ADIOS 1.2.1 User's Manual

August 2010

ADIOS 1.2.1 USER'S MANUAL

Prepared for the
Office of Science
U.S. Department of Energy

S. Hodson, S. Klasky, Q. Liu, J. Lofstead, N. Podhorszki, F. Zheng, M. Wolf,
T. Kordenbrock, H. Abbasi, N. Samatova

Aug. 2010

Prepared by

OAK RIDGE NATIONAL LABORATORY
Oak Ridge, Tennessee 37831-6070
managed by
UT-BATTELLE, LLC
for the
U.S. DEPARTMENT OF ENERGY
under contract DE-AC05-00OR22725

Contents

Figures

Abbreviations

ADIOS	Adaptive Input/Output System
API	***A****pplication* ***P****rogram* ***I****nterface*
DART	Decoupled and Asynchronous Remote Transfers
GTC	Gyrokinetic Turbulence Code
HPC	high-performance computing
I/O	input/output
MDS	metadata server
MPI	Message-Passing Interface
NCCS	National Center for Computational Sciences
ORNL	Oak Ridge National Laboratory
OS	operating system
PG	process group
POSIX	Portable Operating System Interface
RDMA	remote direct memory access
XML	Extensible Markup Language

Acknowledgments

The Adaptive Input/Output (I/O) system (ADIOS) is a joint product of the National Center of Computational Sciences (NCCS) at Oak Ridge National Laboratory (ORNL) and the Center for Experimental Research in Computer Systems at the Georgia Institute of Technology. This work is being led by Scott Klasky (ORNL); Jay Lofstead (Georgia Tech, funded from Sandia Labs) is the main contributor. ADIOS has greatly benefited from the efforts of the following ORNL staff: Steve Hodson, who gave tremendous input and guidance; Chen Jin, who integrated ADIOS routines into multiple scientific applications; Norbert Podhorszki, who integrated ADIOS with the Kepler workflow system and worked with Qing Gary Liu on the read API. ADIOS also benefited from the efforts of the Georgia Tech team, including Prof. Karsten Schwan, Prof. Matt Wolf, Hassan Abbasi, and Fang Zheng. Wei Keng Liao, Northwestern University, and Wang Di, SUN, have also been invaluable in our coding efforts of ADIOS, writing several important code parts. Essentially, ADIOS is componentization of I/O transport methods. Among the suite of transport methods, Decoupled and Asynchronous Remote Transfers (DART) was developed by Prof. Manish Parashar and his student Ciprian Docan of Rutgers University.

Without a scientific application, ADIOS would not have come this far. Special thanks go to Stephane Ethier at the Princeton Plasma Physics Laboratory (GTS); Researcher Yong Xiao and Prof. Zhihong Lin from the University of California, Irvine (GTC); Julian Cummings at the California Institute of Technology; Seung-Hoe and Prof. C. S. Chang at New York University (XGC); Jackie Chen and Ray Grout at Sandia (S3D); and Luis Chacon at ORNL (Pixie3D).

This project is sponsored by ORNL, Georgia Tech, The Scientific Data Management Center (SDM) at Lawrence Berkeley National Laboratory, and the U.S. Department of Defense.

ADIOS contributors

ANL: Rob Ross
Georgia Tech: Hasan Abbasi, Jay Lofstead, Karsten Schwan, Fang Zheng,
NCSU: Xiaosong Ma, Sriram Lakshminarasimhan, Abhijit Sachidananda, Michael Warren
Northwestern University: Alok Choudhary, Wei Keng Liao, Chen Jin
ORNL: Steve Hodson, Scott Klasky, Qing Gary Liu, Norbert Podhorszki, Steve Poole, Nagiza Samatova, Matthew Wolf
Rutgers University: Ciprian Docan, Fan Zhang, Manish Parashar
Sandia: Todd Kordenbrock
SUN: Wang Di

1 Introduction

1.1 Goals

As computational power has increased dramatically with the increase in the number of processors, input/output (IO) performance has become one of the most significant bottlenecks in today's high-performance computing (HPC) applications. With this in mind, ORNL and the Georgia Institute of Technology's Center for Experimental Research in Computer Systems have teamed together to design the Adaptive I/O System (ADIOS) as a componentization of the IO layer, which is scalable, portable, and efficient on different clusters or supercomputer platforms. We are also providing easy-to-use, high-level application program interfaces (APIs) so that application scientists can easily adapt the ADIOS library and produce science without diving too deeply into computer configuration and skills.

1.2 What Is ADIOS?

ADIOS is a state-of-the-art componentization of the IO system that has demonstrated impressive IO performance results on leadership class machines and clusters; sometimes showing an improvement of more than 1000 times over well known parallel file formats. ADIOS is essentially an I/O componentization of different I/O transport methods. This feature allows flexibility for application scientists to adopt the best I/O method for different computer infrastructures with very little modification of their scientific applications. ADIOS has a suite of simple, easy-to-use APIs. Instead of being provided as the arguments of APIs, all the required metadata are stored in an external Extensible Markup Language (XML) configuration file, which is readable, editable, and portable for most machines.

1.3 The Basic ADIOS Group Concept

The ADIOS "group" is a concept in which input variables are tagged according to the functionality of their respective output files. For example, a common scientific application has checkpoint files prefixed with `restart` and monitoring files prefixed with `diagnostics`. In the XML configuration file, the user can define two separate groups with tag names of adios-group as "restart" and "diagnostic." Each group contains a set of variables and attributes that need to be written into their respective output files. Each group can choose to have different I/O transport methods, which can be optimal for their I/O patterns.

1.4 Other Interesting Features of ADIOS

ADIOS contains a new self-describing file format, BP. The BP file format was specifically designed to support delayed consistency, lightweight data characterization, and resilience. ADIOS also contains python scripts that allow users to easily write entire "groups" with the inclusion of one include statement inside their Fortran/C code. Another interesting feature of ADIOS is that it allows

users to use multiple I/O methods for a single group. This is especially useful if users want to write data out to the file system, simultaneously capturing the metadata in a database method, and visualizing with a visualization method.

The read API enables reading arbitrary subarrays of variables in a BP file and thus variables written out from N processor can be read in on arbitrary number of processors. ADIOS also takes care of the endianness problem at converting to the reader's architecture automatically at reading time. Matlab reader is included in the release while the VisIt parallel interactive visualization software can read BP files too (from version 2.0).

ADIOS is fully supported on Cray XT and IBM BlueGene/P computers as well as on Linux clusters and Mac OSX.

1.5 Future ADIOS 2.0 Goals

One of the main goals for ADIOS 2.0 is to produce faster reads via indexing methods. Another goal is to provide more advanced data types via XML in ADIOS so that it will be compatible with F90/c/C++ structures/objects.

We will also work on the following advanced topics for ADIOS 2.0:

- A link to an external database for provenance recording.
- Autonomics through a feedback mechanism from the file system to optimize I/O performance. For instance, ADIOS can be adaptively changed from a synchronous to an asynchronous method or can decide when to write restart to improve I/O performance.
- A staging area for data querying, analysis, and in situ visualization.

2 Installation

2.1 Obtaining ADIOS

You can download the latest version from the following website

```
http://www.nccs.gov/user-support/adios
```

2.2 Quick Installation

To get started with ADIOS, the following steps can be used to configure, build, test, and install the ADIOS library, header files, and support programs.

```
cd trunk/

./configure –prefix=<install-dir> --with-mxml=<mxml-location>

make

make install
```

Note: There is a `runconf` batch script in the trunk set up for our machines. Studying it can help you setting up the appropriate environment variables and configure options for your system.

2.2.1 Linux cluster

The following is a snapshot of the batch scripts on Ewok, an Intel-based Infiniband cluster running Linux:

```
export MPICC=mpicc
export MPIFC=mpif90
export CC=pgcc
export FC=pgf90
export CFLAGS="-fPIC"
./configure --prefix = <location for ADIOS software installation>
            --with-mxml=<location of mini-xml installation>
            --with-hdf5=<location of HDF5 installation>
            --with-netcdf=<location of netCDF installation>
```

The compiler pointed by MPICC is used to build all the parallel codes and tools using MPI, while the compiler pointed by CC is used to build the sequential tools. In practice, mpicc uses the compiler pointed by CC and adds the MPI library automatically. On clusters, this makes no real difference, but on Bluegene, or Cray XT, parallel codes are built for compute nodes, while the sequential tools are built for the login nodes. The –fPIC compiler flag is needed only if you build the Matlab tools.

2.2.2 Cray XT5

To install ADIOS on a Cray XT5, the right compiler commands and configure flags need to be set. The required commands for ADIOS installation on Jaguar are as follows:

```
export CC=cc
export FC=ftn
./configure --prefix = <location for ADIOS software installation>
            --with-mxml=<location of mini-xml installation>
            --with-hdf5=<location of HDF5 installation>
            --with-netcdf=<location of netCDF installation>
```

2.2.3 Support for Matlab

Matlab requires ADIOS be built with the GNU C compiler. It also requires relocatable codes, so you need to add the –fPIC flag to CFLAGS before configuring ADIOS. The matlab reader is not built automatically at make and is not installed with ADIOS. You need to compile it with Matlab's MEX compiler after the make and copy the files manually to somewhere where Matlab can see them.

```
cd tools/matlab
make matlab
```

2.3 ADIOS Dependencies

2.3.1 Mini-XML parser (required)

The Mini-XML library is used to parse XML configuration files. Mini-XML can be downloaded from

`http://www.minixml.org/software.php`

2.3.2 MPI and MPI-IO (required)

MPI and MPI-IO is required for the ADIOS 1.2 release.

Currently, most large-scale scientific applications rely on the Message Passing Interface (MPI) library to implement communication among processes. For instance, when the Portable Operating System Interface (POSIX) is used as transport method, the rank of each processor in the same communication group, which needs to be retrieved by the certain MPI APIs, is commonly used in defining the output files. MPI-IO can also be considered the most generic I/O library on large-scale platforms.

2.3.3 Fortran90 compiler (optional)

The Fortran 90 interface and example codes are compiled only if there is an f90 compiler available. By default it is required but you can disable it with the option `--disable-fortran`.

2.3.4 Serial NetCDF-3 (optional)

The `bp2ncd` converter utility to NetCDF format is built only if NetCDF is available. Currently ADIOS uses the NetCDF-3 library. Use the option `--with-netcdf=<path>` or ensure that the NETCDF_DIR environment variable is set before configuring ADIOS.

2.3.5 Serial HDF5 (optional)

The `bp2h5` converter utility to HDF5 format is built only if a HDF5 library is available. Currently ADIOS uses the 1.6 version of the HDF5 API but it can be built and used with the 1.8.x version of the HDF5 library too. Use the option `--with-hdf5=<path>` when co

nfiguring ADIOS.

2.3.6 PHDF5 (optional)

The transport method writing files in the Parallel HDF5 format is built only if a parallel version of the HDF5 library is (also) available. You need to use the option `--with-phdf5=<path>` to build this transport method.

If you define Parallel HDF5 and do not define serial HDF5, then bp2h5 will be built with the parallel library.

Note that if you build this transport method, ADIOS will depend on PHDF5 when you link any application with ADIOS even if you the application does not intend to use this method.

If you have problems compiling ADIOS with PHDF5 due to missing flags or libraries, you can define them using

```
--with-phdf5-incdir=<path>,
--with-phdf5-libdir=<path> and
--with-phdf5-libs=<link time flags and libraries>
```

2.3.7 NetCDF-4 Parallel

The NC4 transport method writes files using the NetCDF-4 library which in turn is based on the parallel HDF5 library. You need to use the option `--with-nc4par=<path>` to build this transport method. Also, you need the parallel HDF5 library.

2.3.8 Read-only installation

If you just want the read API to be compiled for reading BP files, use the `--disable-write` option.

2.4 Full Installation

The following list is the complete set of options that can be used with configure to build ADIOS and its support utilities:

```
--help              print the usage of ./configure command
--with-tags[=TAGS]  include additional configurations [automatic]
--with-mxml=DIR     Location of Mini-XML library
--with-hdf5=<location of HDF5 installation>
--with-hdf5-incdir=<location of HDF5 includes>
--with-hdf5-libdir=<location of HDF5 library>
--with-phdf5=<location of PHDF5 installation>
--with-phdf5-incdir=<location of PHDF5 includes>
--with-phdf5-libdir=<location of PHDF5 library>
```

```
  --with-netcdf=<location of NetCDF installation>
  --with-netcdf-incdir=<location of NetCDF includes>
  --with-netcdf-libdir=<location of NetCDF library>
  --with-nc4par=<location of NetCDF 4 Parallel installation>
  --with-nc4par-incdir=<location of NetCDF 4 Parallel includes>
  --with-nc4par-libdir=<location of NetCDF 4 Parallel library>
  --with-nc4par-libs=<linker flags besides -L<nc4par_libdir>, e.g. -
lnetcdf
```

Some influential environment variables are lists below:

```
  CC          C compiler command
  CFLAGS      C compiler flags
  LDFLAGS     linker flags, e.g. -L<lib dir> if you have libraries in a
              nonstandard directory <lib dir>
  CPPFLAGS    C/C++ preprocessor flags, e.g. -I<include dir> if you
              have headers in a nonstandard directory <include dir>
  CPP         C preprocessor
  CXX         C++ compiler command
  CXXFLAGS    C++ compiler flags
  FC          Fortran compiler command
  FCFLAGS     Fortran compiler flags
  CXXCPP      C++ preprocessor
  F77         Fortran 77 compiler command
  FFLAGS      Fortran 77 compiler flags
  MPICC       MPI C compiler command
  MPIFC       MPI Fortran compiler command
```

2.5 Compiling applications using ADIOS

Adios configuration creates a text file that contains the flags and library dependencies that should be used when compiling/linking user applications that use ADIOS. This file is installed as `bin/adios_config.flags` under the installation directory by `make install`. A script, named `adios_config` is also installed that can print out selected flags. Moreover, if you copy the adios_config.flags file and remove all " characters from it, you can include that file in your Makefile and use the flags.

3 ADIOS Write API

As mentioned earlier, ADIOS writing is comprised of two parts: the XML configuration file and APIs. In this section, we will explain the functionality of the writing API in detail and how they are applied in the program.

3.1 Write API Description

3.1.1 Introduction

ADIOS provides both Fortran and C routines. All ADIOS routines and constants begin with the prefix "adios_". For the remainder of this section, only the C versions of ADIOS APIs are presented. The primary differences between the C and Fortran routines is that error codes are returned in a separate argument for Fortran as opposed to the return value for C routines.

A unique feature of ADIOS is group implementation, which is constituted by a list of variables and associated with individual transport methods. This flexibility allows the applications to make the best use of the file system according to its own different I/O patterns.

3.1.2 ADIOS-required functions

This section contains the basic functions needed to integrate ADIOS into scientific applications. ADIOS is a lightweight I/O library, and there are only seven required functions from which users can write scalable, portable programs with flexible I/O implementation on supported platforms:

adios_init—initialize ADIOS and load the configuration file
adios_open—open the group associated with the file
adios_group_size—pass the group size to allocate the memory
adios_write—write the data either to internal buffer or disk
adios_read—associate the buffer space for data read into
adios_close—commit write/read operation and close the data
adios_finalize—terminate ADIOS

You can add functions to your working knowledge incrementally without having to learn everything at once. For example, you can achieve better I/O performance on some platforms by simply adding the asynchronous functions adios_start_calculation, adios_end_calculation, and adios_end_iteration to your repertoire. These functions will be detailed below in addition to the seven indispensable functions.

The following provides the detailed descriptions of required APIs when users apply ADIOS in the Fortran or C applications.

3.1.2.1 adios_init

This API is required only once in the program. It loads XML configuration file and establishes the execution environment. Before any ADIOS operation starts, `adios_init` is required to be called to create internal representations of various data types and to define the transport methods used for writing.

int adios_init (const char *`xml_fname`)
Input:
`xml_fname` – string containing the name of the XML configuration file

Fortran example:
call adios_init ("config.xml", ierr)

3.1.2.2 adios_open

This API is called whenever a new output file is opened. adios_open, corresponding to `fopen` (not surprisingly), opens an adios-group given by `group_name` and associates it with one or a list of transport methods, which can be identified in future operations by the File structure whose pointer is returned as `fd_p`. The group name should match the one defined in the XML file. The I/O handle `fd_p` prepares the data types for the subsequent calls to write data using the io_handle. The third argument, `file_name`, is a string representing the name of the file. As the last argument, `mode` is a string containing a file access mode. It can be any of these three mode specifiers: "`r`," "`w`," or "`a`." Currently, ADIOS supports three access modes: "write or create if file does not exist," "read," and "append file." The call opens the file only if no coordination is needed among processes for transport methods that the users have chosen for this adios_group, such as POSIX method. Otherwise, the actual file will be opened in adios_group_size based on the provided argument `comm`, which will be examined in Sect. 4.1.2.3. As the last argument, we pass the pointer of coordination communicator down to the transport method layer in ADIOS. This communicator is required in MPI-IO–based methods such as collective and independent MPI-IO.

int adios_open (int64_t * `fd_p`, const char * `group_name`
,const char * `file_name`, const char * `mode`, `void *comm`)
Input:
`fd_p`—pointer to the internal file structure
`group_name`—string containing the name of the group
`file_name`—string containing the name of the file to be opened
`mode`—string containing a file access mode
comm— communicator for multi-process coordination
Fortran example:
call adios_open (handle, "restart", "restart.bp", "w", comm, ierr)

3.1.2.3 adios_group_size

This function passes the size of the group to the internal ADIOS transport structure to facilitate the internal buffer management and to construct the group index table. The first argument is the file handle. The second argument is the size of the payload for the group opened in the adios_open routine. This value can be calculated manually or through our python script. It does not affect read operation because the size of the data can be retrieved from the file itself. The third argument is the returned value for the total size of this group, including payload size and the metadata overhead. The value can be used for performance benchmarks, such as I/O speed.

int adios_group_size (int64_t * `fd_p`, uint64_t `group_size`, uint64_t * `total_size`)
Input:
`fd_p`—pointer to the internal file structure
`group_size`—size of data payload in bytes to be written out. If there is an integer 2 × 3 array, the payload size is 4*2*3 (4 is the size of integer)
output :
`total_size`—the total sum of payload and overhead, which includes name, data type, dimensions and other metadata)

Fortran example:
call adios_group_size (handle, groupsize, totalsize, ierr)

3.1.2.4 adios_write

The adios_write routine submits a data element `var` for writing and associates it with the given `var_name`, which has been defined in the adios group opened by adios_open. If the ADIOS buffer is big enough to hold all the data that the adios group needs to write, this API only copies the data to buffer. Otherwise, adios_write will write to disk without buffering. Currently, adios_write supports only the address of the contiguous block of memory to be written. In the case of a noncontiguous array comprising a series of subcontiguous memory blocks, `var` should be given separately for each piece.

In the next XML section, we will further explain that `var_name` is the value of attribute "name" while `var` is the value of attribute "gwrite," both of which are defined in the corresponding <var> element inside adios_group in the XML file. By default, it will be the same as the value of attribute "name" if "gwrite" is not defined.

int adios_write (int64_t `fd_p`, const char * `var_name`, void * `var`)
Input:
`fd_p`—pointer to the internal file structure
`var_name`—string containing the annotation name of scalar or vector in the XML file
`var` —the address of the data element defined need to be written

Fortran example:
call adios_write (handle, "myvar", v, ierr)

3.1.2.5 adios_read

The write API contains a read function (historically, the first one) that uses the same transport method and the xml config file to read in data. It works only on the same number of processes as the data was written out. Typically, checkpoint/restart files are written and read on the same number of processors and this function is the simplest way to read in data. However, if you need to read in on a different number of processors, or you do not want to carry the xml config file with the reading application, you should use the newer and more generic read API discussed in Section 6.

Similar to adios_write, adios_read submits 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. In other words, the value(s) of `var` can only be utilized after adios_close is performed. Here, `var_name` corresponds to the value of attribute "gread" in the <var> element declaration while `var` is mapped to the value of attribute "name." By default, it will be as same as the value of attribute "name" if "gread" is not defined.

int adios_read (int64_t `fd_p`, const char * `var_name`, uint64_t `read_size`,
void * `var`
)
Input:
`fd_p` – pointer to the internal file structure
`var_name` – the name of variable recorded in the file
`var` – the address of variable defined in source code
`read_size` – size in bytes of the data to be read in

Fortran example:
call adios_read (handle, "myvar", 8, v, ierr)

3.1.2.6 adios_close

The adios_close routine commits the writing buffer to disk, closes the file, and releases the handle. At that point, all of the data that have been copied during adios_write will be sent as-is downstream. If the handle were opened for read, it would fetch the data, parse it, and populate it into the provided buffers. This is currently hard-coded to use posix I/O calls.

int adios_close (int64_t * `fd_p`);
Input:
`fd_p` – pointer to the internal file structure

Fortran example:
call adios_close (handle, ierr)

3.1.2.7 adios_finalize

The `adios_finalize` routine releases all the resources allocated by ADIOS and guarantees that all remaining ADIOS operations are finished before the code exits. The ADIOS execution environment is terminated once the routine is fulfilled. The `proc_id` parameter provides users the opportunity to customize special operation on `proc_id`—usually the ID of the head node.

int adios_finalize (int `proc_id`)

Input:
`proc_id` – the rank of the processe in the communicator or the user-defined coordination variable

Fortran example:
call adios_finalize (rank, ierr)

3.1.3 Nonblocking functions

3.1.3.1 adios_end_iteration

The adios_end_iteration provides the pacing indicator. Based on the entry in the XML file, it will tell the transport method how much time has elapsed in a transfer.

3.1.3.2 adios_start_ calculation/ adios_end_calculation

Together, adios_start_calculation and adios_end_calculation indicate to the scientific code when nonblocking methods should focus on engaging their I/O communication efforts because the process is mainly performing intense, stand-alone computation. Otherwise, the code is deemed likely to be communicating heavily for computation coordination. Any attempts to write or read during those times will negatively impact both the asynchronous I/O performance and the interprocess messaging.

3.1.4 Other function

One of our design goals is to keep ADIOS APIs as simple as possible. In addition to the basic I/O functions, we provide another routine listed below.

3.1.4.1 adios_get_write_buffer

The adios_get_write_buffer function returns the buffer space allocated from the ADIOS buffer domain. In other words, instead of allocating memory from free memory space, users can directly use the allocated ADIOS buffer area and thus avoid copying memory from the ADIOS buffer to a user-defined buffer.

int adios_get_write_buffer (int64_t *fd_p*, const char * var_na*me*, uint64_t * *size*, void ** *buffer*)

Input:

fd_p – pointer to the internal File structure

`var_name` – name of the variable that will be read

size – size of the buffer to request

output:

buffer – initial address of read-in buffer for storing the data of `var_name`

3.1.5 Create a first ADIOS program

Figure 1 is a programming example that illustrates how to write a double-precision array `t` and a double-precision array with size of `NX` into file called "test.bp," which is organized in BP, our native tagged binary file format. This format allows users to include rich metadata associated with the block of binary data as well the indexing mechanism for different blocks of data (see Chap. 5).

```
/*example of parallel MPI write into a single file */
#include <stdio.h>    // ADIOS header file required
#include "adios.h"
int main (int argc, char *argv[])
{
    int i, rank, NX;
    double t [NX];
    // ADIOS variables declaration
    int64_t handle;
    uint_64 total_size;
    MPI_Comm comm = MPI_COMM_WORLD;

    MPI_Init ( &argc, &argv);
    MPI_Comm_rank (comm, &rank);

    // data initialization
    for ( i=0; i<NX; i++)
        t [i] = i * (rank+1) + 0.1;

    // ADIOS routines
    adios_init ( "config.xml");
    adios_open (&handle, "temperature", "data.bp", "w",&comm);
    adios_group_size (handle, 4, total_size);
    adios_write (handle, "NX", &NX);
    adios_write (handle, "temperature", t);
    adios_close (handle);
    adios_finalize (rank);
    MPI_Finalize();
    return 0;
}
```

Figure 1. ADIOS programming example.

4 XML Config File Format

4.1 Overview

XML is designed to allow users to store as much metadata as they can in an external configuration file. Thus the scientific applications are less polluted and require less effort to be verified again.

First, we present the XML template. Second, we demonstrate how to construct the XML file from the user's own source code. Third, we note how to troubleshoot and debug the errors in the file.

Abstracting metadata, data type, and dimensions from the source code into an XML file gives users more flexibility to annotate the arrays or variables and centralizes the description of all the data structures, which in return, allows I/O 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. It is useful that the XML name attributes 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 commas and must contain at least one non-numeric character. This is useful for incorporating expressions as various array dimensions elements. Figure 2 illustrates the corresponding XML configuration for the example we demonstrated in Figure 1.

At a minimum, a configuration document must declare an `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 main elements of the xml file format are of the format

<element-name attr1 attr2 …>

```
<adios-config>
    <adios-group>
        <var />
        ……
        <attribute />
        ……
    </adios-group>
    …
    <method>
    ……
    <buffer>
</adios-config>
```

Figure 2. Example XML configuration

4.2 adios-group

The adios-group element represents a container for a list of variables that share the common I/O pattern as stated in the basic concepts of ADIOS in first chapter. In this case, the group domain division logically corresponds to the different functions of output in scientific applications, such as restart, diagnosis, and snapshot. Depending on the different applications, adios-group can occur as many times as is needed.

4.2.1 Declaration

The following example illustrates how to declare an adios group inside an XML file. First we start with adios-group as our tag name, which is case insensitive. It has an indispensable attribute called "name," whose value is usually defined as a descriptive string indicating the function of the group. In this case, the string is called "restart," because the files into which this group is written are used as checkpoints. The second attribute "host-language" indicates the language in which this group's I/O operations are written. The value of attribute "coordination-communicator" is used to coordinate the operations on a shared file accessed by multiple processes in the same communicator domain. "Coordination-var" provides the ability to use the user-defined variable, for example mype, rather than an MPI communicator for file coordination.

<**adios-group name**="restart"
host-language="C"
coordination-communicator="comm"
coordination-var="mype"
time-index="iter"/>

Required:

- name—containing a descriptive string to name the group

Optional:

- host-language—language in which the source code for group is written
- coordination-communicator—MPI-IO writing to a shared file
- coordination-var—coordination variables for non-MPI methods, such as Datatap method
- time-index—time attribute variable

4.2.2 Variables

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

4.2.2.1 Declaration

The following is an example showing how to define a variable in the XML file.

```
<var name="z-plane ion particles"
     gwrite="zion"
     gread="zion_read"
     type="adios_real"
     dimensions="7,mimax"
     read="yes"/>
```

4.2.2.2 Attribute list

The attributes associated with var element as follows:

Required:

- name – the string name of variable stored in the output file
- type – the data type of the variable

Optional:

- gwrite – the value will be used in the python scripts to generate adios_write routines; the default value will be the same as attribute *name* if gwrite is not defined.
- gread – the value will be used in the python scripts to generate adios_read routines' the default value will be the same as attribute *name* if gread is not defined.
- path - HDF-5-style path for the element or path to the HDF-5 group or data item to which this attribute is attached. The default value is "/".
- dimensions - a comma-separated list of numbers and/or names that correspond to integer var elements determine the size of this item. If not specified, the variable is scalar.
- read – value is either *yes* or *no*; in the case of no, the adios_read routine will not be generated for this var entry. If undefined, the default value will be yes.

4.2.3 Attributes

The attribute element for adios_group provides the users with the ability to specify more descriptive information about the variables or group. The attributes can be defined in both static or dynamic fashions.

4.2.3.1 Declaration

The static type of attributes can be defined as follows:

<**attribute** name="experimental date"
path="/zion"
value="Sep-19-2008"
type="adios_real"/>

If an attribute has dynamic value that is determined by the runtime execution of the program, it can be specified as follows:
<**attribute** name="experimental date"
path="/zion"
var="time"/>

where var "time" need to be defined in the same adios-group.

4.2.3.2 Attribute list

Required:

- name - name of the attribute
- path – hierarchical path inside the file for the attribute
- value – attribute has static value of the attribute, mutually exclusive with the attribute *var*
- type – string or numeric type, paired with attribute *value*, in other words,, mutually exclusive with the attribute *var* also
- var – attribute has dynamic value that is defined by a variable in *var*

4.2.4 Gwrite/src

The element <Gwrite/src> is unlike <var> or <attribute>, which are parsed and stored in the internal file structure in ADIOS. The element <gwrite> only affects the execution of python scripts (see Chap. 10). Any content (usually comments, conditional statements, or loop statements) in the value of attribute "src" is copied identically into generated pre-processing files. Declaration

<gwrite src=" "/>

Required:

- src - any statement that needs to be added into the source code. This code must will be inserted into the source code, and must be able to be compiled in the host language, C or Fortran.

4.2.5 Global arrays

The **global-bounds** element is an optional nested element for the adios-group. It specifies the global space and offsets within that space for the enclosed variable elements. In the case of writing to a shared file, the global-bounds information is recorded in BP file and can be interpreted by converters or other postprocessing

tools or used to write out either HDF5 or NetCDF files by using PHDF5 or the PnetCDF method.

4.2.6 Time-index

ADIOS allows a dataset to be expanded in the space domain given by global bounds and in time domain. It is very common for scientific applications to write out a monitoring file at regular intervals. The file usually contains a group of time-based variables that have undetermined dimensional value on the time axis. ADIOS is Similar to NetCDF in that it accumulates the time-index in terms of the number of records, which theoretically can be added to infinity.

If any of variables in an adios group are time based, they can be marked out by adding the time-index variable as another dimension value.

4.2.6.1 Declaration

<**global-bounds dimensions**="nx_g, ny_g" **offsets**="nx_o,0"/>
… variable declarations …
</**global-bounds**>

Required:

- dimensions – the dimension of global space
- offsets – the offset of the data set in global space

Any variables used in the global-bounds element for dimensions or offsets declaration need to be defined in the same adios-group as either variables or attributes.

For detailed global arrays use, see the example illustrated in Section 11.8.

Changing I/O Without Changing Source: The method element provides the hook between the adios-group and the transport methods. The user employs a different transport method simply by changing the method attribute of the method element. If more than one method element is provided for a given group, each element 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, the user makes two element entries for the analysis adios-group. The first indicates how to write to disk, and the second performs 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.

4.2.7 Declaration

The transport element is used to specify the mapping of an I/O transport method, including optional initialization parameters, to the respective adios-group. There are two major attributes required for the method element:

<**transport group**="restart"

method="MPI"
priority="1"
iteration="100"/>

Required:

- group - corresponds to an adios-group specified earlier in the file.
- method – a string indicating a transport method to use with the associated adios-group

Optional:

- priority– a numeric priority for the I/O method to better schedule this write with others that may be pending currently
- base-path–the root directory to use when writing to disk or similar purposes
- iterations– a number of iterations between writes of this group used to gauge how quickly this data should be evacuated from the compute node

4.2.8 Methods list

As the componentization of the IO substrate, ADIOS supports a list of transport methods, described in Section 5:

- NULL
- POSIX
- MPI
- MPI-LUSTRE
- MPI-AMR
- PHDF5
- NC4 (NETCDF4)
- NSSI
- DATATAP
- DART
- DIMES
- MPI-CIO (as research method, not published in 1.2)
- ADAPTIVE (as research method, not published in 1.2)

4.3 Buffer specification

The buffer element defines the attributes for internal buffer size and creating time that apply to the whole application (Figure 3). The attribute allocate-time is identified as being either "now" or "oncall" to indicate when the buffer should be allocated. An "oncall" attribute waits until the programmer decides that all memory needed for calculation has been allocated. It then calls upon ADIOS to allocate buffer. There are two alternative attributes for users to define the buffer size: MB and free-memory-percentage.

4.3.1 Declaration

<**buffer size-MB**="100"
allocate-time="now" />

Required:

- size-MB – the user-defined size of buffer in megabytes. ADIOS can at most allocate from compute nodes. It is exclusive with free-memory-percentage.
- free-memory percentage – the user-defined percentage from 0 to 100% of freememory available on the machine. It is exclusive with size-MB.
- allocate-time – indicates when the buffer should be allocated

4.4 Enabling Histogram

ADIOS 1.2 has the ability to compute a histogram of the given variable's data values at write time. This is specified via the **<analysis>** tag in the XML file. The parameters "**adios-group**" and "**var**" specify the variable for which the histogram is to be performed. "**var**" is the name of the variable and "**adios-group**" is the name of the adios group to which the variable belongs to.

4.4.1 Declaration

The histogram binning intervals can be input in two ways via the XML file:

- By listing the break points as a list of comma separated values in the parameter "break-points"

  <**analysis** **adios-group**="temperature" **var**="temperature"
  break-points="0, 100, 200, 300" />

- By specifying the boundaries of the breaks, and the number of intervals between variable's min and max values

  <**analysis** **adios-group**="temperature" **var**="temperature"
  min="0" **max**="300" **count**="3"/>

Both inputs create the bins (-Inf, 0), [0, 100), [100, 200), [200, 300), [300, Inf). For this example, the final set of frequencies for these 5 binning intervals will be calculated.

Required:

- adios-group – corresponds to an adios-group specified earlier in the file.
- var - corresponds to a variable in adios-group specified earlier in the file.

Optional:

- break-points - list of comma separated values **sorted** in ascending order
- min - minimum value of the binning boundary
- max - maximum value of the binning boundary (it should be greater than min)
- count - number of break points between the min and max values

A valid set of binning intervals must be provided either by specifying "min," "max," and "count" parameters or by providing the "break-points." The intervals given under "break-points" will take precedence when calculating the histogram intervals, if "min," "max," and "count" as well as “break-points” are provided.

4.5 An Example XML file

```
<adios-config host-language="C">

  <adios-group name="temperature" coordination-communicator="comm">
    <var name="NX" type="integer"/>
    <var name="t" type="double" dimensions="NX"/>
    <attribute name="recorded date" path="/" value="Sep 19, 2008" type="string"/>
  </adios-group>

  <method group=" temperature " method="MPI"/>

  <buffer size-MB="1" allocate-time="now"/>
  <analysis adios-group="temperature" var="t" break-points="0, 100, 200, 300"/>

</adios-config>
```

Figure 3. Example XML file for time allocation.

5 Transport methods

Because of the time it can take to move data from one process to another or to write and read data to and from a disk, it is often advantageous to arrange the program so that some work can be done while the messages are in transit. So far, we have used non-blocking operations to avoid waiting. Here we describe some details for arranging a program so that computation and I/O can take place simultaneously.

5.1 Synchronous methods

5.1.1 NULL

The ADIOS NULL method allows users to quickly comment out an ADIOS group by changing the transport method to "NULL," users can test the speed of the routine by timing the output against no I/O. This is especially useful when working with asynchronous methods, which take an indeterminate amount of time. Another useful feature of this I/O is that it quickly allows users to test out the system and determine whether bugs are caused by the I/O system or by other places in the codes.

5.1.2 POSIX

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

Additional features may be added to the ADIOS POSIX transport method over time. A new transport method with a related name, such as POSIX-ASCII, may be provided to perform I/O 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.

5.1.3 MPI

Many large-scale scientific simulations generate a large amount of data, spanning thousands of files or datasets. The use of MPI-IO reduces the amount of files and thus is helpful for data management, storage, and access.

The original MPI method was developed based on our experiments with generating the better MPI-IO performance on the ORNL Jaguar machine. Many of the insights have helped us achieve excellent performance on both the Jaguar XT4 machine and on the other clusters. 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 grouping generates one file per coordination-communicator with the data stored sequentially based on the process rank within the communicator. Figure 4 presents in the example of GTC code, 32 processes in the same Toroidal zone write to one integrated file. Additional serialization of the MPI_File_open calls is done using this communicator as well because each process may have a different size data payload. Rank 0 calculates the size that 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_open so that each process has its own file pointer for file seek and other I/O operations.

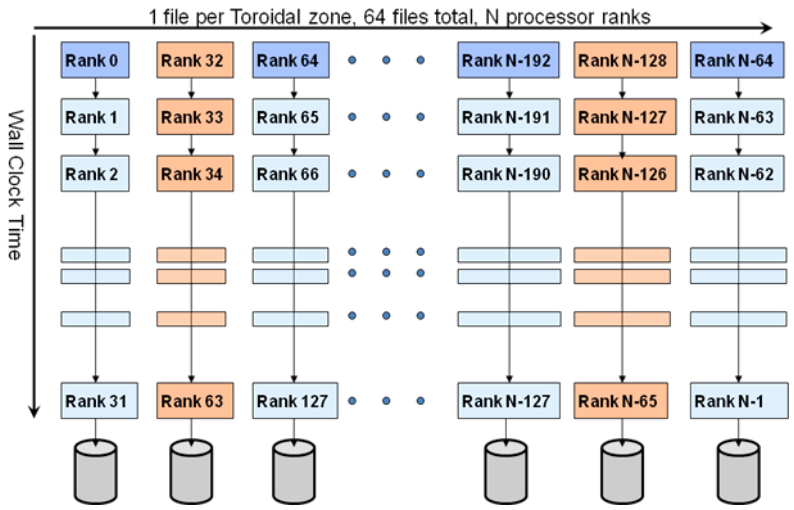

Figure 4. Server-friendly metadata approach: offset the create/open in time

We built the MPI transport method, mainly with Lustre in mind because it has been the primary parallel storage service we have available. However, 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.

The MPI transport method is the most mature, fully featured, and well tested method in ADIOS. We recommend to anyone creating a new transport method

that they study it as a model of full functionality and some of the advantages that can be made through careful management of the storage resources.

5.1.4 MPI_LUSTRE

The MPI_LUSTRE method is the MPI method with stripe alignment to achieve even greater write performance on the Lustre file system. Each writing process' data is aligned to Lustre stripes. This results in better parallelization of the storage elements. The drawback of using this method is that empty chunks are created between the data sets of the separate processes in the output file, and thus the file size is larger than with using the MPI method. The size of an empty space is the difference between the size of the output data of one writing process and the total size of Lustre stripes that can hold that amount of data, so that the next writing process' output starts aligned with another stripe. Choose the stripe size for the output file therefore carefully, to make the empty space as small as possible.

The following XML snippet shows how to use the MPI_LUSTRE method in ADIOS.

```
<method group="temperature" method="MPI_LUSTRE">
  stripe_count=16,stripe_size=4194304,block_size=4194304
</method>
```

There are three key parameters used in this method.

- **stripe_count** specifies how many storage targets to use for the whole output file. If not set, the default value is 4.
- **stripe_size** specifies Lustre stripe size in bytes. If not set, the default value is 1048576 (i.e. 1 MB).
- **block_size** specifies the size of each I/O write request. As an example, if total data size to be written from one process is 800 MB at a time, and you want ADIOS to issue twenty I/O write requests issued from one process to Lustre during the writing, then the block_size should be 40MB.

5.1.5 MPI_AMR

The MPI_AMR method is designed to maximize write performance for applications such as adaptive mesh refinement (AMR) on the Lustre file system. In AMR-like applications, each processor outputs varying amount of data and some can output very small size data. Based upon MPI_LUSTRE method, MPI_AMR further improves the write speed by

1. aggregating data from multiple MPI processors into large chunks. This effectively increases the size of each request and reduces the number of I/O requests.
2. threading the metadata operations such as file open. Users are encouraged to call adios_open and adios_group_size API as early as possible. In case Lustre MDS has a performance hit, the overall metadata performance

won't be affected. The following code snippet shows a typical way of using this method to improve metadata performance.

```
adios_open(...);
adios_group_size(...);
……
//do your computation
……
adios_write(..);
adios_write(..);
adios_close(..);
```

3. further removing communication and wide striping overhead by writing out subfiles. Please refer to POSIX method on how to read data from subfiles.

The following XML snippet shows how to use MPI_AMR method in ADIOS.
There are five key parameters used in this method.

```
<method group="tracers" method="MPI_AMR">
  stripe_count=1;stripe_size=10485760;block_size=10485760;
  num_aggregators=2400;merging_pgs=0
</method>
```

- **stripe_count** specifies how many storage targets to stripe across for each subfile. If not set, the default value is Lustre's default value (i.e. 4). It is recommended that this value set to 1 in the ADIOS 1.2 release.
- **stripe_size** specifies Lustre stripe size in bytes. If not set, the default value is 1048576 (i.e. 1 MB).
- **block_size** specifies the size of each I/O write request. As an example, if block_size is 4 MB and the total data to write out is 8 MB, there will be two I/O write requests issued.
- **num_aggregators** specifies the number of aggregators to use.
- **merging_pgs** is a flag that specifies whether ADIOS process groups are merged during aggregation operation. It is recommended that this flag set to 0 in the ADIOS 1.2 release.

Now for the selection of num_aggregators parameter, suppose you have a MPI job with 120,000 processors and the number of aggregator is set to 2400. Then each aggregator will aggregate the data from 120,000/2400=50 processors. Carefully note that setting num_aggregators too small can incur out-of-memory issue.

5.1.6 PHDF5

HDF5, as a hierarchical File structure, has been widely adopted for data storage in various scientific research fields. Parallel HDF5 (PHDF5) provides a series of APIs to perform the I/O operations in parallel from multiple processors, which dramatically improves the I/O performance of the sequential approach to

read/write an HDF5 file. In order to make the difference in transport methods and file formats transparent to the end users, we provide a mechanism that write/read an HDF5 file with the same schema by keeping the same common adios routines with only one entry change in the XML file. This method provides users with the capability to write out exactly the same HDF5 files as those generated by their original PHDF5 routines. Doing so allows for the same analysis tool chain to analyze the data.

Currently, HDF5 supports two I/O modes: independent and Collective read or write, which can use either the MPI or the POSIX driver by specifying the dataset transfer property list in H5Dwrite function calls. In this release, only the MPI driver is supported in ADIOS; later on, both I/O drivers will be supported by changing the attribute information for PHDF5 method elements in XML.

5.1.7 NetCDF4

Another widely accepted standard file format is NetCDF, which is the most frequently used file format in the climate and weather research communities. Beginning with the NetCDF 4.0 release, NetCDF has added PHDF5 as a new option for data storage called the "netcdf-4 format". When a NetCDF4 file is opened in this new format, NetCDF4 inherits PHDF5's parallel I/O capabilities.

The NetCDF4 method creates a single shared filed in the "netcdf-4 format" and uses the parallel I/O features. The NetCDF4 method supports multiple open files. To select the NetCDF4 method use "NC4" as the method name in the XML file.

Restrictions: Due to the collective nature of the NetCDF4 API, there are some legal XML files that will not work with the NetCDF4 method. The most notable incompatibility is an XML fragment that creates an array variable without a surrounding global-bounds. Within the application, a call to adios_set_path() is used to add a unique prefix to the variable name. A rank-based prefix is an example.

```
<?xml version="1.0"?>
<adios-config host-language="C">
  <adios-group name="atoms " coordination-communicator="comm">
    <var name="nparam" type="integer"/>
    <var name="ntracked" type="integer"/>
    <var name="atoms " type="real" dimensions="nparam,ntracked"/>
  </adios-group>
<method group="atoms" method="NC4 "/>
<buffer size-MB="1" allocate-time="now"/>
</adios-config>
```

Figure 5. Example XML

```
char path[1024];
adios_init ("config.xml");
adios_open (&adios_handle, "atoms", filename, "w", &comm);
sprintf(path, "node_%d_", myrank);
adios_set_path(adios_handle, path);
#include "gwrite_atoms.ch"
adios_close (adios_handle);
adios_finalize (myrank);
```

Figure 6. Example C source

This technique is an optimization that allows each rank to creates a variable of the exact dimensions of the data being written. In this example, each rank may be tracking a different number of atoms.

The NetCDF4 collective API expects each rank to write the same variable with the same dimensions. The example violates both of these expectations.

Note: NetCDF4 files created in the new "netcdf-4 format" cannot be opened with existing tools linked with NetCDF 3.x. However, NetCDF4 provides a backward compatibility API, so that these tools can be relinked with NetCDF4. After relink, these tools can open files in the "netcdf-4 format".

5.1.8 Other methods

ADIOS provides an easy plug-in mechanism for users or developers to design their own transport method. A step-by-step instruction for inserting a new I/O method is given in Section 12.1. Users are likely to choose the best method from among the supported or customized methods for the running their platforms, thus avoiding the need to verify their source codes due to the switching of I/O methods.

5.2 Asynchronous methods

5.2.1 Network Scalable Service Interface (NSSI)

The Network Scalable Service Interface (NSSI) is a client-server development framework for large-scale HPC systems. NSSI was originally developed out of necessity for the Lightweight File Systems (LWFS) project, a joint effort between researchers at Sandia National Laboratories and the University of New Mexico. The LWFS approach was to provide a core set of fundamental capabilities for security, data-movement, and storage, and allow extensibility through the development of additional services. The NSSI framework was designed to be the vehicle to enable the rapid development of such services.

The NSSI method is composed of two components – a client method and a staging service. The client method does not perform any file I/O. Instead, all ADIOS operations become requests to the staging service. The staging service is an ADIOS application, which allows the user to select any ADIOS method for output.

Client requests fall into two categories – pass-through and cached. Pass-through requests are requests that are synchronous on the staging service and return an error immediately on failure. adios_open() is an example of a pass-through request. Cached requests are requests that are asynchronous on the staging service and return an error at a later time on failure. adios_write() is an example of a cached request. All data cached for a particular file is aggregated and flushed when the client calls adios_close().

Each component requires its own XML config file. The client method can be selected in the client XML config using "NSSI" as the method. The service XML config must be the same as the client XML config except that the method is "NSSI_FILTER". When the NSSI_FILTER method is selected, the "submethod" parameter is required. The "submethod" parameter specifies the ADIOS method that the staging service will use for output. Converting an existing XML config file for use with NSSI is illustrated in the following three Figures.

```
<method method="MPI" group="atoms">max_storage_targets=160</method>
```

Figure 7. Example Original Client XML

```
<method method="NSSI" group="atoms/>
```

Figure 8. Example NSSI Client XML

```
<method method="NSSI_FILTER" group="atoms">
  submethod="MPI";subparameters="max_storage_targets=160"
</method>
```

Figure 9. Example NSSI Staging Service XML

After creating new config files, the application's PBS script (or other runtime script) must be modified to start the staging service prior to application launch and stop the staging service after application termination. The ADIOS distribution includes three scripts to help with these tasks.

The start.nssi.staging.sh script launches the staging service. start.nssi.staging.sh takes two arguments – the number of staging services and an XML config file.

The create.nssi.config.sh script creates an XML file that the NSSI method uses to locate the staging services. create.nssi.config.sh takes two arguments – the name of the output config file and the name of the file containing a list of service contact info. The service contact file is created by the staging service at startup. The

staging service uses the ADIOS_NSSI_CONTACT_INFO environment variable to determine the pathname of the contact file.

The kill.nssi.staging.sh script sends a kill request to the staging service. kill.nssi.staging.sh takes one argument – the name of the file containing a list of service contact info (ADIOS_NSSI_CONTACT_INFO). The staging service will gracefully terminate.

```
#!/bin/bash
#PBS -l walltime=01:00:00,size=128

export RUNTIME_PATH=/tmp/work/$USER/genarray3d.$PBS_JOBID
mkdir -p $RUNTIME_PATH
cd $RUNTIME_PATH

export ADIOS_NSSI_CONTACT_INFO=$RUNTIME_PATH/nssi_contact.xml
export ADIOS_NSSI_CONFIG_FILE=$RUNTIME_PATH/nssi_config.xml
$ADIOS_DIR/scripts/start.nssi.staging.sh 4 $RUNTIME_PATH/genarray3d.server.xml >server.log 2>&1 &
sleep 3
$ADIOS_DIR/scripts/create.nssi.config.sh $ADIOS_NSSI_CONFIG_FILE $ADIOS_NSSI_CONTACT_INFO

aprun -n 64 $ADIOS_SRC_PATH/tests/genarray/genarray $RUNTIME_PATH/test.output 4 4 4 128 128 80 >runlog

$ADIOS_DIR/scripts/kill.nssi.staging.sh $ADIOS_NSSI_CONTACT_INFO
```

Figure 10. Example PBS script with NSSI Staging Service

Figure 10 is a example PBS script that highlights the changes required to launch the NSSI staging service.

Required Environment Variables. The NSSI Staging Service requires that the ADIOS_NSSI_CONTACT_INFO variable be set. This variable specifies the full pathname of the file that the service uses to save its contact information. Depending on the platform, the contact information is a NID/PID pair or a hostname/port pair. Rank0 is responsible for gathering the contact information from all members of the job and writing the contact file. The NSSI method requires that the ADIOS_NSSI_CONFIG_FILE variable be set. This variable specifies the full pathname of the file that contains the complete configuration information for the NSSI method. A configuration file with contact information and reasonable defaults for everything else can be created with the create.nssi.config.sh script.

Calculating the Number of Staging Services Required. Remember that all adios_write() operations are cached requests. This implies that the staging service must have enough RAM available to cache all data written by its clients between adios_open() and adios_close(). The current aggregation algorithm requires a buffer equal to the size of the data into which the data is aggregated. The start.nssi.staging.sh script launches a single service per node, so the largest amount of data that can be cached per service is 50% of the memory on a node minus system overhead. System overhead can be estimated at 500MB. If a node has 16GB of memory, the amount of data that can be cached is 7.75GB ((16GB-

500MB)/2). To balance the load on the staging services, the number of clients should be evenly divisible by the number of staging services.

Calculating the Number of Additional Cores Required for Staging. The NSSI staging services run on compute nodes, so additional resources are required to run the job. For each staging service required, add the number of cores per node to the size of the job. If each node has 12 cores and the job requires 16 staging services, add 192 cores to the job.

The NSSI transport method is experimental and is not included with the public version of the ADIOS source code in this release; however it is available for use on the XT4 and XT5 machines at ORNL.

5.2.2 DataTap

DataTap is an asynchronous data transport method built to ensure very high levels of scalability through server-directed I/O. It is implemented as a request-read service designed to bridge the order-of-magnitude difference between available memories on the I/O partition compared with 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") (see Figure 11).

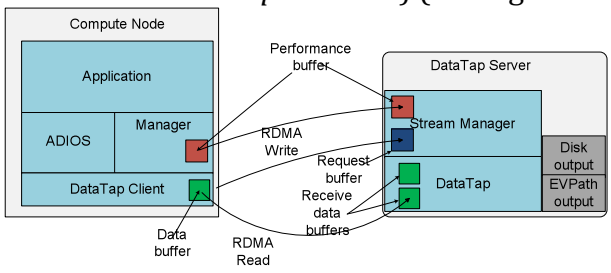

Figure 11. DataTap architecture

Upon application request, the compute node marks up the data in PBIO 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 runtime. When the server has sufficient buffer space, a remote direct memory access (RDMA) read request is issued to the client to read the remote data into a local buffer. The data are then written out to disk or transmitted over the network as input for further processing in the I/O Graph.

We used the Gyrokinetic Turbulence Code (GTC) as an experimental tested 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.

The DataTap transport method is experimental and is not included with the public version of the ADIOS source code in this release; however it is available for use on the XT4 and XT5 machines at ORNL.

5.2.3 Decoupled and Asynchronous Remote Transfers (DART)

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

The DARTClient module is a light library that provides 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 encode the data written by the application into a local transport buffer. Once it has collected data from a simulation, DARTClient notifies the DARTServer through a coordination channel that it has data available to send out. DARTClient then returns and allows the application to continue its computations while data are asynchronously extracted by the DARTServer.

The DARTServer module is a stand-alone service that runs independently of a simulation on a set of dedicated nodes in the staging area. It transfers data from the DARTClient and can save it to local storage system, e.g., Lustre file system, stream it to remote sites, e.g., Ewok cluster, or serve it directly from the staging area to other applications. One instance of the DARTServer can service multiple DARTClient 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 from the clients in parallel. 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.

DART is an asynchronous method available in ADIOS, that can be selected by specifying the transport method in the external ADIOS XML configuration file as “DART”.

```
<method priority=”3” method="DART" group="fluxdiag"/>
```

Figure 12. Select DART as a transport method in the configuration file example.

To make use of the DART transport, an application job needs to also run the DARTServer component together with the application. The server should be

configured and started before the application as a separate job in the system. For example:

```
aprun  -n  $SPROC ./dart_server –s $SPROC –c $PROC &> log.server &
```

Figure 13. Start the server component in a job file first.

The variable $SPROC represents the number of server instances to run, and the variable $PROC represents the number of application processes. For example if the job script runs a coupling scenario with two applications that run on 128 and 432 processors respectively, then the value of $PROC is 560. The '&' character at the end of the line would place the 'aprun' command in the background, and will allow the job script to continue and run the other applications. The server processes produce a configuration file, i.e., 'conf' that is used by the DARTClient component to connect to the servers. This file contains the 'nid' (network identifier), and 'pid' (process identifier) of the master server, which coordinates the client registration and discovery process. The job script should wait for the servers to start-up and produce the 'conf' file, which it can then export to environment variables, e.g., P2TNID, and P2TPID. The clients can use these variables to connect to the server. Exporting the master server identifier through environment variable prevents the larger number of clients from accessing the file system at once.

```
while [ ! –f conf ]; do
    echo "Waiting for servers to start-up"
    sleep 2s
done

while read line; do
    export set "${line}"
done < conf
```

Figure 14. Wait for server start-up completion and export the configuration to environment variables.

The server component will terminate automatically when the applications will finish. The DARTClient components will send an unregister message to the server before they finish execution, and the servers will exit after they receive $PROC unregister messages.

The DART transport method is experimental and is not included with the public version of the ADIOS source code in this release; however it is available for use on the XT4 and XT5 machines at ORNL.

5.2.4 (DIMES)

5.3 Other research methods at ORNL

5.3.1 MPI-CIO

MPI-IO defines a set of portable programming interfaces that enable multiple processes to have 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 generates 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 the <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 the offset into this global space to which 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 the 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.

This research method is installed on Jaguar at ORNL only but is not part of the public release.

5.3.2 MPI-AIO

The initial implementation of the asynchronous MPI-IO method (MPI-AIO) is patterned after the MPI-IO method. Scheduled metadata commands are performed with the same serialization of MPI_Open calls as given in Figure 4 on page 22.

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

Another important factor is the amount of available ADIOS buffer space. In the MPI-IO method, data are 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 buffer allocation is 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 asynchronous I/O with a computation. In ADIOS version 1.2, 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 and adios_end_calculation calls, and complete I/O on adios_end_iteration calls.

This research module is not released in ADIOS 1.2.

6 ADIOS Read API

6.1 Introduction

We can read in any variable and any sub-array of a variable with the read API as well as the attributes. There were three design choices when creating this API:

1. Groups in the BP files are handled separately

   Most BP files contain a single group and the variables and attributes in that group have their paths so it looks like they are organized into a hierarchy. If a BP file contains more than one groups, the second group can have a variable with the same path and name as a variable in the first group. We choose not to add the name of the groups to the root of all paths because that is inconvenient for the majority of the BP files containing a single group.

2. Dimensions of arrays are reported differently for C and Fortran

   When reading from a different language than writing, the storage order of the dimensions is the opposite. Instead of transposing multidimensional arrays in memory to order the data correctly at read time, simply the dimensions are reported reversed.

3. The C API returns structures filled with information while the Fortran API returns information in individual arguments

   Since the BP file format is metadata rich, and the metadata is immediately accessible in the footer of the file, we can have an easy to use API with few functions. The open function returns information on the number of elements and timesteps and the list of groups in the file. The group open returns the list of variables and attributes in the group. The inquiry of a variable returns not just the type and dimensionality of a variable but also the global minimum and maximum of it without reading in the content of the variable from the file.

The read API library has two versions. The MPI version should be used in parallel programs. Only the rank=0 process reads the footer of the file and broadcasts it to the other processes in adios_fopen(). File access is handled through MPI-IO functions. Sequential programs can use any of the two versions but if you do not want dependency on MPI, link your program with the non-MPI version, which uses POSIX I/O functions. In this case, you need to compile your code with the -D_NOMPI option. There is no difference in performance or functionality in the two versions (in sequential applications).

Note that the write API contains the adios_read() function, which is useful to read in data from the same number of processors as the data was written from, like handling checkpoint/restart data (see Section 3.1.2.5.). However, if you need to

read in from a different number of processors or to read in only a subset of an array variable, you need to use this read API.

6.2 Read C API description

Note: for Fortran, please read section 6.4 on page 40.

The sequence of reading in a variable from the BP file is

- open file
- open a group
- inquiry the variable to get type and dimensions
- allocate memory for the variable
- read in variable (whole or part of it)
- free varinfo data structure
- close group
- close file

Example codes using the C API are

- examples/C/read_all/read_all.c
- examples/C/global-array/adios_read_global

6.2.1 adios_errmsg / adios_errno

```
int    adios_errno
char * adios_errmsg()
```

If an error occurrs during the call of a C api function, it either returns NULL (instead of a pointer to an allocated structure) or a negative number. It also sets the integer `adios_errno` variable (the negative return value is actually -1 times the errno value). Moreover, it prints the error message into an internal buffer, which can be retrieved by `adios_errmsg()`.

Note that adios_errmsg() returns the pointer to the internal buffer instead of duplicating the string, so refrain from writing anything into it. Also, only the last error message is available at any time.

6.2.2 adios_fopen

```
ADIOS_FILE * adios_fopen (const char * fname, MPI_Comm comm)
```

ADIOS FILE is a struct of

uint64_t	fh;	File handler
int	groups_count;	Number of adios groups in file
int	vars_count;	Number of variables in all groups
int	attrs_count;	Number of attributes in all groups
int	tidx_start;	First timestep in file, usually 1

int	ntimesteps;	Number of timesteps in file. There is always at least one timestep
int	version;	ADIOS BP version of file format
uint64_t	file_size;	Size of file in bytes
int	endianness;	0: little endian, 1: big endian You do not need to care about this.
char **	group_namelist;	Names of the adios groups in the file (cf. groups_count)

The array for the list of group names is allocated in the function and is freed in the close function.

If you use the MPI version of the library, pass the communicator, which is the communicator of all processes that call the open function. Rank=0 process broadcasts the metadata to the other processes so that we avoid opening the file from many processes at once. If you use the non-MPI version of the library, just pass on an arbitrary integer value, which is not used at all.

6.2.3 adios_fclose

```
int adios_fclose (ADIOS_FILE *fp)
```

You are expected to close a file when you do not need it anymore. This function releases a lot of internal memory structures.

6.2.4 adios_gopen / adios_gopen_byid

```
ADIOS_GROUP * adios_gopen (ADIOS_FILE *fp, const char * grpname)
ADIOS_GROUP * adios_gopen_byid (ADIOS_FILE *fp, int grpid)
```

You need to open a group to get access to its variables and attributes. You can open a group either by its name returned in the ADIOS_FILE struct's group_namelist list of strings or by its index, which is the index of its name in this list of names.

You *can* have several groups open at the same time.

ADIOS_GROUP is a struct of

uint64_t	gh;	Group handler
int	grpid;	group index (0..ADIOS_FILE.groups_count-1)
int	vars_count;	Number of variables in this adios group
char **	var_namelist;	Variable names in a char* array
int	attrs_count;	Number of attributes in this adios group
char **	attr_namelist;	Attribute names in a char* array
ADIOS_FILE *	fp;	pointer to the parent ADIOS_FILE struct

The arrays for the list of variable names and attribute name are allocated in the function and are freed in the group close function.

Note that one can modify the ADIOS_GROUP's namelists because they are not used in the discovery of the variables. However, in index-based queries below, the index of the variable is the index of the variable's position in the original order of the list. If one sorts this list for ordered printouts, one need to remember the original indices of the variables or to identify the variables by name.

6.2.5 adios_gclose

```
int adios_gclose (ADIOS_GROUP *gp)
```

You need to close the group when you do not need it anymore.

6.2.6 adios_inq_var / adios_inq_var_byid

```
ADIOS_VARINFO * adios_inq_var (ADIOS_GROUP *gp, const char * varname)
ADIOS_VARINFO * adios_inq_var_byid (ADIOS_GROUP *gp, int varid)
```

This function should be used if you want to discover the type and dimensionality of a variable or want to get the minimum/maximum/average/standard_deviation values without reading in the data. You can refer to the variable with its name (full path) in the ADIOS_GROUP struct's var_namelist or by the index in that list.

`ADIOS_VARINFO` structure is allocated in the function but there is no corresponding closing function, therefore user has to free the ADIOS_VARINFO* pointer yourself when you do not need it anymore by using the adios_free_varinfo() function.

ADIOS_VARINFO is a struct of

```
int           grpid;          group index (0..ADIOS_FILE.groups_count-1)
int           varid;          variable index (0..ADIOS_GROUP.var_count-1)
enum ADIOS_DATATYPES type;    type of variable
int           ndim;           number of dimensions, 0 for scalars
uint64_t *    dims;           size of each dimension
int           timedim;        -1: variable has no timesteps in file,
                              >=0: which dimension is time
void    *     value;          value of a scalar variable, NULL for array.

void    *     gmin;           minimum value in an array variable.
void    *     gmax;           maximum value of an array variable
void    *     gavg;           average value of an array variable
void    *     gstd_dev;       standard deviation value of an array variable
                              (over all timesteps, for scalars they are = value)

void    *     mins;           minimum per each timestep
void    *     maxs;           maximum per each timestep
void    *     avgs;           average per each timestep
```

```
void    *    std_dev;          standard deviation per each timestep
                               (array of timestep elements)

struct ADIOS_HIST {
  uint32_t     num_breaks;     number of break points of the histogram
  double       min;            minimum of binning boundary
  double       max;            maximum of binning boundary
  double   *   breaks;         break points of the histogram
  uint32_t ** frequencies;     histogram values per timestep
  uint32_t *  gfrequencies;    histogram values for all timesteps
} *hist;                       NULL if histogram binning interval was not
                               formed correctly at write time
```

For complex numbers, the statistics in ADIOS_VARINFO, like gmin, gavg, std_devs etc, are of base type double. They also have an additional dimension that stores the statistics for the magnitude, the real part, and the imaginary part of the complex number, individually. For example, gmin[0] holds the overall minimum value of the magnitude of the complex numbers. gmin[1] and gmin [2] contain the global minimums for the real and the imaginary parts, respectively.

6.2.7 adios_free_varinfo

```
void adios_free_varinfo (ADIOS_VARINFO *cp)
```

Frees up the ADIOS_VARINFO* structure returned by adios_inq_var().

6.2.8 adios_read_var / adios_read_var_byid

```
int64_t adios_read_var (ADIOS_GROUP    * gp,
                        const char     * varname,
                        const uint64_t * start,
                        const uint64_t * count,
                        void           * data)

int64_t adios_read_var_byid (ADIOS_GROUP * gp,
                             int varid,
                             const uint64_t * start,
                             const uint64_t * count,
                             void * data)
```

This function is used to read in the content of a variable, or a subset of it. You need to allocate memory for receiving the data before calling this function. The subset (or the entire set) is defined by the *start* and *count* in each dimension. The *start* and *count* arrays must have as many elements as many dimensions the variable has (i.e. ADIOS_VARINFO.ndim). *Start* contains the starting offsets for each dimension and *count* contains the number of elements to read in a given dimension. If you want to read in the entire variable, *start* should be an array of zeros and *count* should equal to the dimensions of the variable.

Note that *start* and *count* is related to the number of elements in each dimension, not the number of bytes needed for storage. When allocating the data array, multiply the total number of elements with the size of one element. If you need to be generic in this calculation, you can use the `adios_type_size()` function to get the size of one element of a given type (cf. ADIOS_VARINFO.type).

6.2.9 adios_get_attr / adios_get_attr_byid

```
int adios_get_attr (ADIOS_GROUP           * gp,
                    const char            * attrname,
                    enum ADIOS_DATATYPES  * type,
                    int                   * size,
                    void                 ** data)

int adios_get_attr_byid (ADIOS_GROUP           * gp,
                         int                     attrid,
                         enum ADIOS_DATATYPES  * type,
                         int                   * size,
                         void                 ** data)
```

This function retrieves an attribute including its type, memory size and its value. An attribute can only be a scalar value or a string. Memory is allocated in the function to store the value. The allocated size is returned in the size argument.

This function does not read the file usually. The attribute's value is stored in the footer and is already in the memory after the file is opened. However, an attribute can refer to a scalar (or string) variable too. In this case, this function calls adios_read_var internally, so the file will be accessed to read in that scalar.

6.2.10 adios_type_to_string

```
const char * adios_type_to_string (enum ADIOS_DATATYPES type)
```

This function returns the name of a given type.

6.2.11 adios_type_size

```
int adios_type_size(enum ADIOS_DATATYPES type, void *data)
```

This function returns the memory size of one data element of an adios type. If the type is adios_string, and the second argument is the string itself, it returns strlen(data)+1. For other types, *data* is not used and the function returns the size occupied by one element.

6.3 Time series analysis API Description:

ADIOS provides APIs to perform time-series analysis like correlation and covariance on statistics collected in the BP file. As described in Section 6.2.6, adios_inq_var populates characteristics, such as minimum, maximum, average, standard deviation values for an array for each timestep. The following analysis function can be used with ADIOS_VARINFO objects previously defined. This can be performed only for a variable that has a time index.

6.3.1 adios_stat_cor / adios_stat_cov

This function calculates Pearson correlation/covariance of the characteristic data of *vix* and characteristic data of *viy*.

```
double adios_stat_cor (ADIOS_VARINFO * vix,
                       ADIOS_VARINFO * viy,
                       char          * characteristic,
                       uint32_t        time_start,
                       uint32_t        time_end,
                       uint32_t        lag)

double adios_stat_cov (ADIOS_VARINFO * vix,
                       ADIOS_VARINFO * viy,
                       char          * characteristic,
                       uint32_t        time_start,
                       uint32_t        time_end,
                       uint32_t        lag)
```

Required:

- vix - an ADIOS_VARINFO object

Optional:

- viy - either an ADIOS_VARINFO object or NULL
- characteristics - can be any of the following pre-computed statistics: "minimum" or "maximum" or "average" or "standard deviation" (alternatively, "min" or "max" or "avg" or "std_dev" can be given)
- time_start - specifies the start time from which correlation/covariance should be performed
- time_end - specifies the end time up to which correlation/covariance should be performed

  time_start and time_end should be within the time bounds of vix and viy with time_start < time_end

  If time_start and time_end = 0, the entire range of timesteps is considered. In this case, vix and viy should have the same number of timesteps.
- lag - if viy is NULL, and if lag is given, correlation is performed between the data specified by vix, and vix shifted by 'lag' timesteps. If viy is not NULL, lag is ignored.

6.4 Read Fortran API description

The Fortran API does not deal with the structures of the C api rather it requires several arguments in the function calls. They are all implemented as subroutines

like the write Fortran API and the last argument is an integer variable to store the error code output of each function (0 meaning successful operation).

An example code can be found in the source distribution as `tests/bp_read/bp_read_f.F90`.

The most important thing to note is that some functions need integer*8 (scalar or array) arguments. Passing an integer*4 array from your code leads to fatal errors. Please, double check the arguments of the function calls.

Due to the lack of structures and because the Fortran API does not allocate memory for them, you have to inquiry the file after opening it and to inquiry the group after opening it. You also have to inquiry an attribute to determine the memory size and allocate space for it before retrieving it.

Where the API function returns a list of names (inquiry file or inquiry group), you have to provide enough space for them using the counts returned by the preceding open call.

Here is the list of the Fortran subroutines. The GENERIC word indicates that you can use that function with any data type at the indicated argument. Since Fortran90 does not allow defining functions that can take any type of argument, we do not provide an F90 module for this API. The functions are actually defined in C and due to the lack of compiler checking, you can pass any type of array or variable where a GENERIC array is denoted.

```
subroutine adios_errmsg (msg)
     character(*),   intent(out) :: msg
end subroutine

subroutine adios_fopen (fp, fname, comm, groups_count, err)
     integer*8,      intent(out) :: fp
     character(*),   intent(in)  :: fname
     integer,        intent(in)  :: comm
     integer,        intent(out) :: groups_count
     integer,        intent(out) :: err
end subroutine

subroutine adios_fclose (fp, err)
     integer*8,      intent(in)  :: fp
     integer,        intent(out) :: err
end subroutine

subroutine adios_inq_file (fp, vars_count,
                           attrs_count, tstart, ntsteps,
                           gnamelist, err)
     integer*8,      intent(in)  :: fp
     integer,        intent(out) :: vars_count
     integer,        intent(out) :: attrs_count
     integer,        intent(out) :: tstart
```

```
     integer,         intent(out) :: ntsteps
     character(*), dimension(*), intent(inout) :: gnamelist
     integer,         intent(out) :: err
end subroutine

subroutine adios_gopen (fp, gp, grpname, vars_count,
                        attrs_count, err)
     integer*8,       intent(in)  :: fp
     integer*8,       intent(out) :: gp
     character(*),    intent(in)  :: grpname
     integer,         intent(out) :: vars_count
     integer,         intent(out) :: attrs_count
     integer,         intent(out) :: err
end subroutine

subroutine adios_gclose (gp, err)
     integer*8,       intent(in)  :: gp
     integer,         intent(out) :: err
end subroutine

subroutine adios_inq_group (gp, vnamelist, anamelist, err)
     integer*8,       intent(in)  :: gp
     character(*), dimension(*), intent(inout) :: vnamelist
     character(*), dimension(*), intent(inout) :: anamelist
     integer,         intent(out) :: err
end subroutine

subroutine adios_inq_var (gp, varname, vartype, ndim,
                          dims, timedim, err)
     integer*8,       intent(in)  :: gp
     character(*),    intent(in)  :: varname
     integer,         intent(out) :: vartype
     integer,         intent(out) :: ndim
     integer*8, dimension(*), intent(out) :: dims
     integer,         intent(out) :: timedim
     integer,         intent(out) :: err
end subroutine

subroutine adios_read_var (gp, varname, start, count,
                           data, read_bytes)
     integer*8,       intent(in)  :: gp
     character(*),    intent(in)  :: varname
     integer*8, dimension(*), intent(in) :: start
     integer*8, dimension(*), intent(in) :: count
     GENERIC, dimension(*), intent(inout) :: data
     integer*8,       intent(out) :: read_bytes
     ! read_bytes < 0 indicates error
end subroutine

subroutine adios_get_varminmax (gp, varname, value, gmin,
                                gmax, mins, maxs, err)
     integer*8,       intent(in)  :: gp
```

```
      character(*),   intent(in)  :: varname
      GENERIC,        intent(out) :: value
      GENERIC,        intent(out) :: gmin
      GENERIC,        intent(out) :: gmax
      GENERIC, dimension(*), intent(inout) :: mins
      GENERIC, dimension(*), intent(inout) :: maxs
      integer,        intent(out) :: err
end subroutine

subroutine adios_inq_attr (gp, attrname, attrtype,
                           attrsize, err)
      integer*8,      intent(in) :: gp
      character(*),   intent(in)  :: attrname
      integer,        intent(out) :: attrtype
      integer,        intent(out) :: attrsize
      integer,        intent(out) :: err
end subroutine

subroutine adios_get_attr_int1 (gp, attrname, attr, err)
      integer*8,      intent(in)  :: gp
      character(*),   intent(in)  :: attrname
      GENERIC, dimension(*), intent(inout) :: attr
      integer,        intent(out) :: err
end subroutine
```

6.5 Compiling and linking applications

In a C code, include the adios_read.h header file.

In a Fortran 90 code, you do not need to include anything. It is strongly recommended to double check the integer parameters because the read API expects integer*8 arguments at several places and providing an integer will break your code and then debugging it proves to be very difficult.

If you want to use the MPI version of the library, then link your (C or Fortran) application with -ladiosread.

If you want to use the non-MPI version of the library, you need to compile your code with the –D_NOMPI option and link your application with -ladiosread_nompi.

7 BP file format

7.1 Introduction

This chapter describes the file structure of BP, which is the ADIOS native binary file format, to aid in understanding ADIOS performance issues and how files convert from BP files to other scientific file formats, such as netCDF and HDF5.

To avoid the file size limitation of 2 gigabytes by using a signed 32-bit offset within its internal structure, BP format uses an unsigned 64-bit datatype as the file offset. Therefore, it is possible to write BP files that exceed 2 gigabytes on platforms that have large file support.

By adapting ADIOS read routines based on the endianness indication in the file footer, BP files can be easily portable across different machines (e.g., between the Cray-XT4 and BlueGene).

To aid in data selection, we have a low-overhead concept of data characteristics to provide an efficient, inexpensive set of attributes that can be used to identify data sets without analyzing large data content.

As shown in Figure 15, the BP format comprises a series of process groups and the file footer. The remainder of this chapter describes each component in detail and helps the user to better understand (1) why BP is a self -describing and metadata-rich file format and (2) why it can achieve high I/O performance on different machine infrastructures.

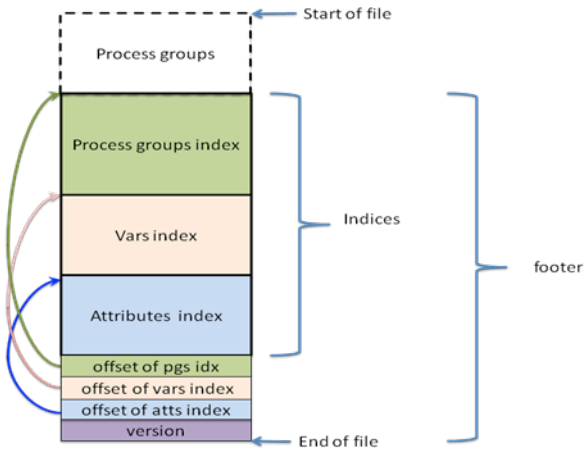

Figure 15. BP file structure

7.2 Footer

One known limitation of the NetCDF format is that the file contents are stored in a header that is exactly big enough for the information provided at file creation. Any changes to the length of that data will require moving data. To avoid this cost, we choose to employ a foot index instead. We place our version identifier and the offset to the beginning of the index as the last few bytes of our file, making it simple to find the index information and to add new and different data to our files without affecting any data already written.

7.2.1 Version

We reserve 4 bytes for the file version, in which the highest bit indicates endianness. Because ADIOS uses a fixed-size type for data, there is no need to store type size information in the footer.

7.2.2 Offsets of indices

In BP format, we store three 8-byte file offsets right before the version word, which allows users or developers to quickly seek any of the index tables for process groups, variables, or attributes.

7.2.3 Indices

7.2.3.1 Characteristics

Before we dive into the structures of the three index tables mentioned earlier, let's first take a look what characteristic means in terms of BP file format. To be able to make a summary inspection of the data to determine whether it contains the feature of greatest interest, we developed the idea of data characteristics. The idea of data characteristics is to collect some simple statistical and/or analytical data during the output operation or later for use in identifying the desired data sets. Simple statistics like array minimum and maximum values can be collected without extra overhead as part of the I/O operation. Other more complex analytical measures like standard deviations or specialized measures particular to the science being performance by require more processing. As part of our BP format, we store these values not only as part of data payload, but also in our index.

7.2.3.2 PG Index table

As shown in Figure 16, the process group (PG) index table encompasses the count and the total length of all the PGs as the first two entries. The rest of the tables contain a set of information for each PG, which contains the group name information, process ID, and time index. The Process ID specifies which process a group is written by. That process will be the rank value in the communicator if the MPI method is used. Most importantly, there is a file-offset entry for each PG, allowing a fast skip of the file in the unit of the process group.

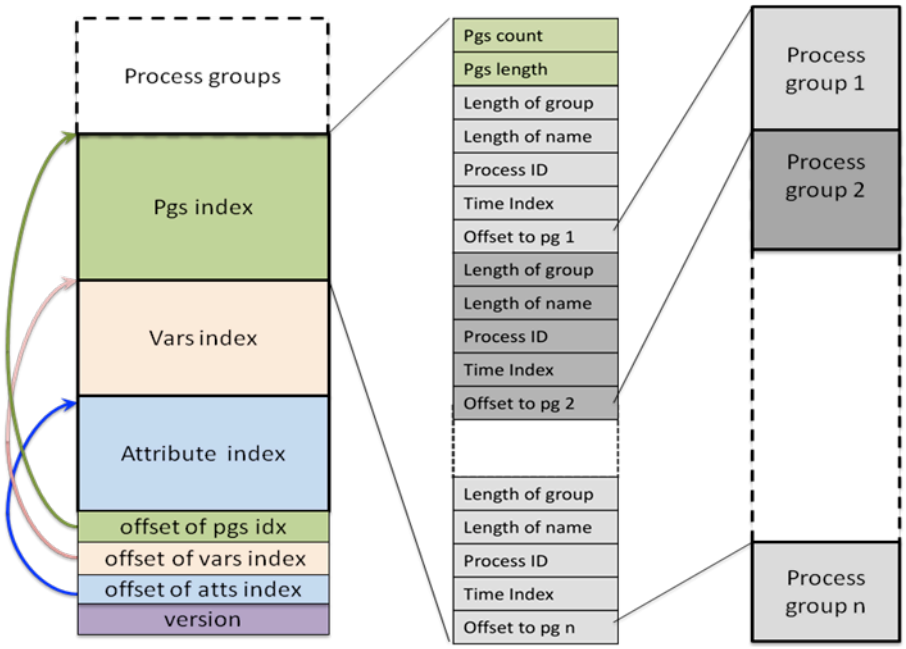

Figure 16. Group index table

7.2.3.3 Variables index table

The variables index table is composed of the total count of variables in the BP file, the size of variables index table, and a list of variable records. Each record contains the size of the record and the basic metadata to describe the variable. As shown in Figure 17, the metadata include the name of the variable, the name of the group the variable is associated with, the data type of the variable, and a series of characteristic features. The structure of each characteristic entry contains an offset value, which is addressed to the certain occurrence of the variable in the BP file. For instance, if n processes write out the variable "d" per time step, and m iterations have been completed during the whole simulation, then the variable will be written (m × n) times in the BP file that is produced. Accordingly, there will be the same number of elements in the list of characteristics. In this way, we can quickly retrieve the single dataset for all time steps or any other selection of time steps. This flexibility and efficiency also apply to a scenario in which a portion of records needs to be collected from a certain group of processes.

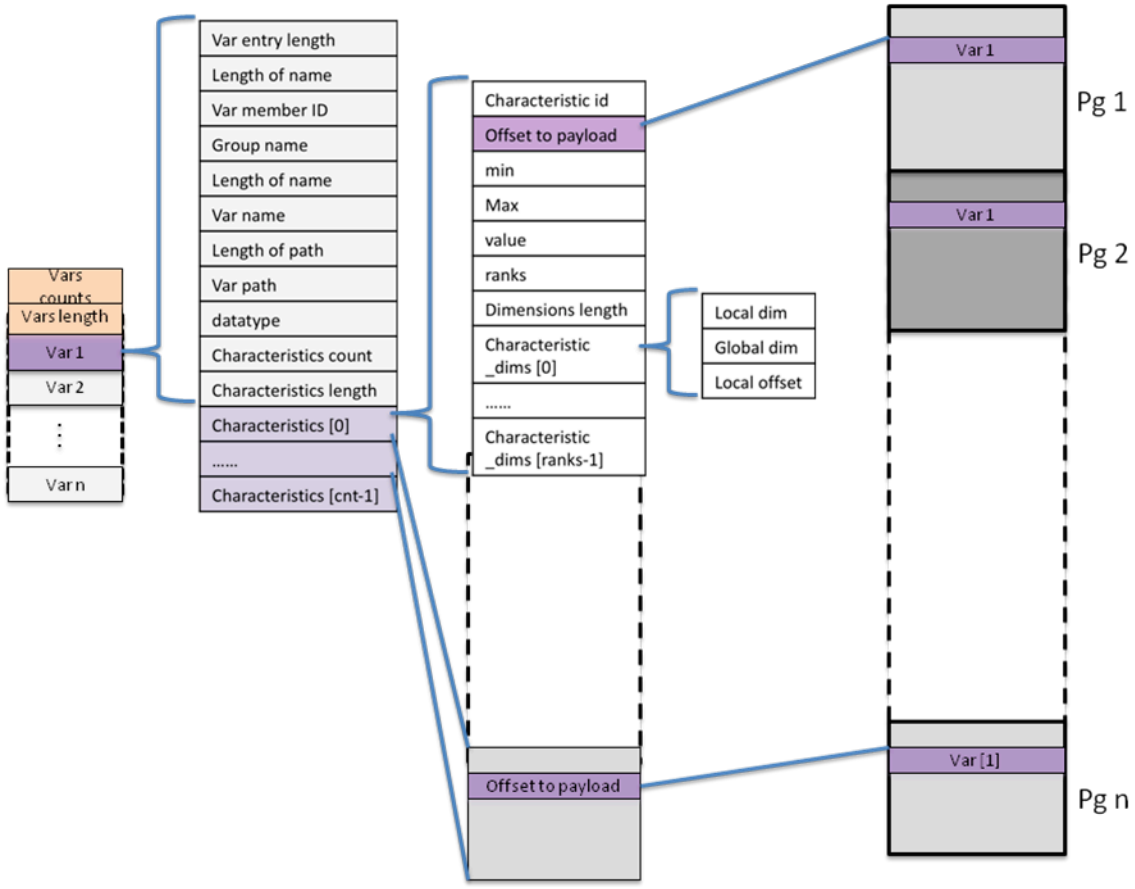

Figure 17. Variables index table

7.2.3.4 Attributes index table

Since an attribute can be considered to be a special type of variable, its index table in BP format is organized in the same way as a variables index table and therefore supports the same types of features mentioned in the previous sections.

7.3 Process Groups

One of the major concepts in BP format is what is called “process group” or PG. The BP file format encompasses a series of PG entries and the BP file footer. Each process group is the entire self-contained output from a single process and is written out independently into a contiguous disk space. In that way, we can enhance parallelism and reduce coordination among processes in the same communication group. The data diagram in Figure 18 illustrates the detailed content in each PG.

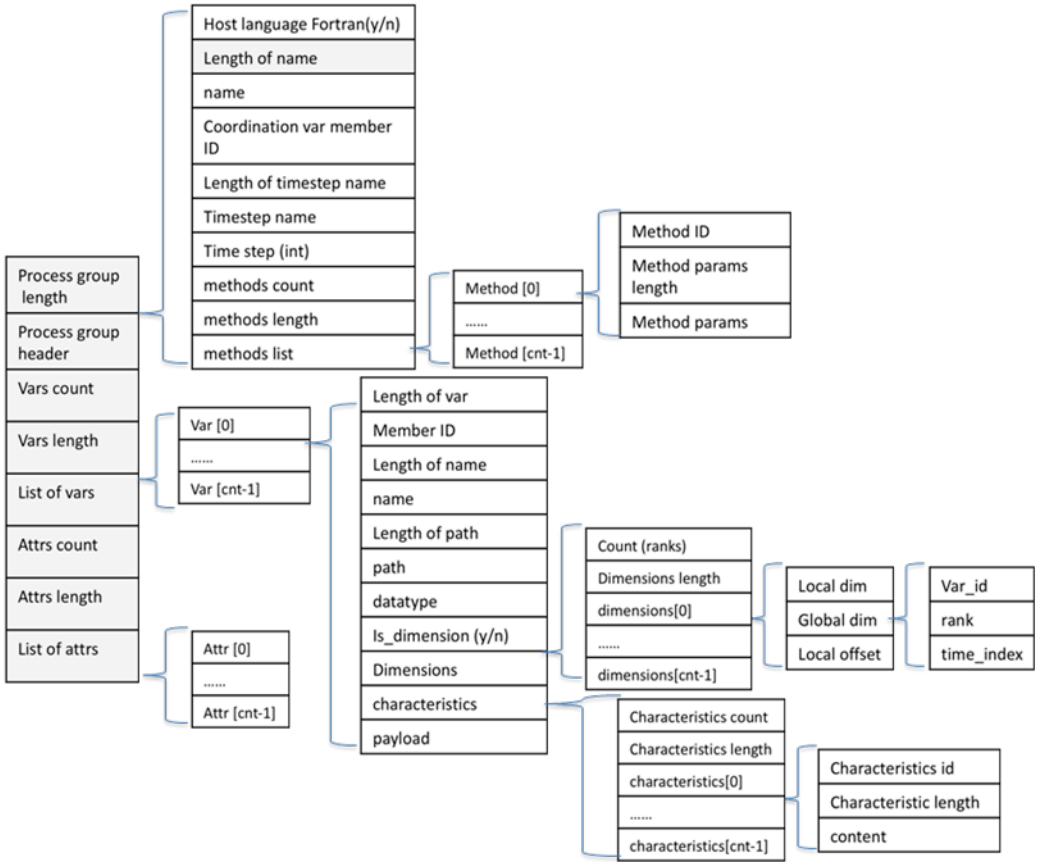

Figure 18. Process group structure

7.3.1 PG header

7.3.1.1 Unlimited dimension

BP format allows users to define an unlimited dimension, which will be specified as the time-index in the XML file. Users can define variables having a dimension with undefined length, for which the variable can grow along that dimension. PG is a self-contained, independent data structure; the dataset in the local space per each time step is not reconstructed at the writing operations across the processes or at time steps. Theoretically, PGs can be appended to infinity; they can be added one after another no matter how many processes or time steps take place during the simulation. Thus ADIOS is able to achieve high I/O performance.

7.3.1.2 Transport methods

One of the advantages of organizing output in terms of groups is to categorize all the variables based on their I/O patterns and logical relationships. It provides flexibility for each group to choose the optimized transport method according to the simulation environment and underlying hardware configuration or the transport methods used for a performance study without even changing the source code. In PG header structure, each entry in the method list has a method

ID and method parameters, such as system-tuning parameters or underneath driver selection.

7.3.2 Vars list

7.3.2.1 Var header

7.3.2.1.1 Dimensions structure

Internal to bp is sufficient information to recreate any global structure and to place the local data into the structure. In the case of a global array, each process writes the size of the global array dimensions, specifies the local offsets into each, and then writes the local data, noting the size in each dimension. On conversion to another format, such as HDF5, this information is used to create hyperslabs for writing the data into the single, contiguous space. Otherwise, it is just read back in and used to note where the data came from. In this way, we can enhance parallelism and reduce coordination. All of our parallel writes occur independently unless the underlying transport specifically requires collective operations. Even in those cases, the collective calls are only for a full buffer write (assuming the transport was written appropriately) unless there is insufficient buffer space.

As shown in Figure 18, the dimension structure contains a time index flag, which indicates whether this variable has an unlimited time dimension. Var_id is used to retrieve the dimension value if the dimension is defined as variable in the XML file; otherwise, the rank value is taken as the array dimension.

7.3.2.2 Payload

Basic statistical characteristics give users the advantage for quick data inspection and analysis. In Figure 18, redundant information about characteristics is stored along with variable payload so that if the characteristics part in the file footer gets corrupted, it can still be recovered quickly. Currently, only simple statistical traits are saved in the file, but the characteristics structure will be easily expanded or modified according to the requirements of scientific applications or the analysis tools.

7.3.3 Attributes list

The layout of the attributes list (see Figure 19) is very similar to that of the variables. However, instead of containing dimensional structures and physical data load, the attribute header has an is_var flag, which indicates either that the value of the attribute is referenced from a variable by looking up the var_id in the same group or that it is a static value defined in the XML file.

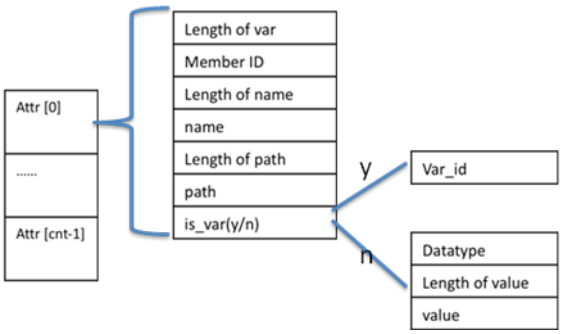

Figure 19. Attribute entry structure

8 Utilities

8.1 adios_lint

We provide a verification tool, called adios_lint, which comes with ADIOS 1.2. It can help users to eliminate unnecessary semantic errors and to verify the integrity of the XML file. Use of adios_lint is very straightforward; enter the adios_lint command followed by the config file name.

8.2 bpls

The bpls utility is used to list the content of a BP file or to dump arbitrary subarrays of a variable. By default, it lists the variables in the file including the type, name, and dimensionality. Here is the description of additional options (use bpls -h to print help on all options for this utility).

-l Displays the global statistics associated with each array (minimum, maximum, average and standard deviation) and the value of each scalar. Note that the detailed listing does not have extra overhead of processing since this information is available in the footer of the BP file.

-t When added to the -l option, displays the statistics associated with the variables for every timestep.

-p Dumps the histogram binning intervals and their corresponding frequencies, if histograms were enabled while writing the bp file. This option generates a "<variable-name>.gpl" file that can be given to the 'gnuplot' program as input.

-a Lists attributes besides the variables

-A Lists only the attributes

-r Sorts the full listing by names. Name masks to list only a subset of the variables/attributes can be given like with the -ls command or as regular expressions (with –e option).

-v Verbose. It prints some information about the file in the beginning before listing the variables.

-S Dump byte arrays as strings instead of with the default numerical listing. 2D byte arrays are printed as a series of strings.

Since bpls is written in C, the order of dimensions is reported with row-major ordering, i.e., if Fortran application wrote an NxM 2D variable, bpls reports it as an MxN variable.

-d Dumps the values of the variables. A subset of a variable can be dumped by using start and count values for each dimension with –s and –c option, e.g.,

–s “10,20,30” –c “10,10,10” reads in a 10x10x10 sub-array of a variable starting from the (10,20,30) element. Indices start from 0. As in Python, -1 denotes the last element of an array and negative values are handled as counts from backward. Thus, -s “-1,-1” –c “1,1” reads in the very last element of a 2D array, or –s “0,0” –c “1,-1” reads in one row of a 2D array. Or –s “1,1” –c “-2,-2” reads in the variable without the edge elements (row 0, colum 0, last row and last column).

Time is handled as an additional dimension, i.e., if a 2D variable is written several times into the same BP file, bpls lists it as a 3D array with the time dimension being the first (slowest changing) dimension.

In the example below, a 4 process application wrote a 4x4 array (each process wrote a 2x2 subset) with values from 0 to 15 once under the name /var/int_xy and 3 times under the name /var/int_xyt.

```
$ bpls -latv g_2x2_2x2_t3.bp
File info:
  of groups:     1
  of variables:  11
  of attributes: 7
  time steps:    3 starting from 1
  file size:     779 KB
  bp version:    1
  endianness:    Little Endian

Group genarray:
  integer    /dimensions/X             scalar = 4
  integer    /dimensions/Y             scalar = 4
  integer    /info/nproc               scalar = 4
  string     /info/nproc/description   attr   = "Number of writers"
  integer    /info/npx                 scalar = 2
  string     /info/npx/description     attr   = "Number of processors
in x dimension"
  integer    /info/npy                 scalar = 2
  string     /info/npy/description     attr   = "Number of processors
in y dimension"
  integer    /var/int_xy               {4, 4} = 0 / 15
  string     /var/int_xy/description   attr   = "2D array with 2D
decomposition"
  integer    /var/int_xyt              {3, 4, 4} = 0 / 15
  string     /var/int_xyt/description  attr   = "3D array with 2D
decomposition with time in 3rd dimension"
```

Figure 20. bpls utility

The content of /var/int_xy can be dumped with

```
$ bpls  g_2x2_2x2_t3.bp -d -n 4 var/int_xy
  integer    /var/int_xy    {4, 4}
```

```
    (0,0)    0 1 2 3
    (1,0)    4 5 6 7
    (2,0)    8 9 10 11
    (3,0)    12 13 14 15
```

The “central” 2x2 subset of /var/int_xy can be dumped with

```
$ bpls g_2x2_2x2_t3.bp -d -s "1,1" -c "2,2" -n 2 var/int_xy
  integer    /var/int_xy    {4, 4}
    slice (1:2, 1:2)
    (1,1)    5 6
    (2,1)    9 10
```

The last element of /var/int_xyt for each timestep can be dumped with

```
$ bpls g_2x2_2x2_t3.bp -d -s "0,-1,-1" -c "-1,1,1" -n 1 var/int_xyt
  integer    /var/int_xyt   {3, 4, 4}
    slice (0:2, 3:3, 3:3)
    (0,3,3)    15
    (1,3,3)    15
    (2,3,3)    15
```

8.3 bpdump

The bpdump utility enables users to examine the contents of a bp file more closely to the actual BP format than with bpls and to display all the contents or selected variables in the format on the standard output. Each writing process’ output is printed separately.

It dumps the bp file content, including the indexes for all the process groups, variables, and attributes, followed by the variables and attributes list of individual process groups (see Figure 21).

---

bpdump [-d var|--dump var] <filename>

```
==========================================================
Process Groups Index:
Group: temperature
     Process ID: 0
     Time Name:
     Time: 1
     Offset in File: 0
==========================================================
Vars Index:
Var (Group) [ID]: /NX (temperature) [1]
     Datatype: integer
     Vars Characteristics: 20
Offset(46)          Value(10)
Var (Group) [ID]: /size (temperature) [2]
     Datatype: integer
```

```
        Vars Characteristics: 20
        Offset(77)          Value(20)
...
Var (Group) [ID]: /rank (temperature) [3]
        Datatype: integer
        Vars Characteristics: 20
            Offset(110)         Value(0)
...
Var (Group) [ID]: /temperature (temperature) [4]
        Datatype: double
        Vars Characteristics: 20
            Offset(143)         Min(1.000000e-01)           Max(9.100000e+00)
Dims (l:g:o): (1:20:0,10:10:0)
...
=========================================================
Attributes Index:
Attribute (Group) [ID]: /recorded-date (temperature) [5]
        Datatype: string
        Attribute Characteristics: 20
        Offset(363)         Value(Sep-19-2008)
...
```

Figure 21. bpdump utility

9 Converters

To make BP files compatible with the popular file formats, we provide a series of converters to convert BP files to HDF5, NETCDF, or ASCII. As long as users give the required schema via the configuration file, the different converter tools currently in ADIOS have the features to translate intermediate BP files to the expected HDF5, NetCDF, or ASCII formats.

9.1 bp2h5

This converter, as indicated by its name, can convert BP files into HDF5 files. Therefore, the same postprocessing tools can be used to analyze or visualize the converted HDF5 files, which have the same data schema as the original ones. The converter can match the row-based or column-based memory layout for datasets inside the file based on which language the source codes are written in. If the XML file specifies global-bounds information, the individual sub-blocks of the dataset from different process groups will be merged into one global the dataset in HDF file.

9.2 bp2ncd

The bp2ncd converter is used to translate bp files into NetCDF files. In Chap. 5, we describe the time-index as an attribute for adios-group. If the variable is time-based, one of its dimensions needs to be specified by this time-index variable, which is defined as an unlimited dimension in the file into which it is to be converted. a NetCDF dimension has a name and a length. If the constant value is declared as a dimension value, the dimension in NetCDF will be named varname_n, in which varname is the name of the variable and n is the nth dimension for that variable. To make the name for the dimension value more meaningful, the users can also declare the dimension value as an attribute whose name can be picked up by the converter and used as the dimension name.

Based on the given global bounds information in a BP file, the converter can also reconstruct the individual pieces from each process group and create the global space array in NetCDF. A final word about editing the XML file: the name string can contain only letters, numbers or underscores ("_"). Therefore, the attribute or variable name should conform to this rule.

9.3 bp2ascii

Sometimes, scientists want to extract one variable with all the time steps or want to extract several variables at the same time steps and store the resulting data in ASCII format. The Bp2ascii converter tool allows users to accomplish those tasks.

Bp2ascii bp_filename –v x1 … xn [–c/-r] –t m,n

-v – specify the variables need to be printed out in ASCII file

-c –print variable values for all the time steps in column

-r – print variable values for all the time steps in row

-t – print variable values for time step m to n, if not defined, all the time steps will be printed out.

9.4 Parallel Converter Tools

Currently, all of the converters mentioned above can only sequentially parse bp files. We will work on developing parallel versions of all of the converters for improved performance. As a result, the extra conversion cost to translate bp into the expected file format can be unnoticeable compared with the file transfer time.

10 Group read/write process

In ADIOS 1.2, we provide a python script, which takes a configuration file name as an input argument and produces a series of preprocessing files corresponding to the individual adios-group in the XML file. Depending on which language (C or FORTRAN) is specified in XML, the python script either generates files gwrite_groupname.ch and gread_groupname.ch for C or files with extension .fh for Fortran. These files contain the size calculation for the group and automatically print adios_write calls for all the variables defined inside adios-group. One need to use only the "`#include filename.ch`" statement in the source code between the pair of adios_open and adios_close.

Users either type the following command line or incorporate it into a Makefile:

```
python gpp.py <config_fname>
```

10.1 Gwrite/gread/read

Below are a few example of the mapping from var element to adios_write/read:

In adios-group "weather", we have a variable declared in the following forms:

1) <var name="temperature" gwrite="t" gread="t_read" type="adios_double" dimensions="NX"/>

When the python command is executed, two files are produced, gwrite_weather.ch and gread_weather.ch. The gwrite_weather.ch command contains

```
adios_write (adios_handle, "temperature", t);
```

while gread_weather.ch contains

```
adios_read (adios_handle, "temperature", t_read).
```

2) <var name="temperature" gwrite="t" gread="t_read" type="adios_double" dimensions="NX" *read="no"*/>

In this case, only the adios_write statement is generated in gwrite_weather.ch. The adios_read statement is not generated because the value of attribute read is set to "no".

3) <var name="temperature" gread="t_read" type="adios_double" dimensions="NX" />

```
adios_write (adios_handle, "temperature", temperature)

adios_read (adios_handle, "temperature", t_read).
```

4) <var name="temperature" gwrite="t" type="adios_double" dimensions="NX" />

```
adios_write (adios_handle, "temperature", t)

adios_read (adios_handle, "temperature", temperature)
```

10.2 Add conditional expression

Sometimes, the adios_write routines are not perfectly written out one after another. There might be some conditional expressions or loop statements. The following example will show you how to address this type of issue via XML editing.

<gwrite src="if (rank == 0) {"/>

<var name="temperature" gwrite="t" gread="t_read" type="adios_double" dimensions="NX" read="no"/>

<gwrite src="}"/>

Rerun the python command; the following statements will be generated in gwrite_weather.ch,

```
if (mype==0) {

adios_write (adios_handle, "temperature", t)

}
```

gread_weather.ch has same condition expression added.

10.3 Dependency in Makefile

Since we include the header files in the source, the users need to include the header files as a part of dependency rules in the Makefile.

11 C Programming with ADIOS

This chapter focuses on how to integrate ADIOS into the users' source code in C and how to write into separate files or a shared file from multiple processes in the same communication domain. These examples can be found in the source distribution under the `examples/C/manual` directory.

In the following steps we will create programs that use ADIOS to write

- a metadata-rich BP file per process
- one large BP file with the arrays from all processes
- N files from P processes, where N << P
- the data of all processes as one global array into one file
- a global-array over several timesteps into one file

The strength of the componentization of I/O in ADIOS allows us to switch between the first two modes by selecting a different transport method in a configuration file and run the program without recompiling it.

11.1 Non-ADIOS Program

The starting programming example, shown in Figure 22, writes a double-precision array `t` with size of `NX` into a separate file per process (the array is uninitialized in the examples).

```
#include <stdio.h>
#include "mpi.h"
#include "adios.h"
int main (int argc, char ** argv)
{
    char        filename [256];
    int         rank;
    int         NX = 10;
    double      t[NX];
    FILE        * fp;

    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    sprintf (filename, "restart_%5.5d.dat", rank);
    fp = open (filename, "w");
    fwrite ( &NX, sizeof(int), 1, fp);
    fwrite (t,  sizeof(double), NX, fp);
    fclose (fp);

    MPI_Finalize ();
    return 0;
```

```
}
```

Figure 22. Original program (examples/C/manual/1_nonadios_example.c).

```
$ mpirun -np 4 1_nonadios_example
$ ls restart_*
restart_00000.dat  restart_00001.dat  restart_00002.dat
restart_00003.dat
```

11.2 Construct an XML File

In the example above, the program is designed to write a file for each process. There is a double-precision one-dimensional array called "t". We also need to declare and write all variables that are used for dimensions (i.e. NX in our example). Therefore, our configuration file is constructed as shown in Figure 23.

```
/* config.xml*/
<?xml version="1.0"?>
<adios-config host-language="C">
  <adios-group name="temperature" coordination-communicator="comm">
    <var name="NX" type="integer"/>
    <var name="temperature" gwrite="t" type="double" dimensions="NX"/>
    <attribute name="description" path="/temperature"  type="string"
          value="Temperature array" />
  </adios-group>

<method group="temperature" method="POSIX"/>

<buffer size-MB="1" allocate-time="now"/>

</adios-config>
```

Figure 23. Example config.xml file

11.3 Generate .ch file (s)

The adios_group_size function and a set of adios_write functions can be automatically generated in gwrite_temperature.ch file by using the following python command:

```
gpp.py config.xml
```

The generated gwrite_temperature.ch file is shown in Figure 24.

```
/* gwrite_temperature.ch */
adios_groupsize = 4 \
          + 8 * (NX);
```

```
adios_group_size (adios_handle, adios_groupsize, &adios_totalsize);
adios_write (adios_handle, "NX", &NX);
adios_write (adios_handle, "temperature", t);
```

Figure 24. Example gwrite_temperature.ch file

11.4 POSIX transport method (P writers, P subfiles + 1 metadata file)

For our first program, we simply translate the program of Figure 22, so that all of the I/O operations are done with ADIOS routines. The POSIX method can be used to write out separate files for each processor in Figure 25. The changes to the original example are highlighted. We need to use an MPI communicator in adios_open() because the subprocesses need to know the rank to create unique subfile names.

```
/*write Separate file for each process by using POSIX*/
#include <stdio.h>
#include "mpi.h"
#include "adios.h"
int main (int argc, char ** argv)
{
    char        filename [256];
    int         rank;
    int         NX = 10;
    double      t[NX];

    /* ADIOS variables declarations for matching gwrite_temperature.ch */
    int             adios_err;
    uint64_t        adios_groupsize, adios_totalsize;
    int64_t         adios_handle;
    MPI_Comm  * comm =  MPI_COMM_WORLD;

    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    sprintf (filename, "restart.bp");
    adios_init ("config.xml");
    adios_open (&adios_handle, "temperature", filename, "w", &comm);
    #include "gwrite_temperature.ch"
    adios_close (adios_handle);
    adios_finalize (rank);
    MPI_Finalize ();
    return 0;
}
```

Figure 25. Example adios program to write P files from P processors (examples/C/manual/2_adios_write.c)

The POSIX method makes a directory to store all subfiles. As for the naming of the directory, it appends ".dir" to the name the file, e.g., restart.bp.dir. For each subfile, it appends the rank of the process (according to the supplied communicators) to the name of the file (here restart.bp), so for example process 2 will write a file restart.bp.dir/restart.bp.2. To facilitate reading of subfiles, the method also generates a global metadata file (restart.bp) which tracks all the variables in each subfile.

```
$ mpirun -np 4 2_adios_write

$ ls restart.bp
restart.bp

restart.bp.dir:
restart.bp.0  restart.bp.1  restart.bp.2  restart.bp.3

$ bpls -lad restart.bp.dir/restart.bp.2 -n 10
  integer    /NX                        scalar = 10
  double     /temperature               {10} = 20 / 29
    (0)    20 21 22 23 24 25 26 27 28 29

  string     /temperature/description  attr   = "Temperature array"
```

11.5 MPI-IO transport method (P writers, 1 file)

Based on the same group description in the configure file and the header file (.ch) generated by python script, we can switch among different transport methods without changing or recompiling the source code.

One entry change in the config.xml file can switch from POSIX to MPI:

<method group="temperature" method="**MPI**"/>

The MPI communicator is passed as an argument of adios_open(). Because it is defined as MPI_COMM_WORLD in the posix example already, the program does not need to be modified or recompiled.

```
$ mpirun -np 4 2_adios_write
```

```
$ ls restart.bp
restart.bp
$ bpls -l restart.bp
Group temperature:
  integer    /NX           scalar = 10
  double     /temperature  {10} = 0 / 39
```

There are several ways to verify the binary results. We can either choose bpdump to display the content of the file or use one of the converters (bp2ncd, bp2h5, or bp2ascii), to produce the user's preferred file format (NetCDF, HDF5 or ASCII, respectively) and use its dump utility to output the content in the standard output. Bpls cannot list the individual arrays written by the processes because the generic read API it uses does not support this (it can see only one of them as the size of /temperature suggest in the listing above). It is suggested to use global arrays (see example below) to present the data written by many processes as one global array, which then can be listed and any slice of it can be read/dumped.

This example, however, can be used for checkpoint/restart files where the application would only read in data from the same number of processes as it was written (see next example). The transparent switch between the POSIX and MPI methods allows the user choose the better performing method for a particular system without changing the source code.

11.6 Reading data from the same number of processors

Now let's move to examples of how to read the data from BP or other files. Assuming that we still use the same configure file shown in Figure 23, the following steps illustrate how to easily change the code and xml file to read a variable.

1. add another variable adios_buf_size specifying the size for read.
2. call adios_open with "r" (read only) mode.
3. Insert #include "gread_temperature.ch"

```
/*Read in data on same number of processors */
#include <stdio.h>
#include "mpi.h"
#include "adios.h"
int main (int argc, char ** argv)
{
    char        filename [256];
    int         rank;
    int         NX = 10;
    double      t[NX];

    /* ADIOS variables declarations for matching gread_temperature.ch */
```

```
    int             adios_err;
    uint64_t        adios_groupsize, adios_totalsize, adios_buf_size;
    int64_t         adios_handle;
    MPI_Comm  comm =  MPI_COMM_WORLD;

    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    sprintf (filename, "restart.bp");
    adios_init ("config.xml");
    adios_open (&adios_handle, "temperature", filename, "r", &comm);
    #include "gread_temperature.ch"
    adios_close (adios_handle);
    adios_finalize (rank);
    MPI_Finalize ();
    return 0;
}
```

Figure 26. Read in data generated by 2_adios_write using gread_temperature.ch (examples/C/manual/3_adios_read.c)

The gread_temperature.ch file generated by gpp.py is the following:

```
/* gread_temperature.ch */
adios_group_size (adios_handle, adios_groupsize, &adios_totalsize);
adios_buf_size = 4;
adios_read (adios_handle, "NX", &NX, adios_buf_size);
adios_buf_size = NX;
adios_read (adios_handle, "temperature", t, adios_buf_size);
```

Figure 27. Example of a generated gread_temperature.ch file

11.7 Writing to Shared Files (P writers, N files)

As the number of processes increases to tens or hundreds of thousands, the amount of files will increase by the same magnitude if we use the POSIX method or a single shared file may be too large if we use the MPI method. In this example we address a scenario in which multiple processes write to N files. In the following example (Figure 28), we write out N files from P processes. This is achieved by creating a separate communicator for N subsets of the processes using MPI_Comm_split().

```
#include <stdio.h>
#include "mpi.h"
#include "adios.h"
int main (int argc, char ** argv)
```

```
{
    char        filename [256];
    int         rank, size;
    int         NX = 10;
    int         N = 3;
    double      t[NX];

    /* ADIOS variables declarations for matching gwrite_temperature.ch */
    int             adios_err;
    uint64_t        adios_groupsize, adios_totalsize;
    int64_t         adios_handle;
    MPI_Comm comm;
    /*
    int         color, key;
    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);

    /* MPI_Comm_split partitions the world group into N disjointed  subgroups,
     * the processes are ranked in terms of the argument key
     *  a new communicator comm is returned for this specific grid configuration
     */
    color = rank % N;
    key = rank / N;
    MPI_Comm_split (MPI_COMM_WORLD, color, key, &comm);

    /* every P/N processes write into the same file
     * there are N files generated.
     */
    sprintf (filename, "restart_%5.5d.bp", color);
    adios_init ("config.xml");
    adios_open (&adios_handle, "temperature", filename, "w", &comm);
    #include "gwrite_temperature.ch"
    adios_close (adios_handle);
    adios_finalize (rank);
    MPI_Finalize ();
    return 0;
}
```

Figure 28. Example ADIOS program writing N files from P processors (N)

The reconstructed MPI communicator `comm` is passed as an argument of the adios_open() call. Therefore, in this example, each file is written by the processes in the same communication domain.

There is no need to change the XML file in this case because we are still using the MPI method.

11.8 Global Arrays

If each process writes out a sub-array that belongs to the same global space, ADIOS provides the way to write out global information so the generic read API can see a single global array (and also the HDF5 or NetCDF file when using our converters). This example demonstrates how to write global arrays, where the number of processes becomes a separate dimension. Each process is writing the one dimensional temperature array of size NX and the result is a two dimensional array of size PxNX. Figure 29 shows how to define a global array in the XML file.

```
<?xml version="1.0"?>
<adios-config host-language="C">
  <adios-group name="temperature" coordination-communicator="comm">
    <var name="NX" type="integer"/>
    <var name="size" type="integer"/>
    <var name="rank" type="integer"/>
    <global-bounds dimensions="size,NX" offsets="rank,0">
      <var name="temperature" gwrite="t" type="double" dimensions="1,NX"/>
    </global-bounds>
    <attribute name="description" path="/temperature"
           value="Global array written from 'size' processes" type="string"/>
  </adios-group>

<method group="temperature" method="MPI"/>
<buffer size-MB="2" allocate-time="now"/>

</adios-config>
```

Figure 29. Config.xml for a global array (examples/C/global-array/adios_global.xml)

The variable is inserted into a <global-bounds>…</global-bounds> section. The global array's global dimension is defined by the variables size and NX, available in all processes and all with the same value. The offset of a local array written by a process is defined using the rank variable, which is different on every process. The variable itself is defined as an 1xNX two dimensional array, although in the C code it is still a one dimensional array.

The gwrite header file generated by gpp.py is the following:

```
/* gwrite_temperature.ch */
adios_groupsize = 4 \
                  + 4 \
```

```
                + 4 \
                + 8 * (1) * (NX);
adios_group_size (adios_handle, adios_groupsize, &adios_totalsize);
adios_write (adios_handle, "NX", &NX);
adios_write (adios_handle, "size", &size);
adios_write (adios_handle, "rank", &rank);
adios_write (adios_handle, "temperature", t);
```

Figure 30. gwrite header file generated from config.xml

The program code is not very different from the one used in the above example. It needs to have the size and rank variables in the code defined (see examples/C/global-array/adios_global.c)

11.8.1 MPI-IO transport method (P writers, 1 file)

```
$ mpirun -np 4 ./adios_global
$ ls adios_global.bp
adios_global.bp

$  bpls -latd adios_global.bp -n 10

 integer    /NX                            scalar = 10
 integer    /rank                          scalar = 0
 integer    /size                          scalar = 4
 double     /temperature                   {4, 10} = 0 / 39 / 19.5 /
11.5434  {MIN / MAX / AVG / STD_DEV}
    (0,0)    0 1 2 3 4 5 6 7 8 9
    (1,0)    10 11 12 13 14 15 16 17 18 19
    (2,0)    20 21 22 23 24 25 26 27 28 29
    (3,0)    30 31 32 33 34 35 36 37 38 39

 string    /temperature/description  attr   = "Global array written from 'size'
processes"
```

The bp2ncd utility can be used to convert the bp file to an NetCDF file:

```
$ bp2ncd adios_global.bp
$ ncdump adios_global.nc
netcdf adios_global {
dimensions:
        NX = 10 ;
        size = 4 ;
        rank = 1 ;
variables:
        double temperature(size, NX) ;
                temperature:description = "Global array written
from \'size\' processes" ;
data:
```

```
 temperature =
  0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
  10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
  20, 21, 22, 23, 24, 25, 26, 27, 28, 29,
  30, 31, 32, 33, 34, 35, 36, 37, 38, 39 ;
}
```

11.8.2 POSIX transport method (P writers, P Subfiles + 1 Metadata file)

To list variables output from POSIX transport, user only needs to specify the global metadata file (e.g., adios_global.bp) as a parameter to bpls, not each individual files (e.g., adios_global.bp.dir/adios_global.bp.0). The output of the POSIX and the MPI methods are equivalent from reading point of view.

```
$ mpirun -np 4 ./adios_global
$ ls adios_global.bp
adios_global.bp

$  bpls -latd adios_global.bp -n 10

 integer    /NX                          scalar = 10
 integer    /rank                        scalar = 0
 integer    /size                        scalar = 4
 double     /temperature                 {4, 10} = 0 / 39 / 19.5 /
11.5434  {MIN / MAX / AVG / STD_DEV}
    (0,0)    0 1 2 3 4 5 6 7 8 9
    (1,0)    10 11 12 13 14 15 16 17 18 19
    (2,0)    20 21 22 23 24 25 26 27 28 29
    (3,0)    30 31 32 33 34 35 36 37 38 39

 string    /temperature/description  attr   = "Global array written from 'size'
processes"
```

The examples/C/global-array/adios_read_global.c program shows how to use the generic read API to read in the global array from arbitrary number of processes.

11.9 Writing Time-Index into a Variable

The time-index allows the user to define a variable with an unlimited dimension, along which the variable can grow in time. Let's suppose the user wants to write out temperature after a certain number of iterations. First, we add the "time-index" attribute to the adios-group with an arbitrary name, e.g. "iter". Next, we find the (global) variable temperature in the adios-group and add "iter" as an extra dimension for it; the record number for that variable will be stored every time it gets written out. Note that we do not need to change the dimensions and offsets in the global bounds, only the individual variable. Also note, that the time dimension must be the slowest changing dimension, i.e. in C, the first one and in Fortran, it must be the last one.

/* config.xml*/
<adios-config host-language="C">

```
  <adios-group name="temperature" coordination-communicator="comm" time-
index="iter">
    <var name="NX" type="integer"/>
    <var name="size" type="integer"/>
    <var name="key" type="integer"/>
    <global-bounds dimensions="size,NX" offsets="key,0">
      <var name="temperature" gwrite="t" type="double"
           dimensions="iter,1,NX"/>   (Note, for Fortran, "iter" needs to be
put in the end, i.e., dimension="NX,1,iter")
    </global-bounds>
    <attribute name="description" path="/temperature"
      value="Global array written from 'size' processes over several timesteps"
      type="string"/>
  </adios-group>
<method group="temperature" method="MPI"/>
<buffer size-MB="1" allocate-time="now"/>
</adios-config>
```

Figure 31. Config.xml for a global array with time (examples/C/global-array-time/adios_globaltime.xml)

The examples/C/global-array-time/adios_globaltime.c is similar to the previous example adios_global.c code. The only difference is that it has an iteration loop where each process writes out the data in each of its 13 iterations.

```
$ mpirun -np 4 ./adios_read_globaltime
$ bpls -la adios_globaltime.bp
Group temperature:
  integer    /NX                          scalar = 10
  integer    /size                        scalar = 4
  integer    /rank                        scalar = 0
  double     /temperature                 {13, 4, 10} = 100 / 1339
/ 719.5 / 374.344  {MIN / MAX / AVG / STD_DEV}
  string     /temperature/description  attr   = "Global array
written from 'size' processes over several timesteps"
```

A slice of two timesteps (6th and 7th), dumped with bpls:

```
$ bpls adios_globaltime.bp -s "5,0,0" -c "2,-1,-1" -n 10 -d
temperature
  double     /temperature  {13, 4, 10}
    slice (5:6, 0:3, 0:9)

    (5,0,0)    600 601 602 603 604 605 606 607 608 609
    (5,1,0)    610 611 612 613 614 615 616 617 618 619
    (5,2,0)    620 621 622 623 624 625 626 627 628 629
```

```
    (5,3,0)    630 631 632 633 634 635 636 637 638 639
    (6,0,0)    700 701 702 703 704 705 706 707 708 709
    (6,1,0)    710 711 712 713 714 715 716 717 718 719
    (6,2,0)    720 721 722 723 724 725 726 727 728 729
    (6,3,0)    730 731 732 733 734 735 736 737 738 739
```

11.10 Reading statistics

In Adios 1.2, statistics like minimum, maximum, average and standard deviation can be aggregated inexpensively. This section shows how these statistics can be accessed from the BP file. The examples/C/stat/stat_write.c is similar to the previous example adios_globaltime.c. It writes an additional variable "complex" of type adios_double_complex along with "temperature." It also has histogram enabled for the variable "temperature." Comparing it with the XML in the previous example, stat.xml has the following additions:

```
/* stat.xml*/

<?xml version="1.0"?>
<adios-config host-language="C">
    <adios-group name="temperature" coordination-communicator="comm"
                time-index="iter">
        <var name="NX" type="integer"/>
        <var name="rank" type="integer"/>
        <var name="size" type="integer"/>
        <global-bounds dimensions="size,NX" offsets="rank,0">
                <var name="temperature" gwrite="t" type="double"
                    dimensions="iter,1,NX"/>
                <var name="complex" gwrite="c" type="double complex"
                    dimensions="iter,1,NX"/>
        </global-bounds>
        </adios-group>

    <method group="temperature" method="MPI"/>
    <buffer size-MB="5" allocate-time="now"/>
    <analysis adios-group="temperature" var="temperature"
        break-points="0, 100, 1000, 10000" />
</adios-config>
```

Figure 32. Config.xml for creating histogram for an array variable (examples/C/stat/stat.xml)

To include histogram calculation, only the XML file needs to be updated, and no change is required in the C code. The examples/C/stat/gwrite_stat.ch requires an

additional 8 * (2) * NX to be added to adios_groupsize and an adios_write (adios_handle, "complex", &c) to handle the complex numbers.

```
$ mpirun -np 2 ./stat_write
[1]: adios_stat.bp written successfully
[0]: adios_stat.bp written successfully
```

The examples/C/stat/stat_read.c shows how to read back the statistics from the bp file. First, the statistics need to be populated into an ADIOS_VARINFO object. This is done with the following set of commands.

```
ADIOS_FILE * f = adios_fopen ("adios_stat.bp", comm);
ADIOS_GROUP * g = adios_gopen (f, "temperature");
ADIOS_VARINFO * v = adios_inq_var (g, "temperature");
```

The object 'v' now contains all the statistical information for the variable "temperature." To access the histogram for temperature, we need to access the ADIOS_HIST data structure inside the ADIOS_VARINFO object. The code below prints the break points and the interval frequencies for the global histogram. For 'n' break points there are 'n + 1' intervals.

```
/* Break points */
for (j = 0; j < v->hist->num_breaks; j++)
        printf ("%lf ", v->hist->breaks[j]);
/* Frequencies */
for (j = 0; j <= v->hist->num_breaks; j++)
        printf ("%d\t", v->hist->gfrequencies[j]);
adios_free_varinfo(v);
```

To access the statistics related to the variable "complex," we need:

```
v = adios_inq_var (g, "complex");
```

The code below describes how to print the minimum values of the magnitude, real and imaginary part of complex data at each timestep. For complex variables alone, all statistics need to be typecasted into a double format.

```
double ** Cmin = (double **) v->mins;
printf ("\nMagnitude  Real  Imaginary\n");
for (j = 0; v->ndim >= 0 &&  (j < v->dims[0]); j ++)
     printf ("%lf %lf %lf\n",
              Cmin[j][0], Cmin[j][1], Cmin[j][2]);
adios_free_varinfo(v);
```

12 Developer Manual

12.1 Create New Transport Methods

One of ADIOS's important features is the componentization of transport methods. Users can switch among the typical methods that we support or even create their own methods, which can be easily plugged into our library. The following sections provide the procedures for adding the new transport method called "abc" into the ADIOS library. In this version of ADIOS, all the source files are located in /trunk/src/.

12.1.1 Add the new method macros in adios_transport_hooks.h

The first file users need to examine is adios_transport_hooks.h, which basically defines all the transport methods and interface functions between detailed transport implementation and user APIs. In the file, we first find the line that defines the enumeration type Adios_IO_methods_datatype add the declaration of method ID ADIOS_METHOD_ABC, and, because we add a new method, update total number of transport methods ADIOS_METHOD_COUNT from 9 to 10.

1. enum Adios_IO_methods datatype

```
enum ADIOS_IO_METHOD {
        ADIOS_METHOD_UNKNOWN   = -2
       ,ADIOS_METHOD_NULL      = -1
       ,ADIOS_METHOD_MPI       = 0
       ……
       ,ADIOS_METHOD_PHDF5     = 8
                                  ← ADIOS_METHOD_ABC  = 9
       ,ADIOS_METHOD_COUNT     = 9  ← ADIOS_METHOD_COUNT  = 10
       };
```

2. Next, we need to declare the transport APIs for method "abc," including init/finalize, open/close, should_buffer, and read/write. Similar to the other methods, we need to add

```
FORWARD_DECLARE (abc)
```

3. Then, we add the mapping of the string name "abc" of the new transport method to the method ID - ADIOS_METHOD_ABC, which has been already defined in enumeration type Adios_IO_methods_datatype. As the last parameter, "1" here means the method requires communications, or "0" if not.

```
MATCH_STRING_TO_METHOD ("abc", ADIOS_METHOD_ABC, 1)
```

4. Lastly, we add the mapping of the string name needed in the initialization functions to the method ID, which will be used by adios_transport_struct variables defined in adios_internals.h.

ASSIGN_FNS (abc, ADIOS_METHOD_ABC)

12.1.2 Create adios_abc.c

In this section, we demonstrate how to implement different transport APIs for method "abc." In adios_abc.c, we need to implement at least 11 required routines:

1. "adios_abc_init" allocates the method_data field in adios_method_struct to the user-defined transport data structure, such as adios_abc_data_struct, and initializes this data structure. Before the function returns, the initialization status can be set by statement "adios_abc_initialized = 1."

2. "adios_abc_open" opens a file if there is only one processor writing to the file. Otherwise, this function does nothing; instead, we use adios_abc_should_buffer to coordinate the file open operations.

3. "adios_abc_should_buffer," called by the "common_adios_group_size" function in adios.c, needs to include coordination of open operations if multiple processes are writing to the same file.

4. "adios_abc_write", in the case of no buffering or overflow, writes data directly to disk. Otherwise, it verifies whether the internally recorded memory pointer is consistent with the vector variable's address passed in the function parameter and frees that block of memory if it is not needed any more.

5. "adios_abc_read" associates the internal data structure's address to the variable specified in the function parameter.

6. "adios_abc_close" simply closes the file if no buffering scheme is used. However, in general, this function performs most of the actual disk writing/reading the buffers to/from the file by one or more processors in the same communicator domain and then close the file.

7. "adios_abc_finalize" resets the initialization status back to 0 if it has been set to 1 by adios_abc_init.

If you are developing asynchronous methods, the following functions need to be implemented as well; otherwise you can leave them as empty implementation.

8. adios_abc_get_write_buffer,

9. "adios_abc_end_iteration" is a tick counter for the I/O routines to time how fast they are emptying the buffers.

10. "adios_abc_start_calculation" indicates that it is now an ideal time to do bulk data transfers because the code will not be performing I/O for a while.

11. “adios_abc_stop_calculation“ indicates that bulk data transfers should cease because the code is about to start communicating with other nodes.

The following is One of the most important things that needs to be noted:

fd->shared_buffer = adios_flag_no,

which means that the methods do not need a buffering scheme, such as PHDF5, and that data write out occurs immediately once adios_write returns.

If fd->shared_buffer = adios_flag_yes, the users can employ the self-defined buffering scheme to improve I/O performance.

12.1.3 A walk-through example

Now let’s look at an example of adding an unbuffered POSIX method to ADIOS. According to the steps described above, we first open the header file -- “adios_transport_hooks.h,” and add the following statements:

- **enum ADIOS_IO_METHOD** {

  ADIOS_METHOD_UNKNOWN = -2
  ,ADIOS_METHOD_NULL = -1
  ,ADIOS_METHOD_MPI = 0
  …
  ,ADIOS_METHOD_PROVENANCE = 8
  // method ID for binary transport method
  ,ADIOS_METHOD_POSIX_ASCII_NB = 9
  // total method number
  ,ADIOS_METHOD_COUNT = 10
  };

- **FORWARD_DECLARE (posix_ascii_nb);**

- **MATCH_STRING_TO_METHOD ("posix_ascii_nb"**

  , ADIOS_METHOD_ POSIX_ASCII_NB, 0)

- **ASSIGN_FNS (binary, ADIOS_METHOD_ POSIX_ASCII_NB)**

Next, we must create adios_posix_ascii_nb,c, which defines all the required routines listed in Sect. 12.1.2 The blue highlights below mark out the data structures and required functions that developers need to implement in the source code.

```
static int adios_posix_ascii_nb_initialized = 0;
struct adios_POSIX_ASCII_UNBUFFERED_data_struct
{
   FILE *f;
```

```
  uint64_t file_size;
};

void adios_posix_ascii_nb _init (const char *parameters
                                 , struct adios_method_struct * method)
{
  struct adios_POSIX_ASCII_UNBUFFERED_data_struct * md;
  if (!adios_posix_ascii_nb_initialized)
  {
    adios_posix_ascii_nb_initialized = 1;
  }
  method->method_data = malloc (
                          sizeof(struct adios_POSIX_ASCII_UNBUFFERED_data_struct)
                                 );
  md = (struct adios_POSIX_ASCII_UNBUFFERED_data_struct *)
                            method->method_data;
  md->f = 0;
  md->file_size = 0;
}

int adios_posix_ascii_nb _open (struct adios_file_struct * fd
                        , struct adios_method_struct * method)
{
  char * name;
  struct adios_POSIX_ASCII_UNBUFFERED_data_struct * p;
  struct stat s;
  p = (struct adios_POSIX_ASCII_UNBUFFERED_data_struct *)
                                  method->method_data;
  name = malloc (strlen (method->base_path) + strlen (fd->name) + 1);
  sprintf (name, "%s%s", method->base_path, fd->name);
  if (stat (name, &s) == 0)
    p->file_size = s.st_size;
  switch (fd->mode)
  {
    case adios_mode_read:
    {
      p->f = fopen (name, "r");
      if (p->f <= 0)
      {
        fprintf (stderr, "ADIOS POSIX ASCII UNBUFFERED: "
                 "file not found: %s\n", fd->name);
        free (name);
        return 0;
      }
      break;
    }
    case adios_mode_write:
    {
      p->f = fopen (name, "w");
      if (p->f <= 0)
```

```
        {
          fprintf (stderr, "adios_posix_ascii_nb_open "
                   "failed for base_path %s, name %s\n"
                   ,method->base_path, fd->name
                  );
          free (name);
          return 0;
        }
        break;
      }
      case adios_mode_append:
      {
        int old_file = 1;
        p->f = fopen (name, "a");
        if (p->f <= 0)
        {
          fprintf (stderr, "adios_posix_ascii_nb_open"
                     " failed for base_path %s, name %s\n"
                ,method->base_path, fd->name
                );
          free (name);
          return 0;
        }
        break;
      }
      default:
      {
        fprintf (stderr, "Unknown file mode: %d\n", fd->mode);
        free (name);
        return 0;
      }
    }
    free (name);
    return 0;
}

enum ADIOS_FLAG adios_posix_ascii_nb_should_buffer
                                (struct adios_file_struct * fd
                                ,struct adios_method_struct * method
                                ,void * comm)
{
    //in this case, we don't use shared_buffer
    return adios_flag_no;
}

void adios_posix_ascii_nb_write (struct adios_file_struct * fd
                            ,struct adios_var_struct * v
                            ,void * data
                            ,struct adios_method_struct * method )
{
```

```
struct adios_POSIX_ASCII_UNBUFFERED_data_struct * p;
p = (struct adios_POSIX_ASCII_UNBUFFERED_data_struct *)
                        method->method_data;
if (!v->dimensions) {
   switch (v->type)
   {
      case adios_byte:
      case adios_unsigned_byte:
         fprintf (p->f,"%c\n", *((char *)data));
         break;
      case adios_short:
      case adios_integer:
      case adios_unsigned_short:
      case adios_unsigned_integer:
         fprintf (p->f,"%d\n", *((int *)data));
         break;
      case adios_real:
      case adios_double:
      case adios_long_double:
         fprintf (p->f,"%f\n", *((double *)data));
         break;
      case adios_string:
         fprintf (p->f,"%s\n", (char *)data);
         break;
      case adios_complex:
         fprintf (p->f,"%f+%fi\n", *((float *)data),*((float *)(data+4)));
         break;
      case adios_double_complex:
         fprintf (p->f,"%f+%fi\n", *((double *)data),*((double *)(data+8)));
         break;
      default:
         break;
       }
}
else
{
   uint64_t j;
   int element_size = adios_get_type_size (v->type, v->data);
   printf("element_size: %d\n",element_size);
   uint64_t var_size = adios_get_var_size (v, fd->group, v->data)/element_size;
   switch (v->type)
   {
      case adios_byte:
      case adios_unsigned_byte:
         for (j = 0;j < var_size; j++)
            fprintf (p->f,"%c ", *((char *)(data+j)));
         printf("\n");
         break;
      case adios_short:
      case adios_integer:
```

```
        case adios_unsigned_short:
        case adios_unsigned_integer:
          for (j = 0;j < var_size; j++)
            fprintf (p->f,"%d ", *((int *)(data+element_size*j)));
          printf("\n");
          break;
        case adios_real:
        case adios_double:
        case adios_long_double:
          for (j = 0;j < var_size; j++)
            fprintf (p->f,"%f ", * ( (double *)(data+element_size*j)) );
          printf("\n");
          break;
        case adios_string:
          for (j = 0;j < var_size; j++)
            fprintf (p->f,"%s ", (char *)data);
          printf("\n");
          break;
        case adios_complex:
          for (j = 0;j < var_size; j++)
            fprintf (p->f, "%f+%fi ", *((float *)(data+element_size*j))
                 ,*((float *)(data+4+element_size*j))
                 );
          printf("\n");
          break;
        case adios_double_complex:
          for (j = 0;j < var_size; j++)
            fprintf (p->f,"%f+%fi ", *((double *)(data+element_size*j))
                 ,*((double *)(data+element_size*j+8)));
          printf("\n");
          break;
        default:
          break;
      }
    }
}

void adios_posix_ascii_nb_get_write_buffer
                                (struct adios_file_struct * fd
                                 ,struct adios_var_struct * v
                                 ,uint64_t * size
                                 ,void ** buffer
                                 ,struct adios_method_struct * method)
{
  *buffer = 0;
}

void adios_posix_ascii_nb_read (struct adios_file_struct * fd
                     ,struct adios_var_struct * v, void * buffer
                      ,uint64_t buffer_size
```

```
                    ,struct adios_method_struct * method )
{
  v->data = buffer;
  v->data_size = buffer_size;
}

int adios_posix_ascii_nb_close (struct adios_file_struct * fd
                                , struct adios_method_struct * method)
{
struct adios_POSIX_ASCII_UNBUFFERED_data_struct * p;
  p = (struct adios_POSIX_ASCII_UNBUFFERED_data_struct *)
                                    method->method_data;
  if (p->f <= 0)
  {
    fclose (p->f);
  }
  p->f = 0;
  p->file_size = 0;
}

void adios_posix_ascii_nb_finalize (int mype, struct adios_method_struct * method)
{
  if (adios_posix_ascii_nb_initialized)
    adios_posix_ascii_nb_initialized = 0;
}
```

The binary transport method blocks methods for simplicity. Therefore, no special implementation for the three functions below is necessary and their function bodies can be left empty:

adios_posix_ascii_nb_end_iteration (struct adios_method_struct * method) {}
adios_posix_ascii_nb_start_calculation (struct adios_method_struct * method) {}
adios posix_ascii_nb stop_calculation (struct adios_method_struct * method) {}

Above, we have implemented the POSIX_ASCII_NB transport method. When users specify POSIX_ASCII_NB method in xml file, the users' applications will generate ASCII files by using common ADIOS APIs. However, in order to achieve better I/O performance, a buffering scheme needs to be included into this example.

12.2 Profiling the Application and ADIOS

There are two ways to get profiling information of ADIOS I/O operations. One way is for the user to explicitly insert a set of profiling API calls around ADIOS API calls in the source code. The other way is to link the user code with a renamed ADIOS library and an ADIOS API wrapper library.

12.2.1 Use profiling API in source code

The profiling library called libadios_timing.a implements a set of profiling API calls. The user can use these API calls to wrap the ADIOS API calls in the source code to get profiling information.

The adios-timing.h header file contains the declarations of those profiling functions.

```
/*
 * initialize profiling
 *
 * Fortran interface
 */
int init_prof_all_(char *prof_file_name, int prof_file_name_size);

/*
 * record open start time for specified group
 *
 * Fortran interface
 */
void open_start_for_group_(int64_t *gp_prof_handle, char *group_name, int
*cycle, int *gp_name_size);

/*
 * record open end time for specified group
 *
 * Fortran interface
 */
void open_end_for_group_(int64_t *gp_prof_handle, int *cycle);

/*
 * record write start time for specified group
 *
 * Fortran interface
 */
void write_start_for_group_(int64_t *gp_prof_handle, int *cycle);

/*
 * record write end time for specified group
 *
 * Fortran interface
 */
void write_end_for_group_(int64_t *gp_prof_handle, int *cycle);

/*
 * record close start time for specified group
```

```
 *
 * Fortran interface
 */
void close_start_for_group_(int64_t *gp_prof_handle, int *cycle);

/*
 * record close end time for specified group
 *
 * Fortran interface
 */
void close_end_for_group_(int64_t *gp_prof_handle, int *cycle);

/*
 * Report timing info for all groups
 *
 * Fortran interface
 */
int finalize_prof_all_();

/*
 * record start time of a simulation cycle
 *
 * Fortran interface
 */
void cycle_start_(int *cycle);

/*
 * record end time of a simulation cycle
 *
 * Fortran interface
 */
void cycle_end_(int *cycle);
```

An example of using these functions is given below.

```
…
! initialization ADIOS
CALL adios_init ("config.xml"//char(0))
! initialize profiling library; the parameter specifies the file where profiling
information is written
CALL init_prof_all("log"//char(0))
…
CALL MPI_Barrier(toroidal_comm, error )
```

```
! record start time of open
! group_prof_handle is an OUT parameter holding the handle for the group
'output3d.0'
! istep is iteration no.
CALL open_start_for_group(group_prof_handle, "output3d.0"//char(0),istep)

CALL adios_open(adios_handle, "output3d.0"//char(0), "w"//char(0))

! record end time of open
CALL open_end_for_group(group_prof_handle,istep)

! record start time of write
CALL write_start_for_group(group_prof_handle,istep)

#include "gwrite_output3d.0.fh"

! record end time of write
CALL write_end_for_group(group_prof_handle,istep)

! record start time of close
CALL cose_start_for_group(group_prof_handle,istep)

CALL adios_close(adios_handle,adios_err)

! record end time of close
CALL close_end_for_group(group_prof_handle,istep)

…
CALL adios_finalize (myid)

! finalize; profiling information are gathered and min/max/mean/var are
calculated for each IO dump
CALL finalize_prof()

CALL MPI_FINALIZE(error)
```

When the code is run, profiling information will be saved to the file "./log" (specified in init_prof_all ()). Below is an example.

```
Fri Aug 22 15:42:04 EDT 2008
I/O Timing results
Operations   :      min              max              mean             var
cycle no      3
io count      0
# Open       :     0.107671         0.108245         0.108032         0.000124
# Open start :     1219434228.866144    1219434230.775268    1219434229.748614    0.588501
# Open end   :     1219434228.974225    1219434230.883335    1219434229.856646    0.588486
```

```
# Write     :       0.000170          0.000190          0.000179          0.000005
# Write start :     1219434228.974226     1219434230.883336     1219434229.856647     0.588486
# Write end  :      1219434228.974405     1219434230.883514     1219434229.856826     0.588484
# Close     :       0.001608          0.001743          0.001656          0.000036
# Close start :     1219434228.974405     1219434230.883514     1219434229.856826     0.588484
# Close end  :      1219434228.976040     1219434230.885211     1219434229.858482     0.588489
# Total     :       0.109484          0.110049          0.109868          0.000137
cycle no       6
io count       1
# Open      :       0.000007          0.000011          0.000009          0.000001
# Open start :      1219434240.098444     1219434242.007951     1219434240.981075     0.588556
# Open end   :      1219434240.098452     1219434242.007962     1219434240.981083     0.588556
# Write     :       0.000175          0.000196          0.000180          0.000004
# Write start :     1219434240.098452     1219434242.007962     1219434240.981083     0.588557
# Write end  :      1219434240.098631     1219434242.008158     1219434240.981264     0.588558
# Close     :       0.000947          0.003603          0.001234          0.000466
# Close start :     1219434240.098631     1219434242.008158     1219434240.981264     0.588558
# Close end  :      1219434240.099665     1219434242.009620     1219434240.982498     0.588447
# Total     :       0.001132          0.003789          0.001423          0.000466
```

The script "post_script.sh" extracts "open time", "write time", "close time", and "total time" from the raw profiling results and saves them in separate files: open, write, close, and total, respectively.

To compile the code, one should link the code with the *–ladios_timing -ladios* option.

12.2.2 Use wrapper library

Another way to do profiling is to link the source code with a renamed ADIOS library and a wrapper library.

The renamed ADIOS library implements "real" ADIOS routines, but all ADIOS public functions are renamed with a prefix "P". For example, adios_open() is renamed as Padios_open(). The routine for parsing config.xml file is also changed to parse extra flags in config.xml file to turn profiling on or off.

The wrapper library implements all adios pubic functions (e.g., adios_open, adios_write, adios_close) within each function. It calls the "real" function (Padios_xxx()) and measure the start and end time of the function call.

There is an example wrapper library called libadios_profiling.a. Developers can implement their own wrapper library to customize the profiling.

To use the wrapper library, the user code should be linked with *–ladios_profiling –ladios*. the wrapper library should precede the "real" ADIOS library. There is no need to put additional profiling API calls in the source code. The user can turn profiling on or off for each ADIOS group by setting a flag in the config.xml file.

```
<adios-group name="restart.model" profiling="yes|no">
  ...
</adios-group>
```

13 Appendix

Datatypes used in the ADIOS XML file

size	Signed type	Unsigned type
1	byte, interger*1	unsigned byte, unsigned integer*1
2	short, integer*2	unsigned short, unsigned integer*2
4	integer, integer*4, real, real*4, float	unsigned integer, unsigned integer*4
8	long, integer*8, real*8, double, long float, complex, complex*8	
16	real*16, long double, double complex, complex*16	
	string	

ADIOS APIs List

Function	Purpose
adios_init	Load the XML configuration file creating internal representations of the various data types and defining the methods used for writing.
adios_finalize	Cleanup anything remaining before exiting the code
adios_open	Prepare a data type for subsequent calls to write data using the io_handle. Mode is one of “r” (read), “w” (write) and “a” (append).
adios_close	Commit all the write to disk, close the file and release adios file handle
adios_group_size	Passing the required buffer size to the transport layer and returned the total size back to the source code
adios_write	Submit a data element for writing. This does NOT actually perform the write in buffered mode. In the overflow case, this call writes to buffer directly.
adios_read	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

adios_set_path	Set the HDF5-style path for all variables in a adios-group. This will reset whatever is specified in the XML file.
adios_set_path_var	Set the HDF-5-style path for the specified var in the group. This will reset whatever is specified in the XML file.
adios_get_write_buffer	For the given field, get a buffer that will be used at the transport level for it of the given size. If size == 0, then auto calculate the size based on what is known from the datatype in the XML file and any provided additional elements (such as array dimension elements). To return this buffer, just do a normal call to adios_write using the same io_handle, field_name, and the returned buffer.
adios_start_calculation	An indicator that it is now an ideal time to do bulk data transfers as the code will not be performing IO for a while.
adios_end_ calculation	An indicator that it is no longer a good time to do bulk data transfers as the code is about to start doing communication with other nodes causing possible conflicts
adios_end_iteration	A tick counter for the IO routines to time how fast they are emptying the buffers.