Adaptive IO System (ADIOS)

Chen Jin, Scott Klasky, Stephen Hodson, Weikuan Yu (Oak Ridge National Laboratory)
Jay Lofstead, Hasan Abbasi, Karsten Schwan, Matthew Wolf (Georgia Tech)
Wei-keng Liao, Alok Choudhary(Northwestern University)
Manish Parashar, Ciprian Docan, Rutgers University.
Ron Oldfield (Sandia National Laboratories)

ABSTRACT:*ADIOS is a state of the art componentization of the IO system that has demonstrated impressive IO performance results on the Cray XT system at ORNL. ADIOS separates the selection and implementation of any particular IO routines from the scientific code offering unprecedented flexibility in the choices for processing and storing data. The API was modelled on F90 IO routines emphasizing simplicity and clarity using external metadata for richness. The metadata is described in a stand-along XML file that is parsed once on code startup and determines what IO routines and parameters are used by the client code for each grouping of data elements generated by the code. By employing this API, a simple change to an entry in the XML file changes the codes to use either synchronous MPI-IO, collective MPI-IO, parallel HDF5, pnetcdf, NULL (no output), or asynchronous transports such as the Rutgers DART implementation and the Georgia Tech DataTap method. Simply by restarting the code, the new IO routines selected in the XML file will be employed. Furthermore, we have been defining additional metadata tags to support in-situ visualization solely through changes in the XML metadata file. The power of this technique is demonstrated on the GTC, GTC_S, XGC1, S3D, Chimera, and Flash codes. We show that when these codes run on a large number of processors, they can sustain high I/O bandwidth when they write out their restart and analysis files.*

KEYWORDS: *I/O, componentization*

1. Introduction

Massive parallel applications running on the next generation supercomputers using 100,000s of cores face severe challenges in IO and data management. The well-known performance and the scalability gap between the computation and the I/O components are enlarging; especially when the trend to many –core architectures further intensifies the load and the contention level on the I/O stack. To overcome these problems, we are building a componentization of the IO layer, the Adaptive IO system (ADIOS), to take the implementation of the IO layer away from the application scientist

Although there are quite a few application programming interfaces (API) such as parallel NetCDF and parallel HDF5, which can deal with large data set storage and access, none of them can prove the best performance for all the different computer architectures and file systems. Researchers find that they often get good performance on a limited number of cores, but poor performance once they scale up to a larger amount of cores. Furthermore, they also find that their solution with one technology works well with one architecture, but breakdown on other machines. There are strategies that researchers can reduce this performance impact, but researchers who code in HDF5 are locked into the performance of this implementation, and can waste a significant amount of time in IO when HDF5 performs poorly.

The scientific codes sometimes consist of a lot of small read/write statements for debug, analysis or annotations. These small IO processes have been proven improper and dramatically affect the IO performance for large-scale scientific simulations. The ADIOS APIs transparently buffer small data to writes or reads so that only large chucks of data block are read/written to disk. Application programmers can avoid user this buffer in ADIOS, but the default is to use this.

Another important factor driving the design of ADIOS is real time monitoring and analysis of large scale simulations. Separating the transport layer from APIs allows programmers to reroute the data flow from disk to any visualization/analysis tools, so that the simulation results can be monitored and debugged on the fly, as well as actively monitor runs which run amok. Such monitoring feature implies the need for backend dataflow engines, such as a scientific workflow system. With the advantageous componentization of ADIOS, an alternative method for integrating a workflow can be easily switched to by modifying the XML configure file.

Conventionally, all of this data must be written in the code, which involves placing these statements inside the code. ADIOS is able to extract the metadata, variable dimensions and types out of the source code into an external configuration file, therefore the original scientific application only need to be modified and verified once by adding ADIOS APIs. The further modification and performance testing can be achieved by only changing the external configure file. To separate metadata away from source code helps the maintenance of the scientific application in that the authors and community are reluctant to redesign and change the code except for the extreme performance requirements.

To provide the broad community of application scientists with a high-performance, easy-to-use interface for I/O processing, we have defined an alternative high-level API with external XML configuration file, allowing for programming without bothering with the details of transport layer implementation. ADIOS is an IO componentization, which allows for fast and scalable IO on small clusters and large petascale supercomputers. Moreover, it provides the flexibility of switching different transport methods by only modifying the xml file without verification of the source code. Last but not the least, ADIOS can be used to help couple codes using both file and memory based methods. In addition, a new ADIOS APIS can be extended for code coupling in ADIOS, which will also send the metadata over to the workflow automation system.

2. ADIOS API's

ADIOS I/O in MPI starts with functions familiar to users of standard "language" I/O or libraries. MPI also has additional features necessary for performance and portability. In this section we focus on the ADIOS counterparts of opening and closing files, as well as read/writing contiguous blocks of data from/to them. At this level the main feature we show is how ADIOS can conveniently express read/write operations in terms of group. Later on in section 4, we can show how easy it is to implement parallelism for these operations. The following programming example illustrates how to write an integer array and a double-precision array with size of *dimx* into file called "test.bp", which is organized in our native tagged binary file format. BP, which stands for binary packed, allows users to include rich metadata associated with the block of binary data. The corresponding XML configuration file required by this program is demonstrated in the next section.

```
    /*example of parallel MPI write into a single file */
    #include "adios.h"
    #include <stdio.h>
    int main(int argc, char *argv[])
    {
      int i, myrank, dimx, *X;
      double *Y;
      dimx=100;
      X=(int*)malloc(sizeof(int)*dimx);
      Y=(double*)malloc(sizeof(double)*dimx);
      for(i=0;i<dimx;i++)
      {
        X[i]=i+myrank*dimx;
        Y[i]=2.5*X[i];
      }
      MPI_Init(&argc,&argv);
           MPI_Comm_rank(mpi_comm_world,&myrank);
adios_init("config.xml",mpi_comm_world,
mpi_comm_self,mpi_info_null);
    adios_get_group(&grp_id,"Potential");
    adios_open(&buf_id,grp_id,"data.bp");
    adios_write(buf_id,"dimx",&dimx);
      adios_write(buf_id,"X",X);
      adios_write(buf_id,"Potential",Y);
    adios_close(buf_id);
    adios_finalize(myrank);
    MPI_Finalize();
    free(Y);free(X);
    }
```

As shown in this example, a pair of `adios_init` and `adios_finalize` should be called between the actual paired `mpi_init` and `mpi_finalize`. Before any adios operation starts, `adios_init` is required to load the XML configuration file creating internal representations of various data types and defining the methods used for writing. Presently, there are additional parameters to define various MPI elements that are supposed to be transparently compatible between Fortran and C, but are not. Similar to `mpi_finalize`, `adios_finalize` releases all the resources allocated by adios and guarantee all the remained adios operations be finished before the code exits. After the basic adios initialization, `adios_get_group` gets called to retrieve a handle for the group with the name of “Potential”, which should be specified in the XML configuration file. The unique feature of adios is the group implementation, which is constituted by a list of variables and associated with individual transport methods. The flexibility allows the applications to make the best use of the file system according to its own different IO patterns.

The ADIOS function corresponding to `fopen` is called `adios_open`. Let us consider the arguments one by one.ADIOS_open(io_handle,group_handle, filename). The first argument is an IO handle. Different from file descriptor, this IO handle only prepare the data type for the subsequent calls to write data using io_handle. The second argument is a string representing the name of file, as in fopen. As the last argument, we pass the address of the ADIOS_FILE variable, which the adios_open will fill in for us. adios_close() triggers the building of the buffer for transfer and then returns control back to the caller. At this point, all of the data is copied and will be sent as-is downstream. If the handle is opened for read, this will cause the fetch of the data, parse it, and populate it into the provided buffers. This is currently hard-coded to use posix io calls.

Adios_write(io_handle,fieldname,&var) submit a data element for writing and associate it with the given filename for this type. This does not actually perform the write. Scalars are duplicated, vectors are referenced. Any changed to vectors before adios_close is called will be reflected in the written data. In the same way, adios_read (io_handle, field_name, &var) - submit a buffer space (var) for reading a data element into. This does NOT actually perform the read. Actual population of the buffer space will happen on the call to adios_close()

As presented in the program, ADIOS provides the application programmers easy-to-use, rich-featured APIs. In the near future, we will implement ADIOS_gwrite(), which will replace all the tedious adios_write function calls and offer cleaner interface for scientists and programmers.

3. ADIOS XML File Description

By abstracting metadata, data type and dimension from source code into XML file, it not only gives users more flexibility to annotate the arrays or variables, but also centralizes the description of all the data structures, which in return allows IO componentization for different implementation of transport methods. By cataloguing 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. Once nice feature of the XML name attributes s 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 following illustrates the corresponding XML configuration for the example we demonstrated in the previous section.

```
<?xml version="1.0"?>
<adios-config host-language="C">
<adios-group name="Potential ">
<global-bounds dimension=”g_x”  offset=”o_x”>
<var name="g_x" type="integer"/>
<var name="o_x" type="integer"/>
<var name="dimx" type="integer"/>
<var name="X" type="integer" dimension=”dimx”/>
<var name="P" type="double" dimension=”dimx”/>
<attribute name=”description” path=”/P” value=”the potential value”/>
</global-bounds>
</adios-group>
<method priority="1" method="MPI-IO" group="Potential"/>
<buffer size-MB="100" allocate-time="now"/>
</adios-config>
```

The main elements of the xml file format are of the format <element-name attr1 attr2 …>. At a minimum, a configuration document must declare adios-config element. It serves as a container for other elements; as such, it MUST be used as the root element. The expected children in any order would be adios-group, method and buffer.

The adios-group element represents a container for a list of variables that share the common IO pattern; in this case, we name it as Potential. Depending on the different scientific application, the occurrence of adios-group can be as many as needed.

Global-bounds are an optional nested element for adios-group, which specifies the global space and offsets within that space for the enclosed variable elements.

The nested var element for adios_group can be either an array or a primitive data type, determined by the dimension attribute provided.

The Attributes associated with var element is listed as below:

• 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.

The method element is used to specify the mapping of an IO transport method to a data type including any initialization parameters. There are two major attributes required for the method element:

• 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.

The buffer element defines the attributes for internal buffer size and creating time, which will apply to the whole application.

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. ADIOS Methods

POSIX Method

The simplest method provided in ADIOS just does binary POSIX IO operations. Currently, it does not support shared file writing or reading and has limited additional functionality. The main purpose for this IO method is to provide a simple way to migrate a one file per process IO routine to ADIOS and test the results without introducing any complexity from MPI-IO or other IO methods. Performance gains just by using this transport method are likely due to our aggressive buffering for better streaming performance to storage.

Additional features may be added to this transport method over time. Most likely is a new transport method with a related name, such as POSIX-ASCII, would be provided to perform IO with additional features. The POSIX-ASCII example would write out a text version of the data formatted nicely according to some parameters provided in the XML file.

MPI-IO Method

Many large-scale scientific simulations generate a large amount of data, spanning thousands of files or datasets. Allowing the use of MPI-IO to reduce the amount of files will be helpful to the data management, storage and access.

The original MPI-IO method was developed by Steve Hodson based on his experiments with generating the better MPI-IO performance on the Jaguar machine at ORNL. Many of his insights have helped us achieve excellent performance on both the Jaguar XT4 machine and on theother clusters, suchas the Ewok end-to-end cluster. Some of the key insights we have taken advantage of include artificially serialized MPI_File_open calls and additional timing delays that can achieve reduced delays due to metadata server (MDS) conflicts on the attached Lustre storage system.

The adapted code takes full advantage of NxM grouping through the coordination-communicator. This will generate one file per coordination-communicator with the data stored sequentially based on the process rank within the communicator. Figure 1 presents in the example of GTC code, 32 processes inthe same Toroidal zonewrite to one integrated file. Additional serialization of the MPI_File_open calls is done using this communicator as well since each process may have a different size data payload. Rank 0 calculates the size it will write, calls MPI_File_open, and then sends its size to rank 1. Rank 1 listens for the offset to start from, adds its calculated size, does an MPI_File_open, and sends the new offset to rank 2. This continues for all processes within the communicator. Additional delays for performance based on the number of processes in the communicator and the projected load on the Lustre MDS can be used to introduce some additional artificial delays that ultimately reduce the amount of time the MPI_File_open calls take by reducing the bottleneck at the MDS.An important fact to be noted is that individual file pointers are retrieved by MPI_File_openso that each process has its own file pointer for file seek and other I/O operations.

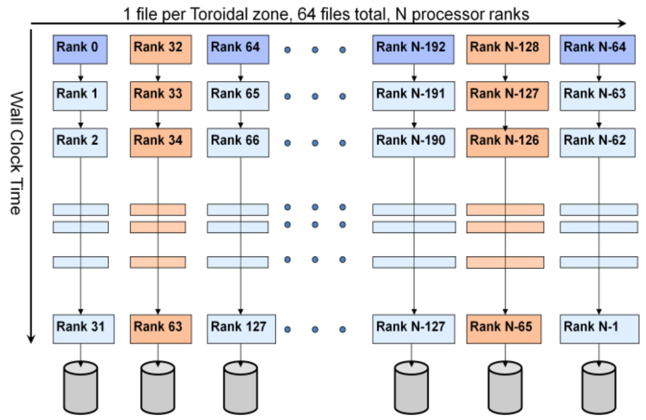

Figure 1: meta-data server friendly approach -- offset the create/open in time

While we have built this mainly with Lustre in mind due to it being the primary parallel storage service we have available, other file system specific tunings are certainly possible and fully planned as part of this transport method system. For each new file system we encounter, a new transport method implementation tuned for that file system, and potentially that platform, can be developed without impacting any of the scientific code.

This transport method is the most mature, full featured, and well tested. We recommend anyone creating a new transport method study this one as a model of full functionality and some of the advantages that can be made through careful management of the storage resources.

MPI-CIO Method

MPI-IO defines a set of portable programming interfaces for multiple processes concurrent access to shared files [1]. It is often used to store and retrieve structured data in their canonical order. The interfaces are split into two types: collective I/O and independent I/O. Collective functions require all processes to participate. Independent I/O, in contrast, requires no process synchronization.

Collective I/O enables process collaboration to rearrange I/O requests for better performance [2,3]. The collective I/O method in ADIOS first defines MPI fileviews for all processes based on the data partitioning information provided in the XML configuration file. ADIOS also generated MPI-IO hints, such as data sieving and I/O aggregators, based on the access pattern and underlying file system configuration. The hints are supplied to the MPI-IO library for further performance enhancement. The syntax to describe the data-partitioning pattern in the XML file uses <global-bounds dimensions offsets> tag, which defines the global array size and the offsets of local subarrays in the global space.

The global-bounds element contains one or more nested var elements each specifying a local array that exists within the described dimensions and offset. Multiple global-bounds elements are permitted and strictly local arrays can be specified outside the context of the global-bounds element.

As with other data elements, each of the attributes of the global-bounds element is provided by the adios_write call. The dimensions attribute is specified by all participating processes and defines how big the total global space is. This value must agree for all nodes. The offset attribute specifies what offset into this global space the local values are addressed. The actual size of the local element is specified in the nested var element(s). For example, if the global bounds dimension were 50 and offset were 10, then the var(s) nested within the global-bounds would all be declared in a global array of 50 elements with each local array starting at an offset of 10 from the start of the array. If more than one var is nested within the global-bounds, they share the declaration of the bounds, but are treated individually and independently for data storage purposes.

MPI-AIO Method

The initial implementations of the asynchronous MPI-IO method (MPI-AIO) is patterned after the MPI-IO method. Scheduled meta-data commands are performed with the same serialization of MPI_Open calls as given in Figure 1.

The degree I/O asynchronicity will depend on several factors. First, the ADIOS library must be built with versions of MPI that are bullt with ansynchronous I/O support through the MPI_File_iwrite, MPI_File_iread, and MPI_Wait calls. If asynchronous I/O is not available, these calls revert to synchronous (read blocking), behaviour, identical to the MPI-IO method described in the previous section.

Another important factor is the amount of ADIOS buffer space available. In the MPI-IO method, data is transported and ADIOS buffer allocation is reclaimed for subsequent use with calls to adios_close(). In the MPI-AIO method, the “close” process can be deferred until the buffer allocation is actually needed for new data. However, if the buffer allocation is exceeded, the data must be synchronously transported before the application can proceed.

The deferral of data transport is key to effectively scheduling ansynchronous I/O with the computation, to be implemented in version 2.0. In ADIOS version 1.0, the application explicitly signals that data transport must be complete with intelligent placement of the adios_close() call to indicate when I/O must be complete. Later versions of ADIOS will perform I/O between adios_begin_calculation andadios_end_calculation calls, and complete I/O on adios_end_interation calls.

DataTap Method

DataTap is an asynchronous data transport method built to ensure very high levels of scalability through server-directed I/O[7,8]. It is implemented as a request-read service designed to bridge the order of magnitude difference between available memory on the I/O partition compared to the compute partition. We assume the existence of a large number of compute nodes producing data (we refer to them as *DataTap clients*) and a smaller number of I/O nodes receiving the data (we refer to them as *DataTap servers*).

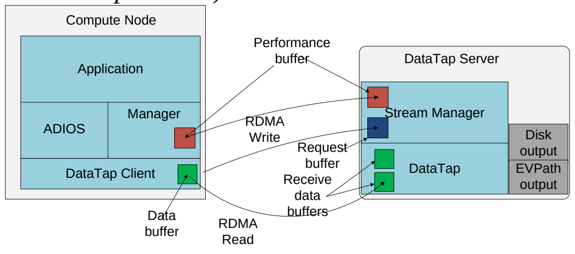

Figure 2: Datatap Architecture

Error! Reference source not found.describes the DataTap architecture. Upon application request the compute node marks up the data in PBIO [9] format and issues a request for a data transfer to the server. The server queues the request until sufficient receive buffer space is available. The major cost associated with setting up the transfer is the cost of allocating the data buffer and copying the data. However, this overhead is small enough to have little impact on the overall application run-time. When the server has sufficient buffer space, an RDMA read request is issued to the client to read the remote data into a local buffer. This data is then written out to disk or transmitted over the network as input for further processing in the I/O Graph.

We used the Gyrokinteic Turbulence Code, GTC[] as an experimental testbed for the DataTap transport. GTC is a particle-in-cell code for simulating fusion within tokamaks, and it is able to scale to multiple thousands of processors. In its default I/O pattern, the dominant I/O cost is from each processor writing out the local particle array into a file. Asynchronous I/O reduces this cost to just a local memory copy, thereby reducing the overhead of I/O in the application.

DART Method

DART is an asynchronous I/O transfer method within ADIOS that enables the low-overhead high-throughput data extraction from a running simulation. The design of DART consists of two main components, (1) DARTClient module, and(2) DARTServer module. Internally, DART system uses Remote Direct Memory Access (RDMA) to implement the communication, coordination and data transport mechanism between the DARTClient and the DARTServer modules.

The DARTClient module is a light library that implements the asynchronous I/O API. It integrates with the ADIOS layer by extending the generic ADIOS data transport hooks. It uses the ADIOS layer features to collect and encodethe data written by the application into a local transport buffer. Once it has collected data from the simulation, DARTClient notifies the DARTServer through a coordination channel that it has data available to send zout. DARTClient then returns and allows the application to continue its computations, while data is asynchronously extracted by the DARTServer.

The DARTServer module is a stand-alone service that runs independently from the simulation. It typically runs on dedicated I/O nodes, and transfers data from the DARTClients and to remote sites, e.g., a remomte a storage system such as the Luster file system. One instance of the DARTServer can service multiple DARTClients instances in parallel. Further, the server can run in cooperative mode, i.e., multiple instances of the server cooperate to service the clients in parallel and to balance load. The DARTServer receives notification messages from the clients, schedules the requests and initiates the data transfers in parallel from the clients. The server schedules and prioritizes the data transfers while the simulation is computing in order to overlap data transfers with computations, to maximize data throughput, and to minimize the overhead on the simulation.

NULL Metod

The ADIOS NULL method allows users to quickly comment out a ADIOS group, by changing the transport method to "NULL". This allows users to test the speed of the routine, by timing the output against no IO. This is especially useful when working with asynchronous methods which have indeterminate amount of time. Another useful feature of this IO is that quickly allows users to also test out the system, and see if bugs are possibly caused by the IO system, or perhaps by other places in the codes.

5. ADIOS in codes

ADIOS in GTC

GTC fusion code provided a variety of different outputs with varying frequency and sizes. From an IO complexity perspective, GTC has seven different groups of output in five categories, each being handled differently. These five categories are restart, tracking, dataout3D, analysis, and diagnosis output. Each of these categories has different IO requirements based on their output patterns. For instance, the large restart data needs to be written as quickly as possible with some annotation. To mitigate the runtime performance impact, it is written infrequently. The analysis, tracking and output3D data, while much smaller, needs to be written out to disk more often. The diagnostic outputs are written very frequently, but are little more than a few kilobytes per time step and always only from the master processor. Therefore, each of these output groups has different requirements for IO performance, annotation and potential tool integration. We also noted that some data was only written as a header by the master process followed by the collection of the payloads from all processors. The dissimilarity between the header data and actual payload can be easily differentiated by specifying different adios-group in XML file. In addition, ADIOS provides the flexibility of selecting how each of these different data groupings performs IO simply by specifying the selected method for each of these groups in the XML. It handles the different sizes for the analysis array outputs through the use of var names for array dimensions. Through the copy-to-write feature in the XML file, we both note and handle the transient nature of the data in the given pointer. This also gives us the ability to take stack-based temporary values and write them properly.

In GTC old restart IO, each MPI process writes/reads a separate file, which is N to N IO complexity. If the new version of ADIOS APIS with MPI-IO transport method, npartdom (the number of processes in particle domain)

MPI processes writes/reads in one integrated file. The total number of files is reduced to mzetamax, shown as below in Figure 3. In this manner, it not only allows each plane has its own file, but is also convenient for programmers to extract arbitrary plane instead of parsing a big binary file if it is collected from all the process in the communication world.

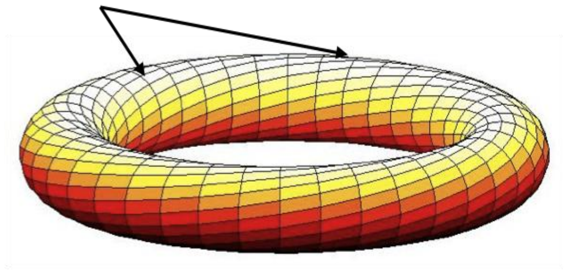
Figure 3 Gyrokinetic Toroidal Code Mesh Space

ADIOS in GTS

GTS is a general geometry model of the GTC code, which has a systematic treatment of plasma rotation and equilibrium EXB flow, realistic plasma profiles and corresponding magnetohydrodynamic (MHD) equilibria. They use a symmetry coordinate system to construct the relatively regular mesh in real space for strongly shaped toroidal plasmas, which facilitates straightforward visualization.

There are two major sources of IO in this code, one from the restarts, and the second is from the particle diagnostics. Unlike GTC, GTS has developed a technique to bin the particles in the code into a five dimensional mesh. The size of the mesh is determined by how much data can be stored and written to disk in a timely fashion. The binning is rather cheap, and can be down in <0.01 of the time in the code, but the IO generated from this can be rather large. Realistic numbers for the mesh size can be over 120GB, which much be written out every 60 seconds for a realistic simulation. This data needs to be written out approximately every other GTS timestep, and we need approximately 2,000 timesteps written to disk to get relevant physics generated from this output. This means that one simulation, which last for 1.5 days can generate over 200 TB of data in 1.5 days.

The other major change to GTS was switching out the 'fort.*" files to ADIOS output, which allows us to place more metadata in the output.

ADIOS in Chimera

The Chimera supernova code provided new challenges as compared with GTC. First, the IO was split into fewer groups than GTC, but the groups were much larger. There are only two groups in Chimera, but a total of around 475 vars. Second, there were other IO operations that generated different formats of data previously output. Our asynchronous methods will take advantage of this by collecting the output once and processing it into multiple streams to storage with one in the compact binary representation and others in the various formatted text forms desired by the users.

By just changing the main restart IO to use ADIOS with the POSIX transport method, we saw a 6.5% reduction in wall clock time for identical runs between the original version and our ADIOS version.

ADIOS in S3D

As a part of the efforts to provide a versatile I/O middleware for scientific applications, we have ported the I/O kernel of the combustion simulation program, S3D, to the initial ADIOS middleware. The porting of the ADIOS to S3D is rather straightforward with the close resemblance of ADIOS interface to the POSIX interface. Rougly speaking, there are three steps involved. In the first step, ADIOS is initialized right after the S3D program starts, i.e., after MPI_Init(). In the second step, the bulk of the S3D I/O code has been replaced with ADIOS I/O routines. An ADIOS configuration file that describes the attributes of various S3D variables is also created. This configuration file offers a great deal of flexibility for ADIOS to choose among its I/O methods. Currently, the POSIX I/O method is activated. Finally in the third step, ADIOS support is finalized before the S3D program reaches MPI_Finalize(). Timing profiles are taken during the ADIOS initialization and finalization steps. One thing to note is that, because S3D uses 8-byte integers, the ADIOS package has been improved to accommodate such scientific applications that make use of variable size data types.

ADIOS in XGC

The XGC code is a Gyrokinetic Turbulence code to understand the edge of the plasma. It is similar to the GTC code, but deploys a full-f algorithm, compared to a delta-f algorithm. This means that the total amount of particles per cell will be much higher than the GTC code, which uses a delta-f method. The simulations also run for a longer amount of time than the GTC code, which requires that the IO system be even more impervious to problems.

ADIOS is currently integrated into the working version of XGC, and has eleven ADIOS groups: restarts, particle diagnostics, electron field diagnostics, ion field diagnostic, 2D diagnostics, and various one dimensional + time diagnostics. By changing all of the IO from XGC (originally in netcdf, hdf5, binary files, and fort.* files), we are able to unify the IO in this code to one single set of API's. The initial performance from ADIOS in this code looks encouraging, although we are waiting for full Cray XT at ORNL to become available to truly test the system.

6. ADIOS Performance

The performance measurements we perform for ADIOS are focused in two areas. First, we want to get the best performance possible out of a transport method to storage. Our 20 GB/sec performance using MPI-IO to Lustre on Jaguar demonstrates this. Second, reducing wall clock time for jobs is our real measure for success. Only by reducing this time will there be a real improvement in terms of cost to users. Our initial Chimera without any optimizations generated a 6.5% reduction in wall clock time. Further optimizations will yield better performance improvements. We have also evaluated the performance up to 1024 processes for the S3D code, and have demonstrated that ADIOS is able to achieve comparable performance to the original S3D I/O implementation.

7. Conclusion

ADIOS is a componentization of IO layer. It has been designed to be easy to program, and to be fast and portable. By allowing users the flexibility to switch between different IO implementations, we can help ensure at least one method works properly on a new platform.

Currently, the GTC, GTS, XGC1, M3D-K, S3D, and Chimera codes are planning to use ADIOS, since it's promise is scalable portable performance. Each group will most likely use different ADIOS methods to enhance their own needs. Preliminary results in our suite of codes on the Cray XT have shown an impressive 20GB/sec for writing the data.

ADIOS 1.0 will be released in the fall of 2008, and will be optimized for extremely fast writes. Most features in ADIOS will continue to evolve. Some of the new features which we will add to the features are group writes, where users will be able to have just one write statement in their code, and ADIOS will look at all of the variables in the XML configuration file and then write out all of the variables with the one write statement.

Another major enhancement that we are currently working on is a header that allows for extremely efficient writes, and allows us to be able to place an indexing table in front of the file. At first, we will implement the indexing, and later we will implement this as part of the writing; allowing the code developer to choose if they want to pay the cost to compute this as the writes are occurring.

Another advantage of ADIOS is that it allows the users to easily create large buffers for the write statements. File systems, such as LUSTRE, can easily take advantage of these large writes, allowing the users to get very fast writes. ADIOS also contains APIs to help schedule IO, allowing the asynchronous IO to only occur during the computational phases of the simulation, and not during the communication phase.

Currently ADIOS is not taking advantage of a feedback mechanism but we envision that ADIOS will take advantage of this in two central ways. First, we can get feedback from the file system to find out if the synchronous writes will be fast. If the file system can quickly tell ADIOS that a restart should not be written, then we can easily allow this, by placing annotations in the XML file for this. We have already designed APIs which can work to count iteration counts in the simulation, and tell the system that certain data must be written out in a given amount of timesteps. Likewise, we can extend this to say, "write out restarts every 100 iterations +- 10. In this way, we can easily allow ADIOS to try to write out the data when the disk system is not performing other IO operations, thus minimizing the amount of time required to write out the data.

8. Acknowledgments

The authors would like to thank the staff at NCCS/ORNL for allowing us to run many test of the Cray XT and Infiniband systems at ORNL. We would like to author thank the code authors of GTC, GTS, XGC, S3D, and Chimera for many useful discussions to help their IO and help form ADIOS into a useful product.

9. References

- Message Passing Interface Forum. MPI-2: Extensions to the Message Passing Interface, July 1997.
- J. del Rosario, R. Brodawekar, and A. Choudhary. Improved Parallel I/O via a Two-Phase Run-time Access Strategy. In the Workshop on I/O in Parallel Computer Systems at IPPS '93, pages 56–70, April 1993.
- R. Thakur, W. Gropp, and E. Lusk. Data Sieving and Collective I/O in ROMIO. In the 7th Symposium on the Frontiers of Massively Parallel Computation, February 1999.
- Z. Lin, S. Ethier, T. S. Hahm, W. M. Tang, "Size Scaling of Turbulent Transport in Magnetically Confied Plasmas," *Phys. Rev. Letters*, vol. 88, 2002.
- C.S. Chang, Ku, M. Adams, F. Hinton, D. Keyes, S. Klasky,W. Lee, Z. Lin, S. Parker, "Gyrokinetic particle simulation of neoclassical transport in the pedestal/scrape-off region of a tokamak plasma," *Institute of Physics Publishing Journal of Physics: Conference Series*,pp. 87-91, 2006.
- W. Wang, Z. Lin, W. Tang, et al., "Global Gyrokinetic Particle Simulation of turbulence and transport in realistic tokamak geometry",*Institute of Physics Publishing Journal of Physics: Conference Series*, pp. 59-64, 2005.
- Seamons,K.,Chen,Y.,Jones,P.,Jozwiak,J.,andWinslett, M., "Server-directed collectiveI/O in Panda", Proceedings of Supercomputing '95, San Diego, CA
- Kotz,D., "Disk-directed I/O for MIMD multiprocessors", ACM Transactions on Computer Systems, pp. 41-74.
- Fabian E. Bustamante, Greg Eisenhauer, Karsten Schwan, and PatrickWidener. Efficient wire formats for high performance computing. InProceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM), pp.39. IEEE Computer Society, 2000.