing the variability factor included in the job request time to ensure sufficient calculation time has elapsed. This not only reduces costs for scientific application runs, but improves the overall use of the machine by affording more applications runs over the same period of time.

Our future work will examine the benefits of adaptive IO on systems beyond Lustre at ORNL, including Franklin at NERSC, PanFS on Sandia's XTP, and perhaps, GPFS on a BlueGene/P machine. Also of interest are other sources of variability, including that of metadata operations like file opens. Finally, there are likely more complex and/or state-rich methods for system adaptation, including those that take into account past usage data.

This chapter examined the issues of variability and scale of large scale parallel file systems. It addressed the two primary sources of variability, internal to the application where

different processes inadvertently compete with each other slowing down IO and external where outside forces are acting upon the file system causing the performance of different areas of the file system to degrade temporarily. With these innovations, the write portion of the write-read cycle for extreme scale science codes is largely addressed. The remaining challenge to be addressed by this thesis is the issue of how these optimizations impact scientist productivity. Only if these optimization for writing do not grossly negatively affect the reading acitivities of application scientists can this effort be deemed a success. The next chapter delves into the impact of these changes on both restart performance and on some common analysis read patterns.

CHAPTER IV

EVALUATION OF IMPACT OF WRITING OPTIMIZATIONS ON READING PERFORMANCE

4.1 Introduction

This chapter focuses on determining how the optimizations made to the writing cycle affect reading performance. In particular, the data layout generated by the BP format imposes a fragmented reading pattern for checkpoint restart and analysis reading patterns.

Scientific productivity is a key goal when running petascale science simulations. Since attaining this goal requires scientists to derive scientific insights from the enormous quantities of data generated by petascale codes, an important technical challenge is to obtain high 'end to end' IO performance, both (1) for writing simulation output to storage and (2) for reading it for subsequent data processing and analysis. Issues include: (i) as HPC applications run with higher process counts, we see increased frequencies and sizes in the checkpoints needed for fault tolerance [94]; (ii) there are proportional increases in the resolution and sizes of the data analysis outputs used to ascertain application progress and health, to extract select scientific insights, or for code coupling in complex simulation systems; and (iii) already, for certain analysis systems in common use, such as the VisIt visualization system [112] used to render scientific data, IO performance has come to dominate all other costs observed in their use [20].

Recent work on extreme-scale IO has developed middleware-based methods to improve write performance [58], has created new output formats for efficient data storage across storage targets [60], has enriched IO software stacks with methods for data staging [3, 76], and has improved IO performance with asynchronous or adaptive IO methods [57] and by explicitly scheduling such data movements [3]. An issue remaining for such work, however, is the aforementioned consideration of 'end to end' IO performance where once data has been written, it must next be efficiently read by restarts and by the analysis or visualization

codes used for data exploration and understanding. Toward that end, this chapter develops and evaluates new methods for scientific data organization and layout on disk, termed *data districts*: (1) each district constitutes a non-overlapping portion of the simulation data space that is organized so as to group together logically 'nearby' simulation data; (2) data districts can vary in size and their totality fully describes each single output step generated by the high performance code; and (3) there is flexibility in how data districts are generated, including where each element of a simulation running on a node of the petascale machine may output multiple and differing data districts.

The use of data districts can substantially improve end to end IO performance:

- it makes it possible to organize output data into multiple dimensional volumes, in keeping with the natural data organizations used in scientific codes; and
- this organization then enables the placement of data onto storage targets in ways that improve the concurrency seen both when data is written and when it is read.

District-based writes and reads are implemented via the ADIOS IO middleware [58], which is aware of both (i) the types and sizes of data used by applications and (ii) the concurrency available in the underlying parallel file storage systems (i.e., its object storage targets) using its BP file format.

The use of data districts is a purposeful departure from current practice in scientific IO, where end users employ standard file formats like NetCDF [71] and HDF5 [35] that have been designed for flexibility in data use and for attaining high levels of portability. In a sense, these standard file formats employ the equivalent of a single district for each output variable, reorganizing data during output to construct a contiguous storage format. In contrast, the use of data districts is akin to a log-based approach to data writing and reading, employing the underlying BP file format for disk storage and access and converting to standard formats as and when needed [60]. With data districts, we observe high end to end IO performance, both for writing data districted files – as also seen in log-structured file systems [88] – and for reading these files, in contrast to common wisdom about log-based file storage. Experimental evaluations demonstrate these facts for representative petascale

simulations and for the write and read patterns seen for such codes, with read patterns derived from checkpoint/restart, analysis, and visualization [112] access patterns seen for these codes. In summary, the key insights that warrant revisiting log-based formats for data storage for extreme scale computing are that (1) by distributing data across as many stripes as possible in the parallel file system, greater concurrency for reading data can be attained, (2) separating considerations of performance from portability can greatly improve performance, and (3) the predominant read patterns are logical slices of the 3-D simulation space rather than sequential reads of the entire file. These three attributes of extreme scale data reading can improve data access times through parallel access and the fact that more useful data is read per operation.

Data districts leverages prior work that has shown improved write performance for log-structured file formats for checkpoint/restarts for both the ADIOS [58] and PLFS [11] systems. ADIOS is the Adaptive Input/Output System (ADIOS), a joint project of Oak Ridge National Laboratory, Georgia Tech and Sandia National Laboratories, and PLFS is the Parallel Log-structured File System developed at Los Alamos National Laboratory. This chapter precisely formulates the notion of data districts and then evaluates it by considering the typical read patterns seen for the analysis applied to simulation output and then using appropriate district-based data organizations for improving read performance. In this fashion, high write performance is combined with as much as a $6\times$ performance improvement in the read performance seen by analysis codes. Specific measurements examine the impacts of data district size and organization on subsequent analysis read patterns for both 2-D and 3-D domain decompositions to determine when, why, and to what extent log-based data districts and their uni-dimensional contiguous data organizations improve read performance. The goal is to determine how to construct and size districts within a log-based format for the lifetime of the simulation data, in contrast to an approach that immediately converts data to the common contiguous formats currently used by common analysis tools.

In the remainder of this chapter, representative end to end IO patterns seen for petascale applications are explained in Section 4.2, followed by a description of the concept of data

districts and the IO architecture in which they are realized in Section 4.3. Experimental evaluations of the concept and its performance appear in Section 4.4. This is followed by detailed discussions and analyses of results in Section 4.5, where we also discuss the implications of data districts on the IO pipeline used with petascale machines. Related work appears in Section 4.6, with conclusions and future work in Section 4.7.

4.2 End to End IO Patterns

Initial performance assessment. With the parallel file systems used in current supercomputing centers, attaining high end to end performance in IO depends both on the way data is written and the way it is read. Well-understood writing and reading patterns are those used for checkpoint/restart [54], coupled with additional writes performed for simple analysis tasks. For such patterns, prior work has established high write performance when using a log-based output format with the PLFS file system [11] as well as when using the BP log-based format provided by the ADIOS IO middleware [60]. In fact, initial measurements showed the ADIOS/BP approach to have an up to 1000× performance advantage compared to data being written with the standard HDF5 output format. These tests evaluated the IO time for performing the complete output of application state for checkpoint restarts for roughly 100 different variables, for a total of around 13 MiB per process. Similarly, PLFS's log-structured approach improved performance by as much as a factor of 150 on large deployments [11]. We note, however, that the performance of the original HDF5 was later improved by a factor of 10, reducing the advantage seen by ADIOS/BP to 100×, and yet more recent improvements may further narrow this gap. Finally, if end users require data to be in some standard format like HDF5, then the ADIOS/BP approach will incur additional costs, for which the single process format conversion time for BP to HDF5 has been measured as being linear with respect to data size [60].

Understanding end to end IO performance. This chapter explores the following ideas: (1) by characterizing and describing the data organizations used – data districts – to obtain performance improvements, (2) by determining and evaluating additional and typical science data read patterns, such as those used by analysis codes, and (3) by diagnosing the sources of

performance improvements derived from using log-based data organizations for large-scale scientific data on the parallel file systems used in supercomputer installations.

For both of our test cases, the simulation space consists of a 3-dimensional space distributed across the processes that comprise the application run. In the case of a 2-D domain decomposition, the 3-D space is decomposed such that one of the three dimensions is not split across processes. For a 3-D domain decomposition, the space is split into rectangular sub-areas that do not span any dimension entirely. For both decompositions, the data stored local to each process is a 3-D piece of the entire space. When this data is written to disk, the shape of this space, particularly the overall data size and shape per data district, plays an important role in determining the performance of common analysis read patterns.

To test these common domain decompositions, we first establish a typical set of analysis read patterns used by petascale science codes. To do so, we interacted with the science users of two petascale science codes that each use different domain decompositions. The union of these results are represented in the set of test cases described below. The 2-D domain decomposition is used with an older version of the Chimera supernova code [65] in production use at petascale run sizes on systems at Oak Ridge National Lab and Texas Advanced Computing Center. Chimera is a code that couples multigroup flux-limited diffusion neutrino transport (a sophisticated approximation of Boltzmann transport) along radial rays (the ray-by-ray-plus approximation) to three-dimensional hydrodynamics, a nuclear burning network, Newtonian self gravity with a spherical general relativistic correction, an industry standard nuclear equation of state (Lattimer-Swesty, Shen, Wilson), and with state of the art neutrino interactions. For a 3-D domain decomposition, the S3D combustion code [19] is examined. S3D is a flow solver for performing direct numerical simulation (DNS) of turbulent combustion. This resulted in the identification of the following write/read patterns:

1. *All data* is written and read, but writes and reads are done by different numbers of processes, and for generality those numbers are not simple multiples of each other.

2. *All of 1 variable* is read from a complete output set. Again, this is performed using different, non-multiple numbers of writers and readers. An example is reading the

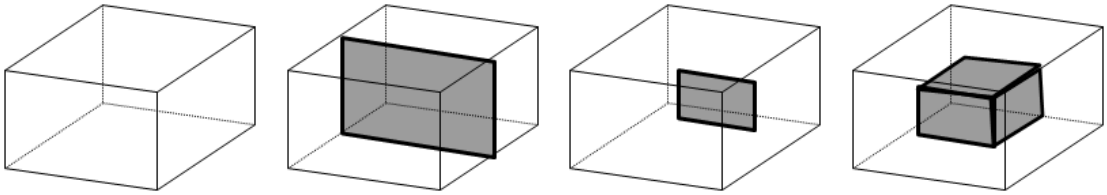
temperature values associated with particles. (see Figure 21(a)).

3. *All of a few variabiles* using different, non-integer multiple numbers of processes. An example is reading three variables to generate a magnetic field vector.

4. *A plane in each dimension* for qualitative exploration (see Figure 21(b)).

5. *An arbitrary rectangular subset* representing a cubic area of interest (see Figure 21(d)).

6. *An arbitrary area on an orthogonal plane* representing one of a collection of read operations to obtain an arbitrary area within the simulation space (see Figure 21(c)).



(a) Whole Domain (b) Whole Plane (c) Partial Plane (d) Sub Area

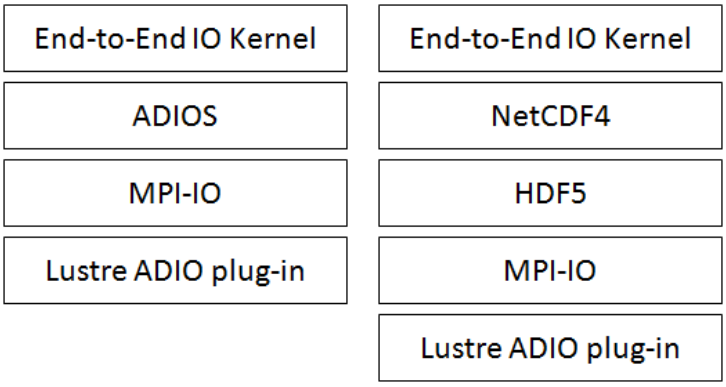
Figure 21: Data Selection Patterns

For analysis reads, Chimera most commonly uses patterns 1-3: all data, all of 1 variable, and all of a few variables. Patterns 4-6 are used as part of an interactive visualization: a plane in each dimension, an arbitrary rectangular subset, an arbitrary area on an orthongonal plane. The S3D code uses the same patterns: pattern 1 for production runs/restarts, patterns 2 and 3 for postprocessing and analysis, pattern 4 for production sanity checks and qualitative exploration, and pattern 5 for postprocessing and in depth analysis. Pattern 6 is not commonly used by S3D. Since the first pattern is essentially the same as the restart performance evaluated in previous work, we do not consider it further in this chapter. Finally, we were not able to elicit from science users other read patterns for the petascale codes under study, causing us to confine our initial study to the patterns identified above.

4.3 Evaluation Architecture

Performance test description. Our goal is to better understand the read performance of write optimized scientific data files. This complements earlier studies and provides a picture of the end to end, i.e., combined write and read, IO performance attainable on

the highly concurrent file systems supporting petascale machines. The IO software stacks evaluated in our work use ADIOS version 1.2 and NetCDF 4.0.1.3 configured to use parallel HDF5, respectively. Specifically, with the ADIOS middleware, the IO stack selects the MPI transport method, resulting in a two layer architecture for IO as illustrated in Figure 22(a). NetCDF uses the MPI transport underneath the HDF5 layer, resulting in a three layer software stack, illustrated in Figure 22(b). This approach helps control inefficiencies in using the HDF5 API, thereby constituting an optimized use of the HDF5 API and files. Further, the additional layer used in the NetCDF stack has negligible additional overhead compared to that of the ADIOS-based stack, as total performance is almost entirely dominated by data movements to and from storage. Finally, to test these different patterns in a consistent way, an end to end (e2e) IO kernel representative of the write and read patterns is created. This e2e IO kernel exhibits a typical writing pattern and then reads data using any of the six patterns.



(a) ADIOS (b) NetCDF

Figure 22: IO Software Architectures Tested

Data Districts and their implementation. For testing, ADIOS [58] and NetCDF [71] are employed. ADIOS is a 64 bit compliant IO componentization. It provides an API almost as simple as POSIX IO, and more importantly, it permits the runtime selection of different IO mechanisms for each IO section of the simulation or host code using ADIOS.

ADIOS has a file format *BP* (Binary Packed) that is decomposed into fragments generally based on the processes that create it rather than on the logical structure of the data.

This format is illustrated in Figure 3. Each of these fragments is referred to as a *Process Group*. The last portions of the file consist of a file version flag and a collection of indices and pointers to the location in the file for each of the indices.

At a detailed level, process groups consist of a short header listing information about the data output grouping being used, including a user-assigned name, such as restart, analysis, or diagnostics, the parameters used for this output method, and a list of variable and attribute entries. Each variable and attribute entry consists of the metadata for the item listing the name, data type, array dimensions, if any, data characteristics, and a payload blob. This payload blob is a memory dump of exactly what was stored in memory with no byte-ordering changes nor restructuring. When reading data back in, each payload that contains part of the data is read and the relevant pieces are extracted to reconstruct the portion of the data requested. If the reading platform uses a different byte ordering, annotations in the file indicate the byte ordering used by the writing system so that the reordering can be performed during the read operation. The indices consist of exact locations of the process groups, metadata about the process group, and the list of variably sized 'pieces' including information about the array dimensions and extents, the data characteristics for each piece, and the list of attributes and the location of each in the file. Each such variable 'piece', then, is a self-contained and self-identifying *data district*.

When writing data, no data reorganization is performed, resulting in high write performance for various logical data organizations. The results reported in this chapter additionally demonstrate that when reading the data, the district-based data decomposition typically improves performance when compared with a canonical order format like the HDF5. The steps taken when reading district-organized data include consulting the index to determine the file offset for reading data for the local process, then reading the relevant district(s), followed by reorganizing the data and discarding the data items that are not needed (i.e., were not requested). There are opportunities for additional optimizations in reading data to avoid unnecessary data discards, but those optimizations have not yet been implemented.

The data district-based organization used with BP is in contrast to the data organization

used by the HDF5 file format, which consists of linked blocks of items, such as a variable, attribute, or metadata. Each logical variable in an HDF5 file (and therefore, also the NetCDF cases studied in this chapter) is linearized into a single blob by reconstructing from all of the pieces provided by the distributed processes in question into a contiguous, in order format. The format uses a standard byte-ordering of the logical space as if walking all of the dimensions, in order, in nested loops.

The current version of NetCDF has replaced the underlying file format with HDF5. It addresses the metadata and data layout limitation limits of the NetCDF3 format while taking advantage of the extensive efforts to build a portable full functionality API and file format by the HDF5 group. Additionally, the API is nearly identical to the older NetCDF3 API making porting of applications to the new API a much simpler task that switching to HDF5. The extensive support for both the NetCDF3 and HDF5-based file format employed by NetCDF affords broad tool compatibility for a portable data format. To our benefit for this evaluation, the relatively simple NetCDF API is a fully optimized set of calls to the HDF5 API, thus avoiding any bias based on inappropriate or inefficient use of the HDF5 API.

The Lustre file system used in all of our evaluations is configured to use 160 storage targets for all files, which is the maximal level of parallelism allowed by Lustre. For the ADIOS/BP approach, the stripe size is adjusted to 4 MiB automatically. For the NetCDF setup, the stripe size is set to the default 1 MiB. The impact this stripe size has on the performance is discussed in Section 4.5.

4.4 Experimental Evaluation

The checkpoint/restart tests are performed on the Cray XT4 partition of Jaguar at ORNL. The XT4 partition contains 7,832 compute nodes in addition to dedicated login/service nodes. Each compute node contains a quad-core AMD Opteron 1354 (Budapest) processor running at 2.1 GHz, 8 GiB of DDR2-800 memory (some nodes use DDR2-667 memory), and a SeaStar2 router. The resulting partition contains 31,328 processing cores, more than 62 TiB of memory, over 600 TiB of disk space, and a peak performance of 263 teraflop/s

(263 trillion floating point operations per second). The SeaStar2 router has a peak bandwidth of 45.6 GiB/s. The routers are connected in a 3D torus topology, which provides an interconnect with very high bandwidth, low latency, and extreme scalability. The output is striped across all 144 Lustre storage targets and using a stripe size of 1 MiB.

The other 5 series of tests are performed on the petascale partition of the Jaguar machine, known as JaguarPF, at Oak Ridge National Laboratory. This Cray XT5 partition contains 18,688 compute nodes in addition to dedicated login/service nodes. Each compute node contains dual hex-core AMD Opteron 2435 (Istanbul) processors running at 2.6GHz, 16 GiB of DDR2-800 memory, and a SeaStar 2+ router. The resulting partition contains 224,256 processing cores, more than 300 TiB of memory, over 6 PiB of disk space, and a peak performance of 2.3 petaflop/s. For all tests, Spider, the ORNL shared scratch space Lustre file system, is employed. The peak IO performance for Spider from JaguarPF is 60-90 GiB/sec for writing.

To evaluate the impact of the log-based data organization of data districts compared with the contiguous data organizations of other formats, the five remaining analysis reading patterns described in Section 4.2 are evaluated over a 3-D domain, using both a 2-D and a 3-D domain decomposition. Pixie3D [16], an MHD fusion code, has a very similar data model to S3D, but has multiple sizes for the data models, yielding small (32 cubes), medium (64 cubes), large (128 cubes), and extra large (256 cubes) data. As a broader evaluation, these four data models are tested for the 3-D domain decomposition. In accordance with that fact, the per process total output data sizes employed are about 13 MiB for 2-D, 2 MiB for 3-D small, 16 MiB for 3-D medium, 128 MiB for 3-D large, and 1 GiB for 3-D extra large. The per process variable sizes are quite different. For the 2-D domain decomposition, it is a mere 2400 bytes. For the 3-D case, it is 256 KiB, 2 MiB, 16 MiB, and 128 MiB respectively. These data sizes are used to generate output files using both 7168 processes and 16384 writing processes.

Tests are deliberately designed to avoid bias. In particular, while the use of 16K processes is advantageous in terms of running at a larger scale, those runs could be perceived

as biased in that these tests can more easily 'hit' process boundaries for reading. For example, if the sub area read neatly falls on the natural boundaries created when the data was written, fewer reads would be required to retrieve the data. This would result in unfairly benefitting the log-structured approach. Our response is to run additional tests with 7K processes as an alternative size, which yields reads that will not easily 'hit' the even process boundaries. For reading, operations are split evenly among the reader processes.

First, the checkpoint/restarts are evaluated. To test the restart read performance, various process counts from 128 to 2048 are employed to write the data with the same or one-half the number of process used to write to read the data back in. PnetCDF formatted data is compared. For all tests, the best results for a series of four runs for each data point is selected for the BP and PnetCDF performance. The horizontal axis represents the number of writers employed to generate the restart data no matter the number of processes used to read the data back in.

To validate the performance impact at various scales, two sets of tests are run. The first set of 'large scale' results is a series of eight runs ranging from 512 process to 4096 processes, at 512 process increments. These tests are each run at least 5 times with the arithmetic means of the elapsed time being shown on the graphs. These steps are chosen for two reasons. First, for S3D, the typical analysis execution is on no fewer than one-fourth of the prior simulation run's process count. Second, at the supercomputing centers, analysis clusters of 512 to 1024 cores are becoming much more common. As this trend continues, 4096 cores for an analysis cluster will soon be common. The second set of 'small scale' experiments examine various process counts ranging from four to 512, in increments of 4 processes. Each of these five tests are run a minimum of 5 times each. The arithmetic means of the resulting times are used for comparisons. This experiment set represents (1) an initial exploration scenario before a more extensive, long-term data analysis run is performed, and (2) analyses performed on small data sets. Tests are run for all cases where sufficient local memory is available. For example, it is not possible to read a 2 TiB variable into 4 processes on the machine tested. The elapsed wall clock time, in seconds, is measured from the opening of the file, through the read operation(s), to the end of closing the file for

the slowest process of the readers.

The location of the planar areas within the 3-D domain is selected to be in the middle for each dimension. For the sub-planes, the same location is selected, but the plane is bounded to one-quarter the size of the plane and centered. That is, the sub-plane boundary is located half-way between the edge and center for each side. The sub-area selection similarly selects a rectangular area bounded by planes half-way between the edge and the center.

For fairness, care is taken to ensure that none of the read patterns provide significant advantages to the log-based format employed by ADIOS. For example, selecting only areas where the data districts map exactly to the process boundaries would give an advantage to the log-based format because the data can more easily be selected. Further, to ensure that no inadvertent advantage is gained based on the in-memory data and/or file data layout, any planar or linear selection of data is performed multiple times, once in each dimension in each test case. For example, when reading a plane, 3 planes are read – one each in X, Y, and Z. The total time for all three reads are used for the results. This approach controls for the on-disk layout and any reorganization required to return the selected data.

Since the aggregate data sizes are consistent across all reads, they are summarized in Tables 5 and 6. Briefly, based on how the domain is constructed for Chimera, the data sizes are very modest, even for the 16K process run. Chimera's complexity comes from the number of variables used rather than the sizes individual variables. This large count guarantees that few, if any, of the data payloads will fall on the beginning of a stripe boundary. For the S3D application, even the smallest sizes yield variables that reach 1.75 GiB.

Pattern	7192	16384
2	16.4 MiB	37.5 MiB
3	49.2 MiB	112.5 MiB
4	468.5 KiB	728 KiB
5	2.05 MiB	4.68 MiB
6	117.1 KiB	182 KiB

Table 5: 2-D Data Sizes Read for Each Analysis Pattern

While this chapter does not evaluate write performance, we note that there was a set

	Small		Medium		Large		Extra Large	
Pattern	7192	16384	7192	16384	7192	16384	7192	16384
2	1.75 GiB	4 GiB	14 GiB	32 GiB	112 GiB	256 GiB	896 GiB	2048 GiB
3	5.25 GiB	12 GiB	42 GiB	96 GiB	336 GiB	768 GiB	2688 GiB	6244 GiB
4	9 MiB	16 MiB	36 MiB	64 MiB	144 MiB	256 MiB	576 MiB	1 GiB
5	224 MiB	512 MiB	1.75 GiB	4 GiB	14 GiB	32 GiB	112 GiB	256 GiB
6	2.25 MiB	4 MiB	9 MiB	16 MiB	36 MiB	64 MiB	144 MiB	256 MiB

Table 6: 3-D Data Sizes Read for Each Analysis Pattern

of tests we could not complete, for the 3-D domain decompositions. For the 2-D domain decomposition, the ADIOS/BP writing tasks took less than 10 minutes to complete, but the NetCDF tasks took nearly 1 hour to complete. The results motivate the use of log-structured checkpoint formats in ADIOS and PLFS. For the 3-D domain decomposition, for ADIOS/BP, all eight data files took less than 90 minutes to complete. This includes the 1 GiB-per-process-by-16384 process extra large case. For NetCDF, just the 7192 process cases took nearly 11 hours. For the 16384 process case, for just the small, medium, and large tasks, NetCDF took nearly 4 hours. The extra large case for 16384 processes did not complete in 24 hours, the upper limit available to the authors for running tests on the Jaguar machine. By estimating the amount of data in the partial output when the task was terminated, another 90 minutes was likely to be necessary for this single output to complete!

The detailed evaluation is split into three parts. The first part examines the checkpoint/restart performance as this is typically performed on some similar number of processes to what wrote the data initially. The next two parts look at large scale and small scale analysis reading results, respectively. The large scale results look at process counts from 512 up to 4096 for reading while the small scale examine from 4 through 508 processes for reading. In both of the latter parts, the 7K and 16K write test files are used as the data source.

4.4.1 Checkpoint/Restart Results

This first evaluation shows the checkpoint/restart results. First are the uniform restart results followed by different ratios of writing to reading processes.

4.4.1.1 Pixie3D Uniform Restarts

To establish a baseline, a uniform restart is tested. This is using the same number of processes to read the restart as wrote it originally. Figure 23 shows the performance results. For large data, the performance approaches the IOR benchmark performance [98] for the machine.

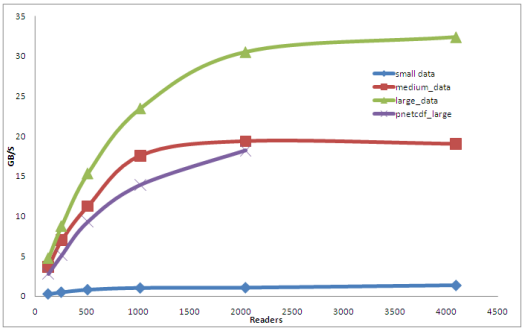


Figure 23: Uniform Reads for the Pixie3D data

Pixie3D Small For the small data, good read performance just cannot be attained no matter the number of processes used to read or write the data. The overall data size is too small to overcome the inherent overhead of the parallel file system. Figure 24 shows the performance for reading the restart output on half as many processes as wrote the restart. The horizontal axis represents the number of processes that wrote the data originally.

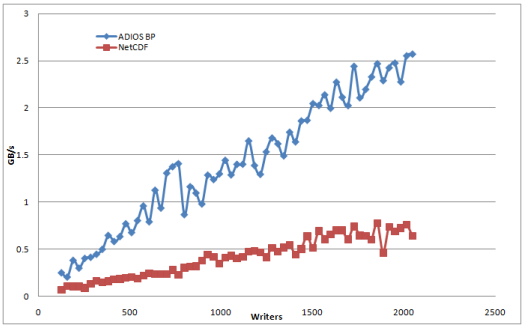


Figure 24: Small Model, Half Process Count

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

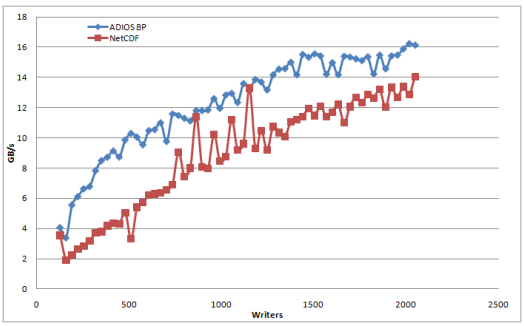


Figure 25: Medium Model, Half Process Count

Pixie3D Medium Restarts using the medium data model can achieve much better performance, but still cannot achieve more than a fraction of the theoretical maximum performance for the system. Figure 25 shows the performance for restarting using half as many processes.

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

Pixie3D Large The large data cases finally reach the maximum general performance seen for applications in production use. Figure 26 shows the performance for using half as many processes to restart.

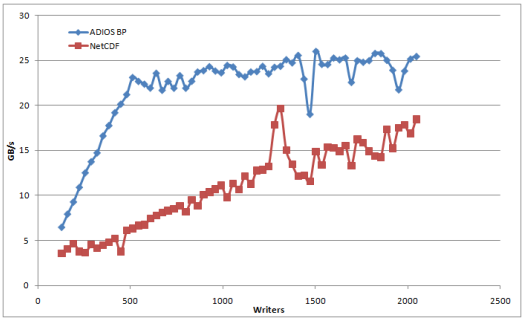


Figure 26: Large Model, Half Process Count

4.4.1.2 Discussion

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

uniform read rate. Small data is about the same performance. We also tested ADIOS read performance on additional domain decompositions, reading back with different numbers of readers (including restarts involving more reading processes than writers). In most cases we were 2× faster compared to reading directly from PnetCDF, and read speed was always superior on non-uniform restarts.

An additional set of tests using a non-integer factor difference between the writers and readers yields similar results. For example, using 64 writers and 80 readers represents moving from a 4×4×4 setup to 4×4×5. For small data, BP is consistently 2× faster compared with PnetCDF, for medium, it is 20% faster, and for large data, the performance is essentially identical.

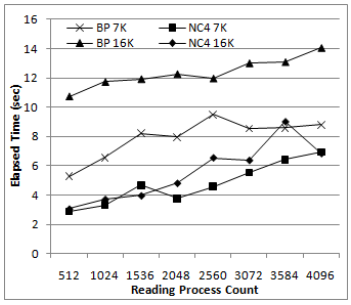
4.4.2 Large Scale Results

Large scale tests examine the performance of the data district-based decomposition compared with a canonical format for common analysis read patterns at representative process counts, again using ADIOS and NetCDF, respectively. Figure 27 shows the 2-D domain decomposition results. The NetCDF approach has less elapsed time for essentially all tests. The explanation for this performance relates to the domain decomposition itself. This Chimera example has very little data on a per process basis, relying on a thin 'pencil' of data. This results in a data domain that consists of 300 doubles, a mere 2400 bytes. When reading the planar areas or sub areas, all 7K or 16K data districts must be read in when using the BP format for the plane across the domain decomposition, demonstrating the need for additional optimizations for BP. Another contributor to the advantages observed for NetCDF is that HDF5 has buffering and caching for read operations that are capable of reading the entire variable into memory once, if it can fit, and then distributes it using messaging. This test case also demonstrates why the contiguous format was favored in the past: because 2-D simulations and analysis were more prevalent, but unfortunately, this is no longer the case, as 3-D simulations now largely prevail.

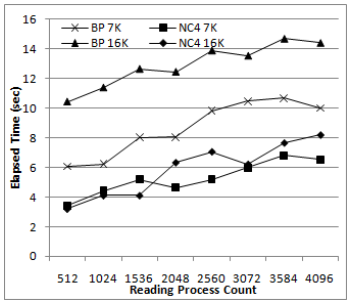
The notion and use of data districts directly reflect the importance of 3-D domain decompositions. Specifically, while 2-D domain decompositions divide data into full extent

'pencils' of the entire domain, 3-D domain decompositions divide the domain into blocks and assemble them in a 3-D pattern to yield the entire simulation space, motivating the 'dimensional' or spatial nature of data districts. As process counts increase with a corresponding growth in the simulation space, fewer codes will be able to use a 2-D domain decomposition due to memory limitations. For these cases, a 3-D domain decomposition will be required.

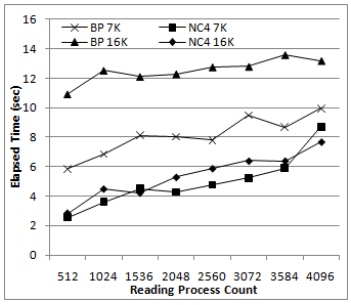
Figures 28, 29, 30 and 31 show the results for the 3-D experiments. Overall, data districts show superior performance for the 3-D domain decomposition for all data sizes, process counts, and tests. Again, by splitting the data into 'dimensional' chunks instead of using a single contiguous logical layout, fewer, larger reads can be performed to retrieve the data for all of the different patterns. Instead of performing many very small reads to obtain a plane, only the chunks that contain the planar pieces will be read, whereupon relevant data is extracted using in-memory operations.



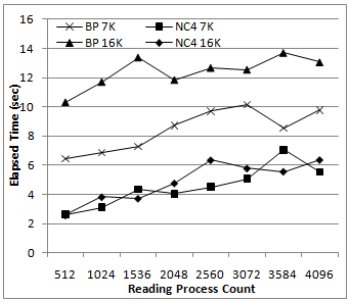
(a) One Variable



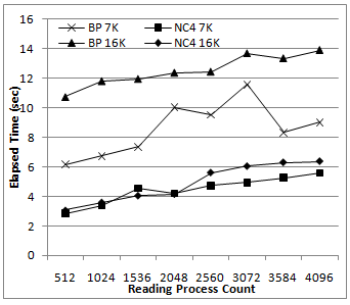
(b) Three Variables



(c) Full Planes

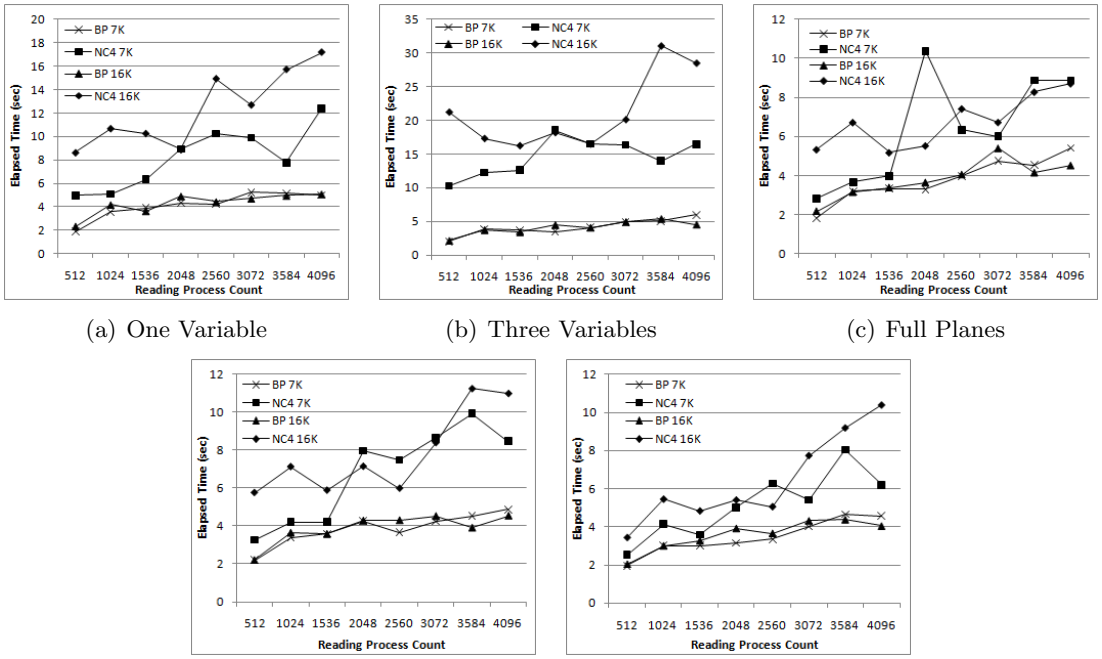


(d) Sub Area



(e) Sub Planes

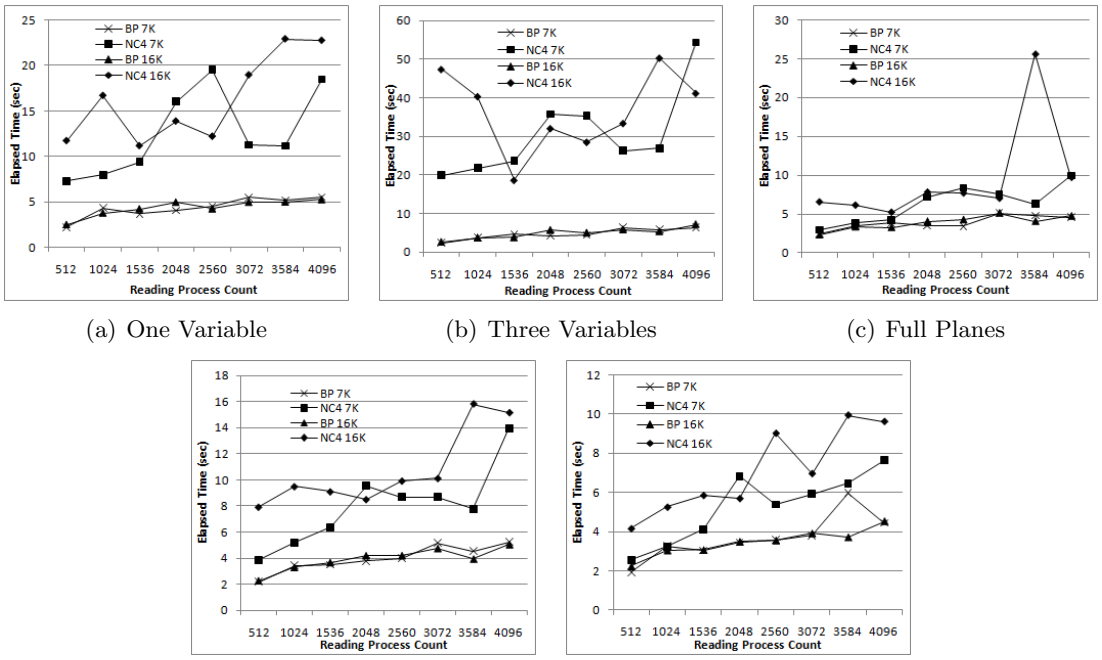
Figure 27: 2-D Large Scale Reading Performance



(a) One Variable (b) Three Variables (c) Full Planes

(d) Sub Area (e) Sub Planes

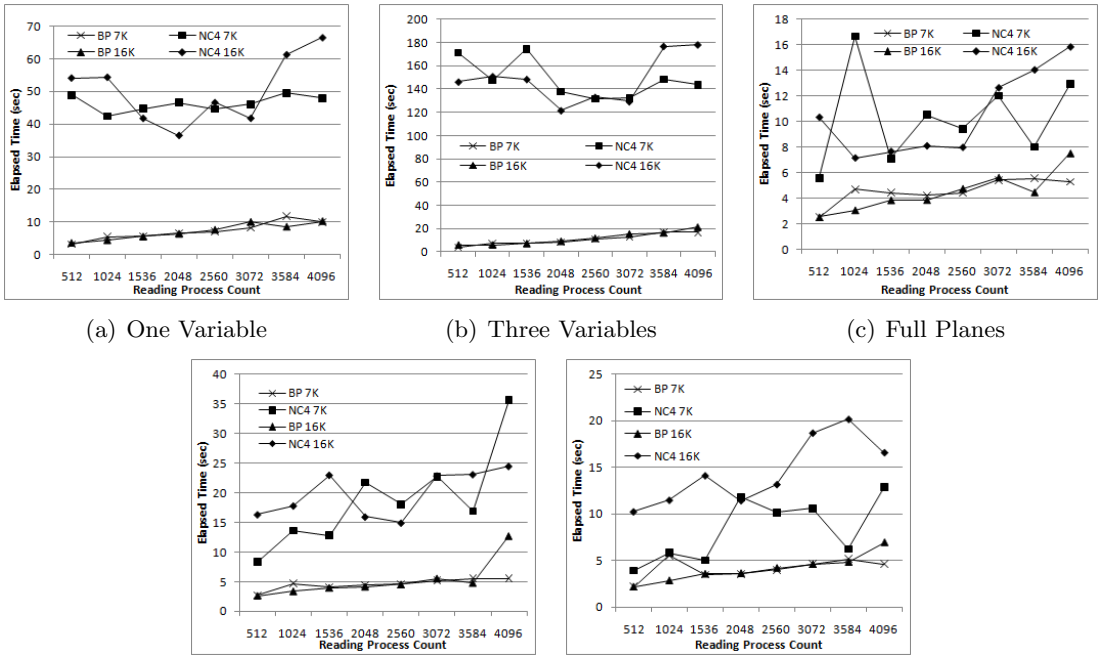
Figure 28: 3-D Small Model Large Scale Reading Performance



(a) One Variable (b) Three Variables (c) Full Planes

(d) Sub Area (e) Sub Planes

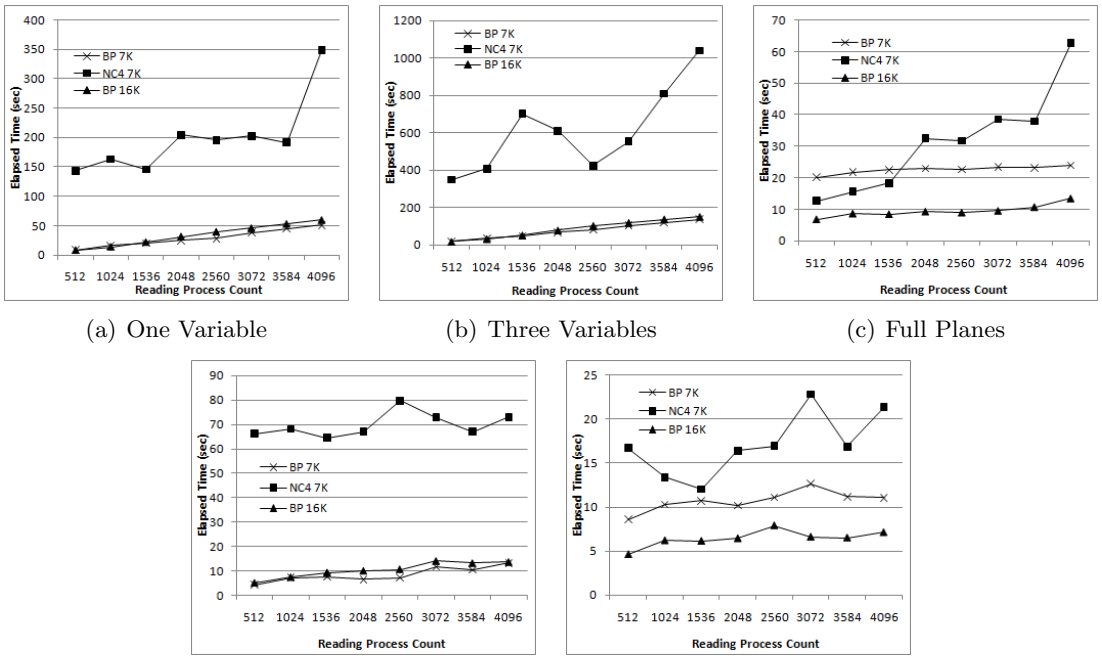
Figure 29: 3-D Medium Model Large Scale Reading Performance



(a) One Variable (b) Three Variables (c) Full Planes

(d) Sub Area (e) Sub Planes

Figure 30: 3-D Large Model Large Scale Reading Performance



(a) One Variable (b) Three Variables (c) Full Planes

(d) Sub Area (e) Sub Planes

Figure 31: 3-D Extra Large Model Large Scale Reading Performance

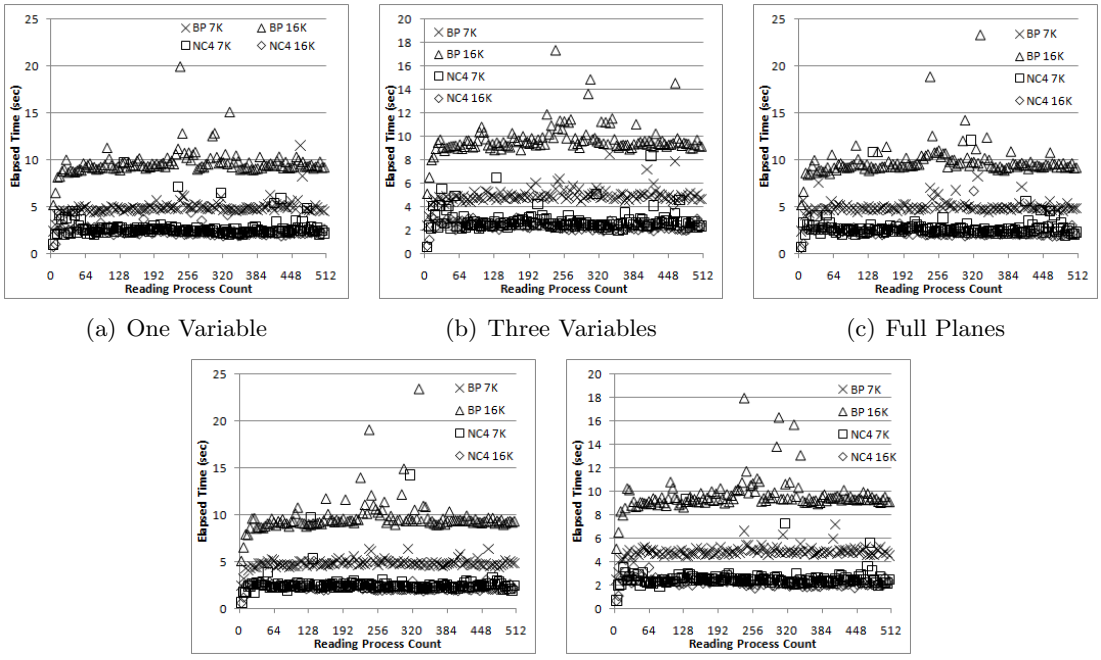
4.4.3 Small Scale Results

For the 2-D domain decomposition, the small scale results mirror those attained with the large scale tests. For the 7K processes case, the performance of ADIOS/BP is between a factor of 2× to 3× worse than that seen with the NetCDF format. For the 16K processes case, the performance worsens to a factor between 4× and 5× worse.

At a more detailed level, the 2-D results break down as follows. For all tests NetCDF performance is essentially identical for both the 7K processes and 16K processes data sets for all process counts. The BP 7K processes set is 2× worse; the 16K processes set is 4× worse.

For the 3-D domain decomposition results, there is a bit more variation. For test 2, the BP performance for both 7K and 16K are essentially identical for all process counts. The performance is essentially identical for the small, medium, and large cases suggesting the observed performance is a minimum time required to perform this test. The much larger data size for the extra large test measures out to the peak performance for the file system at around 8 seconds to read all of the data. While NetCDF has worse performance in all cases, it is only slightly worse for small scale, with the performance degrading progressively as the data size increases. For test 3, the performance for BP follows the same characteristics with the small, medium, and large cases having essentially identical performance around 3.5 seconds with the extra large taking longer at around 13 seconds. The NetCDF performance again is worse in all cases. For the small data model, it is roughly 3× worse and performance progressively degrades from there. Test 4 is more interesting. For the small and medium data models, both NetCDF and BP are essentially the same performance. For the large data model, NetCDF is 2× worse than the BP models. For the extra large model, BP has worse performance by a factor of about 1.6×. If the more detailed data were available, it is likely the additional reads for the pieces is the problem. Do note that as the number of readers increases, the time for BP stays nearly flat while the NetCDF time grows passing BP at 2048 processes and continues to grow rapidly from there. Test 5 shows similar characteristics to tests 2 and 3. The performance for BP is essentially constant for small, medium, and large data sets with a larger time for extra large. NetCDF starts nearly the same for small

data and grows progressively worse as the data size increases. Test 6 is more interesting. For the small, medium, and large cases, the performance for BP and NetCDF is essentially identical. For extra large, the BP 7K performance is about 1.1× worse than NetCDF. The 16K processes performance for BP is better than both at about 0.6× as much time. As is the case with test 4, as the number of readers scales, the performance reverses. In this case, it happens much sooner at 512 processes and the NetCDF performance degrades more slowly as reading processes are added.



(a) One Variable (b) Three Variables (c) Full Planes

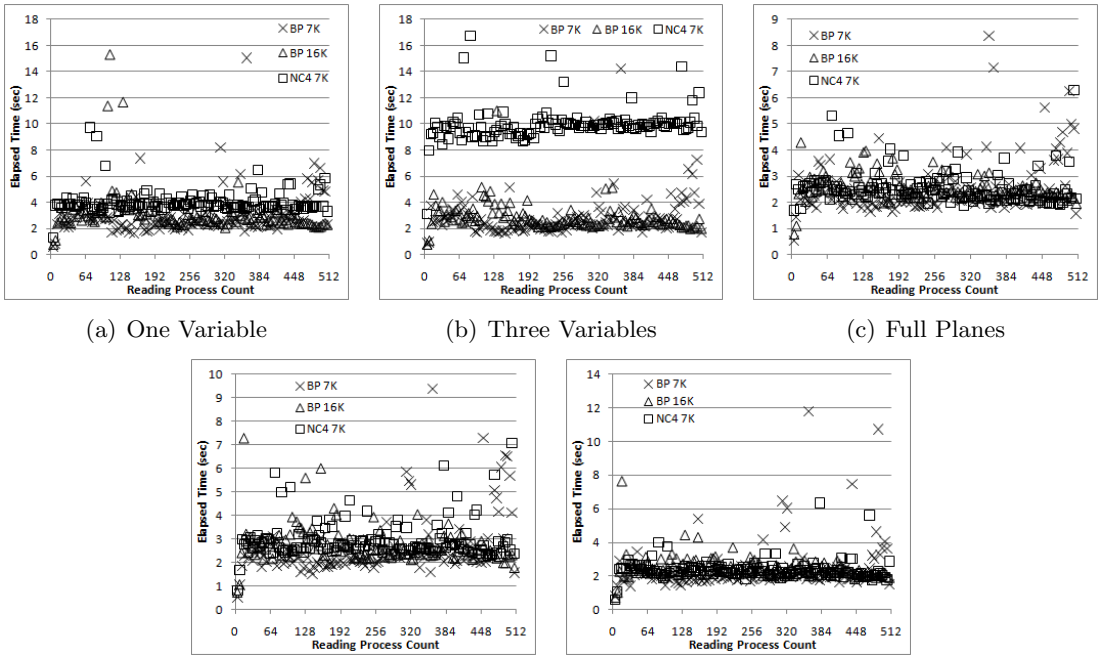
(d) Sub Area (e) Sub Planes

Figure 32: 2-D Small Scale Reading Performance

4.5 Detailed Analysis and Discussion

Examination of the performance for both the large scale and small scale experiments, for the 2-D and 3-D domain decompositions, sheds light on important considerations for both writing and reading performance. To recap the physical distribution of the data sets on disk, for all tests, the output is striped across 160 storage targets. For NetCDF, the stripe size is the default 1 MiB while it is adjusted automatically by ADIOS to 4 MiB.

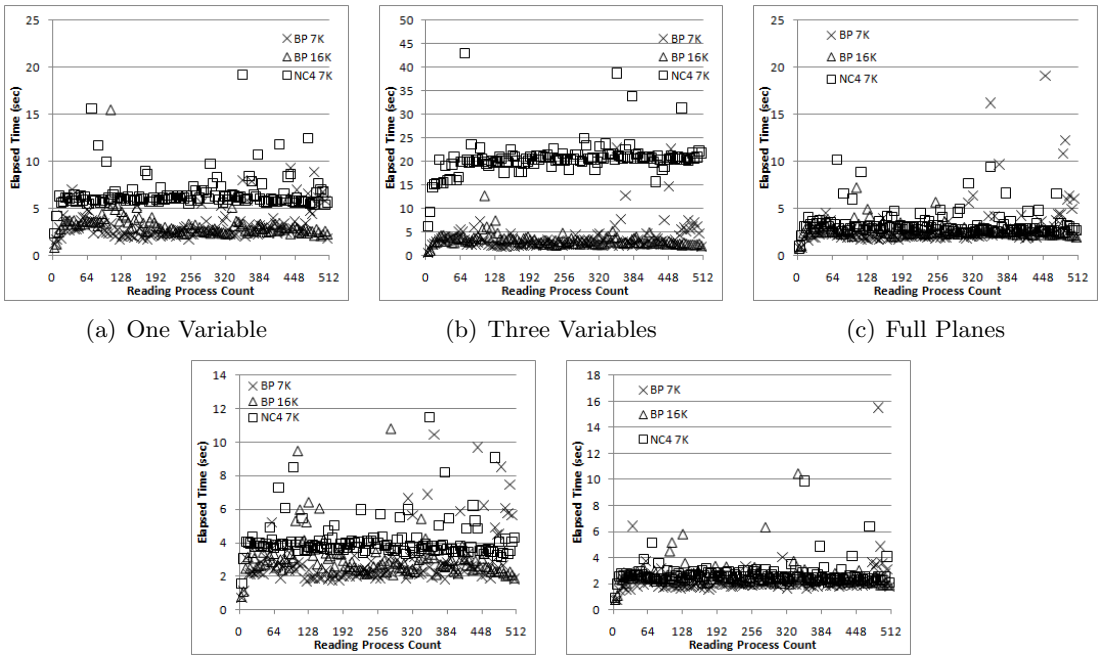
Concurrency is critical for high IO performance. Two factors affect the impact of concurrency on IO performance. First, the amount of data located on a single storage target



(a) One Variable (b) Three Variables (c) Full Planes

(d) Sub Area (e) Sub Planes

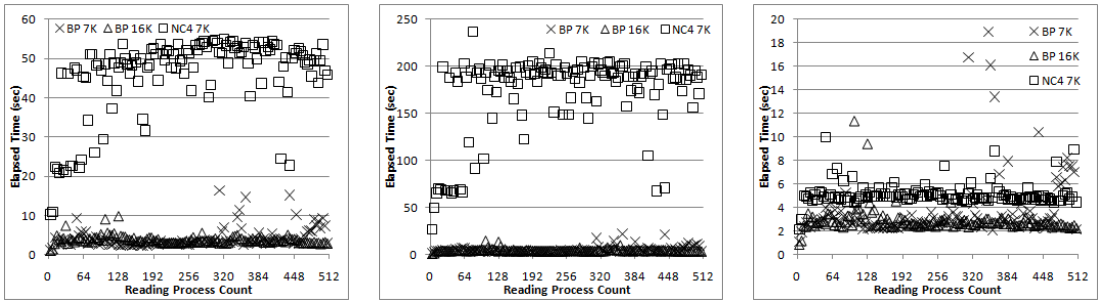
Figure 33: 3-D Small Model Small Scale Reading Performance



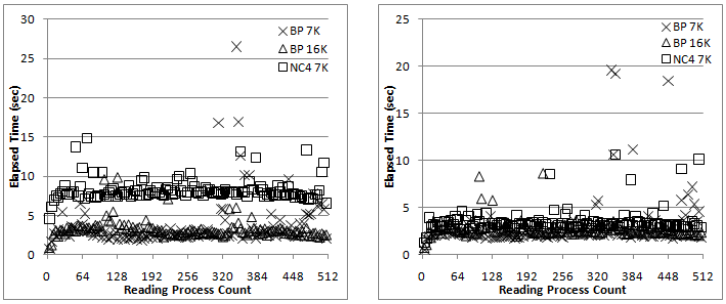
(a) One Variable (b) Three Variables (c) Full Planes

(d) Sub Area (e) Sub Planes

Figure 34: 3-D Medium Model Small Scale Reading Performance

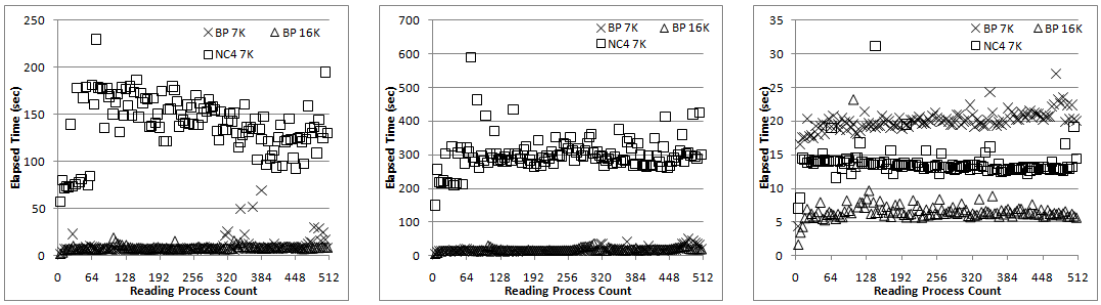


(a) One Variable (b) Three Variables (c) Full Planes

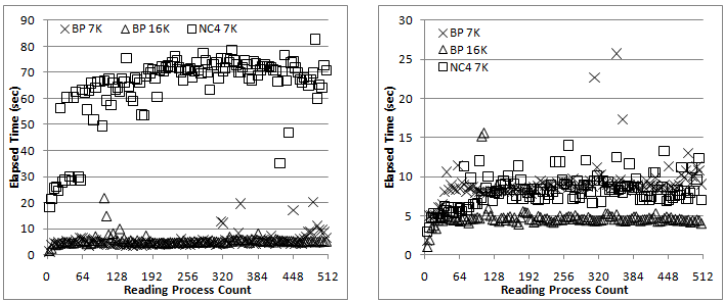


(d) Sub Area (e) Sub Planes

Figure 35: 3-D Large Model Small Scale Reading Performance



(a) One Variable (b) Three Variables (c) Full Planes



(d) Sub Area (e) Sub Planes

Figure 36: 3-D Extra Small Model Large Scale Reading Performance

determines how much data can be accessed at one time. Wider spread of data implies more concurrency in access, but of course, each storage target can only service one client at a time. The goal, therefore, is to balance these two factors so that the largest number of storage targets can be employed for reading at one time, without overloading any target with too many requests that ultimately are serviced serially.

For the 2-D domain decomposition, the processes are arranged in a 112×64 space for 7K processes and 128×128 for 16K processes. For the 3-D domain decomposition, for all per-process data sizes, a 28×16×16 distribution is used for the 7K process runs and a 32×32×16 distribution is used for the 16K process runs. For the 2-D, 7K processes case for ADIOS/BP, the full plane of data test reads three planes: 112 data districts, 64 data districts, and 7192 districts (one set for each dimension). For the 3-D 7K process runs for ADIOS/BP, the full plane of data test reads the three planes: 448 (28×16) data districts, 256 (16×16) data districts, and 448 (16×28) data districts. For the 2-D domain decomposition, reading 7192 blocks from 160 storage targets overwhelms the IO system resulting in poor performance. NetCDF can read all of the data at once and then distribute to the other processes. This one large read, even if it is done serially from a single storage target, is faster than the 7192 reads from all available storage targets. For the 3-D domain decomposition, the use of data districts reduces the number of blocks read to a manageable number across all patterns, resulting in more consistent performance. Because NetCDF spreads the data according to the logical global array dimensions, this results in either many very small reads or in larger reads with poor relevant data density. In essence, the performance penalty seen by ADIOS/BP for 2-D is seen by NetCDF for 3-D, for similar reasons.

The stripe size also plays a role. For the 2-D case, ADIOS/BP only spreads the output from a process across 4 storage targets compared with NetCDF's 13. The reduced concurrency forced on ADIOS further impacts performance. For the 3-D case, again, the opposite is true. The size of the variables causes them to be striped across all of the storage targets. By using a larger stripe size, the number of requests to a single storage target is sharply reduced, which improves response time.

Separating considerations of performance from portability. Performance is strongly

linked to file layout. By carefully sizing data according to inherent buffer sizes and considering the total data size to avoid interference, better performance can be achieved.

For the 2-D case, the failure of addressing the inherent buffer sizes, such as the 1 MiB chunk used in many disk caching systems, incur the full overhead of locating the data for only a small amount of time spent reading it. In particular, because only 2400 bytes for any variable is written contiguously, the most data that can be retrieved when reading a data district is 2400 bytes. Conversely, by storing the entire variable contiguously, NetCDF's single read pays the disk overhead only once (or a small number of times, if it is striped). In contrast, for the 3-D case, until the data per data district becomes enormous (1 GiB per data district), all reading patterns are the same or superior for the log-based approach. Even at the enormous size, once the process count grows, the advantage returns to the log-based approach. The advantages for writing have been clear for a while. With this study, the advantages for reading, for an HPC and parallel file system environment, are apparent as well.

Importance of 'natural' file organizations. Sequential read patterns for scientific data do occur, but they are not the only patterns to consider. Scientific discoveries occur as the data is explored through analysis tools that use select portions of the data. More precisely, data is retrieved and analyzed in regions determined by the types of analysis being done rather than as an entire data set. By arranging the data in smaller blocks – as data districts – increased efficiency can be attained for retrieving the data. In other words, it is no longer necessary to span the entire data storage, skipping large areas of unrequested data, just to get to another few bytes of relevant data.

For the 2-D domain decomposition, each data district contains just 2400 bytes (300 doubles). For reads that stay within those 300 doubles, performance is good. Once a neighbor in the other two dimensions is requested, however, one or more read operation(s) from somewhere else in storage are required. Unless all of the data can be read into memory and parceled out through messaging, many reads are required even for retrieving a small small region. Therefore, for the 3-D domain decomposition with its local dimensions of at least 32×32×32 (the small data model), far more relevant data can be read in a single

operation than in the contiguous layout. This natural organization is a key benefit of the data districts approach. In contrast, with NetCDF, traversing any dimension except for one will require seeking to different areas of the file to read a small amount of data.

4.6 Related Work

Log-based approaches have been used for transactional systems, such as databases [32] for decades. The performance advantages for write-intensive tasks is well documented. More recently, log based formats have been used in general file systems. Early studies, such as the log-structured file system [88, 95, 113], demonstrated the write performance advantage for many scenarios, but all could suffer from penalties for particular patterns of reads. In other words, random write performance was greatly enhanced by employing the log-structured format, but sequential reads suffered. This is not the case with parallel file systems when files are striped across many storage targets. Here, the disk head movement penalty of sequential reads is greatly reduced due to the number of devices that can work independently for each read operation. This means that parallelism in storage systems warrants a re-examination of previous assumptions, particularly for the disk usage patterns seen for petascale science.

Based on the knowledge of the reading penalties for log-structured formats, self-describing data formats like HDF5 [35] and NetCDF [71] were developed. They offer a balanced approach to write and read performance optimization that work well for a variety of situations. The scale introduced by petascale science, however, has revealed limitations of these approaches. By coordinating across the writing processes to achieve a canonical format, writing times suffer. As further demonstrated in this chapter, the reading times can also be impacted at scale. This means that some of the optimizations introduced by these formats can become a liability rather than an advantage. On the other hand, such formats will remain important in terms of portability and their use by third-party tools, warranting investigations of efficient methods for 'on demand' format conversion. In a similar spirit, the SciDB initiative [102] is attempting to provide self-describing data formats for scientific applications through a database, though it is attempting to specify a common API compatibility rather than specifying on-disk layout and is hence, orthogonal to techniques such

as ADIOS.

The efficient access of multidimensional scientific data has been considered at the compiler level as well, usually in the form of intelligent prefetching optimizations [68]. These approaches are orthogonal to our optimizations, however, as they are designed around the movement of data between main memory and the processor cache, whereas we operate at the level of retrieving data from the parallel filesystem.

Other approaches to addressing the mismatch between the output organization and the read pattern needs of analysis codes include the use of a staging area to host data reorganization and pre-analysis routines. Both the synchronous data staging [76] and the IO Forwarding Software Layer [6] effectively manage the writing time spent by an application, through aggregating such requests and thereby partially managing the resulting impact on the storage system. However, such work has not taken advantage of staging areas to accelerate subsequent data use for analysis or other reading tasks. Complementary work pursued in the PreDatA [123] project is to reduce the need for or sizes of subsequent reads, by pre-analyzing data to the extent possible. The costs of the additional resources used are mitigated through write time reductions and improved data preparation for later analysis operations.

The utility of data districts is motivated in part by previous work [92] that has shown that multi-dimensional data may be mapped efficiently to modern disk drives, in ways that preserve spatial locality across multiple dimensions. Here, in addition to data being read efficiently and sequentially in a single dimension, intelligent placement allows other dimensions to be accessed with low positioning costs. Although our goals are similar (efficient access of data across multiple dimensions), our approach is orthogonal and compatible: we operate at the level of the parallel file system, whereas prior work is implemented using the firmware below the device level (albeit exposed to the application).

Zazen [109] from D.E. Shaw Research tries to improve end to end IO performance by bypassing a remote, parallel file system such as Lustre. Instead, Zazen caches simulation data as a series of small files across the multiple disks of an ancillary analysis cluster attached to the high performance machine via a network. The approach yields excellent performance

for appropriate analysis workloads (in particular, molecular dynamic simulations that map well to a time series of small files), but does not consider the simultaneous attainment of high write performance. Further, its architectural assumptions differ in that there is no parallel file system that is attached to the petascale machine.

4.7 Conclusions and Future Work

This chapter documents the general performance advantage, at scale, for the BP file format. The optimizations made for writing have not negatively impacted the performance for reading except in the small data case. For those cases, as is demonstrated in Chapter 2, the file conversion time is reasonable. For these cases, converting to an alternative format for the analysis operations still offers performance advantages. For all others, the optimizations made to improve the write portion of the write-read cycle have also served to help the read portion.

As HPC has moved through terascale into petascale computing, formerly efficient approaches for encoding data and performing IO must be revisited. In particular, the 3-D domain decompositions used by petascale simulation codes demand new approaches in how data is organized and formatted for efficient storage on the parallel file systems used with petascale machines. This chapter describes a log-based approach to data storage that organizes data as 'dimensional' data chunks, termed data districts, that can be efficiently and concurrently written to and read from the many storage targets such systems employ. The outcome is high end to end, i.e., write and read, performance for log-based data organizations, in addition to notable improvements over the performance observed for contiguous encodings used by the standard file formats in current use, such as HDF5 and NetCDF, excepting only cases of 2-D domain decompositions with small variable sizes.

Future work with log-based formats like data districts should develop additional optimizations for 2-D domain decomposition, such as the use of aggregation networks to reduce the number of log records being read. Also of interest is the exploration of trade-offs for achieving improved read performance without unduly impacting that of writing, as partly addressed by data sieving [107]. Also of interest is a study of the impact of alternative in

memory data layout, in row major vs. column major order, on both writing and reading performance. Finally, it would be useful to investigate automated methods for data conversion and/or replication based on 'downstream' usage requirements, with methods that use asynchronous IO coupled with in-flight data manipulation and data staging.

Current work by the HDF Group into using a storage format similar to the data districts used in the ADIOS/BP and PLFS log-based formats has shown excellent writing performance. This chapter's insights should help drive those optimization efforts.

CHAPTER V

PREVIOUS WORK

While the issues identified in this thesis for extreme scale computing are recent, managing IO at scale and IO portability have been addressed in the past, with different parts of the problem space addressed by file systems and IO libraries.

File systems have evolved in two different directions to address storage at scale. First, distributed and network file systems, such as NFS [73], XFS [7], and Ceph [116] aim to provide a single namespace and transparent access to a collection of storage devices existing on the network. These storage devices may or may not be attached to a single machine. By collecting all of the storage devices into a virtual single space, the ability to access and store quantities of data beyond those capable of being hosted by a single device becomes possible. Based on standard file system semantics, a restriction is that each file is written by only one client at a time. As a result, it is not possible for two different processes to both open a single file for writing and then each store data at a different offsets in the file. A notable attribute of these systems and in keeping with the enterprise systems and applications for which they are intended is their excellent performance for typical individual file read/write patterns, due to efficient support for file block read-ahead and caching.

The second development is parallel file systems as demonstrated with systems like Lustre [14], Panasas [82], and GPFS [93]. Parallel file systems permit multi-process applications to simultaneously write data to a single file. Further, they enable distribution of a single file across multiple storage devices for aggregate bandwith gains. Certain limitations of these general purpose file systems prevent them from fully addressing the needs of extreme scale computing. First, all of these file systems must manage the general case for a variety of applications without unduly penalizing any particular access pattern. While this does achieve good general performance, it does not address the need to achieve extreme performance in bursts as is required for the extreme scale computing environment. This also means that

these file systems all support a POSIX compliant interface for broad compatibility. This restriction forces decisions that negatively affect performance. For example, the requirement to enforce consistency semantics prevents avoiding these costs for applications that either implicitly or explicitly manage the consistency directly. The second major limitation is due to the functionality of the metadata layer. While the metadata management layer has some knowledge of the distribution of processes, it does not have any knowledge of the data sizes or data types. Without this additional knowledge the file systems are not able to manage the data effectively. Third, while distributing files so that the expected data distribution does not overload any single storage device, it does not address the dynamic performance requirements of extreme scale applications. Because distribution of data is a static decision made at file creation time, the system is not capable of redistributing the data to different locations within the global storage system dynamically to address transient and severe performance problems.

Researchers have developed experimental systems that address some of the limitations of the general purpose file systems described above. The lightweight file systems (LWFS) [79] project at Sandia National Laboratories strips down the semantics and services to only authentication and authorization. All other services must be provided in a layer on top affording a custom file system build using reusable components. This approach works well to avoid the bottlenecks associated with say, POSIX compliance, but additional functionality, like that described in this thesis, is needed to deal with say, managing multiple client processes accessing the same storage target at the same time. In other words, LWFS provides some of the base functionality on which the solutions described in this thesis can be built.

An entirely different approach to high performance IO is pursued by the Google File System (GFS) [30], which assumes the global file system will consist of very large numbers of inexpensive servers and disks regularly having unrecoverable failures somewhere in the distributed file system. Similar to our work, however, are the facts that (1) GFS also eliminates support for some features like POSIX compliance to gain performance, and (2) GFS also attempts to optimize performance for the usage patterns for which it is constructed, which

concern the acquisition and evaluation of patterns found in large-scale web traces. Since such data is never discarded, append operations are the normal state with essentially no updates in general usage. By taking advantage of this access pattern, GFS achieves higher performance for applications exhibiting this pattern. With the file system and application layer codesigned together, high bandwidth operations are rotuinely achieved. The deployment environment is not latency sensitive affording sacrifices to achieve higher bandwidth to accelerate the movement of the large data chunks into main memory for processing. With the assumptions of constant failures, the metadata server does not keep full track of the location of all of the chunks and replicas for files. Locating all of the chunks for a file may introduce latency for operations, but will not impact the bandwidth once the data movement begins. This architecture is well suited to the data processing environment at Google and other data processing intensive sites, but it does not address the latency sensitive HPC environment where both bandwidth and latency are important considerations.

IO libraries come in different forms providing different levels of abstraction and differing support for data management and interaction with the storage system. At a lower level, MPI-IO provides a common interface for a collection of parallel IO optimizations. Techniques like disk directed IO [44], grouping [77], and collective buffering [13] are the basis for the MPI-IO layer optimizations and have helped grow IO system performance for teraflop systems. With the advent of petascale and development of exascale computing, these techniques can still help, but they do not address the full complexity of the problem at these larger scales. Extra communication and coordination required to perform collective buffering ultimately dominates the IO time. When running an S3D IO kernel for an extra-large data model of 1 GiB per process, representative of a hybrid MPI/OpenMP run typical of many multi-core application runs, running at 16384 processes requires more than 24 hours to perform a single output operation using these techniques (see Chapter 4). These outcomes demand rethinking the approaches. The techniques of scheduling IO operations on storage targets is still quite important [57, 60], but the extra overhead introduced by coordinating all of the processes can be counterproductive at scale. The inclusion of the ADIO layer to insert optimizations for particular file systems helps improve scalability, but

it does not address the end-to-end problem. While many sophisticated features for storing and reading complex data types are available, the extra work required to use these features is a likely cause for the relatively poor adoption by HPC application. In spite of the inherently changable approach of the ADIO layer, the lack of adoption of the full data type system prevents the system from taking full advantage of the optimizations.

At a higher level, APIs like HDF-5 [35], NetCDF4 [71], and P-NetCDF [72] force the user to provide additional details about the IO operations' data types, sizes, and distribution. While this could afford significant opportunities for optimization, these systems have ceded control of the lower level IO to either MPI-IO or standard POSIX calls. This disconnect worked well as long as the techniques embraced by MPI-IO scaled. With the ceiling for scalability being reached, these APIs suffer from the same limitations.

Overall, neither at the file system level nor at the IO library level has any system nor any combination of existing tools adequately addressed the needs of extreme scale data management.

A different approach for extreme scale data management works from the idea that data is too large and/or it is too costly to move to secondary storage. For example, if the analysis output of an HPC application is 2 TiB and needs to be output every 10 minutes, but the file system is only capable of achieving 10 GiB/sec, the application will spend about one-third of the time writing analysis output. To better manage this data volume it is possible to move the data to a staging area either asynchronously [34] or synchronously [76, 6]. The movement to the staging area can address some of the mismatch between the compute area and the storage area ultimately accelerating IO performance. By carefully managing how the IO is scheduled [3], the interference between IO and intra-application communication can be managed reducing the wall clock time overhead of IO to nearly 0. This rapid data movement assumes that it is both reasonable and possible to move the data volumes to appropriate storage resources. This may not always be the case. For cases where the data is too large or when sufficient time is available during the compute phase between output operations, the real benefit of this approach can be achieved. The introduction of in-line data processing operations for filtering, sorting, or

data analysis preparation tasks can improve both the writing *and* and later analysis reading performance of HPC applications [123]. By placing embarassingly parallel operations close to the compute source and communication-required or communication-intensive tasks nearer to storage where there are fewer processes involved, the amount of time spent performing IO is reduced, even when inclduing the additional resources required for staging are included. These staging-style approaches operate at different layers than the approach described in this thesis. First, they operate at a lower layer in that they directly manage data movement from the compute area to a secondary, in memory location for either further processing and/or more efficient movement to storage. The approach outlined in this thesis operates at a higher layer in that it assumes the actual transport of data will be efficient and instead focuses on scheduling the movement operations for the correct time. The asynchronous approaches, such as DataTap, also schedule the data movement, but they do not move directly to storage avoiding the interference and contention issues of the storage subsystem. Second, when in-line operations can be deployed, these staging-style approaches operate at a higher level. The introduction of and flexible placement of processing operations is beyond the scope of the work in this thesis. While the staging approaches can work quite well for achieving good IO performance and introduce additional functionality that actually reduces the cost of IO while enhancing the usefulness of the data, they are not a universal solution. First, to achieve many of the goals described above, asynchronous data movement is required. This may not be desireable for a particular application due to failure patterns and/or the cost of regenerating the data. Second, the data generated will require additional memory to store the data while it is being spooled to secondary storage. The additional memory requirement may not be possible given the platform characteristics and/or the application needs. Third, the amount of compute power required to perform the in-line operations may exceed the available excess capacity on the compute resources. And fourth, even when moving data to a staging area to reduce the IO time, it is still critically important to properly manage IO for optimal performance from the staging area to the storage resources. In spite of these limitations, these staging-related approaches will become increasingly important as multi-core architectures and hybrid CPU/GPU nodes

are introduced. The staging areas may become on-node locations that manage the data movement out of the node while the rest of the compute cores, either CPU, GPU, or both, continues computation.

CHAPTER VI

CONCLUSION AND OPEN ISSUES

6.1 Conclusion

This thesis addresses data management for extreme scale in a high performance computing environment. Data management is not simply writing data with good performance, but instead encompasses writing, annotation, and subsequent reading of data. The development of the ADIOS middleware enables programmer productivity through a simple API and more consistent performance as HPC platforms evolve through the replacement of the underlying transport method without requiring any source code changes. The BP file format is designed for high performance in large process count environments by minimizing coordination operations and focusing on a central index of file contents for later data discovery and locating. By using a middleware approach, ADIOS can leverage additional metadata during output about both the data types, sizes, and distribution, but also the dynamic state of the storage system. This metadata affords intelligent decisions and management of how to organize and perform the output to ensure consistent high performance output. While the output performance advantages are clear, they either benefit or do not penalize subsequent read performance.

During output, by leveraging the embarassingly parallel capabilities of the compute area to annotate data, subsequent reading operations are greatly enhanced. The ability to quickly identify if data values within an output set exceed a threshold becomes trivial. The benefits are not limited to this annotation capability. The BP file format has demonstrated, particularly for extreme scale configurations, that reading operations can be accelerated by as much as a factor of 6. These benefits show that it is possible to both optimize the output operations with techniques that would perhaps counterintuitively also aid later read performance. The key to these advantages is the parallel file system.

The large number of storage devices that can be used in parallel spread the output and

therefore the data subsequently read across more devices in moderately sized chunks. This distribution aids reading by adding additional parallel sources to increase the aggregate bandwidth.

While these advantages clearly demonstrate the advantages for the write-read cycle, other advanges are also realized. The ADIOS middleware has additional features specifically aimed at enhancing the write-read cycle. First, with special calls to aid asynchronous IO, additional mixed IO/computation models are possible. For example, by employing asynchronous IO and scheduling the movement of data to avoid interfering with communication activities [2], extra resources can be used for other operations. These additional resources incur no net increase in the wall clock time spent by the application due to the reduced IO time from the asynchronous IO. By using these resources, operations intended to reorganize, filter, or apply computation to data to accelerate later analysis becomes feasible [123]. Through the use of these Preparatory Data Analytics, data can be sorted or annotated to aid later analysis reads. While this approach can yield great results, the techniques described to accelerate writing, including the output formatting are still important. Ultimately, if the data is written to storage, these techniques are completely relevant.

Second, the ability to specify more than one transport method for an output operation affords the opportunity for special purpose transport methods that trigger disk-based workflows. By placing the workflow transport last in the XML file, the output operations will complete for all transport operations before the 'trigger' transport is invoked. At that time, it is safe to message a workflow system to indicate that a new batch of data is available for processing. This reduces the delay in processing data as it is generated and avoids the metadata server impact of constantly querying for a list of files and the sizes to determine when a new, complete data set has been written. This workflow triggering approach has been demonstrated successfully [83, 22].

Through the ADIOS API, optimized output techniques, a writing *and* reading friendly intermediate file format called BP, and facilities for incorporating 'in flight' data processing operations and the mecahnisms for triggering workflows all yeild a comprehensive approach for addressing the write-read cycle for scientific application in HPC environments.

6.2 Open Issues

While the work presented in this thesis encapsulates a complete system, several open issues are apparent.

First, the output mechanisms and focus are based around a file-oriented structure. Investigations into how to better organize, store, annotate, and move the tremendous data volumes of extreme scale science are becoming increasing important. The traditional POSIX-based file worldview is increasingly overwhelmed. To complement the storage and access research, additional emarassingly parallel statistical metrics should be considered to further aid reading performance. New techniques using partial solutions that can quickly be resolved at read time can greatly benefit applications. For example, arithematic mean of an array cannot be performed without communication. By each writing process computing a local sum and/or mean value, the reading processes can then assemble these pieces together to compute a global mean value without requiring the communication during the output phase. Other statistical measures should also be investigated.

Second, the increasing number of compute cores through the introduction of multi-core CPUs and GPGPUs will lead to increasing output operations. In particular, the acceleration offered by GPGPUs will require far more frequent output operations in order for scientificly valid analysis be performed. These developments will require more advanced output processing to reduce the data volumes just to maintain output performance and scientific validity of the generated data.

Third, solid state storage devices, such as flash memory devices, introduce another layer into the memory hierarchy offering additional opportunities for processing data 'in transit' or simply to stage the data to slower storage. This change requires rethinking the output process from simply writing to storage to considering moving to a temporary area from which the data is either spooled to slower storage or additional operations can be applied to the data before moving to secondary storage.

Fourth, the resilience of the output format has been addressed partially by this thesis. There are more areas to investigate. For example, fully investigating transactional techniques for controlling the acceptance of data sets as they are moved into storage or even the

annotation of partial data sets and the recovery of the pieces successfully saved can lead to reduced losses of compute time due to data set corruption or loss. And finally, the 'in transit' data operations are really a precursor for deep analysis operations to gain scientific insights without having to move all of the simulation data to storage. The data can be analyzed as part of the output only storing data that meets certain scientifically interesting criteria. This helps avoid the storage interface bottleneck while maintaining the standard notion of 'writing' data for later analysis common in scientific applications. Another class of these 'in transit' operations apply to more complex code coupling operations where richer, two-way interactions among a collection of codes require data adjustments to fit the differing models and scales for each code. The requirement to better support these sorts of interfaces and environments such that the applications can change what sort of environment they are currently executing within more transparently. For example, the ADIOS API currently supports changing the destination of an write operation by changing an entry in the XML file. This allows an application to write either to storage or potentially to an in-memory coupled code transparent to the writing application. On the reading, or receiving side, it is possible to switch between reading from a file or from another source, but this change requires minor adjustments to the application code to ensure proper operation. There is also a lack of clear support for blocking either synchronously or asynchronously on reading to better support an in-memory coupled environment. Extreme scale HPC applications require the flexibility to run in either an isolated environment, a disk-based workflow system, or using an in memory coupling scenario. Changing these operating mechanisms should not require any source code changes, including when restarting from a failure or to continue operating when a failure occurs in a tighter coupling environment. These additional topics will extend this work into a more comprehensive extreme scale data management system and will certainly generate additional topics as they are explored.

APPENDIX A

ADIOS API

In addition to the APIs mentioned below, others exist for reading and some other operations. Another entire set of APIs exist for transport method implementers to make that job easier. Neither of these additional sets of functions are described here. For more information, please refer to http://adiosapi.org/.

A.0.0.1 Setup/Cleanup/Main Loop

```
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` () initiates parsing of the configuration file generating all of the internal data type information, configures the mechanisms for each, and potentially sets up the buffer. Buffer creation can be delayed until a subsequent call to `adios_allocate_buffer` if it should be based on a percentage of memory free or other allocation-time sensitive

considerations.

`Adios_begin_calculation` () and `adios_end_calculation` () provide the 'ticker' mechanism for asynchronous IO, providing the asynchronous IO mechanism with information about the compute phases, so that IO can be performed at times when the application is not engaged in communications.

`Adios_end_iteration` () is a pacing function designed to give feedback to asynchronous IO for gauging what progress must be made with data transmission in order to keep up with the code. For example, if a restart is written every 40 iterations, the XML file may indicate an iteration count of 30 to evacuate the data to give some adjustment for storage congestion or other issues.

`Adios_finalize` () indicates the code is about to shut down and any asynchronous operations need to complete. It will block until all of the data has been drained from the compute node.

A.0.0.2 Write Operation

```
adios_open (&handle, "restart", "filename", mode, comm)
adios_group_size (handle, group_size, \&totalsize)
adios_write (handle, "comm", comm)
...
adios_write (handle, "zion", zion)
...
adios_write (handle, "mzeta", mzeta)
...
adios_close (handle)
```

`Adios_open` () generates a handle that manages the transport specific information and serves to collect the data buffers used for writing or reading.

`Adios_group_size` () tells ADIOS how large the data written by this process will be at maximum. This is used to determine file offset calculation and coordination.

`Adios_write` () specifies for a given name what data buffer to use. If it is writing a scalar value, the value is copied enabling the use of expressions as parameters to this call. If it is an array with statically defined dimensions, it can resolve directly the size involved. If it has dynamic dimension elements, those must be defined before the call to `adios_close` in order for the write to succeed.

`Adios_close` () performs three purposes. First, it indicates that all of the data buffers for either writing or reading have been provided. Second, it initiates either the write or read operation. Finally, it indicates to the transport layer to close the connection. By delaying the reading or writing until this point, we eliminate the complexity of processing data values in exactly the same order for reading, writing, or as they are specified in the XML file.

A.1 ADIOS XML Details

The main elements of the XML file format are of the format `<element-name attr1 attr2 ...>` Most of the attributes share a common definition and are therefore collected at the end of the section for brevity. The description below is structured like the XML document:

```
<adios-config>
 <adios-group name>
  <global-bounds dimensions offset>
   <var name path type dimensions/>
  </global-bounds>

  <var name path type dimensions/>

  <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>
```

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.
- `var` - a variable that is either an array or a primitive data type, such as integer or float, depending on the attributes provided.
- `attribute` - attributes attached to a var or var path.
- `transport` - mapping a transport method to a data type including any initialization parameters.
- `buffer` - internal buffer sizing and creation time. Used only once.

Attributes

- `name` - name of this element or attribute. For a datatype, this is used in the code to select this data type for an IO operation.
- `path` - HDF-5-style path for the element or path to the HDF-5 group or data item to which this attribute is attached.
- `type` - data type. Currently supported values (size): `byte` (1-byte), `integer` (4-byte), `integer*4` (4-byte), `integer*8` (8-byte), `long` (8-byte), `real` (4-byte), `real*8` (8-byte), `double` (8-byte), `complex` (16-byte), and `string`.
- `dimensions` - a comma separated list of numbers and/or names that correspond to scalar elements to determine the size of this item

- `value` - value for the attribute
- `priority` - [optional] a numeric priority for the IO methods to better schedule this write with others that may be pending currently
- `method` - a string indicating a transport method to use with the associated adios-group.
- `iterations` - [optional] a number of iterations between writes of this type used to gauge how quickly this data should be evacuated from the compute node
- `base-path` - [optional] the root directory to use when writing to disk or similar purposes
- `group` - corresponds to an adios-group specified earlier in the file.
- parameters - [optional] a string passed to the method for initialization.
- `size-MB` - the number of MiB to allocate for buffering. Either `size-MB` or `free-memory-percentage` is required.
- `free-memory-percentage` - the percentage of free ram to allocate for buffering. Either `size-MB` or `free-memory-percentage` is required.
- `allocate-time` - either "`now`" or "`oncall`" to indicate when the buffer should be allocated. "`oncall`" will wait until the programmer decides that all memory needed for calculation has been allocated and will then `call adios_allocate_buffer` ()

A.1.0.3 Read Operation

```
adios_set_read_method (method)
adios_fopen (&handle, "filename", comm, \&group_count)
adios_gopen (handle, \&group_handle, "group_name", \&var_count, \&attr_count)
adios_inq_var (group_handle, "path/var_name", \&var_type,
                    &var_rank, \&dims, \&vtimed)
adios_read_var (group_handle, "path/var_name", start, readsize,
                    &buffer, \&read_bytes)
```

```
adios_get_attr (group_handle, "attr_name", \&type, \&size, \&buffer)
adios_glocse (group_handle)
adios_fclose (handle)
```

`Adios_set_read_method` () is an optional call to change the default reading method from POSIX IO to another method, such as code coupling

`Adios_fopen` () opens a BP file for reading

`Adios_gopen` () selects an ADIOS group within a BP file for reading

`Adios_inq_var` () queries the BP file for information about the specified variable

`Adios_read_var` () retrieves the portion of the variable specified

`Adios_get_attr` () retrieves the value of the attribute specified

`Adios_gclose` () closes the reference to a particular group in the BP file

`Adios_fclose` () closes the BP file opened by `adios_fclose` ()

REFERENCES

[1] Abbasi, H., Wolf, M., Schwan, K., Eisenhauer, G., and Hilton, A., "XChange: Coupling parallel applications in a dynamic environment," *Cluster Computing, IEEE International Conference on*, vol. 0, pp. 471–480, 2004.

[2] Abbasi, H., Lofstead, J., Zheng, F., Klasky, S., Schwan, K., and Wolf, M., "Extending I/O through high performance data services," in *Cluster Computing*, (Austin, TX), IEEE International, September 2007.

[3] Abbasi, H., Wolf, M., Eisenhauer, G., Klasky, S., Schwan, K., and Zheng, F., "DataStager: Scalable data staging services for petascale applications," in *HPDC '09: Proceedings of the 18th ACM international symposium on High performance distributed computing*, (New York, NY, USA), pp. 39–48, ACM, 2009.

[4] Abbasi, H., Wolf, M., and Schwan, K., "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*, (Washington, DC, USA), pp. 341–348, IEEE Computer Society, 2007.

[5] Ahrens, J., Law, C., Schroeder, W., Martin, K., and Papka, M., "A parallel approach for efficiently visualizing extremely large, Time-Varying datasets," 2000.

[6] Ali, N., Carns, P. H., Iskra, K., Kimpe, D., Lang, S., Latham, R., Ross, R. B., Ward, L., and Sadayappan, P., "Scalable I/O forwarding framework for high-Performance computing systems," in *CLUSTER*, pp. 1–10, 2009.

[7] Anderson, T. E., Dahlin, M. D., Neefe, J. M., Patterson, D. A., Roselli, D. S., and Wang, R. Y., "Serverless network file systems," *ACM Trans. Comput. Syst.*, vol. 14, no. 1, pp. 41–79, 1996.

[8] Antypas, K. and Uselton, A., "MPI-I/O on franklin XT4 system at NERSC," in *Cray User's Group 2009*, Cray User's Group, 2009.

[9] Axboe, J., "Linux block IO–Present and future," *Proceedings of the Ottawa Linux Symposium 2004*, 2004.

[10] Batsakis, A., Burns, R. C., Kanevsky, A., Lentini, J., and Talpey, T., "CA-NFS: A Congestion-Aware network file system," in *FAST*, pp. 99–110, 2009.

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

[12] Bhatelé, A. and Kalé, L. V., "Quantifying Network Contention on Large Parallel Machines," *Parallel Processing Letters (Special Issue on Large-Scale Parallel Processing)*, vol. 19, no. 4, pp. 553–572, 2009.

[13] Bordawekar, R., del Rosario, J. M., and Choudhary, A. N., "Design and evaluation of primitives for parallel I/O," in *SC*, pp. 452–461, 1993.

[14] Braam, P. J., "Lustre: A scalable high-performance file system," Nov. 2002.

[15] bung Chen, H., Grider, G., and Fields, P., "A Cost-Effective, high bandwidth server I/O network architecture for cluster systems," in *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pp. 1–10, March 2007.

[16] Chacón, L., "A Non-Staggered, Conservative, $\nabla \dot{s} B \rightarrow = 0$, Finite-Volume Scheme for 3D Implicit Extended Magnetohydrodynamics in Curvilinear Geometries," *Computer Physics Communications*, vol. 163, pp. 143–171, Nov. 2004.

[17] Chandra, S., Parashar, M., and Ray, J., "Analyzing the impact of computational heterogeneity on runtime performance of parallel scientific components," in *SpringSim '07: Proceedings of the 2007 spring simulation multiconference*, (San Diego, CA, USA), pp. 395–402, Society for Computer Simulation International, 2007.

[18] Chang, C. S. and Ku, S., "Spontaneous rotation sources in a quiescent tokamak edge plasma," *Physics of Plasmas*, vol. 15, no. 6, p. 062510, 2008.

[19] Chen, J. H., Choudhary, A., de Supinski, B., DeVries, M., Hawkes, E. R., Klasky, S., Liao, W. K., Ma, K. L., Mellor-Crummey, J., Podhorszki, N., Sankaran, R., Shende, S., and Yoo, C. S., "Terascale direct numerical simulations of turbulent combustion using S3D," *Computational Science & Discovery*, vol. 2, no. 1, p. 015001 (31pp), 2009.

[20] Childs, H. R., Brugger, E., Bonnell, K. S., Meredith, J. S., Miller, M., Whitlock, B., and Max, N., "A contract based system for large data visualization," in *IEEE Visualization*, p. 25, 2005.

[21] Choudhary, A., keng Liao, W., Gao, K., Nisar, A., Ross, R., Thakur, R., and Latham, R., "Scalable I/O and analytics," 2009.

[22] Cummings, J., Sim, A., Shoshani, A., Lofstead, J., Schwan, K., Docan, C., Parashar, M., Klasky, S., Podhorszki, N., and Barreto, R., "EFFIS: an End-to-end framework for fusion integrated simulation," in *In Proceedings of The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing (PDP 2010)*, 2010.

[23] Dail, H., Casanova, H., and Berman, F., "A decoupled scheduling approach for the grads program development environment," 2002.

[24] Deelman, E., Singh, G., Su, M.-H., Blythe, J., Gil, Y., Kesselman, C., Mehta, G., Vahi, K., Berriman, G. B., Good, J., Laity, A., Jacob, J. C., and Katz, D. S., "Pegasus: A framework for mapping complex scientific workflows onto distributed systems," *Sci. Program.*, vol. 13, no. 3, pp. 219–237, 2005.

[25] Docan, C., Parashar, M., and Klasky, S., "High Speed Asynchronous Data Transfers on the Cray XT3," tech. rep., Rutgers, The State University of New Jersey, CoRE Building, 96 Frelinghuysen Rd, Piscataway, NJ 08854, May 2007.

[26] Docan, C., Parashar, M., and Klasky, S., "DataSpaces: An interaction and coordination framework for coupled simulation workflows," *HPDC '10: Proceedings of the 18th international symposium on High performance distributed computing*, 2010.

[27] Frings, W., Wolf, F., and Petkov, V., "Scalable massively parallel I/O to task-Local files," in *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, (New York, NY, USA), pp. 1–11, ACM, 2009.

[28] FSIO, H., "http://institutes.lanl.gov/hec-fsio/docs/hec-fsio-fy07-gaps_roadmap.pdf."

[29] Fu, G. Y., Park, W., Strauss, H. R., Breslau, J., Chen, J., Jardin, S., and Sugiyama, L. E., "Global hybrid simulations of energetic particle effects on the n = 1 mode in tokamaks: Internal kink and fishbone instability," *Physics of Plasmas*, vol. 13, no. 5, p. 052517, 2006.

[30] Ghemawat, S., Gobioff, H., and Leung, S.-T., "The google file system," *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 29–43, 2003.

[31] Gosink, L., Shalf, J., Stockinger, K., Wu, K., and Bethel, W., "HDF5-Fastquery: Accelerating complex queries on HDF datasets using fast bitmap indices," in *In SSDBM*, pp. 149–158, 2006.

[32] Gray, J., McJones, P., Blasgen, M., Lindsay, B., Lorie, R., Price, T., Putzolu, F., and Traiger, I., "The recovery manager of the system r database manager," *ACM Comput. Surv.*, vol. 13, no. 2, pp. 223–242, 1981.

[33] Gulati, A. and Varman, P. J., "Scheduling multiple flows on parallel disks," in *HiPC*, pp. 477–487, 2005.

[34] Hasan Abbasi, Matthew Wolf, K. S., "LIVE data workspace: A flexible, dynamic and extensible platform for petascale applications," in *Cluster Computing*, (Austin, TX), IEEE International, September 2007.

[35] HDF5, "http://hdf.ncsa.uiuc.edu/products/hdf5/."

[36] http://www.mkomo.com/cost-per gigabyte, "HDD cost per GB graph 1980-2009."

[37] Hua, Y., Jiang, H., Zhu, Y., Feng, D., and Tian, L., "SmartStore: A new metadata organization paradigm with semantic-Awareness for next-Generation file systems," in *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, (New York, NY, USA), pp. 1–12, ACM, 2009.

[38] in Binary, G., "http://www.sesp.cse.clrc.ac.uk/publications/data-management/report/node12.html."

[39] Indices, H.-. P., "http://www.hdfgroup.uiuc.edu/rfc/hdf5 /hdf5indexing/pis.html."

[40] Interface, M. P., "MPI-2: Extensions to the message-passing interface," 1996.

[41] Iyer, S. and Druschel, P., "Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O," *SIGOPS Oper. Syst. Rev.*, vol. 35, no. 5, pp. 117–130, 2001.

[42] Kasick, M. P., Tan, J., Gandhi, R., and Narasimhan, P., "Black-Box problem diagnosis in parallel file systems," in *FAST*, 2010.

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

[44] Kotz, D., "Disk-directed I/O for MIMD multiprocessors," tech. rep., Hanover, NH, USA, 1994.

[45] Kramer, W. T. C. and Ryan, C., "Performance variability of highly parallel architectures," in *International Conference on Computational Science*, pp. 560–569, 2003.

[46] Kumar, V., Cai, Z., Cooper, B. F., Eisenhauer, G., Schwan, K., Mansour, M., Seshasayee, B., and Widener, P., "Implementing diverse messaging models with Self-Managing properties using IFLOW," in *ICAC '06: Proceedings of the 2006 IEEE International Conference on Autonomic Computing*, (Washington, DC, USA), pp. 243–252, IEEE Computer Society, 2006.

[47] Kutare, M., Eisenhauer, G., Wang, C., Schwan, K., Talwar, V., and Wolf, M., "Monalytics: Online monitoring and analytics for managing large scale data centers," *International Conference on Autonomic Computing (ICAC)*, June 2010.

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

[49] Latham, R., Ross, R., and Thakur, R., "The impact of file systems on MPI-IO scalability," in *Proceedings of EuroPVM/MPI 2004*, September 2004.

[50] Leung, A., Adams, I., and Miller, E. L., "Magellan: A searchable metadata architecture for large-scale file systems," Tech. Rep. UCSC-SSRC-09-07, University of California, Santa Cruz, Nov. 2009.

[51] Leung, A., Parker-Wood, A., and Miller, E. L., "Copernicus: A scalable, High-Performance semantic file system," Tech. Rep. UCSC-SSRC-09-06, University of California, Santa Cruz, Oct. 2009.

[52] Leung, A., Shao, M., Bisson, T., Pasupathy, S., and Miller, E. L., "Spyglass: Fast, scalable metadata search for large-scale storage systems," Feb. 2009.

[53] Li, J., keng Liao, W., Choudhary, A., Ross, R., Thakur, R., and Latham, R., "Parallel netCDF: A scientific High-Performance I/O interface," 2003.

[54] Li, K., Naughton, J. F., and Plank, J. S., "Real-Time, concurrent checkpoint for parallel programs," *SIGPLAN Not.*, vol. 25, no. 3, pp. 79–88, 1990.

[55] Liu, B., Rangaswami, R., and Dimitrijevic, Z., "Stream combination: Adaptive IO scheduling for streaming servers," *SIGBED Rev.*, vol. 3, no. 1, pp. 23–28, 2006.

[56] Lofstead, J., Klasky, S., Booth, M., Abbasi, H., Zheng, F., Wolf, M., and Schwan, K., "Petascale IO using the adaptable IO system," Cray User's Group, 2009.

[57] Lofstead, J., Zheng, F., Liu, Q., Klasky, S., Oldfield, R., Kordenbrock, T., Schwan, K., and Wolf, M., "Managing variability in the IO performance of petascale storage systems," in *SC '10: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, (New York, NY, USA), ACM, 2010.

[58] Lofstead, J., Klasky, S., Schwan, K., Podhorszki, N., and Jin, C., "Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS)," in *CLADE 2008 at HPDC*, (Boston, Massachusetts), ACM, June 2008.

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

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

[61] Ludäscher, B., Altintas, I., Berkley, C., Higgins, D., Jaeger, E., Jones, M., Lee, E. A., Tao, J., and Zhao, Y., "Scientific workflow management and the kepler system: Research articles," *Concurr. Comput. : Pract. Exper.*, vol. 18, no. 10, pp. 1039–1065, 2006.

[62] Ludäscher, B., Altintas, I., Berkley, C., Higgins, D., Jaeger, E., Jones, M., Lee, E. A., Tao, J., and Zhao, Y., "Scientific workflow management and the kepler system: Research articles," *Concurr. Comput. : Pract. Exper.*, vol. 18, no. 10, pp. 1039–1065, 2006.

[63] Malewicz, G., Foster, I., Rosenberg, A., and Wilde, M., "A tool for prioritizing DAGMan jobs and its evaluation," *High Performance Distributed Computing, 2006 15th IEEE International Symposium on*, pp. 156–168, 0-0 2006.

[64] Massalin, H. and Pu, C., "Fine-Grain adaptive scheduling using feedback," *Computing Systems*, vol. 3, pp. 139–173, 1990.

[65] Messer, O. E. B., Bruenn, S. W., Blondin, J. M., Hix, W. R., Mezzacappa, A., and Dirk, C. J., "Petascale Supernova Simulation with CHIMERA," *Journal of Physics Conference Series*, vol. 78, pp. 012049–+, July 2007.

[66] Miller, E. L., Katz, R. H., and Katz, Y. H., "Analyzing the I/O behavior of supercomputer applications," in *Eleventh IEEE Symposium on Mass Storage Systems*, pp. 51–55, 1991.

[67] Miller, M. C., Reus, J. F., Matzke, R. P., Arrighi, W. J., Schoof, L. A., Hitt, R. T., and Espen, P. K., "Enabling interoperation of high performance, scientific computing applications: Modeling scientific data with the sets & fields (SAF) modeling system," in *International Conference on Computational Science (2)*, pp. 158–170, 2001.

[68] Mowry, T. C., Lam, M. S., and Gupta, A., "Design and evaluation of a compiler algorithm for prefetching," in *ASPLOS-V: Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*, (New York, NY, USA), pp. 62–73, ACM, 1992.

[69] NERSC, "http://www.nersc.gov/nusers/systems/franklin/monitor.php. obtained 24 march 2010.."

[70] NetCDF, "http://epp.eps.nagoya-u.ac.jp/num-analysis/guidec/netcdf-c-14.html."

[71] NetCDF, "http://www.unidata.ucar.edu/software/netcdf/."

[72] netCDF, P., "http://trac.mcs.anl.gov/projects/parallel-netcdf."

[73] NFS, "http://www.ietf.org/rfc/rfc3010.txt."

[74] NFS, P., "http://tools.ietf.org/wg/nfsv4/."

[75] Nisar, A., keng Liao, W., and Choudhary, A. N., "Scaling parallel I/O performance through I/O delegate and caching system," in *SC*, p. 9, 2008.

[76] Nisar, A., Liao, W.-k., and Choudhary, A., "Scaling parallel I/O performance through I/O delegate and caching system," in *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, (Piscataway, NJ, USA), pp. 1–12, IEEE Press, 2008.

[77] Nitzberg, B., "Performance of the iPSC/860 concurrent file system," tech. rep., NASA Ames Research Center, December 1992.

[78] Nowoczynski, P., Stone, N., Yanovich, J., and Sommerfield, J., "Zest checkpoint storage system for large supercomputers," in *Petascale Data Storage Workshop, 2008. PDSW '08. 3rd*, pp. 1–5, Nov. 2008.

[79] Oldfield, R., Ward, L., Riesen, R., Maccabe, A., Widener, P., and Kordenbrock, T., "Lightweight I/O for scientific applications," *Cluster Computing, 2006 IEEE International Conference on*, pp. 1–11, 25-28 Sept. 2006.

[80] Operators, N., "http://nco.sourceforge.net/."

[81] Otoo, E., Olken, F., and Shoshani, A., "Disk cache replacement algorithm for storage resource managers in data grids," in *In: Proc. of the IEEE/ACM SC 2002 Conf. on Supercomputing. Los Alamitos: IEEE Computer Society*, pp. 1–15, 2002.

[82] Panasas, "http://www.panasas.com/."

[83] Podhorszki, N., Klasky, S., Liu, Q., Abbasi, H., Lofstead, J., Schwan, K., Wolf, M., Zheng, F., Docan, C., Parashar, M., and Cummings, J., "Plasma fusion code coupling using scalable I/O services and scientific workflows," in *In Proceedings of The 4th Workshop on Workflows in Support of Large-Scale Science at Supercomputing 2009*, 2009.

[84] Podhorszki, N., Ludaescher, B., and Klasky, S. A., "Workflow automation for processing plasma fusion simulation data," in *WORKS '07: Proceedings of the 2nd Workshop on Workflows in Support of Large-Scale Science*, (New York, NY, USA), pp. 35–44, ACM, 2007.

[85] Polte, M., Simsa, J., Tantisiriroj, W., Gibson, G., Dayal, S., Chainani, M., and Uppugandla, D., "Fast Log-Based concurrent writing of checkpoints," in *Petascale Data Storage Workshop, 2008. PDSW '08. 3rd*, pp. 1–4, Nov. 2008.

[86] Polte, M., Lofstead, J., Bent, J., Gibson, G., Klasky, S., Liu, Q., Parashar, M., Podhorszki, N., Schwan, K., Wingate, M., and Wolf, M., "...and eat it too: High read performance in Write-Optimized HPC I/O middleware file formats," in *In Proceedings of Petascale Data Storage Workshop 2009 at Supercomputing 2009*, 2009.

[87] Results, S. O. G. T., "http://users.nccs.gov/ oral/jagregtests/gtc128.html."

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

[89] Rosenblum, M. and Ousterhout, J. K., "The design and implementation of a Log-Structured file system," *ACM Trans. Comput. Syst.*, vol. 10, no. 1, pp. 26–52, 1992.

[90] Ross, R., Latham, R., Miller, N., and Carns, P., "A Next-Generation parallel file system for linux clusters," January 2004.

[91] Ross, R., Thakur, R., Loewe, W., and Latham, R., "Parallel I/O in practice," in *SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, (New York, NY, USA), p. 216, ACM, 2006.

[92] Schlosser, S. W., Schindler, J., Papadomanolakis, S., Shao, M., Ailamaki, A., Faloutsos, C., and Ganger, G. R., "On multidimensional data and modern disks," in *FAST'05: Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies*, (Berkeley, CA, USA), pp. 17–17, USENIX Association, 2005.

[93] Schmuck, F. and Haskin, R., "GPFS: A Shared-Disk file system for large computing clusters," in *In Proceedings of the 2002 Conference on File and Storage Technologies (FAST*, pp. 231–244, 2002.

[94] Schroeder, B. and Gibson, G. A., "Understanding failures in petascale computers," *Journal of Physics: Conference Series*, vol. 78, p. 012022 (11pp), 2007.

[95] Seltzer, M., Bostic, K., Mckusick, M. K., and Staelin, C., "An implementation of a log-Structured file system for UNIX," in *USENIX'93: Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, (Berkeley, CA, USA), pp. 3–3, USENIX Association, 1993.

[96] Shafeeq, J. F., Flinn, J., Sinnamohideen, S., Tolia, N., and Satyanarayanan, M., "Data staging on untrusted surrogates,"

[97] Shan, H., Antypas, K., and Shalf, J., "Characterizing and predicting the I/O performance of HPC applications using a parameterized synthetic benchmark," in *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, (Piscataway, NJ, USA), pp. 1–12, IEEE Press, 2008.

[98] Shan, H. and Shalf, J., "Using IOR to analyze the I/O performance for HPC platforms," Cray User's Group, 2007.

[99] SILO, "https://wci.llnl.gov/codes/visit/3rd_party/silo.book.pdf."

[100] Simitci, H. and Reed, D. A., "Adaptive disk striping for parallel Input/Output," in *IEEE Symposium on Mass Storage Systems*, pp. 88–102, 1999.

[101] Sinha, R. R. and Winslett, M., "Multi-Resolution bitmap indexes for scientific data," *ACM Trans. Database Syst.*, vol. 32, no. 3, p. 16, 2007.

[102] Stonebraker, M., Becla, J., DeWitt, D. J., Lim, K.-T., Maier, D., Ratzesberger, O., and Zdonik, S. B., "Requirements for science data bases and SciDB," in *CIDR*, 2009.

[103] System, A. A. V., "http://www.avs.com."

[104] System, H. D., "http://www.sesp.cse.clrc.ac.uk/publications/data-management/report/node14.html."

[105] System, P. D., "http://pds.jpl.nasa.gov/documents/sr/index.html."

[106] Tables, H.-. P., "http://www.carabos.com/docs/opsi-indexes.pdf."

[107] Thakur, R., Gropp, W., and Lusk, E., "Data sieving and collective I/O in ROMIO," in *FRONTIERS '99: Proceedings of the The 7th Symposium on the Frontiers of Massively Parallel Computation*, (Washington, DC, USA), p. 182, IEEE Computer Society, 1999.

[108] Tran, N. and Reed, D. A., "Automatic arima time series modeling for adaptive I/O prefetching," *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, no. 4, pp. 362–377, 2004.

[109] Tu, T., Rendleman, C. A., Miller, P. J., Sacerdoti, F. D., Dror, R. O., and Shaw, D. E., "Accelerating parallel analysis of scientific simulation data via zazen," in *FAST'10: Proceedings of the 8thth conference on USENIX Conference on File and Storage Technologies*, pp. 129–142, USENIX Association, 2010.

[110] Unidata, "http://www.hdfgroup.org/projects/netcdf-4/."

[111] Uttamchandani, S., Yin, L., Alvarez, G. A., Palmer, J., and Agha, G. A., "CHAMELEON: A Self-Evolving, Fully-Adaptive resource arbitrator for storage systems," in *USENIX Annual Technical Conference, General Track*, pp. 75–88, 2005.

[112] Visit, "http://www.llnl.gov/visit/home.html."

[113] Wang, R. Y., Anderson, T. E., and Patterson, D. A., "Virtual log based file systems for a programmable disk," in *OSDI '99: Proceedings of the third symposium on Operating systems design and implementation*, (Berkeley, CA, USA), pp. 29–43, USENIX Association, 1999.

[114] Wang, W. X., Lin, Z., Tang, W. M., Lee, W. W., Ethier, S., Lewandowski, J. L. V., Rewoldt, G., Hahm, T. S., and Manickam, J., "Gyro-Kinetic simulation of global turbulent transport properties in tokamak experiments," *Physics of Plasmas*, vol. 13, no. 9, p. 092505, 2006.

[115] Watson, R. W. and Coyne, R. A., "The parallel I/O architecture of the High-Performance storage system (hpss)," in *MSS '95: Proceedings of the 14th IEEE Symposium on Mass Storage Systems*, (Washington, DC, USA), p. 27, IEEE Computer Society, 1995.

[116] Weil, S., Brandt, S. A., Miller, E. L., Long, D. D. E., and Maltzahn, C., "Ceph: A scalable, High-Performance distributed file system," Nov. 2006.

[117] Widener, P. M., Barrick, M., Pullikottil, J., Bridges, P. G., and Maccabe, A. B., "Metabots: A framework for Out-of-Band processing in Large-Scale data flows," in *Proc. 2007 International Conference on Grid Computing (Grid 2007)*, (Austin, Texas), September 2007.

[118] Widener, P. M., Wolf, M., Abbasi, H., Barrick, M., Lofstead, J., Pullikottil, J., Eisenhauer, G., Gavrilovska, A., Klasky, S., Oldfield, R., Bridges, P. G., Maccabe, A. B., and Schwan, K., "Structured streams: Data services for petascale science environments," Tech. Rep. TR-CS-2007-17, University of New Mexico, Albuquerque, NM, November 2007.

[119] www.newegg.com, "Capacity for $100 retrieved on june 25, 2010."

[120] Xing, J., Xiong, J., Sun, N., and Ma, J., "Adaptive and scalable metadata management to support a trillion files," in *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, (New York, NY, USA), pp. 1–11, ACM, 2009.

[121] Yu, W., Vetter, J. S., and Canon, R. S., "OPAL: An Open-Source MPI-IO library over cray XT," *Storage Network Architecture and Parallel I/Os, IEEE International Workshop on*, vol. 0, pp. 41–46, 2007.

[122] Yu, W., Vetter, J., and Oral, H., "Performance characterization and optimization of parallel I/O on the cray XT," *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pp. 1–11, April 2008.

[123] Zheng, F., Abbasi, H., Docan, C., Lofstead, J., Klasky, S., Liu, Q., Parashar, M., Podhorszki, N., Schwan, K., and Wolf, M., "PreDatA - preparatory data analytics on Peta-Scale machines," in *In Proceedings of 24th IEEE International Parallel and Distributed Processing Symposium, April, Atlanta, Georgia*, 2010.

VITA

Contact Information

College of Computing
Georgia Institute of Technology

cell: (678) 447-4111
home: (505) 508-0175
e-mail: jay@lofstead.org
web: http://www.lofstead.org

Research Interests

High Performance Computing, IO, Data Streaming, Large Scale Data Management, Scientific Visualization.

Education

Georgia Institute of Technology, Atlanta, Georgia

Doctor of Philosophy, Computer Science **August 2004 – present**

- Expected graduation date: December 2010
- Advisor: Professor Karsten Schwan

Master of Science, Computer Science **August 2002 – August 2004**

Bachelor of Science, Computer Science, with honor **August 1988 – June 1993**

Honors and Awards

Georgia Institute of Technology, President's Fellowship, Fall 2006 – present

Sandia National Laboratories Excellence in Engineering 5 Year Fellowship, January 2005 – December 2009

Upsilon Pi Epsilon, ACM honor society, August 2003

McKesson Information Systems' Pinnacle Innovation of the Year, February 2001

Selected Publications

Conference Papers

Jay Lofstead, Fang Zheng, Qing Liu, Scott Klasky, Ron Oldfield, Todd Kordenbrock, Karsten Schwan, Matthew Wolf. Managing Variability in the IO Performance of Petascale Storage Systems. In *SC '10: Proceedings of the 2010 ACM/IEEE conference on Supercomputing*, 2010. To appear.

F. Zheng, H. Abbasi, C. Docan, J. Lofstead, S. Klasky, Q. Liu, M. Parashar, N. Podhorszki, K. Schwan, and M. Wolf. Predata - preparatory data analytics on peta-scale machines. In *In Proceedings of 24th IEEE International Parallel and Distributed Processing Symposium, April, Atlanta, Georgia*, 2010.

J. Cummings, A. Sim, A. Shoshani, J. Lofstead, K. Schwan, C. Docan, M. Parashar, S. Klasky, N. Podhorszki, and R. Baretto. EFFIS: an End-to-end Framework for Fusion Integrated Simulation. In *In Proceedings of 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing, February, Pisa, Italy*, 2010.

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

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

Workshop Papers

M. Polte, J. Lofstead, J. Bent, G. Gibson, S. Klasky, Q. Liu, M. Parashar, N. Podhorszki, K. Schwan, M. Wingate, and M. Wolf. ...and eat it too: High read performance in write-optimized hpc i/o middleware file formats. In *In Proceedings of Petascale Data Storage Workshop 2009 at Supercomputing 2009*, 2009.

N. Podhorszki, S. Klasky, Q. Liu, H. Abbasi, J. Lofstead, K. Schwan, M. Wolf, F. Zheng,

C. Docan, M. Parashar, and J. Cummings. Plasma fusion code coupling using scalable I/O services and scientific workflows. In *In The 4th Workshop on Workflows in Support of Large-Scale Science at Supercomputing 2009, November, Portland, OR*, 2009.

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

J. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin. Flexible io and integration for scientific codes through the adaptable io system (adios). In *CLADE 2008 at HPDC*, Boston, Massachusetts, June 2008. ACM.

Non-Refereed Papers

J. Lofstead, Q. Liu, S. Klasky, M. Booth, R. Oldfield, K. Schwan, and M. Wolf. High performance io on busy systems. In *Poster presented at Petascale Data Storage Workshop 2009 at Supercomputing 2009*, 2009.

J. Lofstead, S. Klasky, M. Booth, H. Abbasi, F. Zheng, M. Wolf, and K. Schwan. Petascale io using the adaptable io system. Cray User's Group, 2009.

C. Jin, S. Klasky, S. Hodson, W. Yu, J. Lofstead, H. Abbasi, K. Schwan, M. Wolf, W. Liao, A. Choudhary, M. Parashar, C. Docan, and R. Oldfield. Adaptive io system (adios). Cray User's Group, 2008.

Book Chapters

A. Shoshani and D. Rotem, editors. *Scientific Data Management–Challenges, Existing Technologies, and Deployment*, chapter High Throughput Data Movement, pages 151–180. Chapman and Hall, Boca Raton, FL, 2009. ISBN 9781420069808.

M. Parashar and S. Hariri, editors. *Autonomic Computing: Concepts, Infrastructure, and Applications*, chapter AutoFlow: Autonomic Information Flows for Critical Information Systems. CRC Press, Boca Raton, FL, 2006. ISBN 978-0849393679.

Physics Papers

Z. Lin, Y. Xiao, I. Holod, W. Zhang, W. Deng, S. Klasky, J. Lofstead, C. Kamath, and N. Wichmann. Advanced simulation of electron heat transport in fusion plasmas. *Journal of Physics: Conference Series*, 180:012059 (10pp), 2009.

C. S. Chang, S. Ku, P. Diamond, M. Adams, R. Barreto, Y. Chen, J. Cummings, E. D'Azevedo, G. Dif-Pradalier, S. Ethier, L. Greengard, T. S. Hahm, F. Hinton, D. Keyes, S. Klasky, Z. Lin, J. Lofstead, G. Park, S. Parker, N. Podhorszki, K. Schwan, A. Shoshani, D. Silver, M. Wolf, P. Worley, H. Weitzner, E. Yoon, and D. Zorin. Whole-volume integrated gyrokinetic simulation of plasma turbulence in realistic diverted-tokamak geometry. *Journal of Physics: Conference Series*, 180:012057 (14pp), 2009.

C. S. Chang, S. Klasky, J. Cummings, R. Samtaney, A. Shoshani, L. Sugiyama, D. Keyes, S. Ku, G. Park, S. Parker, N. Podhorszki, H. Strauss, H. Abbasi, M. Adams, R. Barreto, G. Bateman, K. Bennett, Y. Chen, E. D. Azevedo, C. Docan, S. Ethier, E. Feibush, L. Greengard, T. Hahm, F. Hinton, C. Jin, A. Khan, A. Kritz, P. Krsti, T. Lao, W. Lee, Z. Lin, J. Lofstead, P. Mouallem, M. Nagappan, A. Pankin, M. Parashar, M. Pindzola, C. Reinhold, D. Schultz, K. Schwan, D. Silver, A. Sim, D. Stotler, M. Vouk, M. Wolf, H. Weitzner, P. Worley, Y. Xiao, E. Yoon, and D. Zorin. Toward a first-principles integrated simulation of tokamak edge plasmas. *Journal of Physics: Conference Series*, 125:012042 (7pp), 2008.

Professional Experience

Georgia Institute of Technology, Atlanta, Georgia

Graduate Research Assistant **August 2003 – December 2004, January 2010 – August 2010**

Research in IO, data movement, and super computing topics.

Oak Ridge National Laboratory, Oak Ridge, Tennessee

Intern **Summer 2007, 2008, 2009**

Research and Develop IO middleware technology to increase scientist productivity and increase machine usage through reduced time spent performing IO. Development of ADIOS IO middleware.

Sandia National Laboratories, Albuquerque, New Mexico

Intern **Summer 2006, 2008**

Worked with the Storage Research group on IO middleware technology.

Worldspan (now Travelport), Atlanta, Georgia

Intern **May 2004 – May 2006**

Assisted with the development of a high performance database-based approach for data storage and retrieval for calculating flight pricing. Developed a more efficient database update approach for the 1100+ machines. Improved efficiency of the airfare pricing engine enabling the inclusion of international flight information.

McKesson Information Systems, Atlanta, Georgia

Senior Software Engineering Advisor **April 1997 – August 2002**

Developed a professional services organization for building custom solutions for the enterprise web-based portal for hospital information systems.

Chief Architect/Chief Engineer

Primary architect, designer, and developer for the Horizon Portal web framework for hospital information systems. Developed workflow engine for hospital forms processing including a paper trail for audit purposes.

Senior Technical Education Consultant

Part of small team to rearchitect the corporate-wide technology choices for applications sold to customers. Selected and developed teaching materials for C, C++, Visual C++, Microsoft Foundation Classes (MFC), COM development, Java, and web development classes.

Senior Software Engineer/Team Lead

Developed an alerting system for clinical data. It essentially operates as database triggers built outside of the database engine using a portable rules language specific to the healthcare environment (Arden Syntax).

Siemens Energy & Automation, Johnson City, Tennessee

Software Engineer **June 1993 – April 1997**

Developed a front-end for a software-based Programmable Logic Controllers (PLCs). Created a Visual Basic for Applications integration for constructing and automating a factory.

Part of a multi-national team developing multi-lingual development tools for PLCs. Particular responsibilities included the mapping, abstraction, and translation layer between the graphical editors and the compiler input representation and adjusting language-related User Interface complications.

Palmer and Associates, Norcross, Georgia

Undergraduate Cooperative Education **March 1989 – September 1991**

Assisted with the design and development of custom applications for workflow automation and in developing teaching materials for database design and SQL.

Teaching

ADIOS tutorial at SciDAC 2008, Seattle, Washington

Instructor along with Scott Klasky, Chen Jin **July 18, 2008**

HBO & Company/McKesson/HBOC, Atlanta, Georgia

Senior Technical Education Consultant **November 1998 – November 1999**

Courses Taught: C++, Visual C++, and MFC/COM.

Georgia Institute of Technology, Atlanta, Georgia

Instructor **January 1993 – June 1993**

Course Taught: Introduction to the C Programming Language

Teaching Assistant **August 1992 – March 1993**

Courses Taught: Introduction to the C Programming Language, Assembly Language Programming.

Programming

C, C++, Java, MPI, HDF5, NetCDF, ADIOS, Linux shell scripting, Fortran, SQL.