ORNL/TM-2009/100

ADIOS 1.5.0 User's Manual

June 2013

DOCUMENT AVAILABILITY

Reports produced after January 1, 1996, are generally available free via the U.S. Department of Energy (DOE) Information Bridge:

Web site:http://www.osti.gov/bridge

Reports produced before January 1, 1996, may be purchased by members of the public from the following source:

National Technical Information Service
5285 Port Royal Road
Springfield, VA 22161
Telephone:703-605-6000 (1-800-553-6847)
TDD:703-487-4639
Fax:703-605-6900
E-mail:info@ntis.fedworld.gov
Web site:http://www.ntis.gov/support/ordernowabout.htm

Reports are available to DOE employees, DOE contractors, Energy Technology Data Exchange (ETDE) representatives, and International Nuclear Information System (INIS) representatives from the following source:

Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831
Telephone:865-576-8401
Fax:865-576-5728
E-mail:reports@adonis.osti.gov
Web site:http://www.osti.gov/contact.html

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

ORNL/TM-2009/100

ADIOS 1.5.0 USER'S MANUAL

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

Authors

N. Podhorszki, Q. Liu, J. Logan, H. Abbasi, J.Y. Choi, S. Klasky

Contributors

J. Lofstead, S. Hodson, F. Zheng, M. Wolf, T. Kordenbrock, N. Samatova, J.Q. Mu, J. Dayal, Y. Tian

June 2013

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

1 Introduction **10**
1.1 Goals . . . 10
1.2 What is ADIOS? . . . 10
1.3 The Basic ADIOS Group Concept . . . 10
1.4 Other Interesting Features of ADIOS . . . 10
1.5 What's new in version 1.5.0 . . . 11
1.6 What's new in version 1.4 . . . 11

2 Installation **13**
2.1 Obtaining ADIOS . . . 13
2.2 Quick Installation . . . 13
2.2.1 Quick installation with Automake . . . 13
2.2.2 Quick installation with CMake . . . 14
2.3 ADIOS Dependencies . . . 15
2.3.1 Mini-XML parser (required) . . . 15
2.3.2 MPI and MPI-IO (required) . . . 15
2.3.3 Python (required) . . . 15
2.3.4 Fortran90 compiler (optional) . . . 15
2.3.5 Serial NetCDF-3 (optional) . . . 15
2.3.6 Serial HDF5 (optional) . . . 15
2.3.7 Lustreapi (optional) . . . 16
2.3.8 Staging transport methods (optional) . . . 16
2.3.9 Read-only installation . . . 17
2.3.10 PHDF5 (optional) . . . 17
2.3.11 NetCDF-4 Parallel (optional) . . . 18
2.3.12 Serial HDF5 (optional) . . . 18
2.4 Full Installation . . . 18
2.4.1 Full Installation with Automake . . . 18
2.4.2 Full Installation with CMake . . . 19
2.5 Compiling applications using ADIOS . . . 19
2.5.1 Sequential applications . . . 21
2.6 Language bindings . . . 21
2.6.1 Support for Matlab . . . 21
2.6.2 Support for Java . . . 21
2.6.3 Support for Numpy . . . 22

3 ADIOS Write API **23**
3.1 Write API Description . . . 23
3.1.1 Introduction . . . 23
3.1.2 ADIOS-required functions . . . 23
3.1.3 Asynchronous I/O support functions . . . 27
3.1.4 Other functions . . . 27
3.2 Write Fortran API description . . . 27
3.2.1 Create the first ADIOS program . . . 29

4 ADIOS No-XML Write API **31**
4.1 No-XML Write API Description . . . 31
4.1.1 adios_init_noxml . . . 31
4.1.2 adios_allocate_buffer . . . 32
4.1.3 adios_declare_group . . . 32
4.1.4 adios_define_var . . . 32
4.1.5 adios_write_byid . . . 33
4.1.6 adios_define_attribute . . . 33
4.1.7 adios_select_method . . . 34
4.2 Create a no-XML ADIOS program . . . 34

5 XML Config File Format **37**
5.1 Overview . . . 37
5.2 adios-config . . . 37
5.3 adios-group . . . 38
5.3.1 Declaration . . . 38
5.3.2 Variables . . . 38
5.3.3 Attributes . . . 39
5.3.4 Gwrite src . . . 40
5.3.5 Global arrays . . . 40
5.3.6 Time-index . . . 40
5.4 Transport method . . . 41
5.4.1 Declaration . . . 41
5.4.2 Methods list . . . 41
5.5 Buffer specification . . . 42
5.5.1 Declaration . . . 42
5.6 Enabling Histogram . . . 42
5.6.1 Declaration . . . 42
5.7 An Example XML file . . . 43

6 Transport Methods **44**
6.1 Mainline Transport Methods . . . 44
6.1.1 NULL . . . 44
6.1.2 POSIX . . . 44
6.1.3 MPI . . . 44
6.1.4 MPI_LUSTRE . . . 45
6.1.5 MPI_AGGREGATE . . . 46
6.1.6 VAR_MERGE . . . 47
6.1.7 Dataspaces . . . 48
6.1.8 DIMES . . . 49
6.1.9 Flexpath . . . 49
6.1.10 PHDF5 . . . 49
6.1.11 NetCDF4 . . . 50
6.2 Research Methods . . . 51
6.2.1 Network Scalable Service Interface (NSSI) . . . 51
6.2.2 DataTap . . . 52
6.2.3 MPI-CIO . . . 53
6.2.4 MPI-AIO . . . 54

7 ADIOS Read API **55**
7.1 Introduction . . . 55
7.1.1 Changes from version 1 . . . 55
7.1.2 Concepts . . . 55
7.1.3 Selections . . . 57
7.2 How to use the read functions . . . 57
7.3 Notes . . . 58

7.4 Read C API description . . . 58
7.4.1 adios_errmsg / adios_errno . . . 59
7.4.2 adios_read_init_method . . . 59
7.4.3 adios_read_finalize_method . . . 60
7.4.4 adios_read_open . . . 60
7.4.5 adios_read_open_file . . . 61
7.4.6 adios_read_close . . . 61
7.4.7 adios_advance_step . . . 61
7.4.8 adios_release_step . . . 62
7.4.9 adios_inq_var . . . 62
7.4.10 adios_inq_var_byid . . . 62
7.4.11 adios_free_varinfo . . . 63
7.4.12 adios_inq_var_stat . . . 63
7.4.13 adios_inq_var_blockinfo . . . 63
7.4.14 Selections . . . 63
7.4.15 adios_schedule_read . . . 65
7.4.16 adios_schedule_read_byid . . . 65
7.4.17 adios_perform_reads . . . 66
7.4.18 adios_check_reads . . . 66
7.4.19 adios_free_chunk . . . 66
7.4.20 adios_get_attr . . . 66
7.4.21 adios_get_attr_byid . . . 67
7.4.22 adios_type_to_string . . . 67
7.4.23 adios_type_size . . . 67
7.4.24 adios_get_grouplist . . . 67
7.4.25 adios_group_view . . . 67
7.5 Time series analysis API Description . . . 68
7.5.1 adios_stat_cor / adios_stat_cov . . . 68
7.6 Read Fortran API description . . . 69
7.7 Compiling and linking applications . . . 72
7.7.1 C/C++ applications . . . 73
7.7.2 Fortran applications . . . 73
7.8 Supported scenarios and samples . . . 73
7.9 Reading a file as file . . . 73
7.9.1 Discover and read in a complete variable . . . 73
7.9.2 Multiple steps of a variable . . . 74
7.9.3 Read a bounding box subset of a variable . . . 74
7.9.4 Reading non-global variables written by multiple processes . . . 74
7.10 Reading streams . . . 76
7.10.1 Opening a stream . . . 76
7.10.2 Reading one step at a time, blocking if a new step is late . . . 77
7.10.3 Locking and step advancing scenarios . . . 77
7.10.4 Handling errors due to missing steps . . . 78
7.11 Non-blocking reads . . . 78
7.11.1 Chunk reads: read without pre-allocating buffers . . . 78
7.11.2 Read into user-allocated buffers . . . 79
7.12 More esoteric scenarios . . . 79
7.12.1 In situ read: read data locally available on the node . . . 79
7.12.2 Variable stepping of variables in a stream . . . 80

8 Utilities **81**
8.1 adios_lint . . . 81
8.2 adios_config . . . 81
8.3 bpls . . . 81
8.4 bpdump . . . 83

9 Converters **84**
9.1 bp2h5 . . . 84
9.2 bp2ncd . . . 84
9.3 bp2ascii . . . 84
9.4 Parallel Converter Tools . . . 85

10 Group Read/Write Process **86**
10.1 Gwrite/gread/read . . . 86
10.2 Add conditional expression . . . 86

11 Language bindings **88**
11.1 Java support . . . 88
11.1.1 Adios class . . . 88
11.1.2 AdiosFile, AdiosGroup, and AdiosVarinfo classes . . . 90
11.1.3 AdiosDatatype, AdiosFlag, and AdiosBufferAllocWhen classes . . . 91
11.1.4 Example . . . 91
11.2 Python/Numpy support . . . 92
11.2.1 APIs for Writing and No-XML . . . 92
11.2.2 APIs for Reading . . . 94
11.2.3 Example . . . 96

12 C Programming with ADIOS **99**
12.1 Non-ADIOS Program . . . 99
12.2 Construct an XML File . . . 100
12.3 Generate .ch file (s) . . . 100
12.4 POSIX transport method (P writers, P subfiles + 1 metadata file) . . . 101
12.5 MPI-IO transport method (P writers, 1 file) . . . 102
12.6 Reading data from the same number of processors . . . 102
12.7 Writing to Shared Files (P writers, N files) . . . 103
12.8 Global Arrays . . . 104
12.8.1 MPI-IO transport method (P writers, 1 file) . . . 105
12.8.2 POSIX transport method (P writers, P Subfiles + 1 Metadata file) . . . 106
12.9 Writing Time-Index into a Variable . . . 106
12.10 Reading statistics . . . 107

13 Appendix **110**
13.1 Datatypes used in the ADIOS XML file . . . 110
13.2 ADIOS APIs List . . . 111
13.3 An Example on Writing Sub-blocks using No-XML APIs . . . 111

List of Figures

6.1 Server-friendly metadata approach: offset the create/open in time 45
6.2 Spatial Aggregation of a 4x4 2D Variable From 4 proceeses to Process 0 47
6.3 DataTap architecture 53

Abbreviations

ADIOS Adaptive Input/Output System

API Application Program Interface

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.

Chapter 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 and IBM BlueGene/P supercomputers as well as on Linux clusters and Mac OSX.

1.5 What's new in version 1.5.0

Some small changes to the API have been made in this version.

- `adios_init()` has an MPI_Comm argument
- `adios_open()` also has an MPI_Comm argument instead of a void * argument. This means, existing codes have to be modified to pass the communicator itself instead of a pointer to it. The C compiler gives a warning only when compiling old codes, which can easily be missed.
- `adios_read_open()` is introduced instead of `adios_read_open_stream()` to indicate that this function is to be used equally for files and staged datasets. It opens the file/stream as a stream, see more explanation in the Read API chapter 7.

Two new staging methods, DIMES and FLEXPATH have been added. They require third-party software to be installed.

A new build system using CMake has been added. The two, automake and CMake build will go along for a while but eventually ADIOS will use CMake.

A new write method, VAR_MERGE, has been added, that performs spatial aggregation of small data blocks of processors to write larger chunks to the output file. It improves both the write and read performance of such datasets.

1.6 What's new in version 1.4

With ADIOS 1.4, there are several changes and new functionalities. The four major changes are in the Read API:

- No groups at reading anymore. You get all variables in one list. There are no `adios_gopen` / `adios_gclose` / `adios_inq_group` calls after opening the file.
- No time dimension. A 3D variable written multiple times will be seen as a 3D variable which has multiple steps (and not as single 4D variable as in adios 1.3.1). Read requests should provide the number of steps to be read at once separately from the spatial dimensions.
- Multiple reads should be "scheduled" and then one `adios_perform_reads()` will do all at once.
- Selections. Instead of providing bounding box (offset and count values in each dimension) in the read request itself, a selection has to be created beforehand. Besides bounding boxes, also list of individual points are supported as well as selections of a specific block from a particular writing process.

Overall, a single old `adios_read_var()` becomes three calls, but n reads over the same subdomain requires $1 + n + 1$ calls. All changes were made towards in situ applications, to support streaming, non-blocking, chunking reads. Old codes can use the old read API too, for reading files but new users are strongly encouraged to use the new read API, even if they personally find the old one simpler to use for reading data from a file. The new API allows applications to move to in situ (staged, or memory-to-memory) processing of simulation data when file-based offline processing or code coupling becomes severely limited.

Other new things in ADIOS:

- New read API. Files and streams can be processed step-by-step (or files with multiple steps at once). Multiple read requests are served at once, which enables for superior performance with some methods. Support for non-blocking and for chunked reads in memory-limited applications or for interleaving computation with data movement, although no current methods provide performance advantages in this release.
- Fortran90 modules for write and read API. Syntax of ADIOS calls can be checked by the Fortran compiler.
- Java and Numpy bindings available (they should be built separately).
- Visualization schema support in the XML configuration. Meshes can be described using output variables and data variables can be assigned to meshes. This will allow for automatic visualization from ADIOS-BP files with rich metadata, or to convey the developer's intentions to other users about how to visualize the data. A manual on the schema is separate from this Users' Manual and can be downloaded from the same web page.
- *Skel* I/O skeleton generator for automatic performance evaluation of different methods. The XML configuration, that describes the output of an application, is used to generate code that can be used to test out different methods and to choose the best. Skel is part of ADIOS but it's manual is separate from this Users' Manual and can be downloaded from the same web page.

Chapter 2

Installation

2.1 Obtaining ADIOS

You can download the latest version from the following website

```
http://www.olcf.ornl.gov/center-projects/adios
```

2.2 Quick Installation

2.2.1 Quick installation with Automake

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.

2.2.1.1 Linux cluster

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

```
export MPICC=mpicc
export MPICXX=mpiCC
export MPIFC=mpif90
export CC=pgcc
export CXX=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/XK, 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 language bindings later.

2.2.1.2 Cray supercomputers

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

```
export CC=cc
export CXX=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>
```

2.2.2 Quick installation with CMake

CMake is an alternative way used to configure, build, test, and install the ADIOS library, header files, and support programs. CMake 2.8.0 or higher is required to build ADIOS.

```
cd trunk/
source cmake_init
mkdir build
cd build
cmake ..
make
```

Note: The cmake_init batch script in the trunk set up for our machines. You need to set up the appropriate environment variables and configure options for your system.

The nice and highly recommended feature of CMake is the ability of out-of-source build. More specifically, all generated temporary files, binary executables and new files are within the "build" folder, a completely separate directory, which will make the source tree without cluttering up. To do this, first create a separate build directory (see "mkdir build") and then switch to the build directory and run cmake with an argument for the ADIOS source directory.

If crossing-compiling is required for a certain system, for example the intel compiler on Titan, there is another cmake variable CMAKE_TOOLCHAIN_FILE need to be initialized. This changes the cmake command from

```
cmake ..
```

to

```
cmake -DCMAKE_TOOLCHAIN_FILE=../toolchain.cmake ..
```

To install ADIOS on a Cray XK6, the right compiler commands and configure flags need to be set. The required commands for ADIOS installation on Titan are as follows:

```
export CC=cc
export CXX=CC
export FC=ftn
export MXML_DIR=<location of mini-xml installation>
export SEQ_HDF5_DIR=<location of HDF5 installation>
export SEQ_NC_DIR=<location of NETCDF installation>
cmake <ADIOS_SOURCEDIR>
```

2.2.2.1 Linux cluster

The following is a snapshot of the batch scripts on Sith, an Intel-based Infiniband cluster running Linux. Note the difference between the Automake and CMake builds: CMake uses MPI compilers for the complete build in the current version of ADIOS.

```
export CC=mpicc
export CXX=mpiCC
export FC=mpif90
export CFLAGS="-fPIC"
cmake <ADIOS_SOURCEDIR>
```

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 ADIOS.

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 Python (required)

The XML processing utility `utils/gpp/gpp.py` is a code written in python using xml.dom.minidom. It is used to generate C or Fortran code from the XML configuration files that can be included in the application source code. Examples and tests will not build without Python.

2.3.4 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` (Automake) or `export BUILD_FORTRAN=OFF` (CMake).

2.3.5 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 with Automake. While with CMake, the environment variables should be set before running cmake:

```
export SEQ_NC_DIR=<path>
```

2.3.6 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 configuring ADIOS with Automake or

```
export SEQ_HDF5_DIR=<path>
```

with CMake.

2.3.7 Lustreapi (optional)

The Lustreapi library is used internally by `MPI_LUSTRE` and `MPI_AMR` method to figure out Lustre parameters such as stripe count and stripe size. Without giving this option, users are expected to manually set Lustre parameters from ADIOS XML configuration file (see `MPI_LUSTRE` and `MPI_AMR` method). Use the configuration option `--with-lustre=<path>` to define the path to this library.

2.3.8 Staging transport methods (optional)

In ADIOS 1.5, three transport methods are available for memory-to-memory transfer (staging) of data between two applications: The DataSpaces and DIMES libraries from Rutgers University and the Flexpath library from Georgia Tech.

2.3.8.1 Networking libraries for staging

Staging methods use Remote Direct Memory Access (RDMA) operations, supported by specific libraries on various systems.

Infiniband. If you have an Infininband network with `ibverbs` and `rdmacm` libraries installed, you can configure ADIOS to use it for staging methods with the option `--with-infiniband=DIR` in Automake to define the path to the Infiniband libraries. In CMake, library ibverbs is detected by examining if function `ibv_alloc_pd` exists auomatically without extra effort by the user.

Cray Gemini network. On newer Cray machines (XK6 and XE6) with the Gemini network, the `PMI` and `uGNI` libraries are used by the staging methods. Configure ADIOS with the options in Automake

```
--with-cray-pmi=/opt/cray/pmi/default \
--with-cray-ugni-incdir=/opt/cray/gni-headers/default/include \
--with-cray-ugni-libdir=/opt/cray/ugni/default/lib
```

or in CMake

```
export CRAY_PMI_DIR=/opt/cray/pmi/default
export CRAY_UGNI_DIR=/opt/cray/ugni/default
```

Portals. Portals is an RDMA library from Sandia Labs, and it has been used on Cray XT5 machines with Seastar networks. Configure ADIOS with the option

```
--with-portals=DIR      Location of Portals (yes/no/path_to_portals)
```

2.3.8.2 DataSpaces staging method

The DataSpaces model provides a separate server running on separate compute nodes, into/from which data can be written/read with a geometrical (3D) abstraction. It is an efficient way to stage data from an application to one or more other applications in an asynchronous way. Multiple steps of data outputs can be stored, limited only by the available memory. It can also be used for interactive in-situ visualization, where the visualization can be multiple steps behind the application. DataSpaces can be downloaded from `http://www.dataspaces.org`
Build the DataSpaces method with the option in Automake:

```
--with-dataspaces=DIR  Build the DATASPACES transport method. Point to the
                       DATASPACES installation.
--with-dataspaces-incdir=<location of dataspaces includes>
--with-dataspaces-libdir=<location of dataspaces library>
```

or in CMake

```
export DATASPACES_DIR=<location of DATASPACES installation>
```

2.3.8.3 DIMES staging method

In the DIMES model, the reading application pulls the data directly from the writer application's memory. It provides the same geometrical (3D) abstraction for writing/reading datasets as DataSpaces. It is an efficient way to stage data from one application to another in an asynchronous (and very fast) way. Only a single step of data output can be stored. DIMES is part of the DataSpaces library.
Build the DIMES method with the option:

```
--with-dimes=DIR  Build the DIMES transport method. Point to the
                      DIMES installation.
--with-dimes-incdir=<location of dimes includes>
--with-dimes-libdir=<location of dimes library>
```

2.3.8.4 Flexpath staging method

Flexpath transport requires several packages from the Chaos repository at Georgia Tech: `dill`, `cercs_env`, `gen_thread`, `atl`, `ffs`, and `evpath`. To download and install these libraries, a script (`trunk/chaos_build.pl`) is available at: `http://svn.research.cc.gatech.edu/kaos/chaos_base/` (username: `anon`, password: `anon`). Download the repository using SVN, and follow the instructions at the top of the script.
To use ADIOS with Flexpath, configure adios with the option:

```
--with-flexpath=DIR  Where DIR is the installation directory of
                                        the Chaos libraries.
```

The default network transport for Flexpath is sockets. Additionally, you can use ENET and NNTI (allowing for Infiniband, Portals, and Gemini), both of which are included in the `chaos_build.pl` script. To use these transports, set the `CMTransport` environment variable to either ``nnti'' or ``enet'' before running your applications with Flexpath.

Additionally, in the ADIOS XML file, we allow for the user to specify a `Queuesize` parameter specifying how many timesteps the writer can buffer before it blocks. For example,

```
<method group="arrays" method="FLEXPATH">QUEUE_SIZE=10;</method>
```

2.3.9 Read-only installation

If you just want the read API to be compiled for reading BP files, use the `--disable-write` option with Automake and `export BUILD_WRITE=OFF` with CMake.

2.3.10 PHDF5 (optional)

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

```
export PAR_HDF5_DIR=<path>
```

Note: Do not expect better performance with ADIOS/PHDF5 than with PHDF5 itself. ADIOS does not write differently to a HDF5 formatted file, it simply uses PHDF5 function calls to write out data. Also good to know, that the method in ADIOS uses the collective function calls, that requires that every process participates in the writing of each variable.

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 your 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>
```

2.3.11 NetCDF-4 Parallel (optional)

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. You also need to provide the parallel HDF5 library.

While with CMake, the environment variables are set by the folloing:

```
export PAR_NC_DIR=<path>
```

2.3.12 Serial HDF5 (optional)

Note: Do not expect better performance with ADIOS/NC4 than with NC4 itself. ADIOS does not write differently to a HDF5 formatted file, it simply uses NC4 function calls to write out data. Also good to know, that this method requires that every process participates in the writing of each variable.

2.4 Full Installation

2.4.1 Full Installation with Automake

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-infiniband=DIR      Location of Infiniband
--with-portals=DIR      Location of Portals (yes/no/path_to_portals)
--with-cray-pmi=<location of CRAY_PMI installation>
--with-cray-pmi-incdir=<location of CRAY_PMI includes>
--with-cray-pmi-libdir=<location of CRAY_PMI library>
--with-cray-pmi-libs=<linker flags besides -L<cray-pmi-libdir>, e.g. -lpmi
--with-cray-ugni=<location of CRAY UGNI installation>
--with-cray-ugni-incdir=<location of CRAY UGNI includes>
--with-cray-ugni-libdir=<location of CRAY UGNI library>
--with-cray-ugni-libs=<linker flags besides -L<cray-ugni-libdir>, e.g. -lugni
--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
--with-dataspaces=<location of DataSpaces installation>
--with-dataspaces-incdir=<location of DataSpaces includes>
--with-dataspaces-libdir=<location of DataSpaces library>
--with-dimes=<location of DataSpaces installation>
--with-dimes-incdir=<location of dimes includes>
--with-dimes-libdir=<location of dimes library>
--with-flexpath=<location of the Chaos packages>
--with-lustre=<location of Lustreapi library>
```

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
```

2.4.2 Full Installation with CMake

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

```
export MXML_DIR=<location of mxml installation>
export SEQ_NC_DIR=<location of sequential netcdf installation>
export PAR_NC_DIR=<location of parallel netcdf installation>
export SEQ_HDF5_DIR=<location of sequential hdf5 installation>
export PAR_HDF5_DIR=<location of parallel hdf5 installation>
export CRAY_UGNI_DIR=<location of CRAY UGNI installation>
export CRAY_PMI_DIR=<location of CRAY_PMI installation>
export DATASPACES_DIR=<location of DataSpaces installation>
```

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>
CXX        C++ compiler command
CXXFLAGS   C++ compiler flags
FC         Fortran compiler command
FCFLAGS    Fortran compiler flags
```

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. In a Makefile, if you set `ADIOS_DIR` to the installation directory of ADIOS, you can set the flags for building your code flexibly as shown below for a Fortran application:

```
override ADIOS_DIR := <your ADIOS installation directory>
override ADIOS_INC := $(shell ${ADIOS_DIR}/bin/adios_config -c -f)
override ADIOS_FLIB := $(shell ${ADIOS_DIR}/bin/adios_config -l -f)

example.o : example.F90
```

```
        ${FC} -g -c ${ADIOS_INC} example.F90 $<

example: example.o
        ${FC} -g -o example example.o ${ADIOS_FLIB}
```

The example above is for using write (and read) in a Fortran + MPI application. However, several libraries are built for specific uses:

- `libadios.a` MPI + C/C++ using ADIOS to write and read data
- `libadiosf.a` MPI + Fortran using ADIOS to write and read data
- `libadios_nompi.a` C/C++ without MPI
- `libadiosf_nompi.a` Fortran without MPI
- `libadiosread.a` MPI + C/C++ using ADIOS to only read data
- `libadiosreadf.a` MPI + Fortran using ADIOS to only read data
- `libadiosread_nompi.a` C/C++ without MPI, using ADIOS to only read data
- `libadiosreadf_nompi.a` Fortran without MPI, using ADIOS to only read data

The C libraries include both the old and new API in one library. However, the old read API in Fortran has name clashes with the new API, therefore separate Fortran libraries are built for it:

- `libadiosf_v1.a`
- `libadiosreadf_v1.a`
- `libadiosf_nompi_v1.a`
- `libadiosreadf_nompi_v1.a`

The following options in `adios_config` allows for setting the include and link flags for a specific build:

```
adios_config [-d | -c | -l] [-f] [-r] [-s] [-1] [-v] [-i]
Arguments
   -d   Base directory for ADIOS install
   -c   Compiler flags for C/C++, using ADIOS write/read methods
   -l   Linker flags for C/C++, using ADIOS write/read methods

   -f   Print above flags for Fortran90
   -r   Print above flags for using ADIOS read library only.
   -s   Print above flags for using ADIOS in a sequential code (no MPI).
   -1   Print above flags for using old Read API of ADIOS.

   -v   Version of the installed package
   -i   More installation information about the package

Notes
   - Multiple options of d,c,l are enabled. In such a case, the output is
     a list of FLAG=flags, where FLAG is one of (DIR, CFLAGS, LDFLAGS)
   - If none of d,c,l are given, all of them is printed
   - If none of f,r,s are given, flags for C/C++, using ADIOS write/read
     methods are printed
```

That is, for example, `adios_config -lfrs` will print the *link* flags for building a *sequential Fortran* application that *only reads* data with ADIOS.

2.5.1 Sequential applications

Use the `-D_NOMPI` pre-processor flag to compile your application for a sequential build. ADIOS has a dummy MPI library, `mpidummy.h`, that re-defines all MPI constructs necessary to run ADIOS without MPI. You can declare

`MPI_Comm comm;`

in your sequential code to pass it on to functions that require an `MPI_Comm` variable.

If you want to write a C/C++ parallel code using MPI, but also want to provide it as a sequential tool on a login-node without modifying the source code, then write your application as MPI, do not include `mpi.h` but include `adios.h` or `adios_read.h`. for the sequential build. `adios.h`/`adios_read.h` include the appropriate header file `mpi.h` or `mpidummy.h` (the latter provided by ADIOS) depending on which version you want to build.

2.6 Language bindings

ADIOS comes with various bindings to languages, that are not built with the Automake tools discussed above. After building ADIOS, these bindings have to be manually built.

2.6.1 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. 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 or set the `MATLABPATH` to this directory to let Matlab know where to look for the bindings.

```
cd wrappers/matlab
make matlab
```

2.6.2 Support for Java

ADIOS provides a Java language binding implemented by the Java Native Interface (JNI). The program can be built with CMake (`http://www.cmake.org/`) which will detect your ADIOS installation and related programs and libraries. With CMake, you can create a build directory and run cmake pointing the Java wrapper source directory (wrappers/java) containing CMakeLists.txt. For example,

```
cd wrappers/java
mkdir build
cd build
cmake ..
```

CMake will search installed ADIOS libraries, Java, JNI, MPI libraries (if needed), etc. Once completed, type `make` to build. If you need verbose output, you type as follows:

```
make VERBOSE=1
```

After successful building, you will see libAdiosJava.so (or libAdiosJava.dylib in Mac) and AdiosJava.jar. Those two files will be needed to use in Java. Detailed instructions for using this Java binding will be discussed in Section 11.1.

If you want to install those files, type the following:

```
make install
```

The default installation directory is `/usr/local`. You can change by specifying `CMAKE_INSTALL_PREFIX` value;

```
cmake -DCMAKE_INSTALL_PREFIX=/path/to/install /dir/to/source
```

Or, you can use the `ccmake` command, the CMake curses interface. Please refer to the CMake documents for more detailed instructions.

This program contains a few test programs. To run testing after building, type the following command:

```
make test
```

If you need a verbose output, type the following

```
ctest -V
```

2.6.3 Support for Numpy

ADIOS also provides a Python/Numpy language binding. The source code is located in `wrappers/numpy`. This module is developed by Cython.

Like the Java binding, this Python/Numpy wrapper can be built by using CMake (`http://www.cmake.org/`). You can create a build directory and run cmake by pointing the source directory. For example,

```
cd wrappers/numpy
mkdir build
cd build
cmake ..
```

CMake will search installed ADIOS library and Python/Numpy. Once completed, type `make` to build or type the following for the verbose output.

```
make VERBOSE=1
```

After successful building, you will see adios.so. This file can be loaded in Python. Detailed instructions for using this module in Python will be discussed in Section 11.2.

This program contains a few test programs. To run testing after building, type the following command:

```
make test
```

If you need a verbose output, type the following

```
ctest -V
```

Chapter 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 function is required only once during the program run. It loads the 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. From version 1.5, this function does have an `MPI_Comm comm` argument.

```
int adios_init (const char * xml_fname, MPI_Comm comm)
```

Input:

- xml_fname - string containing the name of the XML configuration file
- comm - MPI communicator. Any process that is going to use ADIOS should call this function and must be a member of this communicator.

Fortran example:

```
call adios_init ("config.xml", comm, ierr)
```

3.1.2.2 adios_open

This function is to open or to append to an output file. adios_open opens an adios-group identified by *group_name* and associates it with one or a list of transport methods. A pointer is returned as *fd_p* for subsequent operations. The group name should match one of the groups defined in the XML file. The I/O handle The third argument, *file_name*, is a string representing the name of the file. The fourth argument *mode* is a string containing a file access mode. It can be one 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 last argument is the MPI communicator *comm* that includes all processes that write to the file. Individual writes can be called by individual processes, but *adios_group_size* and *adios_close* are collective operations, that all processes under this communicator should call.

Note, that a file is not necessarily opened during this call. Some methods postpone the actual file open to *adios_group_size*.

Note, that before version 1.5, this function required a `void * comm` argument, so you need to update existing codes to pass the communicator itself now, not a pointer to it.

```
int adios_open (int64_t * fd_p, const char * group_name,
            const char * file_name, const char * mode, MPI_Comm 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 (in bytes) for the group opened in the adios_open routine that the specific process is going to write into the file. This value can be calculated manually, knowing the sizes of all variables to be written or through our python script `gpp.py` that generates and puts this calculation and *adios_group_size* call along the write operations of all variables into a text file. 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, which is the payload size increased with the metadata overhead. The value can be used for performance benchmarks, such as I/O speed.

Note that in the XML file, you should specify a buffer size for ADIOS. The buffer is used to collect all outputs between one adios_open – adios_close cycle, and all data is written out during adios_close. This

maximizes the benefits of large I/O chunks and is one of the key contributors of the superior performance of ADIOS file I/O. If the buffer is not sufficiently large enough to hold all data for the output, ADIOS methods fall back to write data to the target at each adios_write, that will result in much worse write performance.

```
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 $4 \times 2 \times 3$ (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 XML definition of the corresponding 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. When the function returns, the memory pointed by *var* can be reused by the application. Adios_write expects the address of the contiguous block of memory to be written. A noncontiguous array, comprising a series of subcontiguous memory blocks, should be given separately for each piece.

In the next chapter about the XML file, we will further explain that the var_name argument of this function should correspond to the value of the attribute "name" in the variable definition. Another attribute, "gwrite," is used by `gpp.py` to generate the variable name in the application source code, that is passed as *var* in this call. See the `<var>` element inside the `<adios_group>` element in the XML file. If "gwrite" is not defined, it will be handled as if it were the same as the value of attribute "name".

```
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, use a transport method that does not support read (e.g. the MPI_AMR method) 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 7.

Similar to adios_write, adios_read passes the buffer space in the *var* argument 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 file was opened for read, this function fetches all data and populates it into the buffers provided in the adios_read 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 developers of ADIOS transport methods the opportunity to customize some special operations based on the proc_id—usually on one process.

```
int adios_finalize (int proc_id)
```

Input:

- proc_id - the rank of the process (in the MPI application)

Fortran example:

```
call adios_finalize (rank, ierr)
```

call adios_finalize (rank, ierr)

3.1.3 Asynchronous I/O support 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. Applications usually perform computation in an iterative loop, and write data with a regular frequency (but not at every iteration). This function, if called at each iteration, can provide hints to the ADIOS layer about the progress of the application and thus estimate the remaining time to the next output phase. Asynchronous I/O methods can use this estimate to trickle the data of the previous output as slow as possible to minimize interference with the application.

3.1.3.2 adios_start_ calculation/ adios_end_calculation

Together, adios_start_calculation and adios_end_calculation indicate to asynchronous methods when they 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 collective communication of the application will negatively impact both the asynchronous I/O performance and the interprocess messaging.

3.1.4 Other functions

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.

3.2 Write Fortran API description

A Fortran90 module, `adios_write_mod.mod` provides the ADIOS write subroutines discussed above. They are all interfaced to the C library. Their extra last argument (compared to the corresponding C functions) is an integer variable to store the error code output of each function (0 meaning successful operation).

Here is the list of the Fortran90 subroutines from `adios_write_mod.mod`. In the list below `GENERIC` word indicates that you can use that function with any data type at the indicated argument; it is not a Fortran90 keyword. The actual module source defines all possible combinations of type and dimensionality for such subroutines.

```
subroutine adios_init (config, comm, err)
    character(*),   intent(in)  :: config
    integer,        intent(in)  :: comm
    integer,        intent(out) :: err
end subroutine

subroutine adios_init_noxml (comm, err)
    integer,        intent(in)  :: comm
    integer,        intent(out) :: err
end subroutine

subroutine adios_finalize (mype, err)
    integer,        intent(in)  :: mype
    integer,        intent(out) :: err
end subroutine

subroutine adios_open (fd, group_name, filename, mode, comm, err)
    integer*8,      intent(out) :: fd
    character(*),   intent(in)  :: group_name
    character(*),   intent(in)  :: filename
    character(*),   intent(in)  :: mode
    integer,        intent(in)  :: comm
    integer,        intent(out) :: err
```

```
end subroutine

subroutine adios_group_size (fd, data_size, total_size, err)
    integer*8,      intent(out) :: fd
    integer*8,      intent(in)  :: data_size
    integer*8,      intent(in)  :: total_size
    integer,        intent(out) :: err
end subroutine

subroutine adios_write (fd, varname, data, err)
    integer*8,      intent(in)  :: fd
    character(*),   intent(in)  :: varname
    GENERIC,        intent(in)  :: data
    integer,        intent(in)  :: err
end subroutine

subroutine adios_read (fd, varname, buffer, buffer_size, err)
    integer*8,      intent(in)  :: fd
    character(*),   intent(in)  :: varname
    GENERIC,        intent(out) :: buffer
    integer*8,      intent(in)  :: buffer_size
    integer,        intent(in)  :: err
end subroutine

subroutine adios_set_path (fd, path, err)
    integer*8,      intent(in)  :: fd
    character(*),   intent(in)  :: path
    integer,        intent(out) :: err
end subroutine

subroutine adios_set_path_var (fd, path, varname, err)
    integer*8,      intent(in)  :: fd
    character(*),   intent(in)  :: path
    character(*),   intent(in)  :: varname
    integer,        intent(out) :: err
end subroutine

subroutine adios_end_iteration (err)
    integer,        intent(out) :: err
end subroutine

subroutine adios_start_calculation (err)
    integer,        intent(out) :: err
end subroutine

subroutine adios_stop_calculation (err)
    integer,        intent(out) :: err
end subroutine

subroutine adios_close (fd, err)
    integer*8,      intent(in)  :: fd
    integer,        intent(out) :: err
end subroutine

!
```

```
! No-XML calls
!
subroutine adios_declare_group (id, groupname, time_index, stats_flag, err)
    integer*8,       intent(out) :: id
    character(*),    intent(in)  :: groupname
    character(*),    intent(in)  :: time_index
    integer,         intent(in)  :: stats_flag
    integer,         intent(out) :: err
end subroutine

subroutine adios_define_var (group_id, varname, path, vartype, dimensions, global_dimens
    integer*8,       intent(in)  :: group_id
    character(*),    intent(in)  :: varname
    character(*),    intent(in)  :: path
    integer,         intent(in)  :: vartype
    character(*),    intent(in)  :: dimensions
    character(*),    intent(in)  :: global_dimensions
    character(*),    intent(in)  :: local_offsets
    integer*8,       intent(out) :: id
end subroutine

subroutine adios_define_attribute (group_id, attrname, path, attrtype, value, varname, e
    integer*8,       intent(in)  :: group_id
    character(*),    intent(in)  :: attrname
    character(*),    intent(in)  :: path
    integer,         intent(in)  :: attrtype
    character(*),    intent(in)  :: value
    character(*),    intent(in)  :: varname
    integer,         intent(out) :: err
end subroutine

subroutine adios_select_method (group_id, method, parameters, base_path, err)
    integer*8,       intent(in)  :: group_id
    character(*),    intent(in)  :: method
    character(*),    intent(in)  :: parameters
    character(*),    intent(in)  :: base_path
    integer,         intent(out) :: err
end subroutine

subroutine adios_allocate_buffer (sizeMB, err)
    integer,         intent(in)  :: sizeMB
    integer,         intent(out) :: err
end subroutine
```

3.2.1 Create the first ADIOS program

Listing 3.1 is a programming example that illustrates how to write a double-precision array `t` of 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 Chapter 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;
    int NX = 10;
    double t [NX];
    // ADIOS variables declaration int64_t handle;
    uint_64 group_size, 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", comm);
    adios_open (&handle, "temperature", "data.bp", "w",comm);
    group_size = sizeof(int) \        // int NX
               + sizeof(double) * NX; // double array t
    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;
}
```

Listing 3.1: ADIOS programming example.

Chapter 4

ADIOS No-XML Write API

ADIOS provides an option of writing data without loading an XML configuration file. This set of APIs is designed to cater to output data , which is not definable from the start of the simulation; such as an adaptive code. Using the no-XML API allows users to change their IO setup at runtime in a dynamic fashion. This section discusses the details of no-XML write API's and demonstrates how they can be used in a program.

4.1 No-XML Write API Description

This section lists routines that are needed for ADIOS no-XML functionalities. These routines prepare ADIOS metadata construction, for example, setting up groups, variables, attributes and IO transport method, and hence must be called before any other ADIOS I/O operations, i.e., adios_open, adios_group_size, adios_write, adios_close. A common practice of using no-XML API is to first initialize ADIOS by calling adios_init_noxml and call adios_allocate_buffer to allocate the necessary buffer for ADIOS to achieve best performance. Subsequently, declare a group via adios_declare_group, and then adios_define_var needs to be repetitively called to define every variable for the group. In the end, adios_select_method needs to be called to choose a specific transport method.

adios_init_noxml — initialize no-XML ADIOS
adios_allocate_buffer — specify ADIOS buffer allocation strategy and buffer size in MB
adios_declare_group — declare an ADIOS group
adios_define_var — define an ADIOS variable for an ADIOS group
adios_define_attribute — define an ADIOS attribute for an ADIOS group
adios_write_byid — write a variable, identified by the ID returned by adios_define_var, instead of by name
adios_select_method — associate an ADIOS transport method, such as MPI, POSIX method with a particular ADIOS group. The transport methods that are supported can be found in Chapter 6.

4.1.1 adios_init_noxml

As opposed to adios_init(), adios_init_noxml initializes ADIOS without loading and XML configuration file. Note that adios_init_noxml is required to be called only once and before any other ADIOS calls.

```
int adios_init_noxml (MPI_Comm comm)
```

Input:

- MPI communicator. All processes that uses ADIOS for writing data must be included in the group of this communicator.

Fortran example:

```
call adios_init_noxml (comm, ierr)
```

4.1.2 adios_allocate_buffer

The adios_allocate_buffer routine allocates a memory buffer for ADIOS to buffer all writes before writing all data at once.

```
int adios_allocate_buffer (
    enum ADIOS_BUFFER_ALLOC_WHEN adios_buffer_alloc_when,
    uint64_t buffer_size)
```

Input:

- adios_buffer_alloc_when - indicates when ADIOS buffer should be allocated. The value can be either ADIOS_BUFFER_ALLOC_NOW or ADIOS_BUFFER_ALLOC_LATER. See Section 5.5 for more details on ADIOS buffer.
- buffer_size - the size of ADIOS buffer in MB.

Fortran example:

```
call adios_allocate_buffer (sizeMB, ierr)
```

Note that, as opposed to the C function, the Fortran subroutine doesn't have adios_buffer_alloc_when argument as it supports only the ADIOS_BUFFER_ALLOC_NOW option.

4.1.3 adios_declare_group

This function is used to declare a new ADIOS group. The concept of ADIOS group, variable, attribute is detailed in Chapter 5.

```
int adios_declare_group (int64_t * id,
                         const char * name,
                         const char * time_index,
                         enum ADIOS_FLAG stats)
```

Input:

- name - string containing the annotation name of the group
- time_index - string containing the name of time attribute. If there is no time attribute, an empty string ("") should be passed
- stats - a flag indicating whether or not to generate ADIOS statistics during writing, such as min/max/-standard deviation. The value of *stats* can be either `adios_flag_yes` or `adios_flag_no`. If stats is set to `adios_flag_yes`, ADIOS internally calculates and outputs statistics for each processor automatically. The downside of turning stats on is that it consumes more CPU and memory during writing and the metadata will be larger.

Output:

- id - pointer to the ADIOS group structure

Fortran example:

```
call adios_declare_group (m_adios_group, "restart", "iter", 1, ierr)
```

4.1.4 adios_define_var

This API is used to declare an ADIOS variable for a particular group.

```
int64_t adios_define_var (int64_t group_id,
                          const char * name,
                          const char * path,
                          enum ADIOS_DATATYPES type,
                          const char * dimensions,
                          const char * global_dimensions,
                          const char * local_offsets)
```

Input:

- group_id - pointer to the internal group structure (returned by adios_declare_group call)
- name - string containing the annotation name of a variable
- path - string containing the path of an variable
- type - variable type (e.g., adios_integer or adios_double)
- dimensions - string containing variable local dimension. If the variable is a scalar, an empty string ("") is expected. See Section 5.3.2 for details on variable local dimensions.
- global_dimensions - string containing variable global dimension. If the variable is a scalar or local array, an empty string ("") is expected. See Section 5.3.5 for details on global dimensions.
- local_offsets - string containing variable local offset. If the variable is a scalar or local array, an empty string ("") is expected.

Return value:
A 64bit ID of the definition that can be used when writing multiple sub-blocks of the same variable within one process within one output step.
Fortran example:

```
call adios_define_var (m_adios_group, "temperature", "", 6, &
                       "NX", "G", "O", varid)
```

4.1.5 adios_write_byid

`adios\_write()` finds the definition of a variable by its name. If you write a variable multiple times in an output step, you must define it as many times as you write it and use the returned IDs in `adios_write_byid()` to identify what you are writing.

```
int adios_write_byid (int64_t fd_p, int64_t id, void * var)
```

Input:

- fd_p—pointer to the internal file structure
- id—id returned by the corresponding adios_define_var() call
- var —the address of the data element defined need to be written

Fortran example:

```
call adios_write_byid (handle, id, v, ierr)
```

4.1.6 adios_define_attribute

This API is used to declare an ADIOS attribute for a particular group. See section 5.2.3 for more details on ADIOS attribute.

```
int adios_define_attribute (int64_t group,
                            const char * name,
                            const char * path,
                            enum ADIOS_DATATYPES type,
                            const char * value,
                            const char * var)
```

Input:

- group - pointer to the internal group structure (returned by adios_declare_group)

- name - string containing the annotation name of an attribute
- path - string containing the path of an attribute
- type - type of an attribute
- value - pointer to a memory buffer that contains the value of the attribute
- var - name of the variable which contains the attribute value. This argument needs to be set if argument `value` is null.

Output:

- None

Fortran example:

```
call adios_define_attribute (m_adios_group, "date", "", 9, &
                             "Feb 2010", "" , ierr)
```

4.1.7 adios_select_method

This API is used to choose an ADIOS transport method for a particular group.

```
int adios_select_method (int64_t group,
                         const char * method,
                         const char * parameters,
                         const char * base_path)
```

Input:

- group - pointer to the internal group structure (returned by adios_declare_group call)
- method - string containing the name of transport method that will be invoked during ADIOS write. The list of currently supported ADIOS methods can be found in Chapter 6.
- parameters - string containing user defined parameters that are fed into transport method. For example, in MPI_AMR method, the number of subfiles to write can be set via this argument (see section 6.1.5). This argument will be ignored silently if a transport method doesn't support the given parameters.
- base_path - string specifing the root directory to use when writing to disk. By default, methods open files with relative paths relative to the current directory, but base_path can be used to change this behavior.

Fortran example:

```
call adios_select_method (m_adios_group, "MPI", "", "", ierr)
```

4.2 Create a no-XML ADIOS program

Below is a programming example that illustrates how to write a double-precision array t and a double-precision array with size of NX using no-XML API. A more advanced example on writing out data sub-blocks is listed in the appendix Section 13.3.

```
program adios_global
    use adios_write_mod
    implicit none
    include "mpif.h"
    character(len=256) :: filename = "adios_global_no_xml.bp"
    integer :: rank, size, i, ierr
    integer,parameter :: NX=10
```

```
integer :: O, G
real*8, dimension(NX) :: t
integer :: comm
integer :: ierr
integer*8 :: adios_groupsize, adios_totalsize
integer*8 :: adios_handle
integer*8 :: m_adios_group
integer*8 :: varid ! dummy variable definition ID

call MPI_Init (ierr)
call MPI_Comm_dup (MPI_COMM_WORLD, comm, ierr)
call MPI_Comm_rank (comm, rank, ierr)
call MPI_Comm_size (comm, size, ierr)
call adios_init_noxml (comm, ierr)
call adios_allocate_buffer (10, ierr)
call adios_declare_group (m_adios_group, "restart", "iter", 1, ierr)
call adios_select_method (m_adios_group, "MPI", "", "", ierr)

!
! Define output variables
!

! define integer scalars for dimensions and offsets
call adios_define_var (m_adios_group, "NX","", 2, &
                       "", "", "", varid)
call adios_define_var (m_adios_group, "G", "", 2 &
                       "", "", "", varid)
call adios_define_var (m_adios_group, "O", "", 2 &
                       "", "", "", varid)

! define a global array
call adios_define_var (m_adios_group, "temperature", "", 6 &
                       "NX", "G", "O", varid)

!
! Write data
!
call adios_open (adios_handle, "restart", filename, "w", comm, ierr)

adios_groupsize = 4 + 4 + 4 + NX * 8
call adios_group_size (adios_handle, adios_groupsize, &
                       adios_totalsize, ierr)
G = NX * size
O = NX * rank
do i = 1, NX
    t(i) = rank * NX + i - 1
enddo

call adios_write (adios_handle, "NX", NX, ierr)
call adios_write (adios_handle, "G", G, ierr)
call adios_write (adios_handle, "O", O, ierr)
call adios_write (adios_handle, "temperature", t, ierr)
call adios_close (adios_handle, ierr)

call MPI_Barrier (comm, ierr)
```

```
    call adios_finalize (rank, ierr)
    call MPI_Finalize (ierr)
end program
```

Listing 4.1: ADIOS no-XML example

Chapter 5

XML Config File Format

5.1 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. Listing 5.1 llustrates 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=value1 attr2=value2 ...>
```

```
<?xml version="1.0"?>
<adios-config>
    <adios-group>
        <var ... />
        <attribute .../>
    </adios-group>
    <method ... />
    <buffer ... />
</adios-config>
```

Listing 5.1: Example XML configuration

5.2 adios-config

The adios-config element is the container for all ADIOS elements, and thus practically all configuration file has one of these elements that contain everything. Multiple elements are allowed, however, there is no know use for that.

The only attribute that adios-config has is the `host-language` attribute for language declaration. Fortran or C should be chosen, according to the source language that is going to use ADIOS. The only difference that

it makes is for the use of `time-index` in a group (see `adios_group` below) when multiple output steps are appended to the same file. In Fortran, the time index, as dimension declaration, should be the last dimension (the slowest one), while in C, it should be the first one (again, the slowest dimension).

```
<adios-config host-language="Fortran">
    ...
</adios-config>
```

5.3 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 the 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.

5.3.1 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

5.3.2 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.

5.3.2.1 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"/>
```

5.3.2.2 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 `gpp.py` python script as the variable name in the source code to generate adios_write routines; the default value is the value of the attribute *name* if gwrite is not defined. Use it if the write code is automatically generated and the desired variable name in the output is different from the name that contains the data in the program. The value is substituted 'as is' into the generated code, so arbitrary Fortran/C expressions can be used.
- gread - the value will be used in the python scripts to generate adios_read routines' the default value is the value of the 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.

5.3.3 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. ADIOS supports only scalar attributes, i.e, no arrays or vectors are supported. Note that a string is considered a scalar in ADIOS. From ADIOS 1.4, only the root process of the application writes the attribute into the output, therefore, process-dependent information cannot be save as an attribute (by definition, that is a variable information, which should be stored in a variable).

5.3.3.1 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.

5.3.3.2 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*

5.3.4 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 Chapter 10). Any content (usually comments, conditional statements, or loop statements) in the value of attribute "src" is copied identically into generated pre-processing files. This is the way to write a subset of variables optionally depending on a logical expression in the source-code. Declaration

```
<gwrite src="if (have_ions==1) then"/>
    ...
<gwrite src="endif"/>
```

Required:

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

5.3.5 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.

5.3.5.1 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 12.8.

5.3.6 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. Note, that with the new read API, time is not represented as an extra dimension at reading. If you write a 2D variable over multiple steps, you will see it still as a 2D variable, which has multiple steps. Nevertheless, multiple, consecutive steps can be read at once into a contigous memory with one read request, if needed.

5.4 Transport method

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.

5.4.1 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. Either the term `transport` or `method` can be used for this element. There are two major attributes required for the method element:

```
<transport group="restart"
    method="MPI"
    priority="1"
    base-path="/proj/proj034/simdata/run256"
    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

5.4.2 Methods list

As the componentization of the IO substrate, ADIOS supports a list of transport methods, described in Chapter 6:

- NULL
- POSIX
- MPI
- MPI_LUSTRE
- MPI_AGGREGATE (or MPI_AMR)
- VAR_MERGE
- PHDF5 (for Parallel HDF5)
- NC4PAR (for Parallel NetCDF4)
- DATASPACES (or DART)
- DIMES
- FLEXPATH

5.5 Buffer specification

The buffer element defines the attributes for internal buffer size and creating time that apply to the whole application (Listing 5.2). 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.

5.5.1 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 free memory available on the machine. It is exclusive with size-MB.
- allocate-time - indicates when the buffer should be allocated

5.6 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.

5.6.1 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.

5.7 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"/>

        <!-- conditional writing of a variable -->
        <gwrite src="if (want_x) {" />
            <var name="x" type="integer" dimensions="NX"/>
        <gwrite src="}" />
    </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>
```

Listing 5.2: Example XML file.

Chapter 6

Transport Methods

As described perviously, ADIOS provides a framework for the development and deployment of new techniques for data movement through the use of *Transport Methods*. While the development of transport is not in the scope of this manual, the user is given the option to easily select a specific method for the needs of an application at run time.

ADIOS includes two broad classes of transport method, viz. mainline methods and experimental methods. Mainline methods are supported methods that provide both high performance, reliability and usability. Experimental methods are under-development research techniques to explore new I/O techniques. While we encourage our users to experiment with all available methods, no explicit support is provided for these research methods.

6.1 Mainline Transport Methods

6.1.1 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.

6.1.2 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.

6.1.3 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 6.1 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.

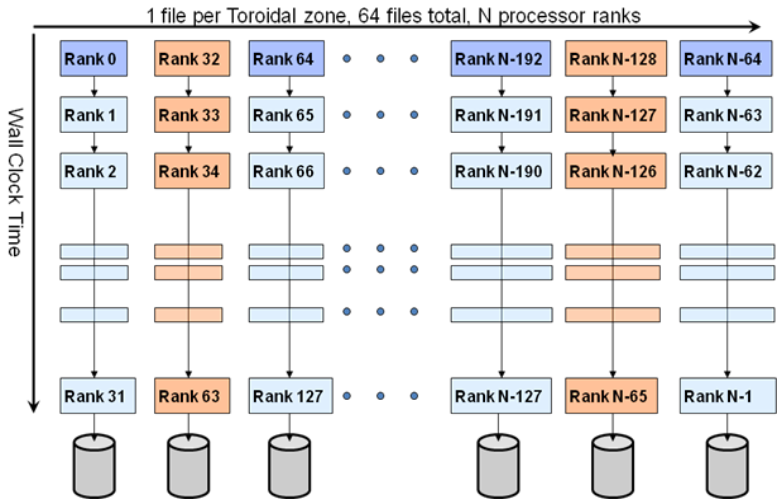

Figure 6.1: Server-friendly metadata approach: offset the create/open in time

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.

6.1.4 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.

Note in 1.3 and later releases, with Lustreapi option enabled in configuration, MPI_LUSTRE sets the parameters automatically and therefore parameters in XML are not required. The method automatically calculates the data size from each processor and sets the proper striping parameters.

6.1.5 MPI_AGGREGATE

The MPI_AGGREGATE method is designed to maximize write performance for large scale applications (more than 10,000 cores) that write out data from a large subset of processors. Based upon MPI_LUSTRE method, MPI_AGGREGATE 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_AGGREGATE method in ADIOS.
There are two key parameters used in this method.

```
<method group="tracers" method="MPI_AGGREGATE">
        num_aggregators=24;num_ost=672
</method>
```

- **num_aggregators** specifies the number of aggregators to use.
- **num_ost** specifies the number of Lustre storage targets available in the file system. Note this parameter is mandatory if "—with-lustre" option is not given during ADIOS configuration.

For example, if you have an 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.

The MPI_AGGREGATE method allocates stand-alone internal buffers for aggregating data. As opposed to ADIOS buffer (the size of which is set from XML file), these buffers are allocated separately and the total

size (on one processor) is twice the ADIOS group size. User needs to make sure each process has enough memory when using this method.

Note that in 1.3 and later releases, with Lustreapi option enabled in configuration, MPI_AGGREGATE sets the parameters automatically and therefore parameters in XML are not required. The method automatically calculates the data size from each processor and sets the proper aggregation parameters. Also note that in previous versions of ADIOS (before 1.4), the MPI_AGGREGATE method was refered to as the MPI_AMR method.

6.1.6 VAR_MERGE

The VAR_MREGE method is designed to extends the capability of current ADIOS methods to address the I/O challenges for applications that write out small variables at scale. During data output, VAR_MERGE merges the small variable blocks from intensive chunking into larger data chunks with their spatial localities are reserved. The benefits are: 1) less contention at storage side during data output; 2) frequent read and seek operations can be avoided during reading. An example of variable merging is given in Figure 6.2.

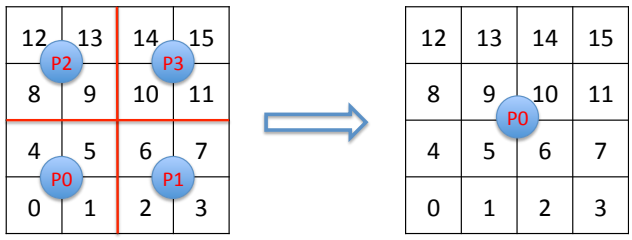

Figure 6.2: Spatial Aggregation of a 4x4 2D Variable From 4 proceeses to Process 0

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

```
<method group="test" method="VAR_MERGE"> chunk_size=2097152; io_method=MPI_AGGREGATE;
    io_parameters=num_aggregators=24,num_ost=672
</method>
```

- **chunk_size** specifies the chunk size after merging. If merging current data chunk leads to a chunk size that is larger than this value, merging will not be performed. The default value is 2MB without specification in XML.
- **io_method** the underlying I/O method for data output. The ADIOS transport method that VAR_MERGE can work with include POSIX, MPI, MPI_LUSTRE and MPI_AGGREGATE.
- **io_parameters** the parameters required for using the io_method. Note that the paramters needs to be separated by ",".

For example, if you want to merge data chunks into 4MB chunks and use MPI_LUSTRE method to write out data, your XML looks like:

```
<method group="test" method="VAR_MERGE"> chunk_size=4194304,
    io_method=MPI_LUSTRE, io_parameters=stripe_count=128,stripe_size=4194304
</method>
```

Here we specified the data to be placed on 128 storage targes, and the stripe size is 4MB. Note that current VAR_MERGE method only supports 1D, 2D and 3D variables. The maximum level of aggregation is 2 due to the consideration of merging overhead.

6.1.7 Dataspaces

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

The dataspaces clients use a light-weight library that provides the asynchronous I/O API to be used by ADIOS. It is integrated with the ADIOS layer and the functionality is exposed through the ADIOS write/read semantics. The ADIOS layer is used to collect and encode the data written by the application into a local transport buffer. Once it has collected data from an application, the transport method notifies the dataspaces_server through a coordination channel that it has data available to send out. At this point, the control is transferred back to the application, while the data is asynchronously extracted by the dataspaces_server.

The dataspaces_server 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 application through RDMA, and can save it to local storage system, e.g., the Lustre file system, stream it to remote sites, e.g., auxilliary clusters, or serve it directly from the staging area to other applications. One instance of the dataspaces_server can service multiple applications in parallel. Further, the server can run in cooperative mode (i.e., multiple instances of the server cooperate to service the application in parallel and to balance load). The dataspaces_server receives notification messages from the transport method, schedules the requests, and initiates the data transfers 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 application.

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

```
<method group="fluxdiag" method="Dataspaces"/>
```

Listing 6.1: Select Dataspaces as a transport method in the configuration file example.

To make use of the Dataspaces transport, an application job needs to also run the dataspaces_server 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 ./dataspaces_server -s $SPROC -c $PROC &> log.server &
```

Listing 6.2: 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 application to connect to the servers. This file contains identifying information 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 before starting the client application processes. Once ADIOS is initialized in the application, this configuration file is parsed to provide the rendevouz information.

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


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

Listing 6.3: Wait for server start-up completion and export the configuration to environment variables.

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

6.1.8 DIMES

DIMES is another asynchronous I/O transfer method within ADIOS that enables low- latency, scalable data extraction from a running simulation. It is designed to enable M×N parallel data redistribution between tightly coupled applications, which essentially performs RDMA-based memory-to-memory transfers of distributed data such as global arrays from M processes of one application to N processes of another application.

Both DIMES and Dataspaces methods are implemented based upon the open source software library DataSpaces. However, unlike Dataspaces method which implements memory-to-memory data sharing using dedicated server nodes in the *staging area*, DIMES method implements the parallel data redistribution through point-to-point transfers directly between processes of data writing and reading applications, and thus bypass the *staging area*.

For applications using the ADIOS DIMES method, the new functionality is exposed through the ADIOS write/read semantics. ADIOS layer is used to collect data written by the application into a local RDMA memory buffer as data objects that can be remotely accessed through the underlying RDMA get operations. Once data is successfully copied from the application space, the control is transferred back to the application, then the simulation data can be asynchronously extracted and redistributed to the coupled reading applications.

Though the DIMES methods does not need simulation data to be transferred to the *staging area*, a stand-alone server module is still required for constructing a distributed directory service to keep track of the in-memory data objects. Internally, client applications would query the service to locate required data objects, and then perform the subsequent RDMA-based parallel data transfers.

DIMES method can be selected by specifying the transport method in the external ADIOS XML configuration file (as below).

```
<method group="fluxdiag" method="DIMES"/>
```

Listing 6.4: Select DIMES as a transport method in the configuration file example.

The dataspaces_server component is extended to support the data locating service mentioned above. To make use of the DIMES method, a job needs to also run the dataspaces_server with the applications. Each dataspaces_server instance needs to run on one dedicated processor core, and multiple sever instances can be started in parallel to support the data locating service for larger scale applications. Instructions on how to configure and run the dataspaces_server can be found in section 6.1.7.

6.1.9 Flexpath

Flexpath is an asynchronous data transport method built to ensure scalable I/O through the use of staging areas. It is built on top of the EVPath event-driven messaging library `http://www.cc.gatech.edu/systems/projects/EVPath/`, which allows for the creation of arbitrary network overlays. Data sent through EVPath is serialized with the Fast, Flexible Serialization (FFS) library, and also allows for additional metadata to be appended to the datastream through the use of separately addressable attributes. To provide high-performance, Flexpath takes advantage of EVPath features such as *multiqueue stones* and *C-on-Demand (COD)* dynamic code generation.

Flexpath also works on top of several popular high-end network interfaces, such as Infiniband, Portals, and Gemini. See the intallation section for information on how to use this functionality.

6.1.10 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. This requires that every process participates in the writing of each variable.

Note: Do not expect better performance with ADIOS/PHDF5 than with PHDF5 itself. ADIOS does not write differently to a HDF5 formatted file, it simply uses PHDF5 function calls to write out data.

6.1.11 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>
```

Listing 6.5: Example XML

```
char path[1024];
adios_init ("config.xml", comm);
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);
```

Listing 6.6: 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".

6.2 Research 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 the Developer's manual. 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.

6.2.1 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>
```

Listing 6.7: Example Original Client XML

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

Listing 6.8: Example NSSI Client XML

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

Listing 6.9: 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
```

Listing 6.10: Example PBS script with NSSI Staging Service

Listing 6.10 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.

6.2.2 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 6.3).

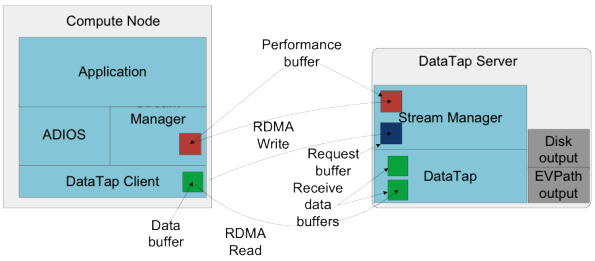

Figure 6.3: DataTap architecture

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.

6.2.3 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.

6.2.4 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 6.1.

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 1.4, 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 1.4.

Chapter 7

ADIOS Read API

7.1 Introduction

The second version of the ADIOS Read API (introduced in ADIOS 1.4) is designed to handle both files on disk and data sets in memory staging areas. Non-blocking and chunking read is introduced to enable concurrent processing of some part of the requested data while other parts are being retrieved for methods that support it. A Selection interface is introduced to define subsets of datasets other than a simple bounding box.

7.1.1 Changes from version 1

The original version of ADIOS Read API (in ADIOS releases 1.0–1.3.1) provides a (1) grouped view of variables in an ADIOS-BP file with (2) time as an extra dimension. ADIOS applications can write multiple, separate groups of variables into one file. They can also write multiple steps of a group into one file. When opening a file for reading, each group is conceptually separated and thus has to be opened separately. Also, an N–dimensional variable with multiple steps is presented as an N+1–dimensional variable, with time represented as the slowest dimension (like in NetCDF). These two representations have been eliminated in the new API, both for the sake of supporting streams with the same API as files and thus enable the transition from file-based analytics and visualization to staging environments where data sets are passed around without touching permanent storage.

In the new API, all variables of all groups are presented at once when opening the file. Nevertheless, some extra functions are provided in the API to get the list of groups written into the file (`adios_get_grouplist()`) and to restrict the view of variables and attributes to a certain group (`adios_group_view()`) in case some application would need this artificial separation.

Time is completely eliminated as a concept from the read API. Now one sees the output steps as they are; just steps. An N–dimensional variable written M times into the same file is represented as an N–dimensional variable with M steps available to read. From permanently stored datasets (files), the user can still read multiple steps at once and store in the user-provided contigous memory but with arguments separate from the spatial dimension specification.

7.1.2 Concepts

- *Reader* is an application reading data using ADIOS
- *Writer* is an application writing output data using ADIOS
- *Reading methods* are different reading codes in ADIOS from which a Reader can choose one, e.g. for reading from a file, or from the memory of the Writer.

7.1.2.1 Staging

Staging here means that data is in memory somewhere and an ADIOS method presents that data to a reader as a stream. That is, it has to be opened, its content can be discovered, it contains variables and attributes,

and it has to be closed afterwards. Different staging scenarios exist and are under development by various teams:

- A staging server with its own compute nodes and memory, stores the output of writing applications, allows connections from several applications and serves read requests.
- A staging library embedded in the reading application, connects directly to a writing application and pulls the requested data out of the writer's memory.
- A staging library like the above with the specialization that both writer and reader occupies the same (multi-core) compute node and shares memory.

ADIOS 1.4 comes with the DataSpaces method that is an implementation of the first scenario. Methods for the other two scenarios are coming in the near future.

7.1.2.2 Streams and Steps

Simulations usually write the same data set regularly over time, so a file or a series of files contains the same set of variables written many times. The dataset written between one adios_open and adios_close calls is called *STEP* and not "time" or "timesteps" to avoid confusing users about what time actually means. A *STREAM* differs from a file containing multiple steps only in the handling of steps. In a file on disk, all steps are available at all times to a reader. In a stream, only one step is available at a time to read, and as newer steps are becoming available from the writer, older steps may disappear. The ADIOS Read API provides two different open functions for streams and files. Nevertheless, a file can be handled in an application as a stream, i.e., by reading one step at a time, processing it, then advancing to the next step. Users are encouraged to write their code with streaming in mind, so that their file processing code can be used in a staging environment without code modification.

Note: A stream in ADIOS is **not** a byte stream! The unit of the stream is one output Step of the Writer, so it still can be much larger than the available memory in the Reader.

In case of opening a file in file mode, each variable may have a different numberof steps, which is known at the time of opening the file. The number of steps value can be inquired for each variable separately (this value is always 1 for all variables in a stream). Then the application can read some steps of a variable at once, which may be different from the "global steps" written into the file. E.g., if a variable is written at every other output steps (1,3,5,...,n), then it will have half as many steps as the file itself has but its steps are addressed as 0,1,2...$\lceil n/2 \rceil - 1$.

In case of streams a step is the feature of the stream, not of the individual variables. There is no individual counting of the variables so the application has to count them itself if needed.

7.1.2.3 Locking strategies

Locking is not used for files, but in a staging environment there are different strategies to deal with disappearing steps of datasets. A daredevil reader may tell the API to not block the writing application at all (*ADIOS_LOCKMODE_NONE*), i.e., to allow for loosing the opened data set any time if space is needed for storing newer steps of the writer's output. A safer way to handle complete steps is to lock the currently opened step (*ADIOS_LOCKMODE_CURRENT*). If the writer has more output in the meantime and there is not enough space for staging, an earlier or even a more recent step can be removed by the staging method. To ensure correct execution of rigid readers, current and all more recent steps should be locked so that they can be read one-by-one without loosing them (*ADIOS_LOCKMODE_ALL*). This strategy, however, can certainly block the writing application if it has to wait for some available space to become free in the staging area.

Note, however, that specific staging methods may not support all locking mechanisms, or the actual locking mechanism would depend on their configuration and runtime set-up. E.g., the DataSpaces method in ADIOS 1.5.1 only supports the *ADIOS_LOCKMODE_CURRENT* locking strategy, although the DataSpaces server can also be started up with a custom locking that enforces synchronized steps of alternating writes and reads. The three locking options are kept in the read API with the hope that some staging methods would support all of them in the future.

7.1.2.4 Chunks

Chunking allows for processing large data in small pieces concurrently while other pieces are being transferred.

Note: In ADIOS 1.5.0, chunking is implemented only "functionally" in any method; all of them will just read the whole variable at once.

A read request of a variable for one step can be served in multiple pieces. The reader should be able to receive parts of the whole requested dataset and process them one by one, instead of expecting the whole data set arriving in one piece, ordered in memory. We call these pieces or parts returned by the reading method to the reader as *chunks*. In ADIOS, chunks are closely related to the individual outputs of writing processes. Therefore, readers should expect to receive one chunk per writer whose output partly matches the query.

7.1.3 Selections

Selection is some subset of a variable. ADIOS reading methods support

- Contiguous *Bounding Box*es (compact ranges in all dimensions of a variable),
- A set of individual *Points*
- Individual selection of a block written by one writer process
- "Auto" selection for a special case of asking for locally available data in in-situ staging frameworks

Note that the bounding box was supported in the ADIOS Read API v1 implicitly, as extra arguments in the adios_read_var calls, and individual blocks were accessible with the special function adios_read_local_var.

All reading methods understand and can serve read requests over such selections.

7.2 How to use the read functions

First, before opening a file/stream, we have to choose a reading method and initialize it with the `adios_read_init_method()`. Also, an `adios_read_finalize_method()` is necessary at the end of the application run. Note, that there is a separate initialization call for each read method the application intends to use.

A file has to be opened with `adios_read_open_file(fname, method, comm)` if the application wants to handle it as file (all steps accessible at once). A name, a read method and an MPI communicator should be provided. A stream (or a file handled as a stream) has to be opened with `adios_read_open(fname, method, comm, lock_mode, timeout_msec)`. A locking strategy has to be specified and some timeout can be specified for waiting for the stream to appear.

In C, a transparent data struct is returned (`ADIOS_FILE`), which enumerates the list of variables and attributes, the current step and the available steps in the staging memory at the time of opening. The available steps are updated whenever seeking with `adios_advance_step()`. Seeking is allowed to the next available step or to the last (newest) available step, with the possible errors of not finding any new step or finding that the stream has terminated. The current step can be released without advancing the step too, to free resources in the staging area. This optimization call is highly encouraged in every application to give free space to the writing application as early as possible.

To read a subset of a variable, a selection object has to be created. The selection is independent from the variable (e.g. a bounding box) and from the opened file/stream, so it can be reused for reading many similar variables.

When reading data, several read operations are first scheduled (`adios_schedule_read_var()`), then `adios_perform_reads()` is called to start/do the actual reading. In blocking mode, this function returns when all reading has been finished and all result is stored in the user provided memory (provided separately for each variable in the schedule step).

In non-blocking mode, this function returns as soon as possible and the application has to check for variables becoming available with `adios_check_reads()`. This function returns zero or one "chunk". If memory was provided to a variable in the schedule call, a single chunk will eventually be returned here that describes the whole variable. If memory was not provided, the chunking read mode should use an internal

buffer in ADIOS to store and return a partial result. It depends on the size of the internal buffer and the number of writers of the requested piece that how many chunks will be needed for each scheduled read. Each chunk is a contiguous subset of the requested variable, but it is the application's own business how to reorganize the chunks into the complete request. This function returns one chunk at a time, which should be processed before calling this function again. It should be called repeatedly until the function tells the application that all reads have been completed.

Reading is concluded with closing the file/stream with `adios_close()` and deleting the ADIOS objects created to schedule the reading (selections with `adios_selection_delete()` and inquired structures with `adios_free_varinfo()`).

7.3 Notes

Dimensions of arrays are reported differently for C and Fortran. When reading from a different language than writing (Fortran vs. C), 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.

Metadata rich footer enables fast information retrieval. Since the BP file format is metadata rich, and the metadata is immediately accessible in the footer of the file, we can get a lot of information without accessing the file again after the open call. The open function returns the list of variables and attributes. Type and dimensionality as well as the actual value of a scalar variable is returned by adios_inq_var. Another inquiry extends the information with statistics (minimum, maximum, average and standard deviation globally and for each writer process separately). Similarly, another inquiry extends the information with dimensionality for each writer process (i.e. the detailed decomposition of a variable).

Steps start from 0, even in Fortran applications (just because ADIOS is written in C, where everything starts from 0).

7.4 Read C API description

Please consult the `adios_read_v2.h` for the data structures and functions discussed here. In the source code, do not include this header file directly, but `adios_read.h`. The sequence of reading in a variable from the BP file is

- initialize the reading method (once per program run)
- open file/stream
- inquiry the variables to get type and dimensions
- allocate memory for the variables
- create a selection object for each variable (reusable for similar subsets)
- schedule reads for all variables (whole or part of it)
- perform the reads
- free varinfo data structure
- close group
- close file
- finalize the read method (once per program run)

Example codes using the C API are

- examples/C/global-array/adios_read_global
- tests/suite/programs/write_read.c

7.4.1 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 is this adios_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. Moreover, only the last error message is available at any time.

7.4.2 adios_read_init_method

Initialize a reading method before opening a file/stream with using the method. Staging methods perform the connection/disconnection to the staging server once during init/finalize.

- **method** Read method to use.
- **comm** MPI communicator of all processes participating in a file/stream operation
- **parameters** A series of name=value pairs separated by ";". E.g. "max_chunk_size=200; app_id = 1". List of parameters is documented for each method separately.

The function returns 0 on success, < 0 on an error.

The methods supported in ADIOS 1.5.0 are

- **ADIOS_READ_METHOD_BP** Read from ADIOS BP file. Every reading process will access the file(s) to serve its own reading needs.
- **ADIOS_READ_METHOD_BP_AGGREGATE** Read from ADIOS BP file. Only the aggregators will access the file(s) to serve all reading requests. They gather the scheduled reads from all reader processes, optimize the read operations and then distribute the requested data to all readers. Specify the number of aggregators by adding `"num_aggregators=<N>"` to the parameters of this function call.
- **ADIOS_READ_METHOD_DATASPACES** Read from staging memory using DataSpaces. The writer applications must use the DATASPACES transport method when writing. See Section 6.1.7 for details on this method.
- **ADIOS_READ_METHOD_DIMES** Read from the staging memory of another application using DIMES. The writer applications must use the DIMES transport method when writing. See Section 6.1.8 for details on this method.
- **ADIOS_READ_METHOD_FLEXPATH** Read from the staging memory of another application using FLEXPATH. The writer applications must use the FLEXPATH transport method when writing. See Section 6.1.9 for details on this method.

Although each read method has a separate initialization, this function can be also used for some global settings:

- **verbose=<integer>** Set the level of verbosity of ADIOS messages: 0=quiet, 1=errors only, 2= warnings, 3=info, 4=debug
- **quiet** Same as verbose=0
- **logfile=<path>** Redirect all ADIOS messages to a file. in ADIOS 1.5.0, there is no process level separation. Note that third-party libraries used by ADIOS will still print their messages to stdout/stderr.
- **abort_on_error** ADIOS will abort the application whenever ADIOS prints an error message. In ADIOS 1.5.0, there are error messages in some write transport methods that still go to stderr and will not abort the code.

```
int adios_read_init_method (enum ADIOS_READ_METHOD method,
                            MPI_Comm comm,
                            const char * parameters);
```

7.4.3 adios_read_finalize_method

Finalize the selected method. Required for all methods that are initialized.

- **method** Read method to finalize.

```
int adios_read_finalize_method(enum ADIOS_READ_METHOD method);
```

7.4.4 adios_read_open

Open an adios file/stream as a stream. In the returned ADIOS_FILE struct, current_step is the currently opened step, which is the oldest step of the stream still available at the time of open. Only data in this current step can be read. The last_step indicates the newest step, which is available in the staging area. It is only an indicator to the reader about how far ahead the writer is in data production. The number and list of variables in the ADIOS_FILE struct reflects the variables in the current step only. The list will change when advancing the step if the writing application writes different variables at different times.

- **fname** Pathname of file/stream to be opened.
- **method** Read method to use for this particular stream.
- **comm** The MPI communicator of all processes that want to read data from the stream. If compiled with -D_NOMPI, pass any integer here.
- **lock_mode** In case of real streams, a step may need to be locked in memory to be able to read all data of the step completely.
- **timeout_sec** >= 0.0: block until the stream becomes available but for max 'timeout_sec' seconds.
  0.0: return immediately if stream is not available
  < 0.0: block possibly forever.
  Note: < 0.0 does not ever return with err_file_not_found error, which is dangerous if the stream name is simply mistyped in the code.

The function returns a pointer to an ADIOS_FILE struct on success, NULL on error with setting adios_errno. Possible errors (adios_errno values)

- **err_file_not_found_error** File/stream does not exist / not yet available.
- **err_end_of_stream** Stream has ended, nothing is available and no more steps should be expected.

```
ADIOS_FILE * adios_read_open (const char * fname,
                              enum ADIOS_READ_METHOD method,
                              MPI_Comm comm,
                              enum ADIOS_LOCKMODE lock_mode,
                              float timeout_sec);
```

The returned ADIOS_FILE structure includes the following information:

- **int nvars** Number of variables in the file (with full path)
- **char ** var_namelist** Variable names in a char* array
- **int nattrs** Number of attributes in the file

- **char ** attr_namelist** Attribute names in a char* array
- **int current_step** The current step in a stream. For a file, it is always 0.
- **int last_step** The currently available latest step in the stream/file.

7.4.5 adios_read_open_file

Open an adios file as a file. Each variable can have different number of steps. Arbitrary steps of a variable can be read at any time. In the returned ADIOS_FILE struct, current_step is always 0, while last_step is the number of global steps - 1. The list of variables include all variables written in all steps.

- **fname** Pathname of file to be opened.
- **method** Read method to use for this particular file.
- **comm** The MPI communicator of all processes that want to read data from the file. If compiled with -D_NOMPI, pass any integer here or use 'mpidummy.h' provided by the ADIOS installation.

The function returns a pointer to an ADIOS_FILE struct, NULL on error (sets adios_errno).
Possible errors (adios_errno values)

- **err_file_not_found_error** File does not exist.

```
ADIOS_FILE * adios_read_open_file (const char * fname,
                                   enum ADIOS_READ_METHOD method,
                                   MPI_Comm comm);
```

7.4.6 adios_read_close

Close an adios file. It will free the content of the underlying data structures and the fp pointer itself.

- **fp** The pointer of the ADIOS_FILE structure returned by the open function.

The function returns 0 on success, $!=0$ on error (also sets adios_errno).

```
int adios_read_close (ADIOS_FILE *fp);
```

7.4.7 adios_advance_step

Advance the current step of a stream. For files opened as file, stepping has no effect. In case of streams,

1. An error should be expected for any step, since a step might not yet be available
2. One can advance to the next available or to the last (newest) available step only. No steps can be hopped over. Nevertheless, one can use the current step's counter to advance many times to get to a certain step.
3. It depends on the locking method, if advancing to the next step advances to the next immediate step (ADIOS_LOCKMODE_ALL) or to the next available step (ADIOS_LOCKMODE_CURRENT). Still, if the reading method in use does not support locking all steps, advancing to the 'next' step may fail if that step is not available anymore and return an error.
4. Advancing to step N informs the read method that all steps before N can be removed if space is needed. There is no way to go back to previous steps.

Arguments:

- **fp** Pointer to an ADIOS_FILE struct.

- **last** 0: next available step, ! = 0: newest available step
- **timeout_sec** >= 0.0: block until the next step becomes available but for max 'timeout_sec' seconds. 0.0 means return immediately if step is not available.
  < 0.0: block forever if necessary.

The function returns 0 on success, ! = 0 on error (also sets adios_errno). Possible errors (adios_errno values):

- **err_end_of_stream** Stream has ended, no more steps should be expected
- **err_step_notready** The requested step is not yet available.
- **err_step_disappeared** The requested step is not available anymore. This error is possible only if the read method does not support LOCKMODE_ALL, you open the stream with LOCKMODE_ALL, request to advance to next and the immediate step after the currently opened one is not available any more, and the method actually returns the error instead of advancing to the next available step.

```
int adios_advance_step (ADIOS_FILE *fp, int last, float timeout_sec);
```

7.4.8 adios_release_step

Release a step in a stream without seeking to the next step. This function is to inform the read method that the current step is no longer needed, but the reader does not yet want to read another step. This function releases the lock on the step only. The current step is not changed in the ADIOS_FILE struct, but resources are freed and thus ADIOS function calls other than advancing or closing the file will fail.

Since `adios_advance_step()` also releases the step from which one advances forward, it is not causing memory leaks if this function is not called. However, it is good practice to release a step after reading all necessary data and before processing it, to let the writer code make progress in the meantime.

```
void adios_release_step (ADIOS_FILE *fp);
```

7.4.9 adios_inq_var

Inquires about a variable. This function does not read anything from the file but processes info already in memory after fopen. It allocates memory for the ADIOS_VARINFO struct and content, so you need to free resources later with adios_free_varinfo().

Note that you can get a scalar variable's value (including strings) with this operation without touching the file/stream. The 'stats' element will be NULL after this call. To get the statistics, another call must be made after this: adios_inq_var_stat(). The 'blocks' element will be NULL after this call. To get the decomposition of a variable in the file/stream, another call must be made after this: adios_inq_var_blockinfo().

- **fp** Pointer to an (opened) ADIOS_FILE struct.
- **varname** Name of the variable.

The function returns a pointer to an ADIOS_VARINFO struct, NULL on error (sets adios_errno).

```
ADIOS_VARINFO * adios_inq_var (ADIOS_FILE *fp, const char * varname);
```

7.4.10 adios_inq_var_byid

This function is the same as adios_inq_var but uses a numerical index instead of a name to reference the variable.

- **varid** index of variable (0..fp->nvars-1) in fp->vars_namelist of ADIOS_FILE struct

The function returns a pointer to an ADIOS_VARINFO struct, NULL on error (sets adios_errno).

```
ADIOS_VARINFO * adios_inq_var_byid (ADIOS_FILE *fp, int varid);
```

7.4.11 adios_free_varinfo

Free memory used by an ADIOS_VARINFO struct.

- **cp** The ADIOS_VARINFO struct that needs to be free'd.

The function does not return any value.

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

7.4.12 adios_inq_var_stat

Get statistics recorded about a variable. The information to calculate the statistics are recorded in the metadata, so no extra file access is necessary after adios_fopen() for this operation. The result is stored in the ADIOS_VARSTAT struct under varinfo.stats. adios_free_varinfo() will free the extra memory allocated in this call. Note that the generation of statistics can be turned off at writing, and then this function will deliver nothing; it is not going to read the data and calculate the statistics.

- **fp** Pointer to an (opened) ADIOS_FILE struct.
- **varinfo** Result of adios_inq_var().
- **per_step_stat** $!=0$: return statistics also per step
- **per_block_stat** $!=0$: return statistics also per writer block

The function returns 0 on success, $!=0$ on error (also sets adios_errno).

```
int adios_inq_var_stat (ADIOS_FILE *fp, ADIOS_VARINFO * varinfo,
                        int per_step_stat, int per_block_stat);
```

7.4.13 adios_inq_var_blockinfo

Get the block-decomposition of the variable about how it is stored in the file or stream. The decomposition information are recorded in the metadata, so no extra file access is necessary after adios_fopen() for this operation. The result is stored in the array of ADIOS_VARBLOCK structs under varinfo.blocks.

adios_free_varinfo() will free the extra memory allocated in this call.

- **fp** Pointer to an (opened) ADIOS_FILE struct.
- **varinfo** Result of adios_inq_var().

Function returns 0 on success, $!=0$ on error (also sets adios_errno).

```
int adios_inq_var_blockinfo (ADIOS_FILE *fp, ADIOS_VARINFO * varinfo);
```

7.4.14 Selections

Before reading some data, one needs to create a selection object, unless a variable is to be read in as a whole by one process. ADIOS supports 4 types of selections: contigous bounding box, list of individual points, the block written separately (by one process), and automatic selection to let the method decide what is optimal to deliver to the specific reader. Note that dimensions and number of points are all 64bit integers as ADIOS supports large datasets.

The functions below return a pointer to the ADIOS_SELECTION struct which can be used to read variables.

7.4.14.1 adios_selection_boundingbox

A boundingbox selection to read a contiguous subset of a multi-dimensional array.

- **ndim** Number of dimensions
- **start** Array of offsets to start reading in each dimension
- **count** Number of data elements to read in each dimension

```
ADIOS_SELECTION * adios_selection_boundingbox (uint64_t ndim,
                                               const uint64_t *start,
                                               const uint64_t *count);
```

7.4.14.2 adios_selection_points

Selection of an enumeration of positions. Each point is described in the N–dimensional (array index) space is described by N offsets. The positions should be enumerated in a 1D array, with the N offsets of each point together.

- **ndim** Number of dimensions
- **npoints** Number of points of the selection
- **points** 1D array of indices, compacted for all dimension (e.g. [i1,j1,k1,i2,j2,k2,...,in,jn,kn] for n points in a 3D space.

```
ADIOS_SELECTION* adios_selection_points (uint64_t ndim,
                                         uint64_t npoints,
                                         const uint64_t *points);
```

7.4.14.3 adios_selection_writeblock

Selection for a block of data coming from a certain producer. A global array consist of many individual, contiguous blocks written out by many writers. One writer may output multiple subsets of a variable. Due to the ADIOS BP format's log-file structure, these blocks are accessible separately, and this selection lets users exploit this fact.

The number of blocks is returned by adios_inq_var(). Indexing of the blocks starts from 0 for the first block written by producer rank 0. Blocks from one writer will have consecutive indices. If each writer outputs one block then the index equals to the rank of the write process. With multi-var writing and multiple steps in a file, the index should be calculated by the reading application using external information beyond what is provided by the ADIOS Read API (e.g. writing this information out into the file as variables).

This selection replaces the adios_read_local_var() function of the old read API. Its main use has been to read files where a variable is not a global array, because the application cannot organize the blocks into an N-dimensional contiguous array. This is the only way to access all writers' blocks of such 'local' variables.

- **index** Index of the written block

```
ADIOS_SELECTION* adios_selection_writeblock (int index);
```

7.4.14.4 adios_selection_auto

Let the method decide what data gets to what reader process. This selection enables each reading method to provide an 'optimal' data transfer from writers to readers. It depends on the method and the circumstances, what this selection actually means. E.g. intra-node in situ processing: readers on a compute node will receive all data from the writers on the same compute node.

- **hints** Method dependent parameters to influence what and how to return (e.g. decomposition; ordering of returned chunks)

```
ADIOS_SELECTION* adios_selection_auto (char * hints);
```

7.4.14.5 adios_selection_delete

Delete a selection and free up memory used by the selection.

```
void adios_selection_delete (ADIOS_SELECTION *selection);
```

7.4.15 adios_schedule_read

Schedule reading a (subset of a) variable from the file. In most cases, you need to allocate the memory for the data and Call adios_perform_reads() to complete the reading of the variables. Multiple reads can/should be scheduled before performing all of them at once. This strategy can improve the use of available I/O bandwidth and possible avoid some seeking on disks. Nevertheless, multiple schedule/perform cycles can be executed on an open file/steam.

In blocking read mode, the memory should be pre-allocated. In non-blocking mode, memory can be allocated or not, and that changes the behavior of the chunked read. If memory is allocated, adios_check_read() returns the whole requested subset of a variable when it is completed. If memory is not allocated, the check returns any chunk already available of a variable (in ADIOS own buffer) and the application has to rearrange the data. The user has to process/copy the data before getting new chunks.

- **fp** Pointer to an (opened) ADIOS_FILE struct.
- **sel** Selection created beforehand with adios_selection...(). sel=NULL means global selection (whole variable)
- **varname** Name of the variable.
- **from_step** File mode only: Read the 'nsteps' consecutive steps from this step of a file variable, instead of from the current (global) step of the file. It is not used in case of a stream.
- **nsteps** Read 'nsteps' consecutive steps from current step. Must be 1 for a stream.
- **data** Pointer to the memory to hold data of the variable. NULL in case of non-blocking, chunked reading.

The function returns 0 on success, ! = 0 on error (also sets adios_errno).

```
int adios_schedule_read (const ADIOS_FILE * fp,
                         const ADIOS_SELECTION * sel,
                         const char            * varname,
                         int                     from_steps,
                         int                     nsteps,
                         void                  * data);
```

7.4.16 adios_schedule_read_byid

This function is the same as adios_schedule_read but uses a numerical index instead of a name to reference the variable.

- **varid** Index of variable (0..fp->nvars-1) in fp->var_namelist of ADIOS_FILE struct.

```
int adios_schedule_read_byid (const ADIOS_FILE * fp,
                              const ADIOS_SELECTION * sel,
                              int                     varid,
                              int                     from_steps,
                              int                     nsteps,
                              void                  * data);
```

7.4.17 adios_perform_reads

Once adios_schedule_read command has been issued for all the variables needed by the reading application, the adios_perform_reads is called to start performing the reads.

- **blocking** If non-zero, return only when all reads are completed. If zero, return immediately and report partial completions through adios_check_reads().

```
int adios_perform_reads (const ADIOS_FILE *fp, int blocking);
```

7.4.18 adios_check_reads

Get a chunk of completed read(s) in a non-blocking or in a non-blocking+chunking read scenario. This function should be called in a loop until all chunks are processed. That is indicated by a 0 return value. A NULL result for chunk only indicates that no chunk is available at the time of call.

One chunk is returned at a time. If memory for a variable is provided in adios_schedule_read (non-blocking scenario), one chunk will be returned for the variable, and the memory will be fully organized (contiguous block). If memory is not provided by the user, a selection of an array specified in a read may be completed in multiple chunks (usually when they come from multiple sources, like different disks or different application processes).

- **fp** Handler to file or stream.
- **chunk** A chunk completed by the time of calling this function. It is NULL if no chunk is returned.

This function returns

- 0: All chunks have been returned previously, no need to call again (chunk is NULL, too).
- 1: Some chunks are/will be available, call again.
- < 0: On error (also sets adios_errno).

```
int adios_check_reads (const ADIOS_FILE  * fp,
                       ADIOS_VARCHUNK   ** chunk);
```

7.4.19 adios_free_chunk

Free the memory of a chunk allocated inside adios_check_reads(). It only frees the ADIOS_VARCHUNK struct and the ADIOS_SELECTION struct pointed by the chunk. The data pointer should never be freed since that memory belongs to the reading method.

```
void adios_free_chunk (ADIOS_VARCHUNK *chunk);
```

7.4.20 adios_get_attr

Get an attribute in a file. This function does not read anything from the file but processes info already in memory after fopen. The memory for the data is allocated within the library. You can use free() to free the memory after use.

- **fp** Pointer to an (opened) ADIOS_FILE struct.
- **attrname** Name of the attribute.
- **type** ADIOS type of attribute (see enum ADIOS_DATATYPES in adios_types.h) filled in by the call.
- **size** Memory size of value (n+1 for a string of n characters) filled in by the call.
- **data** Pointer to the value filled in by the call. You need to cast it afterward according to the type.

Function returns 0 on success, ! = 0 on error (also sets adios_errno).

```
int adios_get_attr (ADIOS_FILE              * fp,
                    const char              * attrname,
                    enum ADIOS_DATATYPES    * type,
                    int                     * size,
                    void                   ** data);
```

7.4.21 adios_get_attr_byid

This function is the same as adios_get_attr but uses a numerical index instead of a name to reference the variable.

- **attrid** Index of attribute (0..fp->nattrs-1) in fp->attr_namelist of ADIOS_FILE struct.

```
int adios_get_attr_byid (ADIOS_FILE              * fp,
                         int                      attrid,
                         enum ADIOS_DATATYPES    * type,
                         int                     * size,
                         void                   ** data);
```

7.4.22 adios_type_to_string

Return the name of an adios type.

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

7.4.23 adios_type_size

Return 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, it does not care about data and returns the size occupied by one element.

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

7.4.24 adios_get_grouplist

Return the list of groups (names) that are written into the file. There is always at least one group there.

- **fp** Pointer to an (opened) ADIOS_FILE struct
- **group_namelist** List of strings. This list is created and filled in by the function call. It should be freed by the user when it is not needed anymore.

Function returns the number of groups, < 0 on error (also sets adios_errno).

```
int adios_get_grouplist (ADIOS_FILE  *fp,
                         char ***group_namelist);
```

7.4.25 adios_group_view

Restrict the view of variables/attributes to a certain group. The provided ADIOS_FILE structure is directly modified but another calls can change to a different group view, or reset back to full view.

- **groupid** Id of the selected group (0..# of groups-1) use -1 to reset to the complete list.

- **fp** Pointer to an (opened) ADIOS_FILE struct nvars, var_namelist, nattrs, and attr_namelist will be modified.

Function returns 0 on success, ! = 0 on error (also sets adios_errno).

Note: A stream does not have groups, only a file can have multiple groups (from separate adios_open/adios_close operations).

```
int adios_group_view (ADIOS_FILE  *fp,
                      int groupid);
```

7.5 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 7.4.9, the `adios_inq_var` and `adios_inq_var_stat` functions populate 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.

7.5.1 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.

7.6 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).

A Fortran90 module, `adios_read_mod.mod` provides the available ADIOS subroutines. 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.

In contrast to the C API, where the open function returns a structure filled with a lot of information, the Fortran API only returns a handle. Therefore, you have to inquiry the file after opening it. You also have to inquiry an attribute to determine the memory size needed to store its value 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.

From functionality point of view, the difference in C and Fortran is that the Fortran API does not allow non-blocking reads in `adios_perform_reads`, and thus chunking is not working either. Memory for all variables should be allocated in advance to store the data.

Here is the list of the Fortran90 subroutines from `adios_read_mod.mod`. In the list below `GENERIC` word indicates that you can use that function with any data type at the indicated argument; it is not a Fortran90 keyword. The actual module source defines all possible combinations of type and dimensionality for such subroutines.

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

subroutine adios_read_init_method (method, comm, parameters, err)
    integer,        intent(in)  :: method
    integer,        intent(in)  :: comm
    character(*),   intent(in)  :: parameters
    integer,        intent(out) :: err
end subroutine

subroutine adios_read_finalize_method (method, err)
    integer,        intent(in)  :: method
    integer,        intent(out) :: err
end subroutine

subroutine adios_read_open (fp, fname, method, comm, lockmode,
                            timeout_msec, err)
    integer*8,      intent(out) :: fp
    character(*),   intent(in)  :: fname
    integer,        intent(in)  :: method
    integer,        intent(in)  :: comm
    integer,        intent(in)  :: lockmode
    integer,        intent(in)  :: timeout_msec
    integer,        intent(out) :: err
end subroutine

subroutine adios_read_open_file (fp, fname, method, comm, err)
    integer*8,      intent(out) :: fp
    character(*),   intent(in)  :: fname
    integer,        intent(in)  :: method
    integer,        intent(in)  :: comm
```

```
    integer,        intent(out) :: err
end subroutine

subroutine adios_advance_step (fp, last, timeout_sec, err)
    implicit none
    integer*8,      intent(in)  :: fp
    integer,        intent(in)  :: last
    real,           intent(in)  :: timeout_sec
    integer,        intent(out) :: err
end subroutine

subroutine adios_release_step (fp, err)
    implicit none
    integer*8,      intent(in)  :: fp
    integer,        intent(out) :: err
end subroutine

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

subroutine adios_inq_file (fp, vars_count, attrs_count,
                           current_step, last_step, err)
    integer*8,      intent(in)  :: fp
    integer,        intent(out) :: vars_count
    integer,        intent(out) :: attrs_count
    integer,        intent(out) :: current_step
    integer,        intent(out) :: last_step
    integer,        intent(out) :: err
end subroutine

subroutine adios_inq_varnames (fp, vnamelist, err)
    integer*8,      intent(in)  :: fp
    character(*), dimension(*), intent(inout) :: vnamelist
    integer,        intent(out) :: err
end subroutine

subroutine adios_inq_attrnames (fp, anamelist, err)
    integer*8,      intent(in)  :: fp
    character(*), dimension(*), intent(inout) :: anamelist
    integer,        intent(out) :: err
end subroutine

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

subroutine adios_inq_attr (fp, attrname, attrtype, attrsize, err)
```

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

subroutine adios_get_scalar (fp, varname, data, err)
    integer*8,       intent(in)  :: fp
    character(*),    intent(in)  :: varname
    GENERIC,         intent(out) :: data
    integer,         intent(out) :: err
end subroutine

subroutine adios_selection_boundingbox (sel, ndim, start, count)
    integer*8,       intent(out)           :: sel
    integer,         intent(in)            :: ndim
    integer*8, dimension(*), intent(in)  :: start
    integer*8, dimension(*), intent(in)  :: count
end subroutine

subroutine adios_selection_points (sel, ndim, npoints, points)
    integer*8,       intent(out)           :: sel
    integer,         intent(in)            :: ndim
    integer*8,       intent(in)            :: npoints
    integer*8, dimension(*), intent(in)  :: points
end subroutine

subroutine adios_selection_writeblock (sel, index)
    integer*8,       intent(out)           :: sel
    integer,         intent(in)            :: index
end subroutine

subroutine adios_selection_auto (sel, hints)
    integer*8,       intent(out)           :: sel
    character(*),    intent(in)            :: hints
end subroutine

subroutine adios_selection_delete (sel)
    integer*8,       intent(in)            :: sel
end subroutine

subroutine adios_schedule_read (fp, sel, varname, from_step, nsteps, data, err)
    integer*8,       intent(in)  :: fp
    integer*8,       intent(in)  :: sel
    character(*),    intent(in)  :: varname
    integer,         intent(in)  :: from_step
    integer,         intent(in)  :: nsteps
    GENERIC, GENERIC_DIMENSIONS, intent(inout) :: data
    integer,         intent(in)  :: err
end subroutine

subroutine adios_perform_reads (fp, err)
    integer*8,       intent(in)  :: fp
    integer,         intent(out) :: err
```

```
end subroutine

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

subroutine adios_get_statistics (gp, varname, value, gmin, gmax, gavg,
                                 gstd_dev, mins, maxs, avgs, std_devs, err)
    integer*8,       intent(in)  :: gp
    character(*),    intent(in)  :: varname
    GENERIC,         intent(out) :: value
    GENERIC,         intent(out) :: gmin
    GENERIC,         intent(out) :: gmax
    real*8,          intent(out) :: gavg
    real*8,          intent(out) :: gstd_dev
    GENERIC, dimension(*), intent(inout) :: mins
    GENERIC, dimension(*), intent(inout) :: maxs
    real*8, dimension(*), intent(inout) :: avgs
    real*8, dimension(*), intent(out) :: std_devs
    integer,dimension(*), intent(out) :: err
end subroutine

!
! Group operations for the case when a file has multiple groups and
! one really wants to see only one of them at a time
!
subroutine adios_inq_ngroups (fp, groups_count, err)
    integer*8,       intent(in)  :: fp
    integer,         intent(out) :: groups_count
    integer,         intent(out) :: err
end subroutine

subroutine adios_inq_groupnames (fp, gnamelist, err)
    integer*8,       intent(in)  :: fp
    character(*), dimension(*), intent(inout) :: gnamelist
    integer,         intent(out) :: err
end subroutine

subroutine adios_group_view (fp, groupid, err)
    integer*8,       intent(in)  :: fp
    integer,         intent(in)  :: groupid
    integer,         intent(out) :: err
end subroutine
```

7.7 Compiling and linking applications

You are encouraged to use the utility `adios_config` to get the compile and link options for your need, using -f option to get the Fortran options, -c for compile, -l for linking, -s for non-MPI applications (see Section 2.5).

7.7.1 C/C++ applications

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

- If you want to use the MPI version of the library, then link your 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`.
- If you have a code using the old (before ADIOS 1.4) read API, compile your code with the `-DADIOS_USE_READ_API_1` and link your application with one of the two libraries above.

7.7.2 Fortran applications

In a Fortran 90 code, use the module `adios_read_mod`. It is strongly recommended to use it 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 application with `-ladiosreadf`.
- 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 `-ladiosreadf_nompi`.
- If you have a code using the old (before ADIOS 1.4) read API, do not use the adios_read_mod module and link your application with one of the two libraries

  `-ladiosreadf_v1` or `-ladiosreadf_nompi_v1`.

7.8 Supported scenarios and samples

For all C examples below the following variables are assumed to be defined:

```
MPI_Comm comm;      // group communicator
ADIOS_FILE *fp;     // file handler
ADIOS_VARINFO *vi;  // information about one variable
double *P;          // array to store variable "P"
```

7.9 Reading a file as file

If a file is opened as a "file" (and not as a stream) than the followings are true:

- All steps in the file are available for reading; there is no "current step" from which to read and therefore, there is no need to advance the step.
- Variables have their own counter for steps. Different variables can have different steps available.
- Multiple consecutive steps of a variable can be read at once, starting from an arbitrary step.
- Multiple groups are allowed to exist in the file. The variables of those groups are presented in one list. This leads to the different number of steps of variables.

7.9.1 Discover and read in a complete variable

Assume we have a file called `mydata.bp` and a 3D array variable `P` of double type in it. We open the file, determine the size of the array, allocate memory for it and then read it in a blocking way. After `adios_perform_reads()`, the data is going to be stored in the allocated memory:

```
fp = adios_read_open_file ("myfile.bp", ADIOS_READ_METHOD_BP, comm);
vi = adios_inq_var (fp, "P");
// vi->ndim tells the number of dimensions
P = (double*) malloc (sizeof(double) *
                          vi->dims[0] * vi->dims[1] * vi->dims[2]);
adios_schedule_read (fp, NULL, "P", 0, 1, P);
adios_perform_reads (fp, 1);
// P contains the data at this point
...
// free ADIOS resources
adios_free_varinfo (vi);
adios_read_close (fp);
```

Listing 7.1: Read a complete array from a file

7.9.2 Multiple steps of a variable

If the file contains more than one step, the array P can have multiple steps too. In case of files, each variable has its own number of steps, provided by `adios_inq_var()`, in the `nsteps` field of the `ADIOS_VARINFO` struct. The example in Listing 7.1 still works but only reads in the first step of P. To read all steps at once, we have to allocate a big enough array for it, and request a read for all steps:

```
...
// vi->nsteps tells the number of steps
P = (double*) malloc (sizeof(double) *
                vi->nsteps * vi->dims[0] * vi->dims[1] * vi->dims[2]);
adios_schedule_read (fp, NULL, "P", 0, vi->nsteps, P);
...
```

7.9.3 Read a bounding box subset of a variable

In parallel codes, a process usually wants to read only a subset of the whole array. If we want to read a rectangular subset from the array, we have to create a boundingbox selection first with `adios_query_boundingbox()`, then pass it as an argument at reading. Let's read a 10x10x10 box from the offset (5,5,5).

```
fp = adios_read_open_file ("myfile.bp", ADIOS_READ_METHOD_BP, comm);
vi = adios_inq_var (fp, "P");
uint64_t count[] = {10,10,10};
uint64_t offs[] = {5,5,5};
P = (double*) malloc (sizeof(double) * count[0] * count[1] * count[2]);
ADIOS_SELECTION *s = adios_selection_boundingbox (3, offs, count);
adios_schedule_read (fp, s, "P", 0, 1, P);
adios_perform_reads (fp, 1);
// P contains the data at this point
...
// free ADIOS resources
adios_free_varinfo (vi);
adios_selection_delete (s);
adios_read_close (fp);
```

Listing 7.2: Read a bounding box of a variable

7.9.4 Reading non-global variables written by multiple processes

ADIOS allows for writing an array from several processes with different sizes, that does not constitute a global array view for reading. A reader still has access to each array in the file although they are named the same. `adios_inq_var()` returns the number of blocks and a flag whether the variable has a global view in

the ADIOS_VARINFO struct. If each process writes only one block of the variable, the MPI rank of the writing process identifies each block. If multiple steps are stored in a file, the second step's indexing starts from 0 again. For stream reading, of course, in each step the block numbering starts from 0. In the most complicated scenario, writers may output multiple blocks per process. In this case, the numbering is continuous for each process, i.e., writer with rank 0 produces block 0, 1, ..., and rank 1 produces the next blocks.

A special query is supported for this kind of reading, which selects one of the writing processes:

```
ADIOS_SELECTION *s = adios_selection_writeblock(5);  // read block 5
```

This special query still allows the Reader for providing an allocated memory to use blocking read. Usually, applications that read checkpoint files, know the size of each piece in advance from their own configuration file. If not, one can get the size of each block by calling adios_inq_var() and then adios_inq_var_blockinfo(). Another way is to read the scalar variables that defined the array size in the writer, using this writeblock selection and use those values. Note that adios_inq_var() provides a scalar variable's value written by one of the writer processes only, so it cannot be used here. To get the scalar value written by a specific process, this rank selection and adios_schedule_read() should be used.

```
/* first read the scalars that define the size of the array written
    by a given process */
int lx, ly, lz;
adios_schedule_read (fp, s, "lx", 0, 1, &lx);
adios_schedule_read (fp, s, "ly", 0, 1, &ly);
adios_schedule_read (fp, s, "lz", 0, 1, &lz);
adios_perform_reads (fp, 1);
// allocate memory to read in the array
P = (double*) malloc (sizeof(double) * lx * ly * lz);
adios_schedule_read (fp, s, "P", 0, 1, P);
adios_perform_reads (fp, 1);
```

Listing 7.3: Read an array written by one specific process, with first reading the scalars that define the size of the array

```
/* first inquire the variable to check the size of the array written
    by a given process */
int lx, ly, lz;
ADIOS_VARINFO * vi = adios_inq_var (fp, "P");
// vi->nblocks[0] tells us how many write blocks are there
// now get per-block size information
adios_inq_var_blockinfo (fp, vi);
lx = vi->blockinfo[5].count[0]; // 5 is block index here
ly = vi->blockinfo[5].count[1];
lz = vi->blockinfo[5].count[2];
// allocate memory to read in the array
P = (double*) malloc (sizeof(double) * lx * ly * lz);
adios_schedule_read (fp, s, "P", 0, 1, P);
adios_perform_reads (fp, 1);
```

Listing 7.4: Read an array written by one specific process, with first checking the size

```
int step = 3; // read step 3 (steps start from 0)
int block = 5; // read block 5 from step 3 (blocks start from 0)
ADIOS_SELECTION *s = adios_selection_writeblock(block);
/* first inquire the variable to check the size of the array written
    by a given process */
int lx, ly, lz;
ADIOS_VARINFO * vi = adios_inq_var (fp, "P");
// vi->nblocks[] tells us how many write blocks are there per step
// vi->sum_nblocks is the total number of blocks for all steps
```

```
// now get per-block size information
adios_inq_var_blockinfo (fp, vi);
int i, gblock =  block; // gblock to hold global block index
for (i=0; i<step; i++)
    gblock += vi->nblocks[i];
lx = vi->blockinfo[gblock].count[0];
ly = vi->blockinfo[gblock].count[1];
lz = vi->blockinfo[gblock].count[2];
// allocate memory to read in the array
P = (double*) malloc (sizeof(double) * lx * ly * lz);
adios_schedule_read (fp, s, "P", step, 1, P);
adios_perform_reads (fp, 1);
```

Listing 7.5: Read an array written by one specific process, when multiple steps are in a file

Of course, a global variable can be read this way, too. A global variable in ADIOS is nothing else than the collection of these individual pieces where metadata is available to tell ADIOS the global dimensions and the offsets of these pieces.

7.10 Reading streams

A file on disk (containing multiple steps) or a stream provided by a staging method can be opened as a stream. In contrast to files opened as files, the following rules apply here:

- Only one step is accessible.
- To read another step, one has to "advance" the step in the stream.
- There is no moving back in the stream, only forward.
- The file open or the advance operations can fail if data is not available any more.
- The end of a stream (last step consumed) is signaled by a different error return value.

The basic read structure is to open a stream, read the first step then advance the step until an error (`err_end_of_stream`) says there is not going to be any more steps. Also, at each advancement, an error may occur if the next step is not available yet (`err_step_notready`) or anymore (`err_step_disappeared`).

7.10.1 Opening a stream

The opening of a stream has to be repeated in case the stream is not yet available. Note, that there is no distinction of the situations where a stream is not yet available vs. the named stream will never exist.

```
fp = adios_read_open ("myfile.bp", ADIOS_READ_METHOD_BP, comm,
                      ADIOS_LOCKMODE_CURRENT, timeout_msec);
while (adios_errno == err_file_not_found) {
    fprintf (stderr, "rank %d: Wait on stream: %s\n", rank, adios_errmsg());
    sleep(1);
    fp = adios_read_open ("myfile.bp", comm,
                          ADIOS_LOCKMODE_CURRENT, timeout_msec);
}
if (adios_errno == err_end_of_stream) {
    // stream has been gone before we tried to open
    fprintf (stderr, "rank %d: Stream terminated before open. %s\n",
             rank, adios_errmsg());
} else if (fp == NULL) {
    // some other error happened
    fprintf (stderr, "rank %d: Error at opening: %s\n",
```

```
            rank, adios_errmsg());
17
} else {
19      // process steps here... see Listing 7.7
        ...
21  }
adios_read_close (fp);
```

Listing 7.6: While loop to open a stream

7.10.2 Reading one step at a time, blocking if a new step is late

In the conditional branch of Listing 7.6 from line 17 is where we can read steps in a loop. Let's assume we read variable P, of which we already know the size and we have allocated the memory before.

```
18  while (adios_errno != err_end_of_stream) {
        // fp->current_step  contains the step we are at
20      adios_schedule_read (fp, NULL, "P", 0, 1, P);
        adios_perform_reads (fp, 1);
22      // this step is no longer needed
        adios_release_step (fp);
24      // ... process P, then advance the step
        // 1) to the next available step (arg 0 as false)
26      // 2) with blocking wait (-1 as timeout)
        adios_advance_step (fp, 0, -1);
28  }
```

Listing 7.7: Read a bounding box of a variable

In the above code snippet we advance to the next available step (second argument in `adios_advance_step()`), possibly skipping other steps if they have appeared and disappeared while we were processing (we asked for locking of only the current step when opening the file). Also we let ADIOS block until a new step becomes available or the stream ends (third parameter in `adios_advance_step()` equals 1). The `fp->current_step` informs us of the step we advanced to.

7.10.3 Locking and step advancing scenarios

1. ADIOS_LOCKMODE_ALL + next step: Read all steps one by one, ensure they are not lost.
2. ADIOS_LOCKMODE_CURRENT + next step: Read each step which is available.
3. last step: Read always the last (newest) step available.
4. ADIOS_LOCKMODE_NONE: reader assumes nothing, even current step can disappear between reads.

If the reader needs to ensure it can process all steps without skipping any, it has to use the strictest locking mode: `ADIOS_LOCKMODE_ALL`, which gives priority to the reader over the performance of the writer. No step will be removed to make space for incoming steps until the reader advances from that step. This may block the writer, so use it only if really needed. Also, when advancing we should ask for the next, and not for the last, step.

If we ask for the last available step, there is no point of locking all steps and thus potentially slowing down the writer.

If we lock nothing at read, the current step can be removed by a staging method if the writer has new data. It is the reader's responsibility to handle errors and ensure its consistent state.

7.10.4 Handling errors due to missing steps

The `adios_advance_step()` gets the next or last available step. In all cases, the `fp->current_step` informs us about the new step. One has to save the previous value and compare with the new one to check if some steps were skipped. This function returns two possible errors. If the writer has terminated the stream and the reader is already at the very last step, an `err_end_of_stream` error will be the result of advancing. This condition should be used to determine when to stop processing the stream. The reader still needs to call `adios_read_close()` to free up resources. On the other hand, if the reader is at the currently latest step and the staging method has not yet received a newer step from the writer, and we try to advance without blocking, an `err_step_notready` error will be returned.

7.11 Non-blocking reads

7.11.1 Chunk reads: read without pre-allocating buffers

Note that this chunked read is partially implemented in ADIOS 1.5.0: memory limits are not considered and each scheduled read is returned in one chunk.

An ADIOS read method can deliver the data in chunks, in its own working memory. The application has to process that data before checking for new chunks. Reader methods are (usually) not using extra threads to perform I/O while the application is doing something else, therefore, the application has to regularly check for chunks until there is one. In this call will the reader method actually perform its work, except for data transfers initiated with RDMA (Remote Direct Memory Access) networking operations, that are executed by the network subsystem independently from the application execution.

First, we need to tell the reading method how much maximum memory it can use for storing data. If we don't provide this, the method will use as much as needed and it might run out of memory. The allowed amount should be enough to store the largest piece of any variable written by any individual process. Reading methods usually do not work with a finer granulation than this size, unless explicitly documented for a given method.

```
adios_read_init_method (ADIOS_READ_METHOD_DATASPACES, comm,
                        "max_chunk_size=100"); // 100 MB
fp = adios_read_open ("myfile.bp", ADIOS_READ_METHOD_BP, comm,
                      ADIOS_LOCKMODE_CURRENT, 0); // 0: wait forever
vi = adios_inq_var (fp, "P");
adios_schedule_read (fp, s, "P", 0, 1, NULL);
adios_perform_reads (fp, 0);
// Loop to get chunks
int ck;
ADIOS_VARCHUNK * chunk;
while ( (ck = adios_check_reads (fp, &chunk)) > 0) {
    if (chunk) {
        // process the chunk first, see Listing 7.9
        ...
        // free memory of chunk (not the data!)
        adios_free_chunk (chunk);
    } else {
        // no chunk was returned, slow down a little
        sleep(1);
    }
}
if (ck < 0) {
    // some error happened
    fprintf (stderr, "rank %d: Error during chunk reads: %s\n",
             rank, adios_errmsg());
}
adios_free_varinfo (vi);
```

```
adios_read_close (fp);
adios_read_finalize_method (ADIOS_READ_METHOD_BP);
```

Listing 7.8: Read variable with auto selection in chunks from a stream

A returned chunk contains the integer *id* of the variable (variable name is `fp->varnamelist[chunk->varid]`), its type, a pointer to the data and a pointer to an `ADIOS_SUBSET` struct, which describes what subset of a variable is returned. ADIOS supports two basic selection types, which can be returned: a single bounding box or a list of points. If the original selection is a bounding box then each chunk will be also a boundingbox representing a subset of the original bounding box. A chunk is usually is the intersection of one writer process' output of the given variable and the original selection. In case of list of points, each chunk will be a list of points too.

```
ADIOS_SUBSET * s = chunk->chunk_subset;
printf ("Variable %s:\n", fp->varnamelist[chunk->varid]);
switch(s->type) {
case ADIOS_SUBSET_BOUNDINGBOX:
    printf ("%d-D Bounding Box offset=(%d %d %d) size=(%d %d %d)\n",
        s->u.bb.ndim;
        s->u.bb.start[0], s->u.bb.start[1], s->u.bb.start[2],
        s->u.bb.count[0], s->u.bb.count[1], s->u.bb.count[2]);
    break;
case ADIOS_SUBSET_POINTS:
    int n;
    for (n=0; n<s->npoints; n++) {
        // One point in 3D is three consecutive values
        //   s->u.points.points[3*n]
        //   s->u.points.points[3*n+1]
        //   s->u.points.points[3*n+2]
    }
    break;
default:
    fprintf (stderr, "rank %d: Error: unexpected chunk type: %d\n",
            rank, s->type);
}
```

Listing 7.9: Processing chunks from a file

7.11.2 Read into user-allocated buffers

If the application provides the memory for each scheduled read, the only difference to the chunked read is that each chunk describes one completed read as it was scheduled. That is, the returned chunk contains the whole subset of a variable. The code structure is thus the same as above, just processing each chunk means processing each variable.

7.12 More esoteric scenarios

7.12.1 In situ read: read data locally available on the node

A special scenario for reading is when the reader application processes data in situ with the writer application, using some of the computing cores of each compute node. Naturally, to avoid cross-node communication, readers want to get data from the writers located on the same node. In Section 7.9.4, we used a rank-based selection to specify from which writer processes we do want to get data. ADIOS does not support providing location based rank information of the writers to the readers, but the writer itself can write such data into the stream and then rank based reading can be applied.

A similar scenario is a file stored on a parallel file system. The best transfer bandwidth can be achieved by the file reading method, if it can decide which piece on what disk goes to which reader. In this case, the writer does not know then what information should be shared with the reader.

Therefore, a special query is defined that lets every staging method to deliver what is considered optimal for that particular method. `adios_query_auto()` lets the reading method to choose which writers' data it will return (in chunks). A staging method will deliver data from those writers that belong to that particular staging process. An in situ method will deliver data from writers that are located on the same compute node that the reader is. Each method has to document how this special case is handled.

7.12.2 Variable stepping of variables in a stream

Usually the number of steps in a file is a global value for all variables and attributes. However, someone may write different variables with different frequencies into a stream. This means that each variable has a different logical step, while in ADIOS the step is the feature of the stream, not of the individual variables. In case of files opened as files, this is straightforward since all read operations use the individual variable's stepping for reading.

In case of streams, however, those individual counters always equal 1. At each advance, the list of variables is updated, which can be used by the application itself to count how many times a given variable has occurred in the steps before that the reader has advanced to.

Let's assume `P` and `Q` are variables written with different frequencies, and `t` is a single real value at each step depicting the simulation time. Here is how we can keep track P and Q, with the extension of Listing 7.7.

```
int varid;
int varid_P, varid_Q;
int count_P = 0, count_Q = 0;
while (adios_errno != err_end_of_stream) {
    // fp->current_step  contains the step we are at
    vi = adios_inq_var (fp, "t");  // get simulation time at this step
    varid_P = varid_Q = -1;
    for (varid=0; varid < fp->nvars) {
        if (!strcmp("P", fp->var_namelist[varid])) {
            adios_schedule_read_byid (fp, NULL, varid, 0, 1, P);
            count_P++;
            varid_P = varid;
        } else if (!strcmp("Q", fp->var_namelist[varid])) {
            adios_schedule_read_byid (fp, NULL, varid, 0, 1, Q);
            count_Q++;
            varid_Q = varid;
        }
        adios_perform_reads (fp, 1);
        adios_release_step (fp); // this step is no longer needed
    // process P, Q, then advance the step
    ...
    // 1) to the next available step (arg 0 as false)
    // 2) with blocking wait (-1 as timeout)
    adios_advance_step (fp, 0, -1);
}
```

Listing 7.10: Processing varying set of variables in a stream

Chapter 8

Utilities

8.1 adios_lint

We provide a verification tool, called adios_lint, which comes with ADIOS. 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 adios_config

This script provides the necessary compile and linking flags to use ADIOS in your application and the version information of the ADIOS installation. See Section 2.5 for how to use it or run "adios_config -h" to see the options.

8.3 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 3r
```

Listing 8.1: 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
```

8.4 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 Listing 8.2).

```
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)
...
=========================================================
```

Listing 8.2: bpdump utility

Chapter 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.

Chapter 10

Group Read/Write Process

In ADIOS, 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
```

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" re

<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.

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.

Chapter 11

Language bindings

ADIOS provides the following wrappers to support various programming environments;

- **Java** – Write and Read ADIOS-BP files, with old read API
- **Python/Numpy** – Write and Read ADIOS-BP files, with old read API

In this chapter, we will describe how one can use ADIOS wrappers and provide a few example codes.

11.1 Java support

The Java wrapper program consists of a set of Java classes defined with a single namespace, `gov.ornl.ccs`. A list of classes is as follows:

- **Adios** – Provides functions to call init/free, write, and no-XML related APIs. All functions are static.
- **AdiosFile** – Related with Read APIs. Represents ADIOS file structure.
- **AdiosGroup** – Related with Read APIs. Represents ADIOS group structure.
- **AdiosVarinfo** – Related with Read APIs. Represents ADIOS varinfo structure.
- **AdiosDatatype** – Enumeration class for ADIOS data types.
- **AdiosFlag** – Enumeration class for ADIOS flags.
- **AdiosBufferAllocWhen** – Enumeration class for ADIOS buffer allocation flags.

11.1.1 Adios class

This class provides static functions for initialization, finalization, writing, and no-XML related APIs. The list of functions and signatures are as follows:

```
/* Call adios_init */
public static int Init(String xml_fname)

/* Call adios_open. Return a group handler */
public static long Open(String group_name, String file_name,
                        String mode, long comm)

/* Call adios_group_size and return the total size */
public static long SetGroupSize(long fh, long group_size)

/* Call adios_write and return the total size */
public static long Write (long fh, String var_name, byte value)
```

```
public static long Write (long fh, String var_name, int value)
public static long Write (long fh, String var_name, long value)
public static long Write (long fh, String var_name, float value)
public static long Write (long fh, String var_name, double value)
public static long Write (long fh, String var_name, byte[] value)
public static long Write (long fh, String var_name, int[] value)
public static long Write (long fh, String var_name, long[] value)
public static long Write (long fh, String var_name, float[] value)
public static long Write (long fh, String var_name, double[] value)

/* Call adios_close */
public static int Close (long fh)

/* Call adios_finalize */
public static int Finalize (int id)

/* Call MPI_Init */
public static int MPI_Init(String[] args)

/* Call MPI_Comm_rank */
public static int MPI_Comm_rank(long comm)

/* Call MPI_Comm_size */
public static int MPI_Comm_size(long comm)

/* Call MPI_Finalize */
public static int MPI_Finalize()

/* Get MPI_COMM_WORLD */
public static long MPI_COMM_WORLD()

/* Call adios_init_noxml */
public static int Init_Noxml()

/* Call adios_allocate_buffer */
public static int AllocateBuffer(AdiosBufferAllocWhen when, long size)

/* Call adios_declare_group */
public static long DeclareGroup(String name, String time_index,
                                AdiosFlag stats)

/* Call adios_define_var */
public static int DefineVar(long group_id, String name, String path,
                            AdiosDatatype type, String dimensions,
                            String global_dimensions,
                            String local_offsets)

/* Call adios_define_attribute */
public static int DefineAttribute(long group_id, String name,
                            String path, AdiosDatatype type,
                            String value, String var)

/* Call adios_select_method */
public static int SelectMethod(long group_id, String method,
```

```
                                String parameters, String base_path)
```

Listing 11.1: Member functions in the Adios class

11.1.2 AdiosFile, AdiosGroup, and AdiosVarinfo classes

AdiosFile, AdiosGroup, and AdiosVarinfo classes represent `ADIOS_FILE`, `ADIOS_GROUP`, `ADIOS_VARINFO` structure, respectively, defined in `adios_read_v1.h`. The following is a skeletal descriptions of those classes and member functions.

```
public class AdiosFile
{
    /* Call adios_fopen */
    public int open(String path, long comm)

    /* Call adios_fclose */
    public int close()

    /* Print contents for debugging purpose */
    public String toString()
}

public class AdiosGroup
{
    /* Constructor. Need AdiosFile instance */
    public AdiosGroup(AdiosFile file)

    /* Call adios_gopen */
    public int open(String grpname)

    /* Call adios_gclose */
    public int close()

    /* Print contents for debugging purpose */
    public String toString()
}

public class AdiosVarinfo
{
    /* Constructor. Need AdiosGroup instance */
    public AdiosVarinfo(AdiosGroup group)

    /* Call adios_inq_var */
    public int inq(String varname)

    /* Call adios_free_varinfo */
    public int close()

    /* Call adios_read_var */
    public double[] read(long[] start, long[] count)

    /* Print contents for debugging purpose */
    public String toString()
}
```

Listing 11.2: Class definitions of AdiosFile, AdiosGroup, and AdiosVarinfo

11.1.3 AdiosDatatype, AdiosFlag, and AdiosBufferAllocWhen classes

AdiosDatatype, AdiosFlag, and AdiosBufferAllocWhen are enumeration classes representing `ADIOS_DATATYPES`, `ADIOS_FLAG`, `ADIOS_BUFFER_ALLOC_WHEN` enum type, respectively, defined in `adios_types.h`. The following is a skeletal descriptions of those classes and member functions.

```
public enum AdiosDatatype {
    UNKNOWN(-1),              /* (SIZE) */
    BYTE(0),                  /* (1) */
    SHORT(1),                 /* (2) */
    INTEGER(2),               /* (4) */
    LONG(4),                  /* (8) */

    UNSIGNED_BYTE(50),        /* (1) */
    UNSIGNED_SHORT(51),       /* (2) */
    UNSIGNED_INTEGER(52),     /* (4) */
    UNSIGNED_LONG(54),        /* (8) */

    REAL(5),                  /* (4) */
    DOUBLE(6),                /* (8) */
    LONG_DOUBLE(7),           /* (16) */

    STRING(9),                /* (?) */
    COMPLEX(10),              /* (8) */
    DOUBLE_COMPLEX(11);       /* (16) */
}

public enum AdiosFlag {
    UNKNOWN(0),
    YES(1),
    NO(2);
}

public enum AdiosBufferAllocWhen {
    UNKNOWN(0),
    NOW(1),
    LATER(2);

}
```

Listing 11.3: Enum classes

11.1.4 Example

An example of Java program to call ADIOS functions is as follows:

```
import gov.ornl.ccs.*;
import java.nio.ByteBuffer;

public class AdiosNoxmlExample
{
    // The main program
    public static void main(String[] args)
    {
        Adios.MPI_Init(new String[0]);
        long comm = Adios.MPI_COMM_WORLD();
        int rank = Adios.MPI_Comm_rank(comm);
```

```
        int size = Adios.MPI_Comm_size(comm);

        Adios.Init_Noxml();
        Adios.AllocateBuffer(AdiosBufferAllocWhen.NOW, 10);

        long group_id = Adios.DeclareGroup("restart", "iter",
                                           AdiosFlag.YES);
        Adios.SelectMethod(group_id, "MPI", "", "");
        Adios.DefineVar(group_id, "NX", "",
                        AdiosDatatype.INTEGER, "", "", "");
        Adios.DefineVar(group_id, "G", "",
                        AdiosDatatype.INTEGER, "", "", "");
        Adios.DefineVar(group_id, "O", "",
                        AdiosDatatype.INTEGER, "", "", "");
        Adios.DefineVar(group_id, "temperature", "",
                        AdiosDatatype.DOUBLE, "NX", "G", "O");

        long adios_handle = Adios.Open("restart", "adios_noxml.bp",
                                       "w", comm);

        int NX = 10;
        int G = NX * size;
        int O = NX * rank;

        double[] t = new double[NX];
        for (int i = 0; i < NX; i++) {
            t[i] = rank * NX + (double) i;
        }

        long groupsize = 4 + 4 + 4 + 8 * (1) * (NX);

        long adios_totalsize = Adios.SetGroupSize(adios_handle, groupsize);

        Adios.Write (adios_handle, "NX", NX);
        Adios.Write (adios_handle, "G", G);
        Adios.Write (adios_handle, "O", O);
        Adios.Write (adios_handle, "temperature", t);
        Adios.Close (adios_handle);

        Adios.Finalize (rank);
        Adios.MPI_Finalize();
    }
}
```

Listing 11.4: Example Java wrapper code

11.2 Python/Numpy support

We developed a ADIOS python wrapper by using Cython. Numpy, a scientific module for Python, is a mandatory requirement.

11.2.1 APIs for Writing and No-XML

The ADIOS python/numpy wrapper provides functions to call ADIOS write and no-XML related APIs as follows (defined in Cython syntax):

```
""" Call adios_init """
cpdef init(char * config)

""" Call adios_open """
cpdef int64_t open(char * group_name,
                   char * name,
                   char * mode,
                   MPI.Comm comm = MPI.COMM_WORLD)

""" Call adios_group_size """
cpdef int64_t set_group_size(int64_t fd_p, uint64_t data_size)

""" Call adios_write """
cpdef int write (int64_t fd_p, char * name, np.ndarray val)
cpdef int write_int (int64_t fd_p, char * name, int val)
cpdef int write_long (int64_t fd_p, char * name, long val)
cpdef int write_float (int64_t fd_p, char * name, float val)

""" Call adios_read """
cpdef int read(int64_t fd_p, char * name, np.ndarray val)

""" Call adios_close """
cpdef int close(int64_t fd_p)

""" Call adios_finalize """
cpdef finalize(int mype = 0)

""" Call adios_init """
cpdef init(char * config)

""" Call adios_open """
cpdef int64_t open(char * group_name,
                   char * name,
                   char * mode,
                   MPI.Comm comm = MPI.COMM_WORLD)

""" Call adios_group_size """
cpdef int64_t set_group_size(int64_t fd_p, uint64_t data_size)

""" Call adios_write """
cpdef int write (int64_t fd_p, char * name, np.ndarray val)
cpdef int write_int (int64_t fd_p, char * name, int val)
cpdef int write_long (int64_t fd_p, char * name, long val)
cpdef int write_float (int64_t fd_p, char * name, float val)

""" Call adios_read """
cpdef int read(int64_t fd_p, char * name, np.ndarray val)

""" Call adios_close """
cpdef int close(int64_t fd_p)

""" Call adios_finalize """
cpdef finalize(int mype = 0)

""" Call adios_init_noxml """
```

```
cpdef int init_noxml():

""" Call adios_allocate_buffer """
cpdef int allocate_buffer(int when,
                          uint64_t buffer_size)

""" Call adios_declare_group """
cpdef int64_t declare_group(char * name,
                            char * time_index,
                            int stats)

""" Call adios_define_var """
cpdef int define_var(int64_t group_id,
                     char * name,
                     char * path,
                     int type,
                     char * dimensions,
                     char * global_dimensions,
                     char * local_offsets)

""" Call adios_define_attribute """
cpdef int define_attribute (int64_t group,
                            char * name,
                            char * path,
                            int type,
                            char * value,
                            char * var)

""" Call adios_select_method """
cpdef int select_method (int64_t group,
                         char * method,
                         char * parameters,
                         char * base_path)
```

Listing 11.5: Functions for writing and No-XML

11.2.2 APIs for Reading

The ADIOS python/numpy wrapper provides ADIOS read related classes as follows (defined in Cython syntax):

```
""" Python class for ADIOS_FILE structure """
cdef class AdiosFile:
    """ Private Memeber """
    cpdef ADIOS_FILE * fp

    """ Public Memeber """
    cpdef public bytes name
    cpdef public int groups_count
    cpdef public int vars_count
    cpdef public int attrs_count
    cpdef public int tidx_start
    cpdef public int ntimesteps
    cpdef public int version
    cpdef public int file_size
    cpdef public int endianness
```

```
    cpdef public dict group

    """ Initialization. Call adios_fopen and populate public members """
    def __init__(self, char * fname, MPI.Comm comm = MPI.COMM_WORLD):
        ...

    """ Call adios_fclose """
    cpdef close(self):
        ...

    """ Print self """
    cpdef printself(self):
        ...

""" Python class for ADIOS_GROUP structure """
cdef class AdiosGroup:
    """ Private Memeber """
    cdef AdiosFile file
    cdef ADIOS_GROUP * gp

    """ Public Memeber """
    cpdef public bytes name
    cpdef public int grpid
    cpdef public int vars_count
    cpdef public int attrs_count
    cpdef public int timestep
    cpdef public int lasttimestep

    cpdef public dict var

    """ Initialization. Call adios_gopen and populate public members """
    def __init__(self, AdiosFile file, char * name):

    """ Call adios_gclose """
    cpdef close(self):
        ...

    """ Print self """
    cpdef printself(self):
        ...

""" Python class for ADIOS_VARINFO structure """
cdef class AdiosVariable:
    """ Private Memeber """
    cdef AdiosGroup group
    cdef ADIOS_VARINFO * vp

    """ Public Memeber """
    cpdef public bytes name
    cpdef public int varid
    cpdef public type type
    cpdef public int ndim
    cpdef public tuple dims
    cpdef public int timedim
```

```
    cpdef public int characteristics_count

    """ Initialization. Call adios_inq_var and populate public members """
    def __init__(self, AdiosGroup group, char * name):
        ...

    """ Call adios_free_varinfo """
    cpdef close(self):
        ...

    """ Call adios_read_var """
    cpdef read(self, tuple offset = (), tuple count = ()):
        ...

    """ Print self """
    cpdef printself(self):
        ...
```

Listing 11.6: Write functions

11.2.3 Example

An example of Python program to call ADIOS functions is shown below. The example is a Python program for converting a NetCDF file to a ADIOS BP file. You can find the code in the source distribution: /wrapper/numpy/example/ncdf2bp.py.

```
#!/usr/bin/env python
from adios import *
from scipy.io import netcdf
import numpy as np
import sys
import os
import operator

def usage():
    print os.path.basename(sys.argv[0]),"netcdf_file","[time dimension name]"

if len(sys.argv) < 2:
    usage()
    sys.exit(0)

fname = sys.argv[1]
fout = '.'.join(fname.split('.')[:-1]) + ".bp"

tname = "time"
if len(sys.argv) > 2:
    tname = sys.argv[2]

## Open NetCDF file
f = netcdf.netcdf_file(fname, 'r')

## Check dimension
assert (all(map(lambda x: x is not None,
                [ val for k, val in f.dimensions.items()
                  if k != tname])))
```

```
## Two types of variables : time-dependent or time-independent
dimvar = {n:v for n,v in f.variables.items() if n in f.dimensions.keys()}
var = {n:v for n,v in f.variables.items() if n not in f.dimensions.keys()}
tdepvar = {n:v for n,v in var.items() if tname in v.dimensions}
tindvar = {n:v for n,v in var.items() if tname not in v.dimensions}

## Time dimension
assert (len(set([v.dimensions.index(tname) for v in tdepvar.values()]))==1)
tdx = tdepvar.values()[0].dimensions.index(tname)

assert (all([v.data.shape[tdx] for v in tdepvar.values()]))
tdim = tdepvar.values()[0].shape[tdx]

## Init ADIOS without xml
init_noxml()
allocate_buffer(BUFFER_ALLOC_WHEN.NOW, 10)
gid = declare_group ("group", tname, FLAG.YES)
select_method (gid, "MPI", "", "")

d1size = 0
for name, val in f.dimensions.items():
    if name == tname:
        continue
    print "Dimension : %s (%d)" % (name, val)
    define_var (gid, name, "", DATATYPE.integer, "", "", "")
    d1size += 4

v2size = 0
for name, var in tdepvar.items():
    print "Variable : %s (%s)" % (name, ','.join(var.dimensions))
    define_var (gid, name, "", DATATYPE.double,
                ','.join(var.dimensions),
                ','.join([dname for dname in var.dimensions
                          if dname != tname]),
                "0,0,0")
    v2size += reduce(operator.mul, var.shape) / tdim * 8

print "Count (dim, var) : ", (d1size, v2size)

## Clean old file
if os.access(fout, os.F_OK):
    os.remove(fout)

for it in range(tdim):
    print
    print "Time step : %d" % (it)

    fd = open("group", fout, "a")
    groupsize = d1size + v2size
    set_group_size(fd, groupsize)

    for name, val in f.dimensions.items():
        if name == tname:
            continue
        print "Dimension writing : %s (%d)" % (name, val)
```

```
        write_int(fd, name, val)

    for name, var in tdepvar.items():
        arr = np.array(var.data.take([it], axis=tdx),
                       dtype=np.float64)
        print "Variable writing : %s %s" % (name, arr.shape)
        write(fd, name, arr)

    close(fd)

f.close()
finalize()

print
print "Done. Saved:", fout
```

Listing 11.7: ncdf2bp.py. An example Python/Numpy wrapper code for converting a NetCDF file to a ADIOS BP file

Chapter 12

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.

12.1 Non-ADIOS Program

The starting programming example, shown in Listing 12.1, 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;
}
```

Listing 12.1: 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
```

12.2 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 Listing 12.2.

```
/* 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
    </adios-group>

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

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

</adios-config>
```

Listing 12.2: Example config.xml

12.3 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 Listing 12.3.

```
/* 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);
```

Listing 12.3: Example gwrite_temperature.ch

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

For our first program, we simply translate the program of Listing 12.1, 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 Listing 12.4. 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", comm);
  adios_open (&adios_handle, "temperature", filename, "w", comm);

  #include "gwrite_temperature.ch"

  adios_close (adios_handle);
  adios_finalize (rank);
  MPI_Finalize ();
  return 0;
}
```

Listing 12.4: 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"
```

12.5 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.

12.6 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 24, 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", comm);
    adios_open (&adios_handle, "temperature", filename, "r", comm);
#include "gread_temperature.ch"
    adios_close (adios_handle);
    adios_finalize (rank);
    MPI_Finalize ();
    return 0;
}
```

Listing 12.5: Example of a generated gread_temperature.ch file 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);
```

Listing 12.6: Example of a generated gread_temperature.ch file

12.7 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 29), 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", comm);
  adios_open (&adios_handle, "temperature", filename, "w", comm);
  #include "gwrite_temperature.ch"
  adios_close (adios_handle);
  adios_finalize (rank);
  MPI_Finalize ();
  return 0;
}
```

Listing 12.7: 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.

12.8 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 30 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>
```

Listing 12.8: 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);
```

Listing 12.9: 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)

12.8.1 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 / AV
    (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' proces
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 ;
}
```

12.8.2 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.

12.9 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>
```

Listing 12.10: 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' proces

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
```

12.10 Reading statistics

In ADIOS, 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:

```
Config.xml for creating histogram for an array variable
```

```
(examples/C/stat/stat.xml)
/* 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>
```

Listing 12.11: 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);
```

Chapter 13

Appendix

13.1 Datatypes used in the ADIOS XML file

size	Signed type	Unsigned type
1	byte, integer*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	

13.2 ADIOS APIs List

13.3 An Example on Writing Sub-blocks using No-XML APIs

This example illustrates both the use of sub blocks in writing, and the usage of the ADIOS non-xml API's. This example will write out two sub blocks of the variable temperature and place these in the global array.

```
#include <stdio.h>
#include <string.h>
#include "mpi.h"
#include "adios.h"
#include "adios_types.h"

#ifdef DMALLOC
#include "dmalloc.h"
#endif

int main (int argc, char ** argv)
{
    char        filename [256];
    int         rank, size, i, block;
    int         NX = 100, Global_bounds, Offsets;
    double      t[NX];
    int         sub_blocks = 3;
    MPI_Comm    comm = MPI_COMM_WORLD;

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

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

    Global_bounds = sub_blocks * NX * size;
```

```
strcpy (filename, "adios_global_no_xml.bp");

adios_init_noxml (comm);
adios_allocate_buffer (ADIOS_BUFFER_ALLOC_NOW, 10);

int64_t       m_adios_group;
int64_t       m_adios_file;

adios_declare_group (&m_adios_group, "restart", "iter", adios_flag_yes);
adios_select_method (m_adios_group, "MPI", "", "");
adios_define_var (m_adios_group, "NX"
                ,"", adios_integer
                ,0, 0, 0);

adios_define_var (m_adios_group, "Global_bounds"
                ,"", adios_integer
                ,0, 0, 0);

for (i=0;i<sub_blocks;i++) {

   adios_define_var (m_adios_group, "Offsets"
                ,"", adios_integer
                ,0, 0, 0);

   adios_define_var (m_adios_group, "temperature"
                ,"", adios_double
                ,"NX", "Global_bounds", "Offsets");
}

adios_open (&m_adios_file, "restart", filename, "w", comm);

adios_groupsize = sub_blocks * (4 + 4 + 4 + NX * 8);

adios_group_size (m_adios_file, adios_groupsize, &adios_totalsize);
adios_write(m_adios_file, "NX", (void *) &NX);
adios_write(m_adios_file, "Global_bounds", (void *) &Global_bounds);
   /* now we will write the data for each sub block */
for (block=0;block<sub_blocks;block++) {

   Offsets = rank * sub_blocks * NX + block*NX;
   adios_write(m_adios_file, "Offsets", (void *) &Offsets);

   for (i = 0; i < NX; i++)
       t[i] = Offsets + i;

   adios_write(m_adios_file, "temperature", t);
}

adios_close (m_adios_file);

MPI_Barrier (comm);

adios_finalize (rank);
```

```
    MPI_Finalize ();
    return 0;
}
```

Note: if local dimension/global dimension/offset of a variable is defined with passing a number, instead of using names of variable as shown in the following code snippet, for example,

```
    varid = adios_define_var (m_adios_group, "temperature",
                              "", adios_double,
                              "100", "400", "0");
```

the returned IDs should be saved and used in calling `adios_write_byid()` instead of `adios_write()`.