Flexible IO and Integration for Scientific Codes Through The Adaptable IO System (ADIOS)

Jay Lofstead
College of Computing
Georgia Institute of Technology
Atlanta, Georgia
lofstead@cc.gatech.edu

Scott Klasky
Oak Ridge National Laboratories
Oak Ridge, Tennessee
klasky@ornl.gov

Karsten Schwan
College of Computing
Georgia Institute of Technology
Atlanta, Georgia
schwan@cc.gatech.edu

Norbert Podhorszki
Oak Ridge National Laboratories
Oak Ridge, Tennessee
pnorbert@ornl.gov

Chen Jin
Oak Ridge National Laboratories
Oak Ridge, Tennessee
cgj@ornl.gov

ABSTRACT

Scientific codes are all subject to variation in performance depending on the runtime platform and/or configuration, the output writing API employed, and the file system for output. Since changing the IO routines to match the optimal or desired configuration for a given system can be costly in terms of human time and machine resources, the Adaptable IO System provides an API nearly as simple as POSIX IO that also provides developers with the flexibility of selection the optimal IO routines for a given platform, without recompilation. As a side effect, we also gain the ability to transparently integrate more tightly with workflow systems like Kepler and Pegasus and visualization systems like Visit with no runtime impact. We achieve this through our library of highly tuned IO routines and other transport methods selected and configured in an XML file read only at startup. ADIOS-based IO has demonstrated high levels of performance and scalability. For example, we have achieved 20 GB/sec write performance using GTC on the Jaguar Cray XT4 system at Oak Ridge National Labs (about 50% of peak performance). We can change GTC output among MPI-IO synchronous, MPI-IO collective, POSIX IO, no IO (for baseline testing), asynchronous IO using the Georgia Tech DataTap system, and Visit directly for in situ visualization with no changes to the source code. We designed this initial version of ADIOS based on the data requirements of 7 major scientific codes (GTC, Chimera, GTS, XGC1, XGC0, FLASH, and S3D) and have successfully adapted all of them to use ADIOS for all of their IO needs.

Categories and Subject Descriptors

D.4.3 [**Operating Systems**]: File Systems Management—*Access Methods*

CLADE'08, June 23, 2008, Boston, Massachusetts, USA.

General Terms

Design, Documentation, Experimentation, Human Factors, Management, Performance, Reliability, Standardization

Keywords

MPI-IO, HDF-5, visualization, workflow, modular IO

1. INTRODUCTION

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 HDF-5 and parallel netCDF with rich data annotation. The tradeoffs for each of these vary depending on the IO patterns, runtime compute system, and IO system being used [1]. 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.

A second factor driving the need for an approach like the ADIOS API is the explosion in the number of cores in future petascale machines, coupled with the need for ever faster file systems able to service them. 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, varying the degrees of metadata annotation for improved scalability [2]. 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.

A third important factor to consider as IO approaches are being reconsidered is the need for online simulation monitoring to ensure the scientific validity of code executions and to prevent 'useless' or problematic runs. Such monitoring typically implies the need to integrate running simulations with analysis workflows, perhaps using workflow systems like Kepler [3] or Pegasus [4]. 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. 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 Fortran POSIX IO. An external XML configuration file describes the data and how to process it using provided, highly tuned IO routines. More importantly, output can simultaneously use multiple IO implementations, using the concept of 'data grouping' embedded into ADIOS. The idea is to facilitate changing IO methods based on the IO patterns of different IO operations and to make it possible to create "dummy" methods that can be trigger events for other systems like workflows. Once the code has been changed to use ADIOS for IO, any of the various IO routines can be selected just by changing the XML file. No source code changes are ever required.

The remainder of this paper will be organized as follows. Section 2 describes related work. Section 3 describes a representative petascale application, the GTC fusion modeling code. Section 4 outlines the ADIOS architecture, with section 5 presenting experimental evaluations of ADIOS. Section 6 contains conclusion and future work.

2. RELATED WORK

Related work falls into three main categories. First, there are the IO APIs attempting to bridge the gap between the code and the storage system. Second are the file systems used for storage and the facilities they provide and/or require and the impact those decisions have on performance. Finally, there are the auxiliary tools and how they can be used in concert with the scientific codes.

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. HDF-5 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 [5] provides support for Visit with no additional support for IO performance tuning or extra data annotation beyond what was needed for Visit.

Parallel file systems universally separate metadata from storage services, to enhance parallel access. Lustre [6] provides custom APIs for configuring the striping, storage server selection, buffer sizes, and other factors likely to impact performance. However, they still are limited to a single metadata server, causing a known bottleneck. In addition, there are known expensive operations, such as 'ls -l', which cause the metadata server to talk with each storage server to calculate the sizes of the pieces of the files stored on that device. PVFS [7] partially addresses the metadata scalability issues of Lustre by having distributed metadata servers, but at the cost of client-side complexity. Each client must use multiple steps to create files [8]. LWFS [9] has taken an extreme position on this topic by eliminating the requirement for online metadata, then using offline methods 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.

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 [10] 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 [11] 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.

None of these examples handle all three problems. The platform independent IO systems all provide either great performance or annotation. With careful use, systems like HDF-5 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. 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.

3. MOTIVATING EXAMPLES

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 have had to be added for each in situ visualization system employed. Each time one of these changes occurred, the base code had to be reevaluated to ensure that it was both operating properly and generating the proper data in the output. These evaluations cost days to weeks of time for the developers and thousands of hours of compute time with no science output. Through a system like ADIOS, the user can quickly test the various IO method available and select one that gives the best combination of performance and required features.

The initial development of ADIOS was motivated by the GTC fusion code and the Chimera supernova code. GTC provided a variety of different outputs with varying frequency and sizes while Chimera offered a vastly larger number of variables output per write with some different reporting/formatting requirements. We have further demonstrated the generality of ADIOS by integrating successfully with XGC0, XGC1, FLASH, GTS, and S3D with at most, only minor tweaks of the system required.

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

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. *Reused buffers must be handled properly* - It is important to support constructed output values in shared buffers. This means that we need to accommodate both copying stack temporary values when they are given to us as well as being able to handle the source code reusing a buffer to construct a second output artifact in an effort to save memory.

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

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 some 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. Each of these has different needs for IO performance, annotation, and potential tool integration. We also noted that some data is only written by a single process as a header with the rest only writing their portion of the payload. 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. As part of one of the analysis outputs to convert to Cartesian coordinates, a single buffer is created and then reused for each of the X, Y, and Z dimensions. That buffer is passed three times to ADIOS for output. Through the copy-on-write feature in the XML file, we both note and handle the transient nature of the data in that pointer. This also gives us the ability to take stack-based temporary values and write them properly. 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, several of the items drove the five requirements above. First, several of the variables output were Fortran array slices created on the stack as parameters to the write. Again, the copy-on-write feature addressed this need. And second, 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.

As mentioned above, since the initial development based on the GTC and Chimera observed requirements, the ADIOS API has successfully integrated with XGC1 fusion, FLASH astrophysics, S3D combustion, GTS fusion, and XGC0 fusion. Global arrays was the only additional feature we needed to add from the initial version of ADIOS to support all of these different codes.

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.

4. ARCHITECTURE

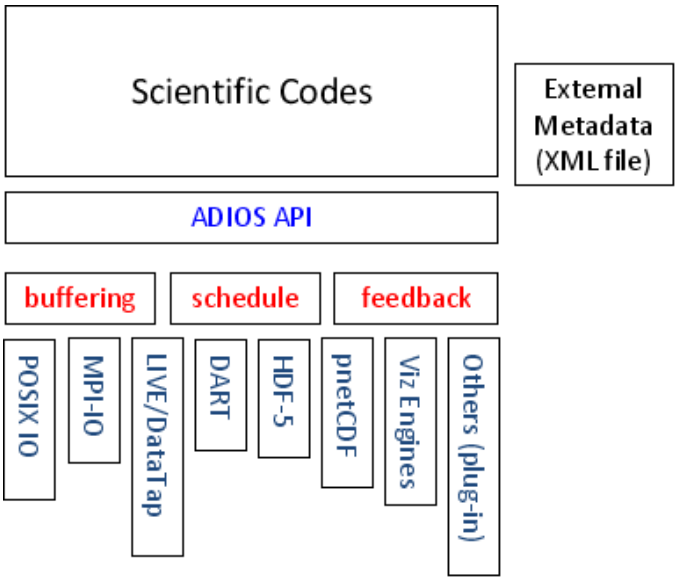

Figure 1: ADIOS Architecture

At a high level, ADIOS structurally looks like Figure 1. The four parts each provide key benefits.

1. *ADIOS API* - The core ADIOS API calls that are used in the scientific codes.
2. *Common services* - Internal services for encoding/decoding, buffering and other shared operations are available for use by any of the transport methods.
3. *Transport methods* - Perform the data operations and auxiliary integrations. For example, MPI-IO, POSIX IO, and Kepler or Visit integration.
4. *External metadata XML file* - Controls how these layers interact.

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

4.1 XML Format

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

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

The main elements of the XML file format are of the format

```
<element-name attr1 attr2 ...>
```

The details of the XML is more fully discussed in Appendix B. The description below is structured like the XML document:

```
<adios-config>
<adios-group name>
<global-bounds dimensions offset
     coordination-communicator
     coordination-var>
  <var name path type
    dimensions write copy-on-write/>
</global-bounds>
<mesh type time-varying>
...
</mesh>
<attribute name path value/>
</adios-group>
<method group method base-path
  priority iterations>
  parameters</method>
<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. Also specifies MxN-style operations through the coordination-communicator and/or coordination-var (more details below).
- var - a variable that is either an array or a primitive data type, depending on the attributes provided.
- mesh - [optional] a mesh description for the data compatible with the VTK data requirements.
- attribute - attributes attached to a var or var path.
- method - mapping a writing method to a data type including any initialization parameters.
- buffer - internal buffer sizing and creation time. Used only once.

Attributes:

- path - HDF-5-style path for the element or path to the HDF-5 group or data item to which this attribute is attached.
- dimensions - a comma separated list of numbers and/or names that correspond to integer var elements to determine the size of this item
- write - [optional] if it is set to "no", then this is an informational element not to be written intended for either grouping or dataset dimension usage
- copy-on-write - [optional] if it is set to "yes", then this is var must be copied when it is provided rather than caching a pointer.
- 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. If the global-bounds element is specified, then we have the ability to coordinate either on the compute nodes using the coordination-communicator or downstream using the coordination-var. Which communication mechanism (e.g., MPI or OpenMP) is used to coordinate is left up to the transport method implementer and potentially selected by the parameters provided in the 'method' element in the XML file. For example, if the MPI synchronous IO method is employed for a particular IO group, it uses MPI to coordinate a group write or even an MPI collective write. Alternatively, a different transport method could use OpenMP. We define that the communicator 'passed in' must make sense to the transport method selected and that the ordering of processes is assumed to be in rank order for that communicator. Similarly, if the coordination-var is provided as well, an asynchronous IO method may choose to send the data downstream annotated with these attributes so that an another process can reassemble the data according to these parameters.

4.1.1 Changing IO Without Changing Source

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

4.2 ADIOS API

Since scientific codes are written in both Fortran and C-style languages, we developed and tested ADIOS from the beginning to have both a Fortran and a C interface. The calls look nearly identical between the two APIs and only differ in the use of pointers in C. The details of these calls will be discussed in more details in Appendix 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.

4.2.1 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")
...
// 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 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. 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.

4.2.2 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 calling close. Since our system focuses on supporting both asynchronous and synchronous operations, the semantics for these calls assume stricter requirements. For example, a supplied buffer is expected to be valid until after the associated adios_close call returns.

```
adios_get_group (&io_group, "restart")
...
adios_open (&group_handle, io_group,
          "restart.01")
adios_write (group_handle, "comm", comm)
...
adios_write (group_handle, "zion", zion)
...
adios_write (group_handle, "mzeta", mzeta)
...
adios_close (group_handle)
```

Adios_get_group retrieves a handle to a structure that knows the members, types, and attributes of a collection of var elements. The second parameter corresponds to an adios-group defined in the XML file. 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. One special note is that adios_close is considered a 'commit' operation. Once it returns, all provided buffers are considered reusable.

4.3 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.

4.4 Transport Methods

We have partnered with experts on each IO method we have at this time. For example, we have a synchronous MPI-IO method based on work done by and verified by Steve Hodson at ORNL, a collective MPI-IO method developed by Wei-keng Liao at Northwestern, a POSIX IO method developed by Jay Lofstead with recommendations for performance enhancements by Wei-keng Liao, DataTap [12] asynchronous IO by Hasan Abbasi at Georgia Tech, and a NULL method for no output, which is useful for benchmarking the performance of the code without any or selectively less IO. For visualization transport methods, we have an initial pass at a VTK interface into Visit and a custom sockets connection into an OpenGL based renderer. We have under development an asynchronous MPI-IO method by Steve Hodson and have planned both a parallel netCDF and HDF-5 methods based on existing tools we have written to convert our default data encoding into these formats.

5. EVALUATION

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

5.1 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_get_group call. This one addition links the IO with metadata in the XML file including the selected IO method. 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.

Before the official release in Q3 2008, we will be adding a new feature that simplifies the source code even more by replacing all of the calls with a single preprocessor string that will be expanded into all of the proper calls. This will further insulate 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.

5.2 Fast IO

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

5.2.1 GTC Performance

To evaluate the system, we have run regular tests with GTC at ORNL just before the machine was partially shutdown for an upgrade. The system is a Cray XT4, dual core AMD x64 chips with 2 GB of RAM per core and around 40-45 GB/sec peak IO bandwidth to a dedicated Lustre parallel file system. Our tests showed a consistent average aggregate write performance of 20 GB/sec for a 128 node job [13]. See Figure 2.

A second evaluation was performed on the ewok system at ORNL. This is an Infiniband cluster of 81 nodes of dual core AMD x64 chips, 2 GB of RAM per core, and about 3 GB/sec peak IO bandwidth to a Lustre file system shared with several other clusters. Two sets of 5 runs for GTC on 128 cores were performed. Each run generated 23 restart outputs for a total of 74.5 GB. The first set was configured to generate

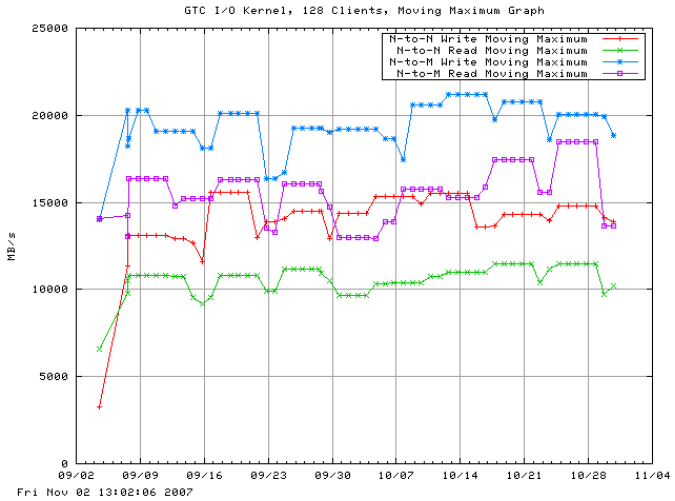

Figure 2: GTC on Jaguar with ADIOS

output using the MPI synchronous transport method while the second set was configured to generate no output using the NULL method. We were able to demonstrate an average 0.46 GB/sec performance. Given the ability to login to various nodes on the machine directly and the shared storage system, there is a large variability in the performance. Two of our runs with IO were faster than one without IO.

5.2.2 *Chimera Performance*

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

5.2.3 *GTS Performance*

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

5.3 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.

6. CONCLUSION AND FUTURE WORK

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 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. Our supplying a variety of expert created and verified transport methods gives the flexibility to select whichever approach is best for the current platform. 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.

There are other approaches for accomplishing these same goals, but we feel they are inferior. For example, a developer could use #ifdefs and have a variety of IO methods for their code. By recompiling with different switches, different IO routines would be employed. Like ADIOS, each of these will have to be tested to see which is best for the platform. This only works well for a code that never changes. However, if a new variable is added, the #ifdef approach would require editing each of the possibilities and then testing each one. With ADIOS, it need only be added to the simulation code in once place and the XML file in one place for use in any and all transport methods. Because we have proven transport method implementations and no recompilation necessary to choose a different method for testing, the difficulty in adding the var and in revalidation of the code is reduced. 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 func-

tion 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.

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. While the system works well, there are many things remaining to experiment with.

First, while we do have written and fully tested support for reading, we have not optimized the performance or evaluated read intensive codes to see how they will impact the philosophical choices made with ADIOS. For example, data reorganization on reading in global arrays will require optimization for best performance. There are many issues to examine that shift the performance to favor either write or read performance to varying degrees. Providing a mechanism for user selection of some of these parameters would be interesting.

Second, we have specialized hint mechanisms to help interleave asynchronous IO with other communication activities, but we need to explore various scheduling and buffering opportunities and tradeoffs within this environment.

Third, while we support global arrays, we have not included any special support striding, ghost zones, blocking, or other automatic array slicing operations. We had initially created some routines and interfaces to handle some of these cases, but found no need for them in our initial test cases. Since they were not fully tested for reading and writing and memory formatting issues (Fortran vs. C memory layouts), we opted to delay these features for the second release.

Forth, online steering and other external forces changing the way the running scientific code operates poses another whole set of challenges we look forward to addressing. For example, it would be nice to be able to change what and how often data is sent to an in situ visualization system while the code is running.

Fifth, we plan on investigating full end-to-end data movement scheduling from the compute nodes to the storage system. For example, if we can detect contention in the storage system through slow performance, what adjustments can be made to give optimal end-to-end performance for all users of the storage system? Second, additional scientific codes, such as adaptive mesh refinement codes, will generate new requirements and challenges for data processing and movement. Third, managing the data artifacts of petascale jobs becomes critical. The size of the restart data collections can quickly outstrip the storage available or even exceed the ability to spool it to offline storage for later manipulation before it is automatically purged. Policies like only maintaining 2 good, complete restart collections at a time, physics-based lossy compression filters, analysis summarization, and file merging stripping out redundant elements, can help manage storage concerns.

Sixth, given our successful integration with visualization systems and the problems we describe with a typical approach for integrating with a workflow system like Kepler, we need to explore various techniques for integration that makes sense with different workflow styles. The initial integration measurements will involve signaling rather than relying on an 'ls -l' process. Others will follow as we identify different workflow integration requirements.

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, 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 [3] and Visit [14], 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.

APPENDIX

A. ADIOS API DETAILS

In addition to the APIs mentioned below, others exist 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://www.cc.gatech.edu/~lofstead/adios.

A.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.0.2 IO Operation

```
adios_get_group (&io_group, "restart")
...
adios_open (&group_handle, io_group,
          "restart.01")
adios_write (group_handle, "comm", comm)
...
adios_write (group_handle, "zion", zion)
...
adios_write (group_handle, "mzeta", mzeta)
...
adios_close (group_handle)
```

Adios_get_group () retrieves a handle to a structure that knows the members, types, and attributes of a collection of var elements. The name corresponds to a adios-group defined in the XML file.

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

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.

B. 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
     coordination-communicator
     coordination-var>
   <var name path type
    dimensions write copy-on-write/>
  </global-bounds>
  <mesh type time-varying>
  ...
  </mesh>
  <attribute name path value/>
 </adios-group>
 <method group method base-path
  priority iterations>
  parameters</method>
 <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.
- mesh - [optional] a mesh description for the data compatible with the VTK data requirements.
- attribute - attributes attached to a var or var path.
- method - 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), 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
- write - [optional, default="yes"] if it is set to "no", then this is an informational element not to be written intended for either grouping or dataset dimension usage

- copy-on-write - [optional, default="no"] if it is set to "yes", then this is var must be copied when it is provided rather than caching a pointer.
- 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 MB 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 ()

C. REFERENCES

[1] R. Ross, R. Thakur, W. Loewe, and R. Latham, "Parallel i/o in practice," in *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM, 2006, p. 216.

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

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

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

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

[6] P. J. Braam, "Lustre: a scalable high-performance file system," Nov. 2002. [Online]. Available: http://www.lustre.org/docs/whitepaper.pdf

[7] R. Ross, R. Latham, N. Miller, and P. Carns, "A next-generation parallel file system for linux clusters," January 2004.

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

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

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

[11] J. Ahrens, C. Law, W. Schroeder, K. Martin, and M. Papka, "A parallel approach for efficiently visualizing extremely large," 2000. [Online]. Available: citeseer.ist.psu.edu/ahrens00parallel.html

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

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

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