ADIOS2 Documentation

Release 2.9.2

William F Godoy

Nov 01, 2023

INTRODUCTION

Funded by the Exascale Computing Project (ECP), U.S. Department of Energy

CHAPTER

ONE

WHAT'S NEW IN 2.9?

1.1 Summary

This is a major release with new features and lots of bug fixes.

1.1.1 General

- GPU-Aware I/O enabled by using Kokkos. Device pointers can be passed to Put()/Get() calls directly. Kokkos 3.7.x required for this release. Works with CUDA, HIP and Kokkos applications. https://adios2.readthedocs.io/en/latest/advanced/gpu_aware.html#gpu-aware-i-o
- GPU-compression. MGARD and ZFP operators can compress data on GPU if they are built for GPU. MGARD operator can be fed with host/device pointers and will move data automaticaly. ZFP operator requires matching data and compressor location.
- Joined Array concept (besides Global Array and Local Array), which lets writers dump Local Arrays (no offsets no global shape) that are put together into a Global Array by the reader. One dimension of the arrays is selected for this join operation, while other dimensions must be the same for all writers. https://adios2.readthedocs.io/en/latest/components/components.html?highlight=Joined#shapes

1.1.2 File I/O

- Default File engine is now BP5. If for some reason this causes problems, manually specify using "BP4" for your application.
- BP5 engine supports multithreaded reading to accelerate read performance for low-core counts.
- BP5 Two level metadata aggregation and reduction reduced memory impact of collecting metadata and therefore is more scalable in terms of numbers of variables and writers than BP4.
- Uses Blosc-2 instead of Blosc for lossless compression. The new compression operator is backward compatible with old files compressed with blosc. The name of the operator remains "blosc".

1.1.3 Staging

- UCX dataplane added for SST staging engine to support networks under the UCX consortium
- MPI dataplane added for SST staging engine. It relies on MPI intercommunicators to connect multiple independent MPI applications for staging purposes. Applications must enable multithreaded MPI for this dataplane.

1.1.4 Experimental features

- Preliminary support for data structs. A struct can have single variables of basic types, and 1D fixed size arrays of basic types. Supported by BP5, SST and SSC engines.

CHAPTER

TWO

INTRODUCTION

ADIOS2 is the latest implementation of the Adaptable Input Output System. This brand new architecture continues the performance legacy of ADIOS1, and extends its capabilities to address the extreme challenges of scientific data IO.

The ADIOS2 repo is hosted at GitHub.

The ADIOS2 infrastructure is developed as a multi-institutional collaboration between

- Oak Ridge National Laboratory
- Kitware Inc.
- Lawrence Berkeley National Laboratory
- Georgia Institute of Technology
- Rutgers University

The key aspects ADIOS2 are

1. **Modular architecture:** ADIOS2 takes advantage of the major features of C++11. The architecture utilizes a balanced combination of runtime polymorphism and template meta-programming to expose intuitive abstractions for a broad range of IO applications.
2. **Community:** By maintaining coding standards, collaborative workflows, and understandable documentation, ADIOS2 lowers the barriers to entry for scientists to meaningfully interact with the code.
3. **Sustainability:** Continuous integration and unit testing ensure that ADIOS2 evolves responsibly. Bug reports are triaged and fixed in a timely manner and can be reported on GitHub.
4. **Language Support:** In addition to the native C++, bindings for Python, C, Fortran and Matlab are provided.
5. **Commitment:** ADIOS2 is committed to the HPC community, releasing a new version every 6 months.

ADIOS2 is funded by the Department of Energy as part of the Exascale Computing Project.

2.1 What ADIOS2 is and isn't

ADIOS2 is:

- **A Unified High-performance I/O Framework**: using the same abstraction API ADIOS2 can transport and transform groups of self-describing data variables and attributes across different media (file, wide-area-network, in-memory staging, etc.) with performance an ease of use as the main goals.
- **MPI-based**: parallel MPI applications as well as serial codes can use it
- **Streaming-oriented**: ADIOS2 favors codes transferring a group of variables asynchronously wherever possible. Moving one variable at a time, in synchronous fashion, is the special case rather than normal.

- **Step-based**: to resemble actual production of data in "steps" of variable groups, for either streaming or random-access (file) media
- **Free and open-source**: ADIOS2 is permissibly licensed under the OSI-approved Apache 2 license.
- **Extreme scale I/O**: ADIOS2 is being used in supercomputer applications that write and read up to several petabytes in a single simulation run. ADIOS2 is designed to provide scalable I/O on the largest supercomputers in the world.

ADIOS2 is not:

- **A file-only I/O library**: Code coupling and in situ analyis is possible through files but special engines are available to achieve the same thing faster through TCP, RDMA and MPI communication. High performance write/read using a file system is a primary goal of ADIOS2 though.
- **MPI-only**
- **A Hierarchical Model**: Data hierarchies can be built on top of the ADIOS2 according to the application, but ADIOS2 sits a layer of abstraction beneath this.
- **A Memory Manager Library**: we don't own or manage the application's memory

2.2 Adaptable IO beyond files in Scientific Data Lifecycles

Performant and usable tools for data management at scale are essential in an era where scientific breakthroughs are collaborative, multidisciplinary, and computational. ADIOS2 is an *adaptable*, *scalable*, and *unified* framework to aid scientific applications when data transfer volumes exceed the capabilities of traditional file I/O.

ADIOS2 provides

- Custom application management of massive data sets, starting from generation, analysis, and movement, as well as short-term and long-term storage.
- Self-describing data in binary-packed (*.bp*) format for rapid metadata extraction
- An ability to separate and extract relevant information from large data sets
- The capability to make real-time decisions based on in-transit or in-situ analytics
- The ability to expand to other transport mechanisms such wide area networks, remote direct memory access, and shared memory, with minimal overhead
- The ability to utilize the full capabilities of emergent hardware technologies, such as high-bandwidth memory and burst buffers

CHAPTER

THREE

INSTALL FROM SOURCE

ADIOS2 uses CMake for building, testing and installing the library and utilities.

3.1 Building, Testing, and Installing ADIOS 2

To build ADIOS v2.x, clone the repository and invoke the canonical CMake build sequence:

```
$ git clone https://github.com/ornladios/ADIOS2.git ADIOS2
$ mkdir adios2-build && cd adios2-build
$ cmake ../ADIOS2
-- The C compiler identification is GNU 7.3.0
-- The CXX compiler identification is GNU 7.3.0
...

ADIOS2 build configuration:
  ...
  Features:
    Library Type: shared
    Build Type:   Release
    Testing: ON
    Build Options:
      BZip2    : ON
      ZFP      : OFF
      SZ       : OFF
      MGARD    : OFF
      MPI      : ON
      DataMan  : ON
      SST      : ON
      ZeroMQ   : ON
      HDF5     : ON
      Python   : ON
      Fortran  : ON
      SysVShMem: ON
      Endian_Reverse: OFF
```

Then compile using

```
$ make -j 16
```

Optionally, run the tests:

```
$ ctest
Test project /home/wgodoy/workspace/build
        Start   1: ADIOSInterfaceWriteTest.DefineVar_int8_t_1x10
  1/295 Test   #1: ADIOSInterfaceWriteTest.DefineVar_int8_t_1x10 ......................
↪.   Passed    0.16 sec
        Start   2: ADIOSInterfaceWriteTest.DefineVar_int16_t_1x10
  2/295 Test   #2: ADIOSInterfaceWriteTest.DefineVar_int16_t_1x10 .....................
↪.   Passed    0.06 sec
        Start   3: ADIOSInterfaceWriteTest.DefineVar_int32_t_1x10

      ...

        Start 294: ADIOSBZip2Wrapper.WrongParameterValue
294/295 Test #294: ADIOSBZip2Wrapper.WrongParameterValue ..............................
↪.   Passed    0.00 sec
        Start 295: ADIOSBZip2Wrapper.WrongBZip2Name
295/295 Test #295: ADIOSBZip2Wrapper.WrongBZip2Name ...................................
↪.   Passed    0.00 sec

100% tests passed, 0 tests failed out of 295

Total Test time (real) =   95.95 sec
```

And finally, use the standard invocation to install:

```
$ make install
```

3.2 CMake Options

The following options can be specified with CMake's `-DVAR=VALUE` syntax. The default option is highlighted.

VAR	VALUE	Description
ADIOS2_USE_MPI	**ON**/OFF	MPI or non-MPI (serial) build.
ADIOS2_USE_ZeroMQ	**ON**/OFF	ZeroMQ for the DataMan engine.
ADIOS2_USE_HDF5	**ON**/OFF	HDF5 engine. If HDF5 is not on the syspath, it can be set using `-DHDF5_ROOT=/path/to/hdf5`
ADIOS2_USE_Python	**ON**/OFF	Python bindings. Python 3 will be used if found. If you want to specify a particular python version use `-DPYTHON_EXECUTABLE=/path/to/interpreter/python`
ADIOS2_USE_Fortran	**ON**/OFF	Bindings for Fortran 90 or above.
ADIOS2_USE_SST	**ON**/OFF	Simplified Staging Engine (SST) and its dependencies, requires MPI. Can optionally use LibFabric/UCX for RDMA transport. You can specify the LibFabric/UCX path manually with the -DLIBFABRIC_ROOT=... or -DUCX_ROOT=... option.
ADIOS2_USE_BZip2	**ON**/OFF	BZIP2 compression.
ADIOS2_USE_ZFP	**ON**/OFF	ZFP compression (experimental).
ADIOS2_USE_SZ	**ON**/OFF	SZ compression (experimental).
ADIOS2_USE_MGARD	**ON**/OFF	MGARD compression (experimental).
ADIOS2_USE_PNG	**ON**/OFF	PNG compression (experimental).
ADIOS2_USE_Blosc	**ON**/OFF	Blosc compression (experimental).
ADIOS2_USE_Endian_Reverse	**ON**/OFF	Reverse endian conversion if a different endianness is detected between write and read.
ADIOS2_USE_IME	**ON**/OFF	DDN IME transport.

In addition to the `ADIOS2_USE_Feature` options, the following options are also available to control how the library gets built:

CMake VAR Options	Values	Description |
`BUILD_SHARED_LIBS`	**ON**/OFF	Build shared libraries.
`ADIOS2_BUILD_EXAMPLES`	**ON**/OFF	Build examples.
`BUILD_TESTING`	**ON**/OFF	Build test code.
`CMAKE_INSTALL_PREFIX`	/path/to/install (`/usr/local`)	Installation location.
`CMAKE_BUILD_TYPE`	Debug/**Release**/RelWithDebInfo/MinSizeRel	Compiler optimization levels.

Example: Enable Fortran, disable Python bindings and ZeroMQ functionality

```
$ cmake -DADIOS2_USE_Fortran=ON -DADIOS2_USE_Python=OFF -DADIOS2_USE_ZeroMQ=OFF ../ADIOS2
```

Notes:

To provide search paths to CMake for dependency searching:

- Use a `PackageName_ROOT` variable to provide the location of a specific package.
- Add an install prefix to the `CMAKE_PREFIX_PATH` which is searched for all packages.
- Both the `PackageName_ROOT` and `CMAKE_PREFIX_PATH` can be used as either environment variables or CMake variables (passed via -D), where the CMake variable takes prescedence.

```
# Several dependencies are installed under /opt/foo/bar and then a
# single dependency (HDF5 in this case) is installed in /opt/hdf5/1.13.0
$ export CMAKE_PREFIX_PATH=/opt/foo/bar
$ cmake -DHDF5_ROOT=/opt/hdf5/1.13.0 ../ADIOS2
```

Example: the following configuration will build, test and install under /opt/adios2/2.9.2 an optimized (Release) version of ADIOS2.

```
$ cd build
$ cmake -DADIOS2_USE_Fortran=ON -DCMAKE_INSTALL_PREFIX=/opt/adios2/2.9.2 -DCMAKE_BUILD_
↪Type=Release ../ADIOS2
$ make -j16
$ ctest
$ make install
```

For a fully configurable build script, click here.

3.3 Building on HPC Systems

1. **Modules:** Make sure all "module" dependencies are loaded and that minimum requirements are satisfied. Load the latest CMake module as many HPC systems default to an outdated version. Build with a C++11-compliant compiler, such as gcc >= 4.8.1, Intel >= 15, and PGI >= 15.
2. **Static/Dynamic build:** On Cray systems such as Titan, the default behavior is static linkage, thus CMake builds ADIOS2 creates the static library `libadios2.a` by default. Read the system documentation to enable dynamic compilation, usually by setting an environment variable such as `CRAYPE_LINK_TYPE=dynamic`. Click here for a fully configurable script example on OLCF systems.
3. **Big Endian and 32-bit systems:** ADIOS2 hasn't been tested on big endian and generally will not build on 32-bit systems. Please be aware before attempting to run.

4. **PGI compilers and C++11 support:** Version 15 of the PGI compiler is C++11 compliant. However it relies on the C++ standard library headers supplied by the system version of GCC, which may or may support all the C++11 features used in ADIOS2. On many systems (Titan at OLCF, for example) even though the PGI compiler supports C++11, the configured GCC and its headers do not (4.3.x on Cray Linux Environment, and v5 systems like Titan). To configure the PGI compiler to use a newer GCC, you must create a configuration file in your home directory that overrides the PGI compiler's default configuration. On Titan, the following steps will re-configure the PGI compiler to use GCC 6.3.0 provided by a module:

```
$ module load gcc/6.3.0
$ makelocalrc $(dirname $(which pgc++)) -gcc $(which gcc) -gpp $(which g++) -g77 $(which␣
↪gfortran) -o -net 1>${HOME}/.mypgirc 2>/dev/null
```

1. **Enabling RDMA for SST data transfers:** The SST engine in ADIOS2 is capable of using RDMA networks for transfering data between writer and reader cohorts, and generally this is the most performant data transport. However, SST depends upon libfabric to provide a generic interface to the underlying RDMA capabilities of the network, and properly configuring libfabric can be a difficult and error-prone task. HPC computing resources tend to be one-off custom resources with their own idiosyncracies, so this documentation cannot offer a definitive guide for every situation, but we can provide some general guidance and some recommendations for specific machines. If you are unable to configure ADIOS2 and libfabric to use RDMA, the best way to get help is to open an issue on the ADIOS2 github repository.

Pre-build concerns of note:

- on some HPC resources, libfabric is available as a loadable module. That should not be taken as an indication that that build of libfabric will work with SST, or even that it is compatible with the system upon which you find it. Your mileage may vary and you may have to build libfabric manually.
- libfabric itself depends upon other libraries like libibverbs and librdmacm. If you build libfabric with a package manager like spack, spack may build custom versions of those libraries as well, which may conflict with the system versions of those libraries.
- MPI on your HPC system may use libfabric itself, and linking your application with a different version of libfabric (or its dependent libraries) may result failure, possibly including opaque error messages from MPI.
- libfabric is structured in such a way that even if it is found during configuration, ADIOS *cannot* determine at compile time what providers will be present at run-time, or what their capabilities are. Therefore even a build that seems to successfully include libfabric and RDMA may be rejected at runtime as unable to support SST data transfer.

Configuration:

ADIOS2 uses the CMake find_package() functionality to locate libfabric. CMake will automatically search system libraries, but if you need to specify a libfabric location other than in a default system location you can add a "-DLIBFABRIC_ROOT=<directory>" argument to direct CMake to libfabric's location. If CMake finds libfabric, you should see the line "RDMA Transport for Staging: Available" near the end of the CMake output. This makes the RDMA DataTransport the default for SST data movement. (More information about SST engine parameters like *DataTransport* appears in the SST engine description.) If instead you see "RDMA Transport for Staging: Unconfigured", RDMA will not be available to SST.

Run-time:

Generally, if RDMA is configured and the libfabric provider has the capabilities that SST needs for RDMA data transfer, SST will use RDMA without external acknowledgement. However, if RDMA is configured, but the libfabric provider doesn't have the capabilities that SST needs, ADIOS will output an error : 'Warning: Preferred DataPlane "RDMA" not found.' If you see this warning in a situation where you expect RDMA to be used, enabling verbose debugging output from SST may provide more information. The SstVerbose environment variable can have values from 1 to 5, with 1 being minimal debugging info (such as confirming which DataTransport is being used), and 5 being the most detailed debugging information from all ranks.

3.4 Installing the ADIOS2 library and the C++ and C bindings

By default, ADIOS2 will build the C++11 `libadios2` library and the C and C++ bindings.

1. **Minimum requirements:**
   - A C++11 compliant compiler
   - An MPI C implementation on the syspath, or in a location identifiable by CMake.
2. **Linking** `make install` will copy the required headers and libraries into the directory specified by `CMAKE_INSTALL_PREFIX`:
   - Libraries:
     - `lib/libadios2.*` C++11 and C bindings
   - Headers:
     - `include/adios2.h` C++11 `namespace adios2`
     - `include/adios2_c.h` C prefix `adios2_`
   - Config file: run this command to get installation info
     - `bin/adios2-config`

3.5 Enabling the Python bindings

To enable the Python bindings in ADIOS2, based on PyBind11, make sure to follow these guidelines:

- **Minimum requirements:**
  - Python 2.7 and above.
  - *numpy*
  - *mpi4py*
- **Running:** If CMake enables Python compilation, an `adios2.so` library containing the Python module is generated in the build directory under `lib/pythonX.X/site-packages/`
  - make sure your `PYTHONPATH` environment variable contains the path to `adios2.so`.
  - make sure the Python interpreter is compatible with the version used for compilation via `python --version`.
  - Run the Python tests with `ctest -R Python`
  - Run helloBPWriter.py and helloBPTimeWriter.py via

```
$ mpirun -n 4 python helloBPWriter.py
$ python helloBPWriter.py
```

3.6 Enabling the Fortran bindings

1. **Minimum requirements:**
   - A Fortran 90 compliant compiler
   - A Fortran MPI implementation
2. **Linking the Fortran bindings:** `make install` will copy the required library and modules into the directory specified by `CMAKE_INSTALL_PREFIX`
   - Library (note that `libadios2` must also be linked) - `lib/libadios2_f.*` - `lib/libadios2.*`
   - Modules - `include/adios2/fortran/*.mod`
3. **Module adios2:** only module required to be used in an application `use adios`

3.7 Running Tests

ADIOS2 uses googletest to enable automatic testing after a CMake build. To run tests just type after building with make, run:

```
$ ctest
  or
$ make test
```

The following screen will appear providing information on the status of each finalized test:

```
Test project /home/wfg/workspace/build
Start  1: ADIOSInterfaceWriteTest.DefineVarChar1x10
1/46 Test  #1: ADIOSInterfaceWriteTest.DefineVarChar1x10 .....................  ␣
↪Passed    0.06 sec
Start  2: ADIOSInterfaceWriteTest.DefineVarShort1x10
2/46 Test  #2: ADIOSInterfaceWriteTest.DefineVarShort1x10 ....................  ␣
↪Passed    0.04 sec
Start  3: ADIOSInterfaceWriteTest.DefineVarInt1x10
3/46 Test  #3: ADIOSInterfaceWriteTest.DefineVarInt1x10 ......................  ␣
↪Passed    0.04 sec
Start  4: ADIOSInterfaceWriteTest.DefineVarLong1x10
...
128/130 Test #128: ADIOSZfpWrapper.UnsupportedCall ..................................
↪....   Passed    0.05 sec
    Start 129: ADIOSZfpWrapper.MissingMandatoryParameter
129/130 Test #129: ADIOSZfpWrapper.MissingMandatoryParameter .........................
↪....   Passed    0.05 sec
    Start 130: */TestManyVars.DontRedefineVars/*
130/130 Test #130: */TestManyVars.DontRedefineVars/* .................................
↪....   Passed    0.08 sec

100% tests passed, 0 tests failed out of 130

Total Test time (real) = 204.82 sec
```

3.8 Running Examples

ADIOS2 is best learned by examples.

A few very basic examples are described below:

Directory	Description
`ADIOS2/examples/hello`	very basic "hello world"-style examples for reading and writing *.bp* files.
`ADIOS2/examples/heatTransfer`	2D Poisson solver for transients in Fourier's model of heat transfer. Outputs `bp.dir` or HDF5.
`ADIOS2/examples/basics`	covers different `Variable` use cases classified by the dimension.

CHAPTER

FOUR

AS PACKAGE

4.1 Conda

Currently ADIOS 2 can be obtained from anaconda cloud:

- x86-64 and MacOS: williamfgc adios2-openmpi adios2-mpich adios2-nompi
- Multiple archs: conda-forge adios2

4.2 Spack

ADIOS 2 is packaged in Spack adios2 package

4.3 Docker

Docker images including building and installation of dependencies and ADIOS 2 containers for Ubuntu 20 and CentOS 7 can be found in: under the directory scripts/docker/

CHAPTER

FIVE

LINKING ADIOS 2

5.1 From CMake

ADIOS exports a CMake package configuration file that allows its targets to be directly imported into another CMake project via the `find_package` command:

```
cmake_minimum_required(VERSION 3.12)
project(MySimulation C CXX)

find_package(MPI REQUIRED)
find_package(ADIOS2 REQUIRED)
#...
add_library(my_library src1.cxx src2.cxx)
target_link_libraries(my_library PRIVATE adios2::cxx11_mpi MPI::MPI_C)
```

When configuring your project you can then set the `ADIOS2_ROOT` or `ADIOS2_DIR` environment variables to the install prefix of ADIOS2.

5.2 From non-CMake build systems

If you're not using CMake then you can manually get the necessary compile and link flags for your project using `adios2-config`:

```
$ /path/to/install-prefix/bin/adios2-config --cxxflags
ADIOS2_DIR: /path/to/install-prefix
-isystem /path/to/install-prefix/include -isystem /opt/ohpc/pub/mpi/openmpi3-gnu7/3.1.0/
↪include -pthread -std=gnu++11
$ /path/to/install-prefix/bin/adios2-config --cxxlibs
ADIOS2_DIR: /path/to/install-prefix
-Wl,-rpath,/path/to/install-prefix/lib:/opt/ohpc/pub/mpi/openmpi3-gnu7/3.1.0/lib /path/
↪to/install-prefix/lib/libadios2.so.2.4.0 -pthread -Wl,-rpath -Wl,/opt/ohpc/pub/mpi/
↪openmpi3-gnu7/3.1.0/lib -Wl,--enable-new-dtags -pthread /opt/ohpc/pub/mpi/openmpi3-
↪gnu7/3.1.0/lib/libmpi.so -Wl,-rpath-link,/path/to/install-prefix/lib
```

CHAPTER

SIX

INTERFACE COMPONENTS

6.1 Components Overview

Note: If you are doing simple tasks where performance is a non-critical aspect please go to the *High-Level APIs* section for a quick start. If you are an HPC application developer or you want to use ADIOS2 functionality in full please read this chapter.

The simple way to understand the big picture for the ADIOS2 unified user interface components is to map each class to the actual definition of the ADIOS acronym.

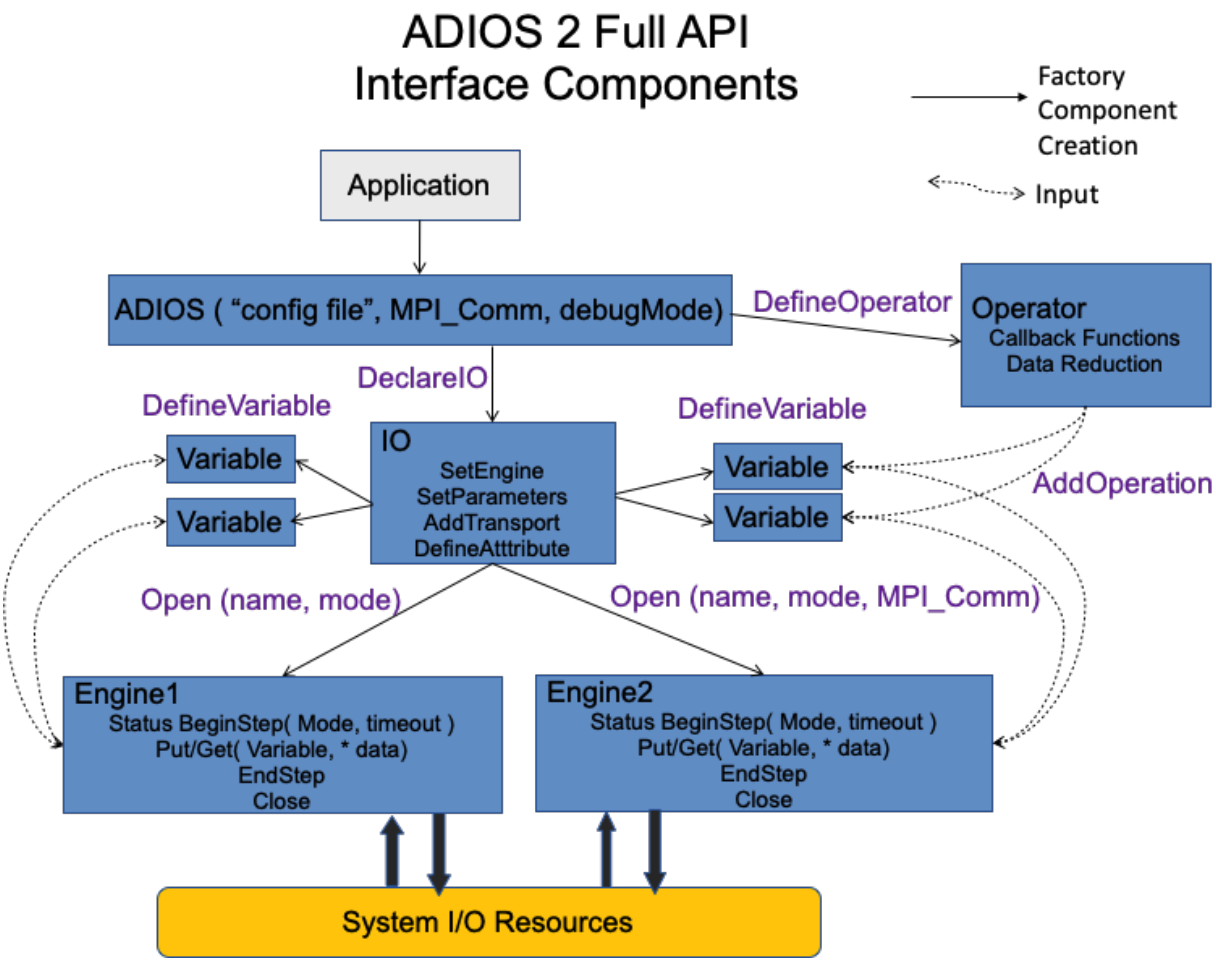
Component	Acronym	Function
ADIOS	ADaptable	Set MPI comm domain Set runtime settings Own other components
IO	I/O	Set engine Set variables/attributes Set compile-time settings
Engine	System	Execute heavy IO tasks Manage system resources

ADIOS2's public APIs are based on the natural choice for each supported language to represent each ADIOS2 components and its interaction with application datatypes. Thus,

Language	Component API	Application Data
C++(11/newer)	objects/member functions	pointers/references/std::vector
C	handler/functions	pointers
Fortran	handler/subroutines	arrays up to 6D
Python	objects/member functions	numpy arrays.

The following section provides a common overview to all languages based on the C++11 APIs. For each specific language go to the *Full APIs* section, but it's highly recommended to read this section as components map 1-to-1 in other languages.

The following figure depicts the components hierarchy from the application's point of view.



- **ADIOS: the ADIOS component is the starting point between an application and the ADIOS2 library. Applications provide:**
    1. the scope of the ADIOS object through the MPI communicator,
    2. an optional runtime configuration file (in XML format) to allow changing settings without recompiling.

  The ADIOS component serves as a factory of adaptable IO components. Each IO must have a unique name within the scope of the ADIOS class object that created them with the DeclareIO function.
- **IO: the IO component is the bridge between the application specific settings, transports. It also serves as a factory of:**
    1. Variables
    2. Attributes
    3. Engines
- **Variable**: Variables are the link between self-describing representation in the ADIOS2 library and data from applications. Variables are identified by unique names in the scope of the particular IO that created them. When the Engine API functions are called, a Variable must be provided along with the application data.
- **Attribute**: Attributes add extra information to the overall variables dataset defined in the IO class. They can be single or array values.
- **Engine**: Engines define the actual system executing the heavy IO tasks at Open, BeginStep, Put, Get, EndStep and Close. Due to polymorphism, new IO system solutions can be developed quickly reusing internal components

and reusing the same API. If IO.SetEngine is not called, the default engine is the binary-pack bp file reader and writer: **BPFile**.

- **Operator**: These define possible operations to be applied on adios2-managed data, for example, compression. This higher level abstraction is needed to provide support for callbacks, transforms, analytics, data models, etc. Any required task will be executed within the Engine. One or many operators can be associated with any of the adios2 objects or a group of them.

6.2 ADIOS

The `adios2::ADIOS` component is the initial contact point between an application and the ADIOS2 library. Applications can be classified as MPI and non-MPI based. We start by focusing on MPI applications as their non-MPI equivalent just removes the MPI communicator.

```
/** ADIOS class factory of IO class objects */
adios2::ADIOS adios("config.xml", MPI_COMM_WORLD);
```

This component is created by passing :

1. **Runtime config file** (optional): ADIOS2 xml runtime config file, see *Runtime Configuration Files*.
2. **MPI communicator** : which determines the scope of the ADIOS library components in an application.

`adios2::ADIOS` objects can be created in MPI and non-MPI (serial) mode. Optionally, a runtime configuration file can be passed to the constructor indicating the full file path, name and extension.

Constructors for MPI applications

```
/** Constructors */

// version that accepts an optional runtime adios2 config file
adios2::ADIOS(const std::string configFile,
              MPI_COMM mpiComm = MPI_COMM_SELF);

adios2::ADIOS(MPI_COMM mpiComm = MPI_COMM_SELF);

/** Examples */
adios2::ADIOS adios(MPI_COMM_WORLD);
adios2::ADIOS adios("config.xml", MPI_COMM_WORLD);
```

Constructors for non-MPI (serial) applications

```
/** Constructors */
adios2::ADIOS(const std::string configFile);

adios2::ADIOS();

/** Examples */
adios2::ADIOS adios("config.xml");
adios2::ADIOS adios; // Do not use () for empty constructor.
```

Factory of IO components: Multiple IO components (IO tasks) can be created from within the scope of an ADIOS object by calling the `DeclareIO` function:

```
/** Signature */
adios2::IO ADIOS::DeclareIO(const std::string ioName);

/** Examples */
adios2::IO bpWriter = adios.DeclareIO("BPWriter");
adios2::IO bpReader = adios.DeclareIO("BPReader");
```

This function returns a reference to an existing IO class object that lives inside the ADIOS object that created it. The `ioName` string must be unique; declaring two IO objects with the same name will throw an exception. IO names are used to identify IO components in the runtime configuration file, *Runtime Configuration Files*.

As shown in the diagram below, each resulting IO object is self-managed and independent, thus providing an adaptable way to perform different kinds of I/O operations. Users must be careful not to create conflicts between system level unique I/O identifiers: file names, IP address and port, MPI Send/Receive message rank and tag, etc.

Tip: The ADIOS component is the only one whose memory is owned by the application. Thus applications must decide on its scope. Any other component of the ADIOS2 API refers to a component that lives inside the ADIOS component(e.g. IO, Operator) or indirectly in the IO component(Variable, Engine)

6.3 IO

The `IO` component is the connection between how applications set up their input/output options by selecting an `Engine` and its specific parameters, subscribing variables to data, and setting supported transport modes to a particular `Engine`. Think of `IO` as a control panel for all the user-defined parameters that applications would like to fine tune. None of the `IO` operations are heavyweight until the `Open` function that generates an `Engine` is called. Its API allows

- generation of `Variable` and `Attribute` components containing information about the data in the input output process
- setting `Engine`-specific parameters and adding supported modes of transport
- generation of `Engine` objects to execute the actual IO tasks.

Note: If two different engine types are needed (*e.g.* `BPFile`, `SST`), you must define two `IO` objects. Also, at reading always define separate IOs to avoid `Variable` name clashes.

6.3.1 Setting a Particular Engine and its Parameters

Engines execute the heavy operations in ADIOS2. Each `IO` may select a type of `Engine` through the `SetEngine` function. If `SetEngine` is not called, then the `BPFile` engine is used.

```
/** Signature */
void adios2::IO::SetEngine( const std::string engineType );

/** Example */
bpIO.SetEngine("BPFile");
```

Each `Engine` allows the user to fine tune execution of buffering and output tasks via parameters passed to the `IO` object. These parameters are then propagated to the `Engine`. For a list of parameters allowed by each engine see *Available Engines*.

Note: `adios2::Params` is an alias to `std::map<std::string,std::string>` to pass parameters as key-value string pairs, which can be initialized with curly-brace initializer lists.

```
/** Signature */
/** Passing several parameters at once */
void SetParameters(const adios2:Params& parameters);
/** Passing one parameter key-value pair at a time */
void SetParameter(const std::string key, const std::string value);

/** Examples */
io.SetParameters( { {"Threads", "4"},
                    {"ProfilingUnits", "Milliseconds"},
                    {"MaxBufferSize","2Gb"},
                    {"BufferGrowthFactor", "1.5" }
                    {"FlushStepsCount", "5" }
                  } );
io.SetParameter( "Threads", "4" );
```

6.3.2 Adding Supported Transports with Parameters

The `AddTransport` function allows the user to specify how data is moved through the system, *e.g.* RDMA, wide-area networks, or files. It returns an `unsigned int` handler for each transport that can be used with the `Engine::Close` function at different times. `AddTransport` must provide library specific settings that the low-level system library interface allows.

```
/** Signature */
unsigned int AddTransport( const std::string transportType,
                           const adios2::Params& parameters );

/** Examples */
const unsigned int file1 = io.AddTransport( "File",
                                            { {"Library", "fstream"},
                                              {"Name","file1.bp" }
                                            } );

const unsigned int file2 = io.AddTransport( "File",
                                            { {"Library", "POSIX"},
                                              {"Name","file2.bp" }
                                            } );

const unsigned int wan = io.AddTransport( "WAN",
                                          { {"Library", "Zmq"},
                                            {"IP","127.0.0.1" },
                                            {"Port","80"}
                                          } );
```

6.3.3 Defining, Inquiring and Removing Variables and Attributes

The template functions `DefineVariable<T>` allows subscribing to data into ADIOS2 by returning a reference to a `Variable` class object whose scope is the same as the `IO` object that created it. The user must provide a unique name, the dimensions: MPI global: shape, MPI local: start and offset, optionally a flag indicating that dimensions are known to be constant, and a data pointer if defined in the application. Note: data is not passed at this stage. This is done by the `Engine` functions `Put` and `Get` for Variables. See the *Variable* section for supported types and shapes.

Tip: `adios2::Dims` is an alias to `std::vector<std::size_t>`, while `adios2::ConstantDims` is an alias to bool `true`. Use them for code clarity.

```
/** Signature */
adios2::Variable<T>
    DefineVariable<T>(const std::string name,
                      const adios2::Dims &shape = {}, // Shape of global object
                      const adios2::Dims &start = {}, // Where to begin writing
                      const adios2::Dims &count = {}, // Where to end writing
                      const bool constantDims = false);

/** Example */
/** global array of floats with constant dimensions */
adios2::Variable<float> varFloats =
    io.DefineVariable<float>("bpFloats",
                             {size * Nx},
                             {rank * Nx},
                             {Nx},
                             adios2::ConstantDims);
```

Attributes are extra-information associated with the current `IO` object. The function `DefineAttribute<T>` allows for defining single value and array attributes. Keep in mind that Attributes apply to all Engines created by the `IO` object and, unlike Variables which are passed to each `Engine` explicitly, their definition contains their actual data.

```
/** Signatures */

/** Single value */
adios2::Attribute<T> DefineAttribute(const std::string &name,
                                     const T &value);

/** Arrays */
adios2::Attribute<T> DefineAttribute(const std::string &name,
                                     const T *array,
                                     const size_t elements);
```

In situations in which a variable and attribute has been previously defined: 1) a variable/attribute reference goes out of scope, or 2) when reading from an incoming stream, the `IO` can inquire about the status of variables and attributes. If the inquired variable/attribute is not found, then the overloaded `bool()` operator of returns `false`.

```
/** Signature */
adios2::Variable<T> InquireVariable<T>(const std::string &name) noexcept;
adios2::Attribute<T> InquireAttribute<T>(const std::string &name) noexcept;

/** Example */
```

(continues on next page)

(continued from previous page)

```
adios2::Variable<float> varPressure = io.InquireVariable<float>("pressure");
if( varPressure ) // it exists
{
  ...
}
```

Note: `adios2::Variable` overloads `operator bool()` so that we can check for invalid states (e.g. variables haven't arrived in a stream, weren't previously defined, or weren't written in a file).

Caution: Since `InquireVariable` and `InquireAttribute` are template functions, both the name and type must match the data you are looking for.

6.3.4 Opening an Engine

The `IO::Open` function creates a new derived object of the abstract `Engine` class and returns a reference handler to the user. A particular `Engine` type is set to the current `IO` component with the `IO::SetEngine` function. Engine polymorphism is handled internally by the `IO` class, which allows subclassing future derived `Engine` types without changing the basic API.

`Engine` objects are created in various modes. The available modes are `adios2::Mode::Read`, `adios2::Mode::Write`, `adios2::Mode::Append`, `adios2::Mode::Sync`, `adios2::Mode::Deferred`, and `adios2::Mode::Undefined`.

```
/** Signatures */
/** Provide a new MPI communicator other than from ADIOS->IO->Engine */
adios2::Engine adios2::IO::Open(const std::string &name,
                                const adios2::Mode mode,
                                MPI_Comm mpiComm );

/** Reuse the MPI communicator from ADIOS->IO->Engine \n or non-MPI serial mode */
adios2::Engine adios2::IO::Open(const std::string &name,
                                const adios2::Mode mode);


/** Examples */

/** Engine derived class, spawned to start Write operations */
adios2::Engine bpWriter = io.Open("myVector.bp", adios2::Mode::Write);

/** Engine derived class, spawned to start Read operations on rank 0 */
if( rank == 0 )
{
    adios2::Engine bpReader = io.Open("myVector.bp",
                                      adios2::Mode::Read,
                                      MPI_COMM_SELF);
}
```

Caution: Always pass `MPI_COMM_SELF` if an `Engine` lives in only one MPI process. `Open` and `Close` are collective operations.

6.4 Variable

An `adios2::Variable` is the link between a piece of data coming from an application and its metadata. This component handles all application variables classified by data type and shape.

Each `IO` holds a set of Variables, and each `Variable` is identified with a unique name. They are created using the reference from `IO::DefineVariable<T>` or retrieved using the pointer from `IO::InquireVariable<T>` functions in *IO*.

6.4.1 Data Types

Only primitive types are supported in ADIOS2. Fixed-width types from <cinttypes> and <cstdint> should be preferred when writing portable code. ADIOS2 maps primitive types to equivalent fixed-width types (e.g. `int -> int32_t`). In C++, acceptable types `T` in `Variable<T>` along with their preferred fix-width equivalent in 64-bit platforms are given below:

```
Data types Variables supported by ADIOS2 Variable<T>

std::string (only used for global and local values, not arrays)
char                        -> int8_t or uint8_t depending on compiler flags
signed char                 -> int8_t
unsigned char               -> uint8_t
short                       -> int16_t
unsigned short              -> uint16_t
int                         -> int32_t
unsigned int                -> uint32_t
long int                    -> int32_t or int64_t (Linux)
long long int               -> int64_t
unsigned long int           -> uint32_t or uint64_t (Linux)
unsigned long long int      -> uint64_t
float                       -> always 32-bit = 4 bytes
double                      -> always 64-bit = 8 bytes
long double                 -> platform dependent
std::complex<float>         -> always  64-bit = 8 bytes = 2 * float
std::complex<double>        -> always 128-bit = 16 bytes = 2 * double
```

Tip: It's recommended to be consistent when using types for portability. If data is defined as a fixed-width integer, define variables in ADIOS2 using a fixed-width type, *e.g.* for `int32_t` data types use `DefineVariable<int32_t>`.

Note: C, Fortran APIs: the enum and parameter adios2_type_XXX only provides fixed-width types.

Note: Python APIs: use the equivalent fixed-width types from numpy. If `dtype` is not specified, ADIOS2 handles numpy defaults just fine as long as primitive types are passed.

6.4.2 Shapes

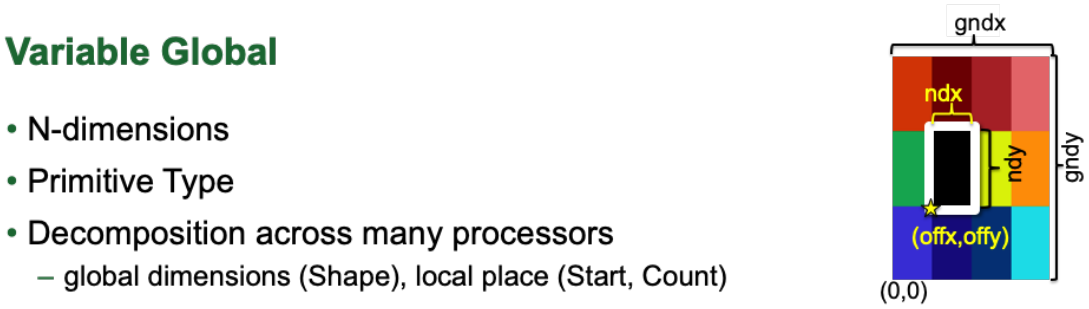
ADIOS2 is designed for MPI applications. Thus different application data shapes must be supported depending on their scope within a particular MPI communicator. The shape is defined at creation from the `IO` object by providing the dimensions: shape, start, count in the `IO::DefineVariable<T>`. The supported shapes are described below.

1. **Global Single Value**: Only a name is required for their definition. These variables are helpful for storing global information, preferably managed by only one MPI process, that may or may not change over steps: *e.g.* total number of particles, collective norm, number of nodes/cells, etc.

```
if( rank == 0 )
{
   adios2::Variable<uint32_t> varNodes = io.DefineVariable<uint32_t>("Nodes");
   adios2::Variable<std::string> varFlag = io.DefineVariable<std::string>(
↪"Nodes flag");
   // ...
   engine.Put( varNodes, nodes );
   engine.Put( varFlag, "increased" );
   // ...
}
```

Note: Variables of type `string` are defined just like global single values. Multidimensional strings are supported for fixed size strings through variables of type `char`.

2. **Global Array**: This is the most common shape used for storing data that lives in several MPI processes. The image below illustrates the definitions of the dimension components in a global array: shape, start, and count.



Warning: Be aware of data ordering in your language of choice (row-major or column-major) as depicted in the figure above. Data decomposition is done by the application, not by ADIOS2.

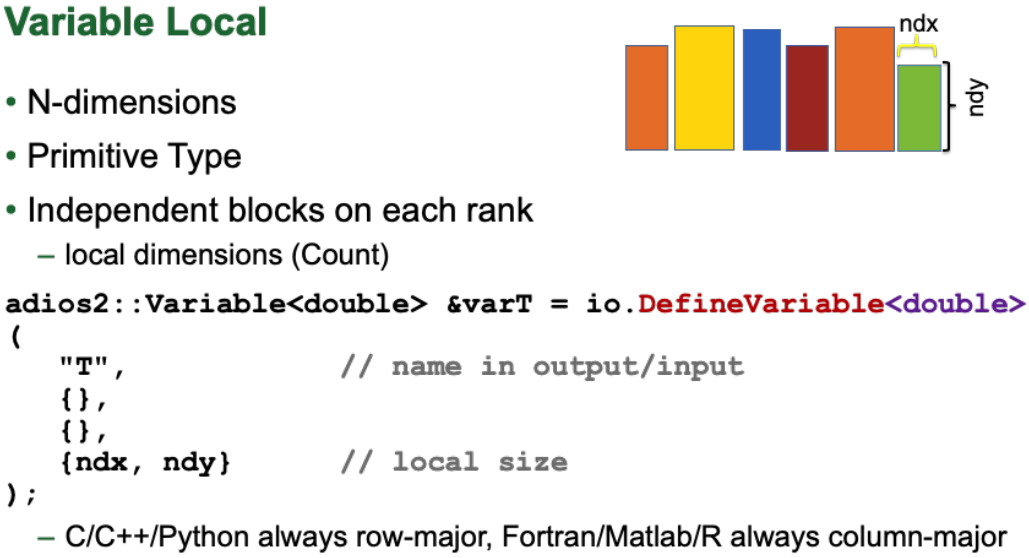
Start and Count local dimensions can be later modified with the `Variable::SetSelection` function if it is not a constant dimensions variable.

3. **Local Value**: Values that are local to the MPI process. They are defined by passing the `adios2::LocalValueDim` enum as follows:

```
adios2::Variable<int32_t> varProcessID =
      io.DefineVariable<int32_t>("ProcessID", {adios2::LocalValueDim})
//...
engine.Put<int32_t>(varProcessID, rank);
```

These values become visible on the reader as a single merged 1-D Global Array whose size is determined by the number of writer ranks.

4. **Local Array**: Arrays that are local to the MPI process. These are commonly used to write checkpoint-restart data. Reading, however, needs to be handled differently: each process' array has to be read separately, using `SetSelection` per rank. The size of each process selection should be discovered by the reading application by inquiring per-block size information of the variable, and allocate memory accordingly.



Note: Constants are not handled separately from step-varying values in ADIOS2. Simply write them only once from one rank.

5. **Joined Array**: Joined arrays are a variation of the Local Array described above. Where LocalArrays are only available to the reader via their block number, JoinedArrays are merged into a single global array whose global dimensions are determined by the sum of the contributions of each writer rank. Specifically: JoinedArrays are N-dimensional arrays where one (and only one) specific dimension is the Joined dimension. (The other dimensions must be constant and the same across all contributions.) When defining a Joined variable, one specifies a shape parameter that give the dimensionality of the array with the special constant `adios2::JoinedDim` in the dimension to be joined. Unlike a Global Array definition, the start parameter must be an empty Dims value. For example, the definition below defines a 2-D Joined array where the first dimension is the one along which blocks will be joined and the 2nd dimension is 5. Here this rank is contributing two rows to this array.

If each of N writer ranks were to declare a variable like this and do a single Put() in a timestep, the reader-side GlobalArray would have shape {2*N, 5} and all normal reader-side GlobalArray operations would be applicable to it.

Note: JoinedArrays are currently only supported by BP4 and BP5 engines, as well as the SST engine with BP5

marshalling.

6.5 Attribute

Attributes are extra information associated with a particular IO component. They can be thought of as a very simplified `Variable`, but with the goal of adding extra metadata. The most common use is the addition of human-readable metadata (*e.g.* `"experiment name"`, `"date and time"`, `"04,27,2017"`, or a schema).

Currently, ADIOS2 supports single values and arrays of primitive types (excluding `complex`<T>) for the template type in the `IO::DefineAttribute`<T> and `IO::InquireAttribute`<T> function (in C++).

The data types supported for ADIOS2 `Attributes` are

```
std::string
char
signed char
unsigned char
short
unsigned short
int
unsigned int
long int
long long int
unsigned long int
unsigned long long int
float
double
long double
```

The returned object (`DefineAttribute` or `InquireAttribute`) only serves the purpose to inspect the current `Attribute`<T> information within code.

6.6 Engine

The Engine abstraction component serves as the base interface to the actual IO systems executing the heavy-load tasks performed when producing and consuming data.

Engine functionality works around two concepts:

1. Variables are published (`Put`) and consumed (`Get`) in "steps" in either "File" random-access (all steps are available) or "Streaming" (steps are available as they are produced in a step-by-step fashion).
2. Variables are published (`Put`) and consumed (`Get`) using a "sync" or "deferred" (lazy evaluation) policy.

Caution: The ADIOS2 "step" is a logical abstraction that means different things depending on the application context. Examples: "time step", "iteration step", "inner loop step", or "interpolation step", "variable section", etc. It only indicates how the variables were passed into ADIOS2 (e.g. I/O steps) without the user having to index this information on their own.

Tip: Publishing and consuming data is a round-trip in ADIOS2. `Put` and `Get` APIs for write/append and read modes aim to be "symmetric", reusing functions, objects, and semantics as much as possible.

The rest of the section explains the important concepts.

6.6.1 BeginStep

Begins a logical step and return the status (via an enum) of the stream to be read/written. In streaming engines `BeginStep` is where the receiver tries to acquire a new step in the reading process. The full signature allows for a mode and timeout parameters. See *Supported Engines* for more information on what engine allows. A simplified signature allows each engine to pick reasonable defaults.

```
// Full signature
StepStatus BeginStep(const StepMode mode,
                     const float timeoutSeconds = -1.f);

// Simplified signature
StepStatus BeginStep();
```

6.6.2 EndStep

Ends logical step, flush to transports depending on IO parameters and engine default behavior.

Tip: To write portable code for a step-by-step access across ADIOS2 engines (file and streaming engines) use `BeginStep` and `EndStep`.

Danger: Accessing random steps in read mode (e.g. `Variable<T>::SetStepSelection` in file engines) will create a conflict with `BeginStep` and `EndStep` and will throw an exception. In file engines, data is either consumed in a random-access or step-by-step mode, but not both.

6.6.3 Close

Close current engine and underlying transports. An `Engine` object can't be used after this call.

6.6.4 Put: modes and memory contracts

`Put` publishes data in ADIOS2. It is unavailable unless the `Engine` is created in `Write` or `Append` mode.

The most common signature is the one that passes a `Variable<T>` object for the metadata, a `const` piece of contiguous memory for the data, and a mode for either `Deferred` (data may be collected at Put() or not until EndStep/PerformPuts/Close) or `Sync` (data is reusable immediately). This is the most common use case in applications.

1. Deferred (default) or Sync mode, data is contiguous memory

```
void Put(Variable<T> variable, const T* data, const adios2::Mode =␣
↪adios2::Mode::Deferred);
```

ADIOS2 Engines also provide direct access to their buffer memory. `Variable<T>::Span` is based on a subset of the upcoming C++20 std::span, which is a non-owning reference to a block of contiguous memory. Spans act as a 1D container meant to be filled out by the application. They provide the standard API of an STL container, providing `begin()` and `end()` iterators, `operator[]` and `at()`, as well as `data()` and `size()`.

`Variable<T>::Span` is helpful in situations in which temporaries are needed to create contiguous pieces of memory from non-contiguous pieces (*e.g.* tables, arrays without ghost-cells), or just to save memory as the returned `Variable<T>::Span` can be used for computation, thus avoiding an extra copy from user memory into the ADIOS2 buffer. `Variable<T>::Span` combines a hybrid `Sync` and `Deferred` mode, in which the initial value and memory allocations are `Sync`, while data population and metadata collection are done at EndStep/PerformPuts/Close. Memory contracts are explained later in this chapter followed by examples.

The following `Variable<T>::Span` signatures are available:

2. Return a span setting a default `T()` value into a default buffer

```
Variable<T>::Span Put(Variable<T> variable);
```

3. Return a span setting an initial fill value into a certain buffer. If span is not returned then the `fillValue` is fixed for that block.

```
Variable<T>::Span Put(Variable<T> variable, const size_t bufferID, const T␣
↪fillValue);
```

In summary, the following are the current Put signatures for publishing data in ADIOS 2:

1. `Deferred` (default) or `Sync` mode, data is contiguous memory put in an ADIOS2 buffer.

```
void Put(Variable<T> variable, const T* data, const adios2::Mode =␣
↪adios2::Mode::Deferred);
```

2. Return a span setting a default `T()` value into a default ADIOS2 buffer. If span is not returned then the default `T()` is fixed for that block (e.g. zeros).

```
Variable<T>::Span Put(Variable<T> variable);
```

3. Return a span setting an initial fill value into a certain buffer. If span is not returned then the `fillValue` is fixed for that block.

```
Variable<T>::Span Put(Variable<T> variable, const size_t bufferID, const T␣
↪fillValue);
```

The following table summarizes the memory contracts required by ADIOS2 engines between `Put` signatures and the data memory coming from an application:

Put	Data Memory	Contract
Deferred	Pointer Contents	do not modify until PerformPuts/EndStep/Close consumed at Put or PerformPuts/EndStep/Close
Sync	Pointer Contents	modify after Put consumed at Put
Span	Pointer Contents	modified by new Spans, updated span iterators/data consumed at PerformPuts/EndStep/Close

Note: In Fortran (array) and Python (numpy array) avoid operations that modify the internal structure of an array (size) to preserve the address.

Each `Engine` will give a concrete meaning to each functions signatures, but all of them must follow the same memory contracts to the "data pointer": the memory address itself, and the "data contents": memory bits (values).

1. **Put in Deferred or lazy evaluation mode (default)**: this is the preferred mode as it allows `Put` calls to be "grouped" before potential data transport at the first encounter of `PerformPuts`, `EndStep` or `Close`.

```
Put(variable, data);
Put(variable, data, adios2::Mode::Deferred);
```

Deferred memory contracts:

- "data pointer" do not modify (e.g. resize) until first call to `PerformPuts`, `EndStep` or `Close`.
- "data contents" may be consumed immediately or at first call to `PerformPuts`, `EndStep` or `Close`. Do not modify data contents after Put.

Usage:

```
// recommended use:
// set "data pointer" and "data contents"
// before Put
data[0] = 10;

// Puts data pointer into adios2 engine
// associated with current variable metadata
engine.Put(variable, data);

// Modifying data after Put(Deferred) may result in different
// results with different engines
// Any resize of data after Put(Deferred) may result in
// memory corruption or segmentation faults
data[1] = 10;

// "data contents" must not have been changed
// "data pointer" must be the same as in Put
engine.EndStep();
//engine.PerformPuts();
//engine.Close();

// now data pointer can be reused or modified
```

Tip: It's recommended practice to set all data contents before `Put` in deferred mode to minimize the risk of modifying the data pointer (not just the contents) before PerformPuts/EndStep/Close.

2. **Put in Sync mode**: this is the special case, data pointer becomes reusable right after `Put`. Only use it if absolutely necessary (*e.g.* memory bound application or out of scope data, temporary).

```
Put(variable, *data, adios2::Mode::Sync);
```

Sync memory contracts:

- "data pointer" and "data contents" can be modified after this call.

Usage:

```
// set "data pointer" and "data contents"
// before Put in Sync mode
data[0] = 10;

// Puts data pointer into adios2 engine
// associated with current variable metadata
engine.Put(variable, data, adios2::Mode::Sync);

// data pointer and contents can be reused
// in application
```

3. **Put returning a Span**: signature that allows access to adios2 internal buffer.

   Use cases:

   - population from non-contiguous memory structures
   - memory-bound applications

   Limitations:

   - does not allow operations (compression)
   - must keep engine and variables within scope of span usage

   Span memory contracts:

   - "data pointer" provided by the engine and returned by `span.data()`, might change with the generation of a new span. It follows iterator invalidation rules from std::vector. Use *span.data()* or iterators, *span.begin()*, *span.end()* to keep an updated data pointer.
   - span "data contents" are published at the first call to `PerformPuts`, `EndStep` or `Close`

   Usage:

```
// return a span into a block of memory
// set memory to default T()
adios2::Variable<int32_t>::Span span1 = Put(var1);

// just like with std::vector::data()
// iterator invalidation rules
// dataPtr might become invalid
// always use span1.data() directly
T* dataPtr = span1.data();

// set memory value to -1 in buffer 0
adios2::Variable<float>::Span span2 = Put(var2, 0, -1);

// not returning a span just sets a constant value
Put(var3);
Put(var4, 0, 2);

// fill span1
span1[0] = 0;
span1[1] = 1;
span1[2] = 2;

// fill span2
```

(continues on next page)

(continued from previous page)

```
span2[1] = 1;
span2[2] = 2;

// here collect all spans
// they become invalid
engine.EndStep();
//engine.PerformPuts();
//engine.Close();

// var1 = { 0, 1, 2 };
// var2 = { -1., 1., 2.};
// var3 = { 0, 0, 0};
// var4 = { 2, 2, 2};
```

The `data` fed to the `Put` function is assumed to be allocated on the Host (default mode). In order to use data allocated on the device, the memory space of the variable needs to be set to Cuda.

```
variable.SetMemorySpace(adios2::MemorySpace::CUDA);
engine.Put(variable, gpuData, mode);
```

Note: Only CUDA allocated buffers are supported for device data. Only the BP4 and BP5 engines are capable of receiving device allocated buffers.

6.6.5 PerformsPuts

Executes all pending `Put` calls in deferred mode and collects span data. Specifically this call copies Put(Deferred) data into internal ADIOS buffers, as if Put(Sync) had been used instead.

Note: This call allows the reuse of user buffers, but may negatively impact performance on some engines.

6.6.6 PerformsDataWrite

If supported by the engine, moves data from prior `Put` calls to disk

Note: Currently only supported by the BP5 file engine.

6.6.7 Get: modes and memory contracts

`Get` is the function for consuming data in ADIOS2. It is available when an Engine is created using `Read` mode at `IO::Open`. ADIOS2 `Put` and `Get` semantics are as symmetric as possible considering that they are opposite operations (*e.g.* `Put` passes `const T*`, while `Get` populates a non-const `T*`).

The `Get` signatures are described below.

1. `Deferred` (default) or `Sync` mode, data is contiguous pre-allocated memory:

```
Get(Variable<T> variable, const T* data, const adios2::Mode =␣
↪adios2::Mode::Deferred);
```

2. In this signature, `dataV` is automatically resized by ADIOS2 based on the `Variable` selection:

```
Get(Variable<T> variable, std::vector<T>& dataV, const adios2::Mode =␣
↪adios2::Mode::Deferred);
```

The following table summarizes the memory contracts required by ADIOS2 engines between `Get` signatures and the pre-allocated (except when using C++11 `std::vector`) data memory coming from an application:

Get	Data Memory	Contract
Deferred	Pointer Contents	do not modify until PerformPuts/EndStep/Close populated at Put or PerformPuts/EndStep/Close
Sync	Pointer Contents	modify after Put populated at Put

1. **Get in Deferred or lazy evaluation mode (default)**: this is the preferred mode as it allows `Get` calls to be "grouped" before potential data transport at the first encounter of `PerformPuts`, `EndStep` or `Close`.

```
Get(variable, data);
Get(variable, data, adios2::Mode::Deferred);
```

   Deferred memory contracts:

   - "data pointer": do not modify (e.g. resize) until first call to `PerformPuts`, `EndStep` or `Close`.
   - "data contents": populated at `Put`, or at first call to `PerformPuts`, `EndStep` or `Close`.

   Usage:`

```
std::vector<double> data;

// resize memory to expected size
data.resize(varBlockSize);
// valid if all memory is populated
// data.reserve(varBlockSize);

// Gets data pointer to adios2 engine
// associated with current variable metadata
engine.Get(variable, data.data() );

// optionally pass data std::vector
// leave resize to adios2
//engine.Get(variable, data);
```

(continues on next page)

(continued from previous page)

```
// "data pointer" must be the same as in Get
engine.EndStep();
// "data contents" are now ready
//engine.PerformPuts();
//engine.Close();

// now data pointer can be reused or modified
```

2. **Put in Sync mode**: this is the special case, data pointer becomes reusable right after Put. Only use it if absolutely necessary (*e.g.* memory bound application or out of scope data, temporary).

```
Get(variable, *data, adios2::Mode::Sync);
```

Sync memory contracts:

- "data pointer" and "data contents" can be modified after this call.

Usage:

```
.. code-block:: c++

std::vector<double> data;

// resize memory to expected size
data.resize(varBlockSize);
// valid if all memory is populated
// data.reserve(varBlockSize);

// Gets data pointer to adios2 engine
// associated with current variable metadata
engine.Get(variable, data.data() );

// "data contents" are ready
// "data pointer" can be reused by the application
```

Note: `Get` doesn't support returning spans.

6.6.8 PerformsGets

Executes all pending `Get` calls in deferred mode.

6.6.9 Engine usage example

The following example illustrates the basic API usage in write mode for data generated at each application step:

```
adios2::Engine engine = io.Open("file.bp", adios2::Mode::Write);

for( size_t i = 0; i < steps; ++i )
{
   // ... Application *data generation

   engine.BeginStep(); //next "logical" step for this application

   engine.Put(varT, dataT, adios2::Mode::Sync);
   // dataT memory already consumed by engine
   // Application can modify dataT address and contents

   // deferred functions return immediately (lazy evaluation),
   // dataU, dataV and dataW pointers and contents must not be modified
   // until PerformPuts, EndStep or Close.
   // 1st batch
   engine.Put(varU, dataU);
   engine.Put(varV, dataV);

   // in this case adios2::Mode::Deferred is redundant,
   // as this is the default option
   engine.Put(varW, dataW, adios2::Mode::Deferred);

   // effectively dataU, dataV, dataW are "deferred"
   // possibly until the first call to PerformPuts, EndStep or Close.
   // Application MUST NOT modify the data pointer (e.g. resize
   // memory) or change data contents.
   engine.PerformPuts();

   // dataU, dataV, dataW pointers/values can now be reused

   // ... Application modifies dataU, dataV, dataW

   //2nd batch
   dataU[0] = 10
   dataV[0] = 10
   dataW[0] = 10
   engine.Put(varU, dataU);
   engine.Put(varV, dataV);
   engine.Put(varW, dataW);
   // Application MUST NOT modify dataU, dataV and dataW pointers (e.g. resize),
   // Contents should also not be modified after Put() and before
   // PerformPuts() because ADIOS may access the data immediately
   // or not until PerformPuts(), depending upon the engine
```

(continues on next page)

(continued from previous page)

```
    engine.PerformPuts();

    // dataU, dataV, dataW pointers/values can now be reused

    // Puts a varP block of zeros
    adios2::Variable<double>::Span spanP = Put<double>(varP);

    // Not recommended mixing static pointers,
    // span follows
    // the same pointer/iterator invalidation
    // rules as std::vector
    T* p = spanP.data();

    // Puts a varMu block of 1e-6
    adios2::Variable<double>::Span spanMu = Put<double>(varMu, 0, 1e-6);

    // p might be invalidated
    // by a new span, use spanP.data() again
    foo(spanP.data());

    // Puts a varRho block with a constant value of 1.225
    Put<double>(varMu, 0, 1.225);

    // it's preferable to start modifying spans
    // after all of them are created
    foo(spanP.data());
    bar(spanMu.begin(), spanMu.end());


    engine.EndStep();
    // spanP, spanMu are consumed by the library
    // end of current logical step,
    // default behavior: transport data
}

engine.Close();
// engine is unreachable and all data should be transported
...
```

Tip: Prefer default `Deferred` (lazy evaluation) functions as they have the potential to group several variables with the trade-off of not being able to reuse the pointers memory space until `EndStep`, `PerformPuts`, `PerformGets`, or `Close`. Only use `Sync` if you really have to (*e.g.* reuse memory space from pointer). ADIOS2 prefers a step-based IO in which everything is known ahead of time when writing an entire step.

Danger: The default behavior of ADIOS2 `Put` and `Get` calls IS NOT synchronized, but rather deferred. It's actually the opposite of `MPI_Put` and more like `MPI_rPut`. Do not assume the data pointer is usable after a `Put` and `Get`, before `EndStep`, `Close` or the corresponding `PerformPuts`/`PerformGets`. Avoid using temporaries, r-values, and out-of-scope variables in `Deferred` mode. Use `adios2::Mode::Sync` in these cases.

6.6.10 Available Engines

A particular engine is set within the `IO` object that creates it with the `IO::SetEngine` function in a case insensitive manner. If the `SetEngine` function is not invoked the default engine is the `BPFile`.

Application	Engine	Description
File	BP5 HDF5	DEFAULT write/read ADIOS2 native bp files write/read interoperability with HDF5 files
Wide-Area-Network (WAN)	DataMan	write/read TCP/IP streams
Staging	SST	write/read to a "staging" area: *e.g.* RDMA

`Engine` polymorphism has two goals:

1. Each `Engine` implements an orthogonal IO scenario targeting a use case (e.g. Files, WAN, InSitu MPI, etc) using a simple, unified API.

2. Allow developers to build their own custom system solution based on their particular requirements in the own playground space. Reusable toolkit objects are available inside ADIOS2 for common tasks: bp buffering, transport management, transports, etc.

A class that extends `Engine` must be thought of as a solution to a range of IO applications. Each engine must provide a list of supported parameters, set in the IO object creating this engine using `IO::SetParameters`, and supported transports (and their parameters) in `IO::AddTransport`. Each Engine's particular options are documented in *Supported Engines*.

6.7 Operator

The Operator abstraction allows ADIOS2 to act upon the user application data, either from a `adios2::Variable` or a set of Variables in an `adios2::IO` object. Current supported operations are:

1. Data compression/decompression, lossy and lossless.

2. Callback functions (C++11 bindings only) supported by specific engines

ADIOS2 enables the use of third-party libraries to execute these tasks.

Operators can be attached onto a variable in two modes: private or shared. In most situations, it is recommended to add an operator as a private one, which means it is owned by a certain variable. A simple example code is as follows.

```
#include <vector>
#include <adios2.h>
int main(int argc, char *argv[])
{
    std::vector<double> myData = {
        0.0001, 1.0001, 2.0001, 3.0001, 4.0001, 5.0001, 6.0001, 7.0001, 8.0001, 9.0001,
        1.0001, 2.0001, 3.0001, 4.0001, 5.0001, 6.0001, 7.0001, 8.0001, 9.0001, 8.0001,
        2.0001, 3.0001, 4.0001, 5.0001, 6.0001, 7.0001, 8.0001, 9.0001, 8.0001, 7.0001,
        3.0001, 4.0001, 5.0001, 6.0001, 7.0001, 8.0001, 9.0001, 8.0001, 7.0001, 6.0001,
        4.0001, 5.0001, 6.0001, 7.0001, 8.0001, 9.0001, 8.0001, 7.0001, 6.0001, 5.0001,
        5.0001, 6.0001, 7.0001, 8.0001, 9.0001, 8.0001, 7.0001, 6.0001, 5.0001, 4.0001,
        6.0001, 7.0001, 8.0001, 9.0001, 8.0001, 7.0001, 6.0001, 5.0001, 4.0001, 3.0001,
        7.0001, 8.0001, 9.0001, 8.0001, 7.0001, 6.0001, 5.0001, 4.0001, 3.0001, 2.0001,
        8.0001, 9.0001, 8.0001, 7.0001, 6.0001, 5.0001, 4.0001, 3.0001, 2.0001, 1.0001,
```

(continues on next page)

(continued from previous page)

```
        9.0001, 8.0001, 7.0001, 6.0001, 5.0001, 4.0001, 3.0001, 2.0001, 1.0001, 0.0001,
    };
    adios2::ADIOS adios;
    auto io = adios.DeclareIO("TestIO");
    auto varDouble = io.DefineVariable<double>("varDouble", {10,10}, {0,0}, {10,10},␣
↪adios2::ConstantDims);

    // add operator
    varDouble.AddOperation("mgard",{{"accuracy","0.01"}});
    // end add operator

    auto engine = io.Open("hello.bp", adios2::Mode::Write);
    engine.Put<double>(varDouble, myData.data());
    engine.Close();
    return 0;
}
```

For users who need to attach a single operator onto multiple variables, a shared operator can be defined using the adios2::ADIOS object, and then attached to multiple variables using the reference to the operator object. Note that in this mode, all variables sharing this operator will also share the same configuration map. It should be only used when a number of variables need *exactly* the same operation. In real world use cases this is rarely seen, so please use this mode with caution.

```
#include <vector>
#include <adios2.h>
int main(int argc, char *argv[])
{
    std::vector<double> myData = {
        0.0001, 1.0001, 2.0001, 3.0001, 4.0001, 5.0001, 6.0001, 7.0001, 8.0001, 9.0001,
        1.0001, 2.0001, 3.0001, 4.0001, 5.0001, 6.0001, 7.0001, 8.0001, 9.0001, 8.0001,
        2.0001, 3.0001, 4.0001, 5.0001, 6.0001, 7.0001, 8.0001, 9.0001, 8.0001, 7.0001,
        3.0001, 4.0001, 5.0001, 6.0001, 7.0001, 8.0001, 9.0001, 8.0001, 7.0001, 6.0001,
        4.0001, 5.0001, 6.0001, 7.0001, 8.0001, 9.0001, 8.0001, 7.0001, 6.0001, 5.0001,
        5.0001, 6.0001, 7.0001, 8.0001, 9.0001, 8.0001, 7.0001, 6.0001, 5.0001, 4.0001,
        6.0001, 7.0001, 8.0001, 9.0001, 8.0001, 7.0001, 6.0001, 5.0001, 4.0001, 3.0001,
        7.0001, 8.0001, 9.0001, 8.0001, 7.0001, 6.0001, 5.0001, 4.0001, 3.0001, 2.0001,
        8.0001, 9.0001, 8.0001, 7.0001, 6.0001, 5.0001, 4.0001, 3.0001, 2.0001, 1.0001,
        9.0001, 8.0001, 7.0001, 6.0001, 5.0001, 4.0001, 3.0001, 2.0001, 1.0001, 0.0001,
    };
    adios2::ADIOS adios;
    auto io = adios.DeclareIO("TestIO");
    auto varDouble = io.DefineVariable<double>("varDouble", {10,10}, {0,0}, {10,10},␣
↪adios2::ConstantDims);

    // define operator
    auto op = adios.DefineOperator("SharedCompressor","mgard",{{"accuracy","0.01"}});
    // add operator
    varDouble.AddOperation(op);
    // end add operator

    auto engine = io.Open("hello.bp", adios2::Mode::Write);
    engine.Put<double>(varDouble, myData.data());
```

(continues on next page)

(continued from previous page)

```
    engine.Close();
    return 0;
}
```

Warning: Make sure your ADIOS2 library installation used for writing and reading was linked with a compatible version of a third-party dependency when working with operators. ADIOS2 will issue an exception if an operator library dependency is missing.

6.8 Runtime Configuration Files

ADIOS2 supports passing an optional runtime configuration file to the *ADIOS* component constructor (`adios2_init` in C, Fortran).

This file contains key-value pairs equivalent to the compile time `IO::SetParameters` (`adios2_set_parameter` in C, Fortran), and `IO::AddTransport` (`adios2_set_transport_parameter` in C, Fortran).

Each `Engine` and `Operator` must provide a set of available parameters as described in the *Supported Engines* section. Prior to version v2.6.0 only XML is supported; v2.6.0 and later support both XML and YAML.

Warning: Configuration files must have the corresponding format extension `.xml`, `.yaml`: `config.xml`, `config.yaml`, etc.

6.8.1 XML

```
<?xml version="1.0"?>
<adios-config>
  <io name="IONAME_1">

    <engine type="ENGINE_TYPE">

      <!-- Equivalent to IO::SetParameters-->
      <parameter key="KEY_1" value="VALUE_1"/>
      <parameter key="KEY_2" value="VALUE_2"/>
      <!-- ... -->
      <parameter key="KEY_N" value="VALUE_N"/>

    </engine>

    <!-- Equivalent to IO::AddTransport -->
    <transport type="TRANSPORT_TYPE">
      <!-- Equivalent to IO::SetParameters-->
      <parameter key="KEY_1" value="VALUE_1"/>
      <parameter key="KEY_2" value="VALUE_2"/>
      <!-- ... -->
      <parameter key="KEY_N" value="VALUE_N"/>
    </transport>
  </io>
```

(continues on next page)

(continued from previous page)

```
  <io name="IONAME_2">
    <!-- ... -->
  </io>
</adios-config>
```

6.8.2 YAML

Starting with v2.6.0, ADIOS supports YAML configuration files. The syntax follows strict use of the YAML node keywords mapping to the ADIOS2 components hierarchy. If a keyword is unknown ADIOS2 simply ignores it. For an example file refer to adios2 config file example in our repo.

```
---
# adios2 config.yaml
# IO YAML Sequence (-) Nodes to allow for multiple IO nodes
# IO name referred in code with DeclareIO is mandatory

- IO: "IOName"

  Engine:
     # If Type is missing or commented out, default Engine is picked up
     Type: "BP5"
     # optional engine parameters
     key1: value1
     key2: value2
     key3: value3

  Variables:

      # Variable Name is Mandatory
    - Variable: "VariableName1"
      Operations:
          # Operation Type is mandatory (zfp, sz, etc.)
        - Type: operatorType
          key1: value1
          key2: value2

    - Variable: "VariableName2"
      Operations:
          # Operations sequence of maps
        - {Type: operatorType, key1: value1}
        - {Type: z-checker, key1: value1, key2: value2}

  Transports:
      # Transport sequence of maps
    - {Type: file, Library: fstream}
    - {Type: rdma, Library: ibverbs}

  ...
```

Caution: YAML is case sensitive, make sure the node identifiers follow strictly the keywords: IO, Engine, Variables, Variable, Operations, Transports, Type.

Tip: Run a YAML validator or use a YAML editor to make sure the provided file is YAML compatible.

CHAPTER

SEVEN

SUPPORTED VIRTUAL ENGINE NAMES

This section provides a description of the Virtual Engines that can be used to set up an actual Engine with specific parameters. These virtual names are used for beginner users to simplify the selection of an engine and its parameters. The following I/O uses cases are supported by virtual engine names:

1. `File`: File I/O (Default engine).

   This sets up the I/O for files. If the file name passed in Open() ends with ".bp", then the BP5 engine will be used starting in v2.9.0. If it ends with ".h5", the HDF5 engine will be used. For old .bp files (BP version 3 format), the BP3 engine will be used for reading (v2.4.0 and below).

2. `FileStream`: Online processing via files.

   This allows a Consumer to concurrently read the data while the Producer is writing new output steps into it. The Consumer will wait for the appearance of the file itself in Open() (for up to one hour) and wait for the appearance of new steps in the file (in BeginStep() up to the specificed timeout in that function).

3. `InSituAnalysis`: Streaming data to another application.

   This sets up ADIOS for transferring data from a Producer to a Consumer application. The Producer and Consumer are synchronized at Open(). The Consumer will receive every single output step from the Producer, therefore, the Producer will block on output if the Consumer is slow.

4. `InSituVisualization`:: Streaming data to another application without waiting for consumption.

   This sets up ADIOS for transferring data from a Producer to a Consumer without ever blocking the Producer. The Producer will throw away all output steps that are not immediately requested by a Consumer. It will also not wait for a Consumer to connect. This kind of streaming is great for an interactive visualization session where the user wants to see the most current state of the application.

5. `CodeCoupling`:: Streaming data between two applications for code coupling.

   Producer and Consumer are waiting for each other in Open() and every step must be consumed. Currently, this is the same as in situ analysis.

These virtual engine names are used to select a specific engine and its parameters. In practice, after selecting the virtual engine name, one can modify the settings by adding/overwriting parameters. Eventually, a seasoned user would use the actual Engine names and parameterize it for the specific run.

7.1 Virtual Engine Setups

These are the actual settings in ADIOS when a virtual engine is selected. The parameters below can be modified before the Open call.

1. `File`. Refer to the parameter settings for these engines of `BP5`, `BP4`, `BP3` and `HDF5` engines earlier in this section.
2. `FileStream`. The engine is `BP5`. The parameters are set to:

Key	**Value Format**	**Default** and Examples
OpenTimeoutSecs	float	**3600** (wait for up to an hour)
BeginStepPollingFrequencySecs	float	**1** (poll the file system with 1 second frequency

3. `InSituAnalysis`. The engine is `SST`. The parameters are set to:

Key	**Value Format**	**Default** and Examples
RendezvousReaderCount	integer	**1** (Producer waits for the Consumer in Open)
QueueLimit	integer	**1** (only buffer one step)
QueueFullPolicy	string	**Block** (wait for the Consumer to get every step)
FirstTimestepPrecious	bool	false (SST default)
AlwaysProvideLatestTimestep	bool	false (SST default)

4. `InSituVisualization`. The engine is `SST`. The parameters are set to:

Key	**Value Format**	**Default** and Examples
RendezvousReaderCount	integer	**0** (Producer does NOT wait for Consumer in Open)
QueueLimit	integer	**3** (buffer first step + last two steps)
QueueFullPolicy	string	**Discard** (slow Consumer will miss out on steps)
FirstTimestepPrecious	bool	**true** (First step is kept around for late Consumers)
AlwaysProvideLatestTimestep	bool	false (SST default)

5. `Code Coupling`. The engine is `SST`. The parameters are set to:

Key	**Value Format**	**Default** and Examples
RendezvousReaderCount	integer	**1** (Producer waits for the Consumer in Open)
QueueLimit	integer	**1** (only buffer one step)
QueueFullPolicy	string	**Block** (wait for the Consumer to get every step)
FirstTimestepPrecious	bool	false (SST default)
AlwaysProvideLatestTimestep	bool	false (SST default)

CHAPTER

EIGHT

SUPPORTED ENGINES

This section provides a description of the *Available Engines* in ADIOS2 and their specific parameters to allow extra-control from the user. Parameters are passed in key-value pairs for:

1. Engine specific parameters
2. Engine supported transports and parameters

Parameters are passed at:

1. Compile time `IO::SetParameters` (`adios2_set_parameter` in C, Fortran)
2. Compile time `IO::AddTransport` (`adios2_set_transport_parameter` in C, Fortran)
3. *Runtime Configuration Files* in the *ADIOS* component.

8.1 BP5

The BP5 Engine writes and reads files in ADIOS2 native binary-pack (bp version 5) format. This was a new format for ADIOS 2.8, improving on the metadata operations and the memory consumption of the older BP4/BP3 formats. BP5 is the default file format as of ADIOS 2.9. As compared to the older format, BP5 provides three main advantages:

- **Lower memory** consumption. Deferred Puts will use user buffer for I/O wherever possible thus saving on a memory copy. Aggregation uses a fixed-size shared-memory segment on each compute node instead of using MPI to send data from one process to another. Memory consumption can get close to half of BP4 in some cases.
- **Faster metadata** management improves write/read performance where hundreds or more variables are added to the output.
- Improved functionality around **appending** many output steps into the same file. Better performance than writing new files each step. Restart can append to an existing series by truncating unwanted steps. Readers can filter out unwanted steps to only see and process a limited set of steps. Just like as in BP4, existing steps cannot be corrupted by appending new steps.

In 2.8 BP5 was a brand new file format and engine. It still does **NOT** support some functionality of BP4:

- **Burst buffer support** for writing data.

BP5 files have the following structure given a "name" string passed as the first argument of `IO::Open`:

```
io.SetEngine("BP5");
adios2::Engine bpFile = io.Open("name", adios2::Mode::Write);
```

will generate:

```
% BP5 datasets are always a directory
name.bp/

% data and metadata files
name.bp/
        data.0
        data.1
        ...
        data.M
        md.0
        md.idx
        mmd.0
```

Note: BP5 file names are compatible with the Unix (/) and Windows (\\) file system naming convention for directories and files.

Note: BP5 has an `mmd.0` file in the directory, which BP4 does not have.

This engine allows the user to fine tune the buffering operations through the following optional parameters:

1. Streaming through file
    1. **OpenTimeoutSecs**: (Streaming mode) Reader may want to wait for the creation of the file in `io.Open()`. By default the Open() function returns with an error if file is not found.
    2. **BeginStepPollingFrequencySecs**: (Streaming mode) Reader can set how frequently to check the file (and file system) for new steps. Default is 1 seconds which may be stressful for the file system and unnecessary for the application.
2. Aggregation
    1. **AggregationType**: *TwoLevelShm*, *EveryoneWritesSerial* and *EveryoneWrites* are three aggregation strategies. See *Aggregation in BP5*. The default is *TwoLevelShm*.
    2. **NumAggregators**: The number of processes that will ever write data directly to storage. The default is set to the number of compute nodes the application is running on (i.e. one process per compute node). TwoLevelShm will select a fixed number of processes *per compute-node* to get close to the intention of the user but does not guarantee the exact number of aggregators.
    3. **AggregatorRatio**: An alternative option to NumAggregators to pick every nth process as aggregator. The number of aggregators will be automatically kept to be within 1 and total number of processes no matter what bad number is supplied here. Moreover, TwoLevelShm will select an fixed number of processes *per compute-node* to get close to the intention of this ratio but does not guarantee the exact number of aggregators.
    4. **NumSubFiles**: The number of data files to write to in the *.bp/* directory. Only used by *TwoLevelShm* aggregator, where the number of files can be smaller then the number of aggregators. The default is set to *NumAggregators*.
    5. **StripeSize**: The data blocks of different processes are aligned to this size (default is 4096 bytes) in the files. Its purpose is to avoid multiple processes to write to the same file system block and potentially slow down the write.
    6. **MaxShmSize**: Upper limit for how much shared memory an aggregator process in *TwoLevelShm* can allocate. For optimum performance, this should be at least *2xM +1KB* where *M* is the maximum size any

process writes in a single step. However, there is no point in allowing for more than 4GB. The default is 4GB.

3. Buffering
    1. **BufferVType**: *chunk* or *malloc*, default is chunking. Chunking maintains the buffer as a list of memory blocks, either ADIOS-owned for sync-ed Puts and small Puts, and user-owned pointers of deferred Puts. Malloc maintains a single memory block and extends it (reallocates) whenever more data is buffered. Chunking incurs extra cost in I/O by having to write data in chunks (multiple write system calls), which can be helped by increasing *BufferChunkSize* and *MinDeferredSize*. Malloc incurs extra cost by reallocating memory whenever more data is buffered (by Put()), which can be helped by increasing *InitialBufferSize*.
    2. **BufferChunkSize**: (for *chunk* buffer type) The size of each memory buffer chunk, default is 128MB but it is worth increasing up to 2147381248 (a bit less than 2GB) if possible for maximum write performance.
    3. **MinDeferredSize**: (for *chunk* buffer type) Small user variables are always buffered, default is 4MB.
    4. **InitialBufferSize**: (for *malloc* buffer type) initial memory provided for buffering (default and minimum is 16Kb). To avoid reallocations, it is worth increasing this size to the expected maximum total size of data any process would write in any step (not counting deferred Puts).
    5. **GrowthFactor**: (for *malloc* buffer type) exponential growth factor for initial buffer > 1, default = 1.05.
4. Managing steps
    1. **AppendAfterSteps**: BP5 enables overwriting some existing steps by opening in *adios2::Mode::Append* mode and specifying how many existing steps to keep. Default value is MAX_INT, so it always appends after the last step. -1 would achieve the same thing. If you have 10 steps in the file,
        - value 0 means starting from the beginning, truncating all existing data
        - value 1 means appending after the first step, so overwrite 2,3…10
        - value 10 means appending after all existing steps
        - value >10 means the same, append after all existing steps (gaps in steps are impossible)
        - -1 means appending after the last step, i.e. same as 10 or higher
        - -2 means removing the last step, i.e. starting from the 10th
        - -11 (and <-11) means truncating all existing data
    2. **SelectSteps**: BP5 reading allows for only seeing selected steps. This is a string of space-separated list of range definitions in the form of "start:end:step". Indexing starts from 0. If 'end' is 'n' or 'N', then it is an unlimited range expression. Range definitions are adding up. Note that in the reading functions, counting the steps is *always 0* to *s-1* where *s* steps are presented, so even after applying this selection, the selected steps are presented as *0* to *s-1*. Examples:
        - "0 6 3 2" selects four steps indexed 0,2,3 and 6 (presented in reading as 0,1,2,3)
        - "1:5" selects 5 consecutive steps, skipping step 0, and starting from 1
        - "2:n" selects all steps from step 2
        - "0:n:2" selects every other steps from the beginning (0,2,4,6…)
        - "0:n:3 10:n:5" selects every third step from the beginning and additionally every fifth steps from step 10.
5. Asynchronous writing I/O
    1. **AsyncOpen**: *true/false* Call the open function asynchronously. It decreases I/O overhead when creating lots of subfiles (*NumAggregators* is large) and one calls *io.Open()* well ahead of the first write step. Only implemented for writing. Default is *true*.

2. **AsyncWrite**: *true/false* Perform data writing operations asynchronously after *EndStep()*. Default is *false*. If the application calls *EnterComputationBlock()/ExitComputationBlock()* to indicate phases where no communication is happening, ADIOS will try to perform all data writing during those phases, otherwise it will write immediately and eagerly after *EndStep()*.

6. Direct I/O. Experimental, see discussion on GitHub.
    1. **DirectIO**: Turn on O_DIRECT when using POSIX transport. Do not use this on parallel file systems.
    2. **DirectIOAlignOffset**: Alignment for file offsets. Default is 512 which is usually
    3. **DirectIOAlignBuffer**: Alignment for memory pointers. Default is to be same as *DirectIOAlignOffset*.
7. Miscellaneous
    1. **StatsLevel**: 1 turns on *Min/Max* calculation for every variable, 0 turns this off. Default is 1. It has some cost to generate this metadata so it can be turned off if there is no need for this information.
    2. **MaxOpenFilesAtOnce**: Specify how many subfiles a process can keep open at once. Default is unlimited. If a dataset contains more subfiles than how many open file descriptors the system allows (see *ulimit -n*) then one can either try to raise that system limit (set it with *ulimit -n*), or set this parameter to force the reader to close some subfiles to stay within the limits.
    3. **Threads**: Read side: Specify how many threads one process can use to speed up reading. The default value is *0*, to let the engine estimate the number of threads based on how many processes are running on the compute node and how many hardware threads are available on the compute node but it will use maximum 16 threads. Value *1* forces the engine to read everything within the main thread of the process. Other values specify the exact number of threads the engine can use. Although multithreaded reading works in a single *Get(adios2::Mode::Sync)* call if the read selection spans multiple data blocks in the file, the best parallelization is achieved by using deferred mode and reading everything in *PerformGets()/EndStep()*.

Key	Value Format	**Default** and Examples
OpenTimeoutSecs	float	**0** for *ReadRandomAccess* mode, **3600** for *Read* mode, `10.0`, `5`
BeginStepPollingFrequency-Secs	float	**1**, 10.0
AggregationType	string	**TwoLevelShm**, EveryoneWritesSerial, EveryoneWrites
NumAggregators	integer >= 1	**0 (one file per compute node)**
AggregatorRatio	integer >= 1	not used unless set
NumSubFiles	integer >= 1	**=NumAggregators**, only used when *AggregationType=TwoLevelShm*
StripeSize	integer+units	**4KB**
MaxShmSize	integer+units	**4294762496**
BufferVType	string	**chunk**, malloc
BufferChunkSize	integer+units	**128MB**, worth increasing up to min(2GB, data-size/process/step)
MinDeferredSize	integer+units	**4MB**
InitialBufferSize	float+units >= 16Kb	**16Kb**, 10Mb, 0.5Gb
GrowthFactor	float > 1	**1.05**, 1.01, 1.5, 2
AppendAfterSteps	integer >= 0	**INT_MAX**
SelectSteps	string	"0 6 3 2", "1:5", "0:n:3 10:n:5"
AsyncOpen	string On/Off	**On**, Off, true, false
AsyncWrite	string On/Off	**Off**, On, true, false
DirectIO	string On/Off	**Off**, On, true, false
DirectIOAlignOffset	integer >= 0	**512**
DirectIOAlignBuffer	integer >= 0	set to DirectIOAlignOffset if unset
StatsLevel	integer, 0 or 1	**1**, 0
MaxOpenFilesAtOnce	integer >= 0	**UINT_MAX**, 1024, 1
Threads	integer >= 0	**0**, 1, 32

Only file transport types are supported. Optional parameters for `IO::AddTransport` or in runtime config file transport field:

Transport type: File

Key	Value Format	**Default** and Examples
Library	string	**POSIX** (UNIX), **FStream** (Windows), stdio, IME

The IME transport directly reads and writes files stored on DDN's IME burst buffer using the IME native API. To use the IME transport, IME must be avaiable on the target system and ADIOS2 needs to be configured with `ADIOS2_USE_IME`. By default, data written to the IME is automatically flushed to the parallel filesystem at every `EndStep()` call. You can disable this automatic flush by setting the transport parameter `SyncToPFS` to `OFF`.

8.2 BP4

The BP4 Engine writes and reads files in ADIOS2 native binary-pack (bp version 4) format. This was a new format for ADIOS 2.5 and improved on the metadata operations of the older BP3 format. Compared to the older format, BP4 provides three main advantages:

- Fast and safe **appending** of multiple output steps into the same file. Better performance than writing new files each step. Existing steps cannot be corrupted by appending new steps.
- **Streaming** through files (i.e. online processing). Consumer apps can read existing steps while the Producer is still writing new steps. Reader's loop can block (with timeout) and wait for new steps to arrive. Same reader code can read the entire data in post or in situ. No restrictions on the Producer.
- **Burst buffer support** for writing data. It can write the output to a local file system on each compute node and drain the data to the parallel file system in a separate asynchronous thread. Streaming read from the target file system are still supported when data goes through the burst buffer. Appending to an existing file on the target file system is NOT supported currently.

BP4 files have the following structure given a "name" string passed as the first argument of `IO::Open`:

```
io.SetEngine("BP4");
adios2::Engine bpFile = io.Open("name", adios2::Mode::Write);
```

will generate:

```
% BP4 datasets are always a directory
name.bp/

% data and metadata files
name.bp/
        data.0
        data.1
        ...
        data.M
        md.0
        md.idx
```

Note: BP4 file names are compatible with the Unix (/) and Windows (\\) file system naming convention for directories and files.

This engine allows the user to fine tune the buffering operations through the following optional parameters:

1. **Profile**: turns ON/OFF profiling information right after a run
2. **ProfileUnits**: set profile units according to the required measurement scale for intensive operations
3. **Threads**: number of threads provided from the application for buffering, use this for very large variables in data size
4. **InitialBufferSize**: initial memory provided for buffering (minimum is 16Kb)
5. **BufferGrowthFactor**: exponential growth factor for initial buffer > 1, default = 1.05.
6. **MaxBufferSize**: maximum allowable buffer size (must be larger than 16Kb). If too large adios2 will throw an exception.

7. **FlushStepsCount**: users can select how often to produce the more expensive collective metadata file in terms of steps: default is 1. Increase to reduce adios2 collective operations footprint, with the trade-off of reducing checkpoint frequency. Buffer size will increase until first steps count if `MaxBufferSize` is not set.
8. **NumAggregators** (or **SubStreams**): Users can select how many sub-files (`M`) are produced during a run, ranges between 1 and the number of mpi processes from `MPI_Size` (`N`), adios2 will internally aggregate data buffers (`N-to-M`) to output the required number of sub-files. Default is 0, which will let adios2 to group processes per shared-memory-access (i.e. one per compute node) and use one process per node as an aggregator. If NumAggregators is larger than the number of processes then it will be set to the number of processes.
9. **AggregatorRatio**: An alternative option to NumAggregators to pick every Nth process as aggregator. An integer divider of the number of processes is required, otherwise a runtime exception is thrown.
10. **OpenTimeoutSecs**: (Streaming mode) Reader may want to wait for the creation of the file in `io.Open()`. By default the Open() function returns with an error if file is not found.
11. **BeginStepPollingFrequencySecs**: (Streaming mode) Reader can set how frequently to check the file (and file system) for new steps. Default is 1 seconds which may be stressful for the file system and unnecessary for the application.
12. **StatsLevel**: Turn on/off calculating statistics for every variable (Min/Max). Default is On. It has some cost to generate this metadata so it can be turned off if there is no need for this information.
13. **StatsBlockSize**: Calculate Min/Max for a given size of each process output. Default is one Min/Max per writer. More fine-grained min/max can be useful for querying the data.
14. **NodeLocal** or **Node-Local**: For distributed file system. Every writer process must make sure the .bp/ directory is created on the local file system. Required when writing to local disk/SSD/NVMe in a cluster. Note: the BurstBuffer* parameters are newer and should be used for using the local storage as temporary instead of this parameter.
15. **BurstBufferPath**: Redirect output file to another location and drain it to the original target location in an asynchronous thread. It requires to be able to launch one thread per aggregator (see SubStreams) on the system. This feature can be used on machines that have local NVMe/SSDs on each node to accelerate the output writing speed. On Summit at OLCF, use "/mnt/bb/<username>" for the path where <username> is your user account name. Temporary files on the accelerated storage will be automatically deleted after the application closes the output and ADIOS drains all data to the file system, unless draining is turned off (see the next parameter). Note: at this time, this feature cannot be used to append data to an existing dataset on the target system.
16. **BurstBufferDrain**: To write only to the accelerated storage but to not drain it to the target file system, set this flag to false. Data will NOT be deleted from the accelerated storage on close. By default, setting the BurstBufferPath will turn on draining.
17. **BurstBufferVerbose**: Verbose level 1 will cause each draining thread to print a one line report at the end (to standard output) about where it has spent its time and the number of bytes moved. Verbose level 2 will cause each thread to print a line for each draining operation (file creation, copy block, write block from memory, etc).
18. **StreamReader**: By default the BP4 engine parses all available metadata in Open(). An application may turn this flag on to parse a limited number of steps at once, and update metadata when those steps have been processed. If the flag is ON, reading only works in streaming mode (using BeginStep/EndStep); file reading mode will not work as there will be zero steps processed in Open().

Key	Value Format	Default and Examples
Profile	string On/Off	**On**, Off
ProfileUnits	string	**Microseconds**, Milliseconds, Seconds, Minutes, Hours
Threads	integer > 1	**1**, 2, 3, 4, 16, 32, 64
InitialBufferSize	float+units >= 16Kb	**16Kb**, 10Mb, 0.5Gb
MaxBufferSize	float+units >= 16Kb	**at EndStep**, 10Mb, 0.5Gb
BufferGrowthFactor	float > 1	**1.05**, 1.01, 1.5, 2
FlushStepsCount	integer > 1	**1**, 5, 1000, 50000
NumAggregators	integer >= 1	**0 (one file per compute node)**, `MPI_Size`/2, ... , 2, (N-to-1) 1
AggregatorRatio	integer >= 1	not used unless set, `MPI_Size`/N must be an integer value
OpenTimeoutSecs	float	**0**, `10.0`, `5`
BeginStepPollingFrequency-Secs	float	**1**, `10.0`
StatsLevel	integer, 0 or 1	**1**, `0`
StatsBlockSize	integer > 0	**a very big number**, `1073741824` for blocks with 1M elements
NodeLocal	string On/Off	**Off**, On
Node-Local	string On/Off	**Off**, On
BurstBufferPath	string	**“”**, /mnt/bb/norbert, /ssd
BurstBufferDrain	string On/Off	**On**, Off
BurstBufferVerbose	integer, 0-2	**0**, `1`, `2`
StreamReader	string On/Off	On, **Off**

Only file transport types are supported. Optional parameters for `IO::AddTransport` or in runtime config file transport field:

Transport type: File

Key	Value Format	Default and Examples
Library	string	**POSIX** (UNIX), **FStream** (Windows), stdio, IME

The IME transport directly reads and writes files stored on DDN's IME burst buffer using the IME native API. To use the IME transport, IME must be avaiable on the target system and ADIOS2 needs to be configured with `ADIOS2_USE_IME`. By default, data written to the IME is automatically flushed to the parallel filesystem at every `EndStep()` call. You can disable this automatic flush by setting the transport parameter `SyncToPFS` to `OFF`.

8.3 BP3

The BP3 Engine writes and reads files in ADIOS2 native binary-pack (bp) format. BP files are backwards compatible with ADIOS1.x and have the following structure given a “name” string passed as the first argument of `IO::Open`:

```
adios2::Engine bpFile = io.Open("name", adios2::Mode::Write);
```

will generate:

```
% collective metadata file
name.bp
```

(continues on next page)

(continued from previous page)

```
% data directory and files
name.bp.dir/
            name.bp.0
            name.bp.1
            ...
            name.bp.M
```

Note: BP3 file names are compatible with the Unix (`/`) and Windows (`\\`) file system naming convention for directories and files.

Caution: The default BP3 engine will check if the `.bp` is the extension of the first argument of `IO::Open` and will add `.bp` and `.bp.dir` if not.

This engine allows the user to fine tune the buffering operations through the following optional parameters:

1. **Profile**: turns ON/OFF profiling information right after a run
2. **ProfileUnits**: set profile units according to the required measurement scale for intensive operations
3. **CollectiveMetadata**: turns ON/OFF forming collective metadata during run (used by large scale HPC applications)
4. **Threads**: number of threads provided from the application for buffering, use this for very large variables in data size
5. **InitialBufferSize**: initial memory provided for buffering (minimum is 16Kb)
6. **BufferGrowthFactor**: exponential growth factor for initial buffer > 1, default = 1.05.
7. **MaxBufferSize**: maximum allowable buffer size (must be larger than 16Kb). If too large adios2 will throw an exception.
8. **FlushStepsCount**: users can select how often to produce the more expensive collective metadata file in terms of steps: default is 1. Increase to reduce adios2 collective operations footprint, with the trade-off of reducing checkpoint frequency. Buffer size will increase until first steps count if `MaxBufferSize` is not set.
9. **NumAggregators** (or **SubStreams**): Users can select how many sub-files (`M`) are produced during a run, ranges between 1 and the number of mpi processes from `MPI_Size` (`N`), adios2 will internally aggregate data buffers (`N-to-M`) to output the required number of sub-files. Default is 0, which will let adios2 to group processes per shared-memory-access (i.e. one per compute node) and use one process per node as an aggregator. If NumAggregators is larger than the number of processes then it will be set to the number of processes.
10. **AggregatorRatio**: An alternative option to NumAggregators to pick every Nth process as aggregator. An integer divider of the number of processes is required, otherwise a runtime exception is thrown.
11. **Node-Local**: For distributed file system. Every writer process must make sure the .bp/ directory is created on the local file system. Required for using local disk/SSD/NVMe in a cluster.

Key	Value Format	**Default** and Examples
Profile	string On/Off	**On**, Off
ProfileUnits	string	**Microseconds**, Milliseconds, Seconds, Minutes, Hours
CollectiveMetadata	string On/Off	**On**, Off
Threads	integer > 1	**1**, 2, 3, 4, 16, 32, 64
InitialBufferSize	float+units >= 16Kb	**16Kb**, 10Mb, 0.5Gb
MaxBufferSize	float+units >= 16Kb	**at EndStep**, 10Mb, 0.5Gb
BufferGrowthFactor	float > 1	**1.05**, 1.01, 1.5, 2
FlushStepsCount	integer > 1	**1**, 5, 1000, 50000
NumAggregators	integer >= 1	**0 (one file per compute node)**, `MPI_Size`/2, ... , 2, (N-to-1) 1
AggregatorRatio	integer >= 1	not used unless set, `MPI_Size`/N must be an integer value
Node-Local	string On/Off	**Off**, On

Only file transport types are supported. Optional parameters for `IO::AddTransport` or in runtime config file transport field:

Transport type: File

Key	Value Format	**Default** and Examples
Library	string	**POSIX** (UNIX), **FStream** (Windows), stdio, IME

8.4 HDF5

In ADIOS2, the default engine for reading and writing HDF5 files is called *"HDF5"*. To use this engine, you can either specify it in your xml config file, with tag `<engine type=HDF5>` or, set it in client code. For example, here is how to create a hdf5 reader:

```
adios2::IO h5IO = adios.DeclareIO("SomeName");
h5IO.SetEngine("HDF5");
adios2::Engine h5Reader = h5IO.Open(filename, adios2::Mode::Read);
```

To read back the h5 files generated with VDS to ADIOS2, one can use the HDF5 engine. Please make sure you are using the HDF5 library that has version greater than or equal to 1.11 in ADIOS2.

The h5 file generated by ADIOS2 has two levels of groups: The top Group, `/` and its subgroups: `Step0` ... `StepN`, where `N` is number of steps. All datasets belong to the subgroups.

Any other h5 file can be read back to ADIOS as well. To be consistent, when reading back to ADIOS2, we assume a default Step0, and all datasets from the original h5 file belong to that subgroup. The full path of a dataset (from the original h5 file) is used when represented in ADIOS2.

We can pass options to HDF5 API from ADIOS xml configuration. Currently we support CollectionIO (default false), and chunk specifications. The chunk specification uses space to seperate values, and by default, if a valid H5ChunkDim exists, it applies to all variables, unless H5ChunkVar is specified. Examples:

```
<parameter key="H5CollectiveMPIO" value="yes"/>
<parameter key="H5ChunkDim" value="200 200"/>
<parameter key="H5ChunkVar" value="VarName1 VarName2"/>
```

We suggest to read HDF5 documentation before appling these options.

After the subfile feature is introduced in HDF5 version 1.14, the ADIOS2 HDF5 engine will use subfiles as the default h5 format as it improves I/O in general (for example, see https://escholarship.org/uc/item/6fs7s3jb)

To use the subfile feature, client needs to support MPI_Init_thread with MPI_THREAD_MULTIPLE.

Useful parameters from the HDF lirbary to tune subfiles are: .. code-block:: xml

H5FD_SUBFILING_IOC_PER_NODE (num of subfiles per node)
set H5FD_SUBFILING_IOC_PER_NODE to 0 if the regular h5 file is prefered, before using ADIOS2 HDF5 engine.

H5FD_SUBFILING_STRIPE_SIZE H5FD_IOC_THREAD_POOL_SIZE

8.5 SST Sustainable Staging Transport

In ADIOS2, the Sustainable Staging Transport (SST) is an engine that allows direct connection of data producers and consumers via the ADIOS2 write/read APIs. This is a classic streaming data architecture where the data passed to ADIOS on the write side (via Put() deferred and sync, and similar calls) is made directly available to a reader (via Get(), deferred and sync, and similar calls).

SST is designed for use in HPC environments and can take advantage of RDMA network interconnects to speed the transfer of data between communicating HPC applications; however, it is also capable of operating in a Wide Area Networking environment over standard sockets. SST supports full MxN data distribution, where the number of reader ranks can differ from the number of writer ranks. SST also allows multiple reader cohorts to get access to a writer's data simultaneously.

To use this engine, you can either specify it in your xml config file, with tag `<engine type=SST>` or, set it in client code. For example, here is how to create an SST reader:

```
adios2::IO sstIO = adios.DeclareIO("SomeName");
sstIO.SetEngine("SST");
adios2::Engine sstReader = sstIO.Open(filename, adios2::Mode::Read);
```

and a sample code for SST writer is:

```
adios2::IO sstIO = adios.DeclareIO("SomeName");
sstIO.SetEngine("SST");
adios2::Engine sstWriter = sstIO.Open(filename, adios2::Mode::Write);
```

The general goal of ADIOS2 is to ease the conversion of a file-based application to instead use a non-file streaming interconnect, for example, data producers such as computational physics codes and consumers such as analysis applications. However, there are some uses of ADIOS2 APIs that work perfectly well with the ADIOS2 file engines, but which will not work or will perform badly with streaming. For example, SST is based upon the *"step"* concept and ADIOS2 applications that use SST must call `BeginStep()` and `EndStep()`. On the writer side, the `Put()` calls between `BeginStep` and `EndStep` are the unit of communication and represent the data that will be available between the corresponding `Begin/EndStep` calls on the reader.

Also, it is recommended that SST-based applications not use the ADIOS2 Get() sync method unless there is only one data item to be read per step. This is because SST implements MxN data transfer (and avoids having to deliver all data to every reader), by queueing data on the writer ranks until it is known which reader rank requires it. Normally this data fetch stage is initiated by `PerformGets()` or `EndStep()`, both of which fulfill any pending `Get()` deferred operations. However, unlike `Get()` deferred, the semantics of `Get()` sync require the requested data to be fetched from the writers before the call can return. If there are multiple calls to `Get()` sync per step, each one may require a communication with many writers, something that would have only had to happen once if `Get()` differed were used instead. Thus the use of `Get()` sync is likely to incur a substantial performance penalty.

On the writer side, depending upon the chosen data marshaling option there may be some (relatively small) performance differences between `Put()` sync and `Put()` deferred, but they are unlikely to be as substantial as between `Get()` sync and `Get()` deferred.

Note that SST readers and writers do not necessarily move in lockstep, but depending upon the queue length parameters and queueing policies specified, differing reader and writer speeds may cause one or the other side to wait for data to be produced or consumed, or data may be dropped if allowed by the queueing policy. However, steps themselves are atomic and no step will be partially dropped, delivered to a subset of ranks, or otherwise divided.

The SST engine allows the user to customize the streaming operations through the following optional parameters:

1. `RendezvousReaderCount`: Default **1**. This integer value specifies the number of readers for which the writer should wait before the writer-side Open() returns. The default of 1 implements an ADIOS1/flexpath style "rendezvous", in which an early-starting reader will wait for the writer to start, or vice versa. A number >1 will cause the writer to wait for more readers and a value of 0 will allow the writer to proceed without any readers present. This value is interpreted by SST Writer engines only.

2. `RegistrationMethod`: Default **"File"**. By default, SST reader and writer engines communicate network contact information via files in a shared filesystem. Specifically, the `"filename"` parameter in the `Open()` call is interpreted as a path which the writer uses as the name of a file to which contact information is written, and from which a reader will attempt to read contact information. As with other file-based engines, file creation and access is subject to the usual considerations (directory components are interpreted, but must exist and be traversable, writer must be able to create the file and the reader must be able to read it). Generally the file so created will exist only for as long as the writer keeps the stream Open(), but abnormal process termination may leave "stale" files in those locations. These stray ".sst" files should be deleted to avoid confusing future readers. SST also offers a **"Screen"** registration method in which writers and readers send their contact information to, and read it from, stdout and stdin respectively. The "screen" registration method doesn't support batch mode operations in any way, but may be useful when manually starting jobs on machines in a WAN environment that don't share a filesystem. A future release of SST will also support a **"Cloud"** registration method where contact information is registered to and retrieved from a network-based third-party server so that both the shared filesystem and interactivity can be avoided. This value is interpreted by both SST Writer and Reader engines.

3. `QueueLimit`: Default **0**. This integer value specifies the number of steps which the writer will allow to be queued before taking specific action (such as discarding data or waiting for readers to consume the data). The default value of 0 is interpreted as no limit. This value is interpreted by SST Writer engines only.

4. `QueueFullPolicy`: Default **"Block"**. This value controls what policy is invoked if a non-zero **QueueLimit** has been specified and new data would cause the queue limit to be reached. Essentially, the **"Block"** option ensures data will not be discarded and if the queue fills up the writer will block on **EndStep** until the data has been read. If there is one active reader, **EndStep** will block until data has been consumed off the front of the queue to make room for newly arriving data. If there is more than one active reader, it is only removed from the queue when it has been read by all readers, so the slowest reader will dictate progress. **NOTE THAT THE NO READERS SITUATION IS A SPECIAL CASE**: If there are no active readers, new timesteps are considered to have completed their active queueing immediately upon submission. They may be retained in the "reserve queue" if the ReserveQueueLimit is non-zero. However, if that ReserveQueueLimit parameter is zero, timesteps submitted when there are no active readers will be immediately discarded.

Besides **"Block"**, the other acceptable value for **QueueFullPolicy** is **"Discard"**. When **"Discard"** is specified, and an **EndStep** operation would add more than the allowed number of steps to the queue, some step is discarded. If there are no current readers connected to the stream, the *oldest* data in the queue is discarded. If there are current readers, then the *newest* data (I.E. the just-created step) is discarded. (The differential treatment is because SST sends metadata for each step to the readers as soon as the step is accepted and cannot reliably prevent that use of that data without a costly all-to-all synchronization operation. Discarding the *newest* data instead is less satisfying, but has a similar long-term effect upon the set of steps delivered to the readers.) This value is interpreted by SST Writer engines only.

5. `ReserveQueueLimit`: Default **0**. This integer value specifies the number of steps which the writer will keep in the queue for the benefit of late-arriving readers. This may consist of timesteps that have already been consumed by any readers, as well as timesteps that have not yet been consumed. In some sense this is target queue minimum size, while QueueLimit is a maximum size. This value is interpreted by SST Writer engines only.

6. `DataTransport`: Default **varies**. This string value specifies the underlying network communication mechanism to use for exchanging data in SST. Generally this is chosen by SST based upon what is available on the current platform. However, specifying this engine parameter allows overriding SST's choice. Current allowed values are **"UCX"**,

"MPI", **"RDMA"**, and **"WAN"**. (**ib** and **fabric** are accepted as equivalent to **RDMA** and **evpath** is equivalent to **WAN**.) Generally both the reader and writer should be using the same network transport, and the network transport chosen may be dictated by the situation. For example, the RDMA transport generally operates only between applications running on the same high-performance interconnect (e.g. on the same HPC machine). If communication is desired between applications running on different interconnects, the Wide Area Network (WAN) option should be chosen. This value is interpreted by both SST Writer and Reader engines.

7. `WANDataTransport`: Default **sockets**. If the SST **DataTransport** parameter is **"WAN**, this string value specifies the EVPath-level data transport to use for exchanging data. The value must be a data transport known to EVPath, such as **"sockets"**, **"enet"**, or **"ib"**. Generally both the reader and writer should be using the same EVPath-level data transport. This value is interpreted by both SST Writer and Reader engines.

8. `ControlTransport`: Default **tcp**. This string value specifies the underlying network communication mechanism to use for performing control operations in SST. SST can be configured to standard TCP sockets, which are very reliable and efficient, but which are limited in their scalability. Alternatively, SST can use a reliable UDP protocol, that is more scalable, but as of ADIOS2 Release 2.4.0 still suffers from some reliability problems. (**sockets** is accepted as equivalent to **tcp** and **udp**, **rudp**, and **enet** are equivalent to **scalable**. Generally both the reader and writer should be using the same control transport. This value is interpreted by both SST Writer and Reader engines.

9. `NetworkInterface`: Default **NULL**. In situations in which there are multiple possible network interfaces available to SST, this string value specifies which should be used to generate SST's contact information for writers. Generally this should *NOT* be specified except for narrow sets of circumstances. It has no effect if specified on Reader engines. If specified, the string value should correspond to a name of a network interface, such as are listed by commands like "netstat -i". For example, on most Unix systems, setting the NetworkInterface parameter to "lo" (or possibly "lo0") will result in SST generating contact information that uses the network address associated with the loopback interface (127.0.0.1). This value is interpreted by only by the SST Writer engine.

10. `ControlInterface`: Default **NULL**. This value is similar to the NetworkInterface parameter, but only applies to the SST layer which does messaging for control (open, close, flow and timestep management, but not actual data transfer). Generally the NetworkInterface parameter can be used to control this, but that also aplies to the Data Plane. Use ControlInterface in the event of conflicting specifications.

11. `DataInterface`: Default **NULL**. This value is similar to the NetworkInterface parameter, but only applies to the SST layer which does messaging for data transfer, not control (open, close, flow and timestep management). Generally the NetworkInterface parameter can be used to control this, but that also aplies to the Control Plane. Use DataInterface in the event of conflicting specifications. In the case of the RDMA data plane, this parameter controls the libfabric interface choice.

12. `FirstTimestepPrecious`: Default **FALSE**. FirstTimestepPrecious is a boolean parameter that affects the queueing of the first timestep presented to the SST Writer engine. If FirstTimestepPrecious is **TRUE**, then the first timestep is effectively never removed from the output queue and will be presented as a first timestep to any reader that joins at a later time. This can be used to convey run parameters or other information that every reader may need despite joining later in a data stream. Note that this queued first timestep does count against the QueueLimit parameter above, so if a QueueLimit is specified, it should be a value larger than 1. Further note while specifying this parameter guarantees that the preserved first timestep will be made available to new readers, other reader-side operations (like requesting the LatestAvailable timestep in Engine parameters) might still cause the timestep to be skipped. This value is interpreted by only by the SST Writer engine.

13. `AlwaysProvideLatestTimestep`: Default **FALSE**. AlwaysProvideLatestTimestep is a boolean parameter that affects what of the available timesteps will be provided to the reader engine. If AlwaysProvideLatestTimestep is **TRUE**, then if there are multiple timesteps available to the reader, older timesteps will be skipped and the reader will see only the newest available upon BeginStep. This value is interpreted by only by the SST Reader engine.

14. `OpenTimeoutSecs`: Default **60**. OpenTimeoutSecs is an integer parameter that specifies the number of seconds SST is to wait for a peer connection on Open(). Currently this is only implemented on the Reader side of SST, and is a timeout for locating the contact information file created by Writer-side Open, not for completing the entire Open() handshake. Currently value is interpreted by only by the SST Reader engine.

15. `SpeculativePreloadMode`: Default **AUTO**. In some circumstances, SST eagerly sends all data from writers to every readers without first waiting for read requests. Generally this improves performance if every reader needs all the data, but can be very detrimental otherwise. The value **AUTO** for this engine parameter instructs SST to apply its own heuristic for determining if data should be eagerly sent. The value **OFF** disables this feature and the value **ON** causes eager sending regardless of heuristic. Currently SST's heuristic is simple. If the size of the reader cohort is less than or equal to the value of the `SpecAutoNodeThreshold` engine parameter (Default value 1), eager sending is initiated. Currently value is interpreted by only by the SST Reader engine.

16. `SpecAutoNodeThreshold`: Default **1**. If the size of the reader cohort is less than or equal to this value *and* the `SpeculativePreloadMode` parameter is **AUTO**, SST will initiate eager data sending of all data from each writer to all readers. Currently value is interpreted by only by the SST Reader engine.

17. `StepDistributionMode`: Default **"AllToAll"**. This value controls how steps are distributed, particularly when there are multiple readers. By default, the value is **"AllToAll"**, which means that all timesteps are to be delivered to all readers (subject to discard rules, etc.). In other distribution modes, this is not the case. For example, in **"RoundRobin"**, each step is delivered only to a single reader, determined in a round-robin fashion based upon the number or readers who have opened the stream at the time the step is submitted. In **"OnDemand"** each step is delivered to a single reader, but only upon request (with a request being initiated by the reader doing BeginStep()). Normal reader-side rules (like BeginStep timeouts) and writer-side rules (like queue limit behavior) apply.

8.6 SSC Strong Staging Coupler

The SSC engine is designed specifically for strong code coupling. Currently SSC only supports fixed IO pattern, which means once the first step is finished, users are not allowed to write or read a data block with a *start* and *count* that have not been written or read in the first step. SSC uses a combination of one sided MPI and two sided MPI methods. In any cases, all user applications are required to be launched within a single mpirun or mpiexec command, using the MPMD mode.

The SSC engine takes the following parameters:

1. `OpenTimeoutSecs`: Default **10**. Timeout in seconds for opening a stream. The SSC engine's open function will block until the RendezvousAppCount is reached, or timeout, whichever comes first. If it reaches the timeout, SSC will throw an exception.
2. `Threading`: Default **False**. SSC will use threads to hide the time cost for metadata manipulation and data transfer when this parameter is set to **true**. SSC will check if MPI is initialized with multi-thread enabled, and if not, then SSC will force this parameter to be **false**. Please do NOT enable threading when multiple I/O streams are opened in an application, as it will cause unpredictable errors. This parameter is only effective when writer definitions and reader selections are NOT locked. For cases definitions and reader selections are locked, SSC has a more optimized way to do data transfers, and thus it will not use this parameter.

Key	**Value Format**	**Default** and Examples
OpenTimeoutSecs	integer	**10**, 2, 20, 200
Threading	bool	**false**, true

8.7 DataMan for Wide Area Network Data Staging

The DataMan engine is designed for data staging over the wide area network. It is supposed to be used in cases where a few writers send data to a few readers over long distance.

DataMan supports compression operators such as ZFP lossy compression and BZip2 lossless compression. Please refer to the operator section for usage.

The DataMan engine takes the following parameters:

1. `IPAddress`: No default value. The IP address of the host where the writer application runs. This parameter is compulsory in wide area network data staging.
2. `Port`: Default **50001**. The port number on the writer host that will be used for data transfers.
3. `Timeout`: Default **5**. Timeout in seconds to wait for every send / receive operation. Packages not sent or received within this time are considered lost.
4. `RendezvousReaderCount`: Default **1**. This integer value specifies the number of readers for which the writer should wait before the writer-side Open() returns. By default, an early-starting writer will wait for the reader to start, or vice versa. A number >1 will cause the writer to wait for more readers, and a value of 0 will allow the writer to proceed without any readers present. This value is interpreted by DataMan Writer engines only.
5. `Threading`: Default **true** for reader, **false** for writer. Whether to use threads for send and receive operations. Enabling threading will cause extra overhead for managing threads and buffer queues, but will improve the continuity of data steps for readers, and help overlap data transfers with computations for writers.
6. `TransportMode`: Default **fast**. Only DataMan writers take this parameter. Readers are automatically synchronized at runtime to match writers' transport mode. The fast mode is optimized for latency-critical applications. It enforces readers to only receive the latest step. Therefore, in cases where writers are faster than readers, readers will skip some data steps. The reliable mode ensures that all steps are received by readers, by sacrificing performance compared to the fast mode.
7. `MaxStepBufferSize`: Default **128000000**. In order to bring down the latency in wide area network staging use cases, DataMan uses a fixed receiver buffer size. This saves an extra communication operation to sync the buffer size for each step, before sending actual data. The default buffer size is 128 MB, which is sufficient for most use cases. However, in case 128 MB is not enough, this parameter must be set correctly, otherwise DataMan will fail.

Key	**Value Format**	**Default** and Examples
IPAddress	string	**N/A**, 22.195.18.29
Port	integer	**50001**, 22000, 33000
Timeout	integer	**5**, 10, 30
RendezvousReaderCount	integer	**1**, 0, 3
Threading	bool	**true** for reader, **false** for writer
TransportMode	string	**fast**, reliable
MaxStepBufferSize	integer	**128000000**, 512000000, 1024000000

8.8 Inline for zero-copy

The `Inline` engine provides in-process communication between the writer and reader, avoiding the copy of data buffers.

This engine is focused on the $N \rightarrow N$ case: N writers share a process with N readers, and the analysis happens 'inline' without writing the data to a file or copying to another buffer. It is similar to the streaming SST engine, since analysis must happen per step.

To use this engine, you can either add `<engine type=Inline>` to your XML config file, or set it in your application code:

```
adios2::IO io = adios.DeclareIO("ioName");
io.SetEngine("Inline");
adios2::Engine inlineWriter = io.Open("inline_write", adios2::Mode::Write);
adios2::Engine inlineReader = io.Open("inline_read", adios2::Mode::Read);
```

Notice that unlike other engines, the reader and writer share an IO instance. Both the writer and reader must be opened before either tries to call `BeginStep()`/`PerformPuts()`/`PerformGets()`. There must be exactly one writer, and exactly one reader.

For successful operation, the writer will perform a step, then the reader will perform a step in the same process. When the reader starts its step, the only data it has available is that written by the writer in its process. The reader then can retrieve whatever data was written by the writer by using the double-pointer `Get` call:

```
void Engine::Get<T>(Variable<T>, T**) const;
```

This version of `Get` is only used for the inline engine. See the example below for details.

Note: Since the inline engine does not copy any data, the writer should avoid changing the data before the reader has read it.

Typical access pattern:

```
// ... Application data generation

inlineWriter.BeginStep();
inlineWriter.Put(var, in_data); // always use deferred mode
inlineWriter.EndStep();
// Unlike other engines, data should not be reused at this point (since ADIOS
// does not copy the data), though ADIOS cannot enforce this.
// Must wait until reader is finished using the data.

inlineReader.BeginStep();
double* out_data;
inlineReader.Get(var, &data);
// Now in_data == out_data.
inlineReader.EndStep();
```

8.9 Null

The `Null` Engine performs no internal work and no I/O. It was created for testing applications that have ADIOS2 output in it by turning off the I/O easily. The runtime difference between a run with the Null engine and another engine tells us the IO overhead of that particular output with that particular engine.

```
adios2::IO io = adios.DeclareIO("Output");
io.SetEngine("Null");
```

or using the XML config file:

```
<adios-config>
    <io name="Output">
        <engine type="Null">
        </engine>
    </io>
</adios-config>
```

Although there is a reading engine as well, which will not fail, any variable/attribute inquiry returns *nullptr* and any subsequent Get() calls will throw an exception in C++/Python or return an error in C/Fortran.

Note that there is also a *Null transport* that can be used by a BP engine instead of the default *File transport*. In that case, the BP engine will perform all internal work including buffering and aggregation but no data will be output at all. A run like this can be used to assess the overhead of the internal work of the BP engine.

```
adios2::IO io = adios.DeclareIO("Output");
io.SetEngine("BP5");
io.AddTransport("Null", {});
```

or using the XML config file

```
<adios-config>
    <io name="Output">
        <engine type="BP5">
        </engine>
        <transport type="Null">
        </transport>
    </io>
</adios-config>
```

8.10 Plugin Engine

For details on using the Plugin Engine, see the *Plugins* documentation.

CHAPTER

NINE

SUPPORTED OPERATORS

The Operator abstraction allows ADIOS2 to act upon the user application data, either from a `adios2::Variable` or a set of Variables in an `adios2::IO` object. Current supported operations are:

1. Data compression/decompression, lossy and lossless.

This section provides a description of the Available Operators in ADIOS2 and their specific parameters to allow extra-control from the user. Parameters are passed in key-value pairs for:

1. Operator general supported parameters.
2. Operator specific supported parameters.

Parameters are passed at:

1. Compile time: using the second parameter of the method `ADIOS2::DefineOperator`
2. *Runtime Configuration Files* in the *ADIOS* component.

9.1 CompressorZFP

The `CompressorZFP` Operator is compressor that uses a lossy but optionally error-bounded compression to achieve high compression ratios.

ZFP provides compressed-array classes that support high throughput read and write random access to individual array elements. ZFP also supports serial and parallel (OpenMP and CUDA) compression of whole arrays, e.g., for applications that read and write large data sets to and from disk.

ADIOS2 provides a `CompressorZFP` operator that lets you compress an decompress variables. Below there is an example of how to invoke `CompressorZFP` operator:

```
adios2::IO io = adios.DeclareIO("Output");
auto ZFPOp    = adios.DefineOperator("CompressorZFP", adios2::ops::LossyZFP);

auto var_r32 = io.DefineVariable<float>("r32", shape, start, count);
var_r32.AddOperation(ZFPOp, {{adios2::ops::zfp::key::rate, rate}});
```

9.1.1 CompressorZFP Specific parameters

The `CompressorZFP` operator accepts the following operator specific parameters:

`CompressorZFP` available parameters	
`accuracy`	Fixed absolute error tolerance
`rate`	Fixed number of bits in a compression unit
`precision`	Fixed number of uncompressed bits per value
`backend`	Backend device: `cuda` `omp` `serial`

9.1.2 CompressorZFP Execution Policy

`CompressorZFP` can run in multiple backend devices: GPUs (CUDA), OpenMP, and in the host CPU. By default `CompressorZFP` will choose its backend following the above order upon the availability of the device adapter.

Exceptionally, if its corresponding ADIOS2 variable contains a CUDA memory address, this is a CUDA buffer, the CUDA backend will be called if available.

In any case, the user can manually set the backend using the ZFPOperator specific parameter `backend`.

9.2 Plugin Operator

For details on using the Plugin Operator, see the *Plugins* documentation.

9.3 Encryption

The Encryption Operator uses the *Plugins* interface. This operator uses libsodium for encrypting and decrypting data. If ADIOS can find libsodium at configure time, this plugin will be built.

This operator will generate a secret key and encrypts the data with the key and a nonce as described in the libsodium secret key cryptography docs. The key is saved to the specified `SecretKeyFile` and will be used for decryption. The key should be kept confidential since it is used to both encrypt and decrypt the data.

Parameters to use with the Encryption operator:

Key	Value Format	Explanation
PluginName	string	Required. Name to refer to plugin, e.g., `MyOperator`
PluginLibrary	string	Required. Name of shared library, `EncryptionOperator`
SecretKeyFile	string	Required. Path to secret key file

CHAPTER

TEN

FULL APIS

Note: Application developers who desire fine-grained control of IO tasks should use the Full APIs. In simple cases (e.g. reading a file for analysis, interactive Python, or saving some data for a small project or tests) please refer to the *High-Level APIs*.

Currently ADIOS2 support bindings for the following languages and their minimum standards:

Language	Standard	Interface
C++	11/newer older	`#include adios2.h` use C bindings
C	99	`#include adios2_c.h`
Fortran	90	`use adios2`
Python	2.7 3	`import adios2` `import adios2`

Tip: Prefer the C++11 bindings if your application C++ compiler supports the 2011 (or later) standard. For code using previous C++ standards (98 or 03) use the C bindings for ABI compatibility.

Caution: Statically linked libraries (`*.a`) might result in conflicting ABIs between an older C++ project, the C bindings, and the adios native C++11 library. Test to make sure it works for your platform.

The current interaction flow for each language binding API with the ADIOS2 library is specified as follows

The following sections provide a summary of the API calls on each language and links to Write and Read examples to put it all together.

10.1 C++11 bindings

Caution: DO NOT use the clause `using namespace adios2` in your code. This is in general a bad practices that creates potential name conflicts. Always use `adios2::` explicitly, *e.g.* `adios2::ADIOS`, `adios2::IO`.

Tip: Prefer the C++11 bindings to take advantage of added functionality (*e.g.* move semantics, lambdas, etc.). If you must use an older C++ standard (98 or 03) to avoid application binary interface (ABI) incompatibilities use the C

bindings.

10.1.1 ADIOS2 components classes

ADIOS2 C++ bindings objects are mapped 1-to-1 to the ADIOS components described in the *Components Overview* section. Only the `adios2::ADIOS` object is "owned" by the developer's program using adios2, all other components are light-weight objects that point internally to a component that lives inside the `adios2::ADIOS` "factory" object.

```
c++11
adios2::ADIOS
adios2::IO
adios2::Variable<T>
adios2::Attribute<T>
adios2::Engine
adios2::Operator
```

The following section provides a summary of the available functionality for each class.

10.1.2 ADIOS class

class **ADIOS**

Public Functions

ADIOS(MPI_Comm comm)

Starting point for MPI apps. Creates an *ADIOS* object. MPI Collective Operation as it call MPI_Comm_dup

Parameters
comm – defines domain scope from application

Throws
`std::invalid_argument` – if user input is incorrect

ADIOS(const std::string &configFile, MPI_Comm comm)

Starting point for MPI apps. Creates an *ADIOS* object allowing a runtime config file. MPI collective and it calls MPI_Comm_dup and MPI_Bcast to pass the configFile contents

Parameters

- **configFile** – runtime config file
- **comm** – defines domain scope from application

Throws
`std::invalid_argument` – if user input is incorrect

ADIOS(const std::string &configFile, MPI_Comm comm, const std::string &hostLanguage)

extra constructor for R and other languages that use the public C++ API but has data in column-major. Pass "" for configfile if there is no config file. Last bool argument exist only to ensure matching this signature by having different number of arguments. Supported languages are "R", "Matlab", "Fortran", all these names mean the same thing: treat all arrays column-major e.g. *adios2::ADIOS*("", comm, "Fortran", false);

ADIOS(const std::string &configFile)

Starting point for non-MPI serial apps. Creates an *ADIOS* object allowing a runtime config file.

Parameters
configFile – runtime config file

Throws
`std::invalid_argument` – if user input is incorrect

ADIOS()

Starting point for non-MPI apps. Creates an *ADIOS* object

Throws
`std::invalid_argument` – if user input is incorrect

ADIOS(const std::string &configFile, const std::string &hostLanguage)

extra constructor for R and other languages that use the public C++ API but has data in column-major. Pass "" for configfile if there is no config file. Last bool argument exist only to ensure matching this signature by having different number of arguments. Supported languages are "R", "Matlab", "Fortran", all these names mean the same thing: treat all arrays column-major e.g. *adios2::ADIOS*("", "Fortran", false);

explicit **operator bool**() const noexcept

object inspection true: valid object, false: invalid object

ADIOS(const *ADIOS*&) = delete

DELETED Copy Constructor. *ADIOS* is the only object that manages its own memory. Create a separate object for independent tasks

ADIOS(*ADIOS*&&) = default

default move constructor exists to allow for auto ad = ADIOS(...) initialization

~ADIOS() = default

MPI Collective calls MPI_Comm_free Uses RAII for all other members

ADIOS &**operator=**(const *ADIOS*&) = delete

copy assignment is forbidden for the same reason as copy constructor

ADIOS &**operator=**(*ADIOS*&&) = default

move assignment is allowed, though, to be consistent with move constructor

IO **DeclareIO**(const std::string name, const ArrayOrdering ArrayOrder = ArrayOrdering::Auto)

Declares a new *IO* class object

Parameters
name – unique *IO* name identifier within current *ADIOS* object

Throws
`std::invalid_argument` – if *IO* with unique name is already declared

Returns
reference to newly created *IO* object inside current *ADIOS* object

IO **AtIO**(const std::string name)

Retrieve an existing *IO* object previously created with DeclareIO.

Parameters
name – *IO* unique identifier key in current *ADIOS* object

Throws
`std::invalid_argument` – if *IO* was not created with DeclareIO

Returns
if *IO* exists returns a reference to existing *IO* object inside *ADIOS*, else throws an exception. *IO* objects can't be invalid.

Operator **DefineOperator**(const std::string name, const std::string type, const Params &parameters = Params())

Defines an adios2 supported operator by its type.

Parameters

- **name** – unique operator name identifier within current *ADIOS* object
- **type** – supported ADIOS2 operator type: zfp, sz
- **parameters** – key/value parameters at the operator object level

Throws
`std::invalid_argument` – if adios2 can't support current operator due to missing dependency or unsupported type

Returns
Operator object

template<class **R**, class ...**Args**>
Operator **DefineOperator**(const std::string name, const std::function<*R*(*Args*...)> &function, const Params &parameters = Params())

Defines an adios2 supported operator by its type. Variadic template version for Operators of type Callback function with signatures suported in ADIOS2. For new signature support open an issue on github.

Parameters

- **name** – unique operator name within *ADIOS* object
- **function** – C++11 callable target
- **parameters** – key/value parameters at the operator level

Throws
`std::invalid_argument` – if adios2 can't support current operator due to missing dependency or unsupported type

Returns
Operator object

Operator **InquireOperator**(const std::string name)

Retrieve an existing *Operator* object in current *ADIOS* object

Parameters
name – *Operator* unique identifier key in current *ADIOS* object

Returns
object to an existing operator in current *ADIOS* object, *Operator* object is false if name is not found

void **FlushAll**()

Flushes all engines in write mode in all IOs created with the current *ADIOS* object. If no *IO* or *Engine* exist, it does nothing.

Throws
`std::runtime_error` – if any engine Flush fails

bool **RemoveIO**(const std::string name)

DANGER ZONE: removes a particular *IO*. This will effectively eliminate any parameter from the config.xml file

Parameters
name – io input name

Returns
true: *IO* was found and removed, false: *IO* not found and not removed

void **RemoveAllIOs**() noexcept

DANGER ZONE: removes all IOs created with DeclareIO. This will effectively eliminate any parameter from the config.xml file also.

void **EnterComputationBlock**() noexcept

Inform *ADIOS* about entering communication-free computation block in main thread. Useful when using Async *IO*

void **ExitComputationBlock**() noexcept

Inform *ADIOS* about exiting communication-free computation block in main thread. Useful when using Async *IO*

10.1.3 IO class

class **IO**

Public Functions

IO() = default

Empty (default) constructor, use it as a placeholder for future *IO* objects from ADIOS::IO functions. Can be used with STL containers.

~IO() = default

Use RAII

explicit **operator bool**() const noexcept

true: valid object, false: invalid object

std::string **Name**() const

Inspects *IO* name

Returns
name

bool **InConfigFile**() const

Checks if *IO* exists in a config file passed to *ADIOS* object that created this *IO*.

Returns
true: in config file, false: not in config file

void **SetEngine**(const std::string engineType)

Sets the engine type for current *IO* object.

Parameters
engineType – predefined engine type, default is bpfile

void **SetParameter**(const std::string key, const std::string value)

Sets a single parameter. Overwrites value if key exists;.

Parameters

- **key** – parameter key
- **value** – parameter value

void **SetParameters**(const adios2::Params &parameters = adios2::Params())

Version that passes a map to fill out parameters initializer list = { "param1", "value1" }, {"param2", "value2"}, Replaces any existing parameter. Otherwise use SetParameter for adding new parameters.

Parameters

parameters – adios::Params = std::map<std::string, std::string> key/value parameters

void **SetParameters**(const std::string &parameters)

Version that passes a single string to fill out many parameters. Replaces any existing parameter. initializer string = "param1=value1 , param2 = value2".

void **ClearParameters**()

Remove all existing parameters. Replaces any existing parameter. initializer string = "param1=value1 , param2 = value2".

adios2::Params **Parameters**() const

Return current parameters set from either SetParameters/SetParameter functions or from config XML for currrent *IO* object

Returns

string key/value map of current parameters (not modifiable)

size_t **AddTransport**(const std::string type, const adios2::Params &parameters = adios2::Params())

Adds a transport and its parameters to current *IO*. Must be supported by current *EngineType()*.

Parameters

- **type** – must be a supported transport type for a particular *Engine*. CAN'T use the keywords "Transport" or "transport"
- **parameters** – acceptable parameters for a particular transport

Throws

`std::invalid_argument` – if type=transport

Returns

transportIndex handler

void **SetTransportParameter**(const size_t transportIndex, const std::string key, const std::string value)

Sets a single parameter to an existing transport identified with a transportIndex handler from AddTransport. Overwrites existing parameter with the same key.

Parameters

- **transportIndex** – index handler from AddTransport
- **key** – parameter key
- **value** – parameter value

Throws

`std::invalid_argument` – if transportIndex not valid, e.g. not a handler from AddTransport.

template<class **T**>
Variable<*T*> **DefineVariable**(const std::string &name, const Dims &shape = Dims(), const Dims &start = Dims(), const Dims &count = Dims(), const bool constantDims = false)

Define a Variable<T> object within *IO*

Parameters

- **name** – unique variable identifier
- **shape** – global dimension
- **start** – local offset
- **count** – local dimension
- **constantDims** – true: shape, start, count won't change, false: shape, start, count will change after definition

Returns
Variable<T> object

template<class **T**>
Variable<*T*> **InquireVariable**(const std::string &name)

Retrieve a *Variable* object within current *IO* object

Parameters
name – unique variable identifier within *IO* object

Returns
if found *Variable* object is true and has functionality, else false and has no functionality

template<class **T**>
Attribute<*T*> **DefineAttribute**(const std::string &name, const *T* *data, const size_t size, const std::string &variableName = "", const std::string separator = "/", const bool allowModification = false)

Define attribute inside io. Array input version.

Parameters

- **name** – unique attribute identifier *IO* object or for a *Variable* if variableName is not empty (associated to a variable)
- **data** – pointer to user data
- **size** – number of data elements
- **variableName** – default is empty, if not empty attributes is associated to a variable
- **separator** – default is "/", hierarchy between variable name and attribute, e.g. variableName/attribute1, variableName::attribute1. Not used if variableName is empty.
- **allowModification** – true allows redefining existing attribute

Throws
`std::invalid_argument` – if *Attribute* with unique name (in *IO* or *Variable*) is already defined

Returns
object reference to internal *Attribute* in *IO*

template<class **T**>

Attribute<*T*> **DefineAttribute**(const std::string &name, const *T* &value, const std::string &variableName = "", const std::string separator = "/", const bool allowModification = false)

Define single value attribute.

Parameters

- **name** – must be unique for the *IO* object or for a *Variable* if variableName is not empty (associated to a variable)
- **value** – single data value
- **variableName** – default is empty, if not empty attributes is associated to a variable
- **separator** – default is "/", hierarchy between variable name and attribute, e.g. variableName/attribute1, variableName::attribute1. Not used if variableName is empty.
- **allowModification** – true allows redefining existing attribute

Throws

`std::invalid_argument` – if *Attribute* with unique name (in *IO* or *Variable*) is already defined

Returns

object reference to internal *Attribute* in *IO*

template<class **T**>
Attribute<*T*> **InquireAttribute**(const std::string &name, const std::string &variableName = "", const std::string separator = "/")

Retrieve an existing attribute.

Parameters

- **name** – must be unique for the *IO* object or for a *Variable* if variableName is not empty (associated to a variable)
- **variableName** – default is empty, if not empty attributes is expected to be associated to a variable
- **separator** – default is "/", hierarchy between variable name and attribute, e.g. variableName/attribute1, variableName::attribute1. Not used if variableName is empty.

Returns

object reference to internal *Attribute* in *IO*, object is false if *Attribute* is not found

bool **RemoveVariable**(const std::string &name)

DANGEROUS! Removes an existing *Variable* in current *IO* object. Might create dangling objects.

Parameters

name – unique *Variable* input

Returns

true: found and removed variable, false: not found, nothing to remove

void **RemoveAllVariables**()

DANGEROUS! Removes all existing variables in current *IO* object. Might create dangling objects.

bool **RemoveAttribute**(const std::string &name)

DANGEROUS! Removes an existing *Attribute* in current *IO* object. Might create dangling objects.

Parameters

name – unique *Attribute* identifier

Returns
true: found and removed attribute, false: not found, nothing to remove

void **RemoveAllAttributes**()

DANGEROUS! Removes all existing attributes in current *IO* object. Might create dangling objects.

Engine **Open**(const std::string &name, const Mode mode)

Open an *Engine* to start heavy-weight input/output operations.

Parameters

- **name** – unique engine identifier
- **mode** – adios2::Mode::Write, adios2::Mode::Read, adios2::Mode::ReadStreaming, or adios2::Mode::Append (BP4 only)

Returns
engine object

Group **InquireGroup**(char delimiter = '/')

Return a *Group* object for hierarchical reading.

Parameters

- **name** – starting path
- **a** – delimiter to separate groups in a string representation

Returns
Group object

Engine **Open**(const std::string &name, const Mode mode, MPI_Comm comm)

Open an *Engine* to start heavy-weight input/output operations. This version allows passing a MPI communicator different from the one used in the *ADIOS* object contructor MPI Collective function as it calls MPI_Comm_dup

Parameters

- **name** – unique engine identifier within *IO*
- **mode** – adios2::Mode::Write, adios2::Mode::Read, or adios2::Mode::Append (BP4 only)
- **comm** – new communicator other than *ADIOS* object's communicator

Returns
engine object

void **FlushAll**()

Flushes all engines created with this *IO* with the Open function

std::map<std::string, Params> **AvailableVariables**(bool namesOnly = false)

Returns a map with variable information.

- key: variable name
- value: Params is a map<string,string>
  - key: "Type", "Shape", "AvailableStepsCount", "Min", "Max", "SingleValue"

value: variable info value as string

Parameters
namesOnly – returns a map with the variable names but with no Parameters. Use this if you only need the list of variable names and call *VariableType()* and *InquireVariable()* on the names individually.

Returns
map<string, map<string, string>>

std::map<std::string, Params> **AvailableAttributes**(const std::string &variableName = "", const std::string separator = "/", const bool fullNameKeys = false)

Returns a map with available attributes information associated to a particular variableName

Parameters

- **variableName** – unique variable name associated with resulting attributes, if empty (default) return all attributes
- **separator** – optional name hierarchy separator (/, ::, _, -, \, etc.)
- **fullNameKeys** – true: return full attribute names in keys, false (default): return attribute names relative to variableName

Returns
map:

std::string **VariableType**(const std::string &name) const

Inspects variable type. This function can be used in conjunction with MACROS in an else if (type == adios2::GetType<T>()) {} loop

Parameters
name – unique variable name identifier in current *IO*

Returns
type as in adios2::GetType<T>() (e.g. "double", "float"), empty std::string if variable not found

std::string **AttributeType**(const std::string &name) const

Inspects attribute type. This function can be used in conjunction with MACROS in an else if (type == adios2::GetType<T>()) {} loop

Parameters
name – unique attribute name identifier in current *IO*

Returns
type as in adios2::GetType<T>() (e.g. "double", "float"), empty std::string if attribute not found

void **AddOperation**(const std::string &variable, const std::string &operatorType, const Params &parameters = Params())

Adds operation and parameters to current *IO* object

Parameters

- **variable** – variable to add operator to
- **operatorType** – operator type to define
- **parameters** – key/value settings particular to the *IO*, not to be confused by op own parameters

std::string **EngineType**() const

Inspect current engine type from SetEngine

Returns
current engine type

10.1.4 Variable <T> class

template<class **T**>

class **Variable**

Public Functions

Variable() = default

Empty (default) constructor, use it as a placeholder for future variables from *IO*:DefineVariable<T> or *IO*:InquireVariable<T>. Can be used with STL containers.

~Variable() = default

Default, using RAII STL containers

explicit **operator bool**() const noexcept

Checks if object is valid, e.g. if(variable) { //..valid }

void **SetMemorySpace**(const MemorySpace mem)

Sets the memory space for all following Puts to either host (default) or device (currently only CUDA supported)

Parameters
mem – memory space where Put buffers are allocated

MemorySpace **GetMemorySpace**()

Get the memory space that was set by the application

Returns
the memory space stored in the *Variable* object

void **SetShape**(const adios2::Dims &shape)

Set new shape, care must be taken when reading back the variable for different steps. Only applies to Global arrays.

Parameters
shape – new shape dimensions array

void **SetBlockSelection**(const size_t blockID)

Read mode only. Required for reading local variables, *ShapeID()* = ShapeID::LocalArray or ShapeID::LocalValue. For Global Arrays it will Set the appropriate Start and Count Selection for the global array coordinates.

Parameters
blockID – variable block index defined at write time. Blocks can be inspected with bpls -D variableName

void **SetSelection**(const adios2::Box<adios2::Dims> &selection)

Sets a variable selection modifying current {start, count} Count is the dimension from Start point

Parameters
selection – input {start, count}

void **SetMemorySelection**(const adios2::Box<adios2::Dims> &memorySelection)

Set the local start (offset) point to the memory pointer passed at Put and the memory local dimensions (count). Used for non-contiguous memory writes and reads (e.g. multidimensional ghost-cells). Currently Get only works for formats based on BP3.

Parameters
memorySelection – {memoryStart, memoryCount}

void **SetStepSelection**(const adios2::Box<size_t> &stepSelection)

Sets a step selection modifying current startStep, countStep countStep is the number of steps from startStep point

Parameters
stepSelection – input {startStep, countStep}

size_t **SelectionSize**() const

Returns the number of elements required for pre-allocation based on current count and stepsCount

Returns
elements of type T required for pre-allocation

std::string **Name**() const

Inspects *Variable* name

Returns
name

std::string **Type**() const

Inspects *Variable* type

Returns
type string literal containing the type: double, float, unsigned int, etc.

size_t **Sizeof**() const

Inspects size of the current element type, sizeof(T)

Returns
sizeof(T) for current system

adios2::ShapeID **ShapeID**() const

Inspects shape id for current variable

Returns
from enum adios2::ShapeID

adios2::Dims **Shape**(const size_t step = adios2::EngineCurrentStep) const

Inspects shape in global variables

Parameters
step – input for a particular Shape if changing over time. If default, either return absolute or in streaming mode it returns the shape for the current engine step

Returns
shape vector

adios2::Dims **Start**() const

Inspects current start point

Returns
start vector

adios2::Dims **Count**() const

Inspects current count from start

Returns
count vector

size_t **Steps**() const

For readRandomAccess mode, inspect the number of available steps

Returns
available steps

size_t **StepsStart**() const

For readRandomAccess mode, inspect the start step for available steps

Returns
available start step

size_t **BlockID**() const

For read mode, retrieve current BlockID, default = 0 if not set with SetBlockID

Returns
current block id

size_t **AddOperation**(const *Operator* op, const adios2::Params &parameters = adios2::Params())

Adds operation and parameters to current *Variable* object

Parameters

- **op** – operator to be added
- **parameters** – key/value settings particular to the *Variable*, not to be confused by op own parameters

Returns
operation index handler in *Operations()*

std::vector<*Operator*> **Operations**() const

Inspects current operators added with AddOperator

Returns
vector of Variable<T>::OperatorInfo

void **RemoveOperations**()

Removes all current Operations associated with AddOperation. Provides the posibility to apply or not operators on a block basis.

std::pair<*T*, *T*> **MinMax**(const size_t step = adios2::DefaultSizeT) const

Read mode only: return minimum and maximum values for current variable at a step. For streaming mode (BeginStep/EndStep): use default (leave empty) for current *Engine* Step At random access mode (File Engines only): default = absolute MinMax

Parameters
step – input step

Returns
pair.first = min pair.second = max

T **Min**(const size_t step = adios2::DefaultSizeT) const

Read mode only: return minimum values for current variable at a step. For streaming mode (within BeginStep/EndStep): use default (leave empty) for current *Engine* Step At random access mode (File Engines only): default = absolute MinMax

Parameters
step – input step

Returns
variable minimum

T **Max**(const size_t step = adios2::DefaultSizeT) const

Read mode only: return minimum values for current variable at a step. For streaming mode (within BeginStep/EndStep): use default (leave empty) for current *Engine* Step At random access mode (File Engines only): default = absolute MinMax

Parameters
step – input step

Returns
variable minimum

std::vector<std::vector<typename *Variable*<*T*>::Info>> **AllStepsBlocksInfo**()

Read mode only and random-access (no BeginStep/EndStep) with file engines only. Allows inspection of variable info on a per relative step (returned vector index) basis

Returns
first vector: relative steps, second vector: blocks info within a step

struct **Info**

Contains block information for a particular Variable<T>

Public Functions

const *T* *__Data__() const

reference to internal block data (used by inline *Engine*). For deferred variables, valid pointer is not returned until EndStep/PerformGets has been called.

Public Members

adios2::Dims **Start**

block start

adios2::Dims **Count**

block count

IOType **Min** = IOType()

block Min, if IsValue is false

IOType **Max** = IOType()

block Max, if IsValue is false

IOType **Value** = IOType()

block Value, if IsValue is true

int **WriterID** = 0

WriterID, source for stream ID that produced this block

size_t **BlockID** = 0

blockID for Block Selection

size_t **Step** = 0

block corresponding step

bool **IsReverseDims** = false

true: Dims were swapped from column-major, false: not swapped

bool **IsValue** = false

true: value, false: array

10.1.5 Attribute <T> class

template<class **T**>

class **Attribute**

Public Functions

Attribute() = default

Empty (default) constructor, use it as a placeholder for future attributes from *IO*:DefineAttribute<T> or *IO*:InquireAttribute<T>. Can be used with STL containers.

explicit **operator bool**() const noexcept

Checks if object is valid, e.g. if(attribute) { //..valid }

std::string **Name**() const

Inspect attribute name

Returns

unique name identifier

std::string **Type**() const

Inspect attribute type

Returns

type

std::vector<*T*> **Data**() const

Inspect attribute data

Returns

data

bool **IsValue**() const

Distinguish single-value attributes from vector attributes

Returns

true if single-value, false otherwise

10.1.6 Engine class

class **Engine**

Public Functions

Engine() = default

Empty (default) constructor, use it as a placeholder for future engines from *IO::Open*. Can be used with STL containers.

~Engine() = default

Using RAII STL containers only

explicit **operator bool**() const noexcept

true: valid engine, false: invalid, not created with *IO::Open* or post IO::Close

std::string **Name**() const

Inspect engine name

Returns
name from *IO::Open*

std::string **Type**() const

From ADIOS2 engine type: "bpfile", "sst", "dataman", "insitumpi", "hdf5"

Returns
engine type as lower case string

Mode **OpenMode**() const

Returns the Mode used at Open for current *Engine*

Returns

StepStatus **BeginStep**()

Begin a logical adios2 step, overloaded version with timeoutSeconds = 0 and mode = Read Check each engine documentation for MPI collective/non-collective behavior.

Returns
current step status

StepStatus **BeginStep**(const StepMode mode, const float timeoutSeconds = -1.f)

Begin a logical adios2 step, overloaded version for advanced stream control Check each engine documentation for MPI collective/non-collective behavior.

Parameters

- **mode** – see enum adios2::StepMode for options, Read is the common use case
- **timeoutSeconds** –

Returns
current step status

size_t **CurrentStep**() const

Inspect current logical step

Returns
current logical step

template<class **T**>
Variable<*T*>::Span **Put**(*Variable*<*T*> variable, const bool initialize, const *T* &value)

Put signature that provides access to the internal engine buffer for a pre-allocated variable including a fill value. Returns a fixed size Span (based on C++20 std::span) so applications can populate data value after this Put and before PerformPuts/EndStep. Requires a call to PerformPuts, EndStep, or Close to extract the Min/Max bounds.

Parameters

- **variable** – input variable
- **initialize** – bool flag indicating if allocated memory should be initialized with the provided value. Some engines (BP3/BP4) may initialize the allocated memory anyway to zero if this flag is false.
- **value** – provide an initial fill value

Returns
span to variable data in engine internal buffer

template<class **T**>
Variable<*T*>::Span **Put**(*Variable*<*T*> variable)

Put signature that provides access to an internal engine buffer (decided by the engine) for a pre-allocated variable. Allocated buffer may or may not be initialized to zero by the engine (e.g. BP3/BP4 does, BP5 does not).

Parameters
variable – input variable

Returns
span to variable data in engine internal buffer

template<class **T**>
void **Put**(*Variable*<*T*> variable, const *T* *data, const Mode launch = Mode::Deferred)

Put data associated with a *Variable* in the *Engine*

Parameters

- **variable** – contains variable metadata information
- **data** – user data to be associated with a variable
- **launch** – mode policy

Throws
`std::invalid_argument` – for invalid variable or nullptr data

template<class **T**>
void **Put**(const std::string &variableName, const *T* *data, const Mode launch = Mode::Deferred)

Put data associated with a *Variable* in the *Engine* Overloaded version that accepts a variable name string.

Parameters

- **variableName** – find variable by name inside *IO* that created this *Engine* with Open
- **data** – user data to be associated with a variable
- **launch** – mode policy

Throws
`std::invalid_argument` – if variable not found or nullptr data

template<class **T**>

void **Put**(*Variable*<*T*> variable, const *T* &datum, const Mode launch = Mode::Deferred)

Put data associated with a *Variable* in the *Engine* Overloaded version that accepts r-values and single variable data.

Parameters

- **variable** – contains variable metadata information
- **datum** – user data to be associated with a variable, r-value or single data value
- **launch** – mode policy, optional for API consistency, internally is always sync

Throws

`std::invalid_argument` – if variable is invalid or nullptr &datum

template<class **T**>
void **Put**(const std::string &variableName, const *T* &datum, const Mode launch = Mode::Deferred)

Put data associated with a *Variable* in the *Engine* Overloaded version that accepts variables names, and r-values and single variable data.

Parameters

- **variableName** – find variable by name inside *IO* that created this *Engine* with Open
- **datum** – user data to be associated with a variable r-value or single data value
- **launch** – mode policy, optional for API consistency, internally is always sync

Throws

`std::invalid_argument` – if variable is invalid or nullptr &datum

template<class **T**, typename **U**, class = typename std::enable_if<std::is_convertible<*U*, AdiosView<*U*>>::value>::type>
inline void **Put**(*Variable*<*T*> variable, *U* const &data, const Mode launch = Mode::Deferred)

The next two Put functions are used to accept a variable, and an AdiosViews which is a placeholder for Kokkos::View

Parameters

- **variable** – contains variable metadata information
- **data** – represents any user defined object that is not a vector (used for an AdiosView)
- **launch** – mode policy, optional for API consistency, internally is always sync

void **PerformPuts**()

Perform all Put calls in Deferred mode up to this point. Specifically, this causes Deferred data to be copied into *ADIOS* internal buffers as if the Put had been done in Sync mode.

void **PerformDataWrite**()

Write already-Put() array data to disk. If supported by the engine, this may relieve memory pressure by clearing *ADIOS* buffers. It is a collective call and can only be called between Begin/EndStep pairs.

template<class **T**>
void **Get**(*Variable*<*T*> variable, *T* *data, const Mode launch = Mode::Deferred)

Get data associated with a *Variable* from the *Engine*

Parameters

- **variable** – contains variable metadata information
- **data** – user data to be associated with a variable, it must be pre-allocated
- **launch** – mode policy

Throws
`std::invalid_argument` – for invalid variable or nullptr data

template<class **T**>
void **Get**(const std::string &variableName, *T* *data, const Mode launch = Mode::Deferred)

Get data associated with a *Variable* from the *Engine*. Overloaded version to get variable by name.

Parameters

- **variableName** – find variable by name inside *IO* that created this *Engine* with Open
- **data** – user data to be associated with a variable. It must be pre-allocated
- **launch** – mode policy

Throws
`std::invalid_argument` – for invalid variableName (variable doesn't exist in *IO*) or nullptr data

template<class **T**>
void **Get**(*Variable*<*T*> variable, *T* &datum, const Mode launch = Mode::Deferred)

Get single value data associated with a *Variable* from the *Engine* Overloaded version that accepts r-values and single variable data.

Parameters

- **variable** – contains variable metadata information
- **datum** – user data to be populated, r-value or single data value
- **launch** – mode policy, optional for API consistency, internally is always sync

Throws
`std::invalid_argument` – if variable is invalid or nullptr &datum

template<class **T**>
void **Get**(const std::string &variableName, *T* &datum, const Mode launch = Mode::Deferred)

Get single value data associated with a *Variable* from the *Engine* Overloaded version that accepts r-values and single variable data.

Parameters

- **variableName** – find variable by name inside *IO* that created this *Engine* with Open
- **datum** – user data to be populated, r-value or single data value
- **launch** – mode policy, optional for API consistency, internally is always sync

Throws
`std::invalid_argument` – for invalid variableName (variable doesn't exist in *IO*) or nullptr data

template<class **T**>
void **Get**(*Variable*<*T*> variable, std::vector<*T*> &dataV, const Mode launch = Mode::Deferred)

Get data associated with a *Variable* from the *Engine*. Overloaded version that accepts a std::vector without pre-allocation.

Parameters

- **variable** – contains variable metadata information
- **dataV** – user data vector to be associated with a variable, it doesn't need to be pre-allocated. *Engine* will resize.

- **launch** – mode policy

Throws

`std::invalid_argument` – for invalid variable

template<class **T**>
void **Get**(const std::string &variableName, std::vector<*T*> &dataV, const Mode launch = Mode::Deferred)

Get data associated with a *Variable* from the *Engine*. Overloaded version that accepts a std::vector without pre-allocation.

Parameters

- **variableName** – find variable by name inside *IO* that created this *Engine* with Open or BeginStep (streaming mode)
- **dataV** – user data vector to be associated with a variable, it doesn't need to be pre-allocated. *Engine* will resize.
- **launch** – mode policy

Throws

`std::invalid_argument` – for invalid variableName (variable doesn't exist in *IO*)

template<class **T**>
void **Get**(*Variable*<*T*> variable, typename *Variable*<*T*>::Info &info, const Mode launch = Mode::Deferred)

Get data associated with a *Variable* from the *Engine*. Data is associated with a block selection, and data is retrieved from variable's BlockInfo.

Note: Preliminary, experimental API, may change soon.

Parameters

- **variable** – contains variable metadata information
- **info** – block info struct associated with block selection, call will link with implementation's block info.
- **launch** – mode policy

Throws

`std::invalid_argument` – for invalid variable or nullptr data

template<class **T**>
void **Get**(const std::string &variableName, typename *Variable*<*T*>::Info &info, const Mode launch = Mode::Deferred)

Get data associated with a *Variable* from the *Engine*. Data is associated with a block selection, and data is retrieved from variable's BlockInfo. Overloaded version to get variable by name.

Note: Preliminary, experimental API, may change soon.

Parameters

- **variableName** – find variable by name inside *IO* that created this *Engine* with Open or BeginStep (streaming mode)
- **info** – block info struct associated with block selection, call will link with implementation's block info.

- **launch** – mode policy

Throws
`std::invalid_argument` – for invalid variableName (variable doesn't exist in *IO*)

template<class **T**>
void **Get**(*Variable*<*T*> variable, *T* **data) const

Assign the value of data to the start of the internal *ADIOS* buffer for variable variable. The value is immediately available.

template<class **T**, typename **U**, class = typename std::enable_if<std::is_convertible<*U*, AdiosView<*U*>>::value>::type>
inline void **Get**(*Variable*<*T*> variable, *U* const &data, const Mode launch = Mode::Deferred)

The next two Get functions are used to accept a variable, and an AdiosViews which is a placeholder for Kokkos::View

Parameters

- **variable** – contains variable metadata information
- **data** – represents any user defined object that is not a vector (used for an AdiosView)
- **launch** – mode policy, optional for API consistency, internally is always sync

void **PerformGets**()

Perform all Get calls in Deferred mode up to this point

void **EndStep**()

Ends current step, by default calls PerformsPut/Get internally Check each engine documentation for MPI collective/non-collective behavior.

bool **BetweenStepPairs**()

Returns True if engine status is between *BeginStep()*/EndStep() pair, False otherwise.

void **Flush**(const int transportIndex = -1)

Manually flush to underlying transport to guarantee data is moved

Parameters
transportIndex –

void **Close**(const int transportIndex = -1)

Closes current engine, after this call an engine becomes invalid MPI Collective, calls MPI_Comm_free for duplicated communicator at Open

Parameters
transportIndex –

template<class **T**>
std::map<size_t, std::vector<typename *Variable*<*T*>::Info>> **AllStepsBlocksInfo**(const *Variable*<*T*> variable) const

Extracts all available blocks information for a particular variable. This can be an expensive function, memory scales up with metadata: steps and blocks per step Valid in read mode only.

Parameters
variable –

Returns
map with all variable blocks information

template<class **T**>

std::vector<typename *Variable*<*T*>::Info> **BlocksInfo**(const *Variable*<*T*> variable, const size_t step) const

Extracts all available blocks information for a particular variable and step. Valid in read mode only.

Parameters

- **variable** – input variable
- **step** – input from which block information is extracted

Returns

vector of blocks with info for each block per step, if step not found it returns an empty vector

template<class **T**>
std::vector<size_t> **GetAbsoluteSteps**(const *Variable*<*T*> variable) const

Get the absolute steps of a variable in a file. This is for information purposes only, because absolute steps cannot be used in any ADIOS2 calls.

size_t **Steps**() const

Inspect total number of available steps, use for file engines in read mode only

Returns

available steps in engine

void **LockWriterDefinitions**()

Promise that no more definitions or changes to defined variables will occur. Useful information if called before the first *EndStep()* of an output *Engine*, as it will know that the definitions are complete and constant for the entire lifetime of the output and may optimize metadata handling.

void **LockReaderSelections**()

Promise that the reader data selections of are fixed and will not change in future timesteps. This information, provided before the *EndStep()* representing a fixed read pattern, may be utilized by the input *Engine* to optimize data flow.

10.1.7 Operator class

class **Operator**

Public Functions

Operator() = default

Empty (default) constructor, use it as a placeholder for future operators from *ADIOS::DefineOperator* functions. Can be used with STL containers.

explicit **operator bool**() const noexcept

true: valid object, false: invalid object

std::string **Type**() const noexcept

Inspect current *Operator* type

Returns

type as string, if invalid returns an empty std::string

void **SetParameter**(const std::string key, const std::string value)

Set a key/value parameters associated with this operator (global parameter from the object it's applied to: *Variable*, *IO*). If key exists, it replace the current value.

Parameters

- **key** – parameter key
- **value** – parameter value

Params &**Parameters**() const

Inspect current operator parameters

Returns

map of key/value parameters

Debugging

For debugging, ADIOS2 C++11 class instances and enums can be passed directly to ostreams, as well as converted to human-readable strings via the ubiquitous `ToString(object)` member variable. You can also directly pass objects to an `ostream`.

Example:

```
auto myVar = io.DefineVariable<double>("myVar");
std::cout << myVar << " has shape id " << myVar.ShapeID() << std::endl;

// will print:
// Variable<double>(Name: "myVar") has shape id ShapeID::GlobalValue

if (myVar.ShapeID() != adios2::ShapeID::GlobalArray)
{
    throw std::invalid_argument("can't handle " +
                                ToString(myVar.ShapeID()) + " in " +
                                ToString(myVar));
}

// will throw exception like this:
// C++ exception with description "can't handle ShapeID::GlobalValue
// in Variable<double>(Name: "myVar")" thrown
```

Group API

class **Group**

Public Functions

std::vector<std::string> **AvailableGroups**()

returns available groups on the path set

Param

std::vector<std::string> **AvailableVariables**()

returns available variables on the path set

Param

std::vector<std::string> **AvailableAttributes**()

returns available attributes on the path set

Param

std::string **InquirePath**()

returns the current path

Param

void **setPath**(std::string path)

set the path, points to a particular node on the tree

Parameters
next – possible path extension

Group **InquireGroup**(std::string group_name)

returns a new group object

Parameters
name – of the group

Returns
new group object

template<class **T**>
Variable<*T*> **InquireVariable**(const std::string &name)

Gets an existing variable of primitive type by name. A wrapper for the corresponding function of the *IO* class.

Parameters
name – of variable to be retrieved

Returns
pointer to an existing variable in current *IO*, nullptr if not found

template<class **T**>
Attribute<*T*> **InquireAttribute**(const std::string &name, const std::string &variableName = "", const std::string separator = "/")

Gets an existing attribute of primitive type by name. A wrapper for the corresponding function of the *IO* class

Parameters
name – of attribute to be retrieved

Returns
pointer to an existing attribute in current *IO*, nullptr if not found

DataType **VariableType**(const std::string &name) const

Inspects variable type. This function can be used in conjunction with MACROS in an else if (type == adios2::GetType<T>()) {} loop

Parameters
name – unique variable name identifier in current *IO*

Returns
type as in adios2::GetType<T>() (e.g. "double", "float"), empty std::string if variable not found

DataType **AttributeType**(const std::string &name) const

Inspects attribute type. This function can be used in conjunction with MACROS in an else if (type == adios2::GetType<T>()) {} loop

Parameters
name – unique attribute name identifier in current *IO*

Returns
type as in adios2::GetType<T>() (e.g. "double", "float"), empty std::string if attribute not found

The Group API can be used for inquiring other group objects, variables, and attributes by reading, provided that variable and attribute names were written in a tree-like way similar that is used in a file system:

```
"group1/group2/group3/variable1"
"group1/group2/group3/attribute1"
```

A group object containing the tree structure is obtained by the `InquireGroup` function of the `IO` object.

```
Group g = io.InquireGroup("group1");
```

Another group object can be generated by the predecessor group object.

```
Group g1 = g.InquireGroup("group2");
```

The absolute path can be inquired or set explicitly

```
std::string path =  g.InquirePath();
g.setPath("group1/group2/group3");
```

Names of available groups, variables and attributes could be inquired:

```
std::vector<std::string> groups = g.AvailableGroups();
std::vector<std::string> variables = g.AvailableVariables();
std::vector<std::string> attributes = g.AvailableVariables();
```

Finally, variables can be inquired

```
auto var = g.InquireVariable("variable1");
```

An extra function is provided that returns a type of a variable

```
DataType varType  = g.VariableType("variable1");
```

Step selection

Steps for reading can be selected using a pre-set parameter with as a file name as a key and a list of selected steps separated by comma as a value.

```
io.SetParameter(filename, "1,3");
```

10.2 Fortran bindings

The Fortran API is a collection of subroutine calls. The first argument is usually a Fortran type (struct) to an ADIOS2 component, while the last argument is an error integer flag, `integer ierr`. `ierr==0` represents successful execution whereas a non-zero value represents an error or a different state. ADIOS2 Fortran bindings provide a list of possible errors coming from the C++ standardized error exception library:

```
! error possible values for ierr
integer, parameter :: adios2_error_none = 0
integer, parameter :: adios2_error_invalid_argument = 1,
```

(continues on next page)

(continued from previous page)

```
integer, parameter :: adios2_error_system_error = 2,
integer, parameter :: adios2_error_runtime_error = 3,
integer, parameter :: adios2_error_exception = 4
```

Click here for a Fortran write and read example to illustrate the use of the APIs calls. This test will compile under your build/bin/ directory.

The following subsections describe the overall components and subroutines in the Fortran bindings API.

10.2.1 ADIOS2 typed handlers

ADIOS2 Fortran bindings handlers are mapped 1-to-1 to the ADIOS2 components described in the *Components Overview* section. For convenience, each type handler contains descriptive components used for read-only inspection.

```
type(adios2_adios)     :: adios
type(adios2_io)        :: io
type(adios2_variable)  :: variable
type(adios2_attribute) :: attribute
type(adios2_engine)    :: engine

!Read-only components for inspection and ( = defaults)

type adios2_adios
     logical :: valid = .false.
 end type

 type adios2_io
     logical :: valid = .false.
     character(len=15):: engine_type = 'BPFile'
 end type

 type adios2_variable
     logical :: valid = .false.
     character(len=4095):: name = ''
     integer :: type = -1
     integer :: ndims = -1
 end type

 type adios2_attribute
     logical :: valid = .false.
     character(len=4095):: name = ''
     integer :: type = -1
     integer :: length = 0
 end type

 type adios2_engine
     logical :: valid = .false.
     character(len=63):: name = ''
     character(len=15):: type = ''
     integer :: mode = adios2_mode_undefined
 end type
```

(continues on next page)

(continued from previous page)

```
type adios2_operator
    logical :: valid = .false.
    character(len=63):: name = ''
    character(len=63):: type = ''
end type
```

Caution: Use the type read-only components for information purposes only. Changing their values directly, *e.g.* `variable%name = new_name` does not have any effect inside the ADIOS2 library

10.2.2 ADIOS subroutines

- **subroutine** `adios2_init` starting point for the ADIOS2 library

```
! MPI versions
subroutine adios2_init(adios, comm, ierr)
subroutine adios2_init(adios, config_file, comm, ierr)

! Non-MPI serial versions
subroutine adios2_init(adios, ierr)
subroutine adios2_init(adios, config_file, ierr)

! WHERE:

! ADIOS2 handler to allocate
type(adios2_adios), intent(out):: adios

! MPI Communicator
integer, intent(in):: comm

! Optional runtime configuration file (*.xml), see Runtime Configuration␣
↪Files
character*(*), intent(in) :: config_file

! error code
integer, intent(out) :: ierr
```

- **subroutine** `adios2_declare_io` spawn/create an IO component

```
subroutine adios2_declare_io(io, adios, io_name, ierr)

! WHERE:

! output ADIOS2 IO handler
type(adios2_io), intent(out):: io

! ADIOS2 component from adios2_init spawning io tasks
type(adios2_adios), intent(in):: adios

! unique name associated with this IO component inside ADIOS2
character*(*), intent(in):: io_name
```

(continues on next page)

(continued from previous page)

```
! error code
integer, intent(out) :: ierr
```

- **subroutine** adios2_at_io retrieve an existing io component. Useful when the original IO handler goes out of scope

```
subroutine adios2_at_io(io, adios, io_name, ierr)

! WHERE:

! output IO handler
type(adios2_io), intent(out):: io

! ADIOS2 component from adios2_init that owns IO tasks
type(adios2_adios), intent(in):: adios

! unique name associated with an existing IO component (created with adios2_
↪declare_io)
character*(*), intent(in):: io_name

! error code
integer, intent(out) :: ierr
```

- **subroutine** adios2_define_operator define an ADIOS2 data compression/reduction operator

```
subroutine adios2_define_operator(op, adios, op_name, op_type, ierr)

! WHERE

! Operator handler
type(adios2_operator), intent(out) :: op

! ADIOS2 handler
type(adios2_adios), intent(in) :: adios

! Operator name
character*(*), intent(in)  :: op_name

! Operator type
character*(*), intent(in)  :: op_type

! error code
integer, intent(out) :: ierr
```

- **subroutine** adios2_inquire_operator inquire an ADIOS2 data compression/reduction operator

```
subroutine adios2_inquire_operator(op, adios, op_name, ierr)

! WHERE

! Operator handler
type(adios2_operator), intent(out) :: op
```

(continues on next page)

(continued from previous page)

```
! ADIOS2 handler
type(adios2_adios), intent(in) :: adios

! Operator name
character*(*), intent(in)  :: op_name

! error code
integer, intent(out) :: ierr
```

- **subroutine** adios2_flush_all flush all current engines in all IO objects

```
subroutine adios2_flush_all(adios, ierr)

! WHERE:

! ADIOS2 component from adios2_init owning IO objects and engines
type(adios2_adios), intent(in):: adios

! error code
integer, intent(out) :: ierr
```

- **subroutine** adios2_remove_io DANGER ZONE: remove an IO object. This will effectively eliminate any parameter from the config xml file

```
subroutine adios2_remove_io(result, adios, name, ierr)

! WHERE

! Returns True if IO was found, False otherwise
logical, intent(out):: result

! ADIOS2 handler
type(adios2_adios), intent(in) :: adios

! IO input name
character*(*), intent(in):: name

! error code
integer, intent(out) :: ierr
```

- **subroutine** adios2_remove_all_ios DANGER ZONE: remove all IO objects created for this ADIOS2 handler. This will effectively eliminate any parameter from the config xml file as well.

```
subroutine adios2_remove_all_ios(adios, ierr)

! WHERE

! ADIOS2 handler
type(adios2_adios), intent(in) :: adios

! error code
integer, intent(out) :: ierr
```

- **subroutine** adios2_finalize final point for the ADIOS2 component

```
subroutine adios2_finalize(adios, ierr)

! WHERE:

! ADIOS2 handler to be deallocated
type(adios2_adios), intent(in):: adios

! error code
integer, intent(out) :: ierr
```

Caution: Make sure that for every call to `adios2_init` there is a call to `adios2_finalize` for the same ADIOS2 handler. Not doing so will result in memory leaks.

- **subroutine** adios2_enter_computation_block inform ADIOS2 about entering communication-free computation block in main thread. Useful when using Async IO.

```
subroutine adios2_enter_computation_block(adios, ierr)

! WHERE

! ADIOS2 handler
type(adios2_adios), intent(in) :: adios

! error code
integer, intent(out) :: ierr
```

- **subroutine** adios2_exit_computation_block inform ADIOS2 about exiting communication-free computation block in main thread. Useful when using Async IO.

```
subroutine adios2_exit_computation_block(adios, ierr)

! WHERE

! ADIOS2 handler
type(adios2_adios), intent(in) :: adios

! error code
integer, intent(out) :: ierr
```

10.2.3 IO subroutines

- **subroutine** adios2_set_engine set the engine type, see *Supported Engines* for a list of available engines

```
subroutine adios2_set_engine(io, engine_type, ierr)

! WHERE:

! IO component
type(adios2_io), intent(in):: io
```

(continues on next page)

(continued from previous page)

```
! engine_type: BP (default), HDF5, DataMan, SST, SSC
character*(*), intent(in):: engine_type

! error code
integer, intent(out) :: ierr
```

- **subroutine** adios2_in_config_file checks if an IO object exists in a config file passed to ADIOS2.

```
subroutine adios2_in_config_file(result, io, ierr)

! WHERE

! Output result to indicate whether IO exists
logical, intent(out):: result

! IO handler
type(adios2_io), intent(in):: io

! error code
integer, intent(out):: ierr
```

- **subroutine** adios2_set_parameter set IO key/value pair parameter in an IO object, see *Supported Engines* for a list of available parameters for each engine type

```
subroutine adios2_set_parameter(io, key, value, ierr)

! WHERE:

! IO component owning the attribute
type(adios2_io), intent(in):: io

! key in the key/value pair parameter
character*(*), intent(in):: key

! value in the key/value pair parameter
character*(*), intent(in):: value

! error code
integer, intent(out) :: ierr
```

- **subroutine** adios2_set_parameters set a map of key/value parameters in an IO object. Replaces any existing parameters. Otherwise use set_parameter for adding single parameters.

```
subroutine adios2_set_parameters(io, parameters, ierr)

! WHERE

! IO handler
type(adios2_io), intent(in) :: io

! Comma-separated parameter list. E.g. "Threads=2, CollectiveMetadata=OFF"
character*(*), intent(in) :: parameters
```

(continues on next page)

(continued from previous page)

```
! error code
integer, intent(out) :: ierr
```

- **subroutine** adios2_get_parameter get parameter value from IO object for a given parameter name

```
subroutine adios2_get_parameter(value, io, key, ierr)

! WHERE

! parameter value
character(len=:), allocatable, intent(out) :: value

! IO handler
type(adios2_io), intent(in) :: io

! parameter key to look for in the IO object
character*(*), intent(in) :: key

! error code
integer, intent(out) :: ierr
```

- **subroutine** adios2_clear_parameters clear all parameters from the IO object

```
subroutine adios2_clear_parameters(io, ierr)

! WHERE

! IO handler
type(adios2_io), intent(in) :: io

! error code
integer, intent(out) :: ierr
```

- **subroutine** adios2_add_transport add a transport to current IO. Must be supported by the currently used engine.

```
subroutine adios2_add_transport(transport_index, io, type, ierr)

! WHERE

! returns a transport_index handler
integer, intent(out):: transport_index

! IO handler
type(adios2_io), intent(in) :: io

! transport type. must be supported by the engine. CAN'T use the keywords␣
↪"Transport" or "transport"
character*(*), intent(in) :: type

! error code
integer, intent(out) :: ierr
```

- **subroutine** adios2_set_transport_parameter set a parameter for a transport. Overwrites existing parameter with the same key.

```
subroutine adios2_set_transport_parameter(io, transport_index, key, value,␣
↪ierr)

! WHERE

! IO handler
type(adios2_io), intent(in):: io

! transport_index handler
integer, intent(in):: transport_index

! transport key
character*(*), intent(in) :: key

! transport value
character*(*), intent(in) :: value

! error code
integer, intent(out):: ierr
```

- **subroutine** adios2_available_variables get a list of available variables

```
subroutine adios2_available_variables(io, namestruct, ierr)

! WHERE

! IO handler
type(adios2_io), intent(in) :: io

! name struct handler
type(adios2_namestruct), intent(out) :: namestruct

! error code
integer, intent(out) :: ierr
```

- **subroutine** adios2_retrieve_names retrieve variable names from namestruct obtained from adios2_available_variables. namelist must be pre-allocated.

```
subroutine adios2_retrieve_names(namestruct, namelist, ierr)

! WHERE

! namestruct obtained from adios2_available_variables
type(adios2_namestruct), intent(inout) :: namestruct

! namelist that will contain variable names
character(*), dimension(*), intent(inout) :: namelist

! error code
integer, intent(out) :: ierr
```

- **subroutine** adios2_available_attributes get list of attributes in the IO object

```
subroutine adios2_available_attributes(io, namestruct, ierr)

! WHERE

! IO handler
type(adios2_io), intent(in) :: io

! list of available attributes
type(adios2_namestruct), intent(out) :: namestruct

! error code
integer, intent(out) :: ierr
```

- **subroutine** adios2_flush_all_engines flush all existing engines opened by this IO object

```
subroutine adios2_flush_all_engines(io, ierr)

! WHERE:

! IO in which search and flush for all engines is performed
type(adios2_io), intent(in) :: io

! error code
integer, intent(out) :: ierr
```

- **subroutine** adios2_remove_variable remove an existing variable from an IO object

```
subroutine adios2_remove_variable(io, name, result, ierr)

! WHERE:

! IO in which search and removal for variable is performed
type(adios2_io), intent(in) :: io

! unique key name to search for variable
character*(*), intent(in) :: name

! true: variable removed, false: variable not found, not removed
logical, intent(out) :: result

! error code
integer, intent(out) :: ierr
```

- **subroutine** adios2_remove_all_variables remove all existing variables from an IO object

```
subroutine adios2_remove_all_variables(io, ierr)

! WHERE:

! IO in which search and removal for all variables is performed
type(adios2_io), intent(in) :: io

! error code
integer, intent(out) :: ierr
```

- **subroutine** adios2_remove_attribute remove existing attribute by its unique name

```
subroutine adios2_remove_attribute(io, name, result, ierr)

! WHERE:

! IO in which search and removal for attribute is performed
type(adios2_io), intent(in) :: io

! unique key name to search for attribute
character*(*), intent(in) :: name

! true: attribute removed, false: attribute not found, not removed
logical, intent(out) :: result

! error code
integer, intent(out) :: ierr
```

- **subroutine** adios2_remove_all_attributes remove all existing attributes

```
subroutine adios2_remove_all_attributes(io, ierr)

! WHERE:

! IO in which search and removal for all attributes is performed
type(adios2_io), intent(in) :: io

! error code
integer, intent(out) :: ierr
```

10.2.4 Variable subroutines

- **subroutine** adios2_define_variable define/create a new variable

```
! Global array variables
subroutine adios2_define_variable(variable, io, variable_name, adios2_type,␣
↪&
                                  ndims, shape_dims, start_dims, count_dims,
↪ &
                                  adios2_constant_dims, ierr)
! Global single value variables
subroutine adios2_define_variable(variable, io, variable_name, adios2_type,␣
↪ierr)

! WHERE:

! handler to newly defined variable
type(adios2_variable), intent(out):: variable

! IO component owning the variable
type(adios2_io), intent(in):: io
```

(continues on next page)

(continued from previous page)

```
! unique variable identifier within io
character*(*), intent(in):: variable_name

! defines variable type from adios2 parameters, see next
integer, intent(in):: adios2_type

! number of dimensions
integer, value, intent(in):: ndims

! variable shape, global size, dimensions
! to create local variables optional pass adios2_null_dims
integer(kind=8), dimension(:), intent(in):: shape_dims

! variable start, local offset, dimensions
! to create local variables optional pass adios2_null_dims
integer(kind=8), dimension(:), intent(in):: start_dims

! variable count, local size, dimensions
integer(kind=8), dimension(:), intent(in):: count_dims

! error code
integer, intent(out) :: ierr

! .true. : constant dimensions, shape, start and count won't change
!          (mesh sizes, number of nodes)
!          adios2_constant_dims = .true. use for code clarity
! .false. : variable dimensions, shape, start and count could change
!           (number of particles)
!           adios2_variable_dims = .false. use for code clarity
logical, value, intent(in):: adios2_constant_dims
```

- available `adios2_type` parameters in **subroutine** `adios2_define_variable`

```
integer, parameter :: adios2_type_character = 0
integer, parameter :: adios2_type_real = 2
integer, parameter :: adios2_type_dp = 3
integer, parameter :: adios2_type_complex = 4
integer, parameter :: adios2_type_complex_dp = 5

integer, parameter :: adios2_type_integer1 = 6
integer, parameter :: adios2_type_integer2 = 7
integer, parameter :: adios2_type_integer4 = 8
integer, parameter :: adios2_type_integer8 = 9

integer, parameter :: adios2_type_string = 10
integer, parameter :: adios2_type_string_array = 11
```

Tip: Always prefer using `adios2_type_xxx` parameters explicitly rather than raw numbers. *e.g.* use `adios2_type_dp` instead of `3`

- **subroutine** `adios2_inquire_variable` inquire and get a variable. See *variable%valid* to check if variable

exists.

```
subroutine adios2_inquire_variable(variable, io, name, ierr)

! WHERE:

! output variable handler:
! variable%valid = .true. points to valid found variable
! variable%valid = .false. variable not found
type(adios2_variable), intent(out) :: variable

! IO in which search for variable is performed
type(adios2_io), intent(in) :: io

! unique key name to search for variable
character*(*), intent(in) :: name

! error code
integer, intent(out) :: ierr
```

- **subroutine** adios2_set_shape set new shape_dims for a variable if its dims are marked as varying in the define call adios2_define_variable

```
subroutine adios2_set_shape(variable, ndims, shape_dims, ierr)

! WHERE

! variable handler
type(adios2_variable), intent(in) :: variable

! number of dimensions in shape_dims
integer, intent(in) :: ndims

! new shape_dims
integer(kind=8), dimension(:), intent(in):: shape_dims

! error code
integer, intent(out) :: ierr
```

- **subroutine** adios2_set_selection selects part of a variable through start_dims and count_dims

```
subroutine adios2_set_selection(variable, ndims, start_dims, count_dims,␣
↪ierr)

! WHERE

! variable handler
type(adios2_variable), intent(in) :: variable

! number of dimensions in start_dims and count_dims
integer, intent(in) :: ndims

! new start_dims
integer(kind=8), dimension(:), intent(in):: start_dims
```

(continues on next page)

(continued from previous page)

```
! new count_dims
integer(kind=8), dimension(:), intent(in):: count_dims

! error code
integer, intent(out) :: ierr
```

- **subroutine** adios2_set_steps_selection set a list of steps by specifying the starting step and the step count

```
subroutine adios2_set_selection(variable, step_start, step_count, ierr)

! WHERE

! variable handler
type(adios2_variable), intent(in) :: variable

! new step_start
integer(kind=8), intent(in):: step_start

! new step_count (or number of steps to read from step_start)
integer(kind=8), intent(in):: step_count

! error code
integer, intent(out) :: ierr
```

- **subroutine** adios2_variable_max get the maximum value in the variable array

```
subroutine adios2_variable_max(maximum, variable, ierr)

! WHERE

! scalar variable that will contain the maximum value
Generic Fortran types, intent(out) :: maximum

! variable handler
type(adios2_variable), intent(in) :: variable

! error code
integer, intent(out) :: ierr
```

- **subroutine** adios2_variable_min get the minimum value in the variable array

```
subroutine adios2_variable_min(minimum, variable, ierr)

! WHERE

! scalar variable that will contain the minimum value
Generic Fortran types, intent(out) :: minimum

! variable handler
type(adios2_variable), intent(in) :: variable
```

(continues on next page)

(continued from previous page)

```
! error code
integer, intent(out) :: ierr
```

- **subroutine** adios2_add_operation add an operation to a variable

```
subroutine adios2_add_operation(operation_index, variable, op, key, value,␣
↪ierr)

! WHERE

! reference to the operator handle that will be created
integer, intent(out):: operation_index

! variable handler
type(adios2_variable), intent(in):: variable

! Operator handler
type(adios2_operator), intent(in):: op

! Operator key
character*(*), intent(in):: key

! Operator value
character*(*), intent(in):: value

! error code
integer, intent(out):: ierr
```

- **subroutine** adios2_set_operation_parameter set a parameter for a operator. Replaces value if parameter already exists.

```
subroutine adios2_set_operation_parameter(variable, operation_index, key,␣
↪value, ierr)

! WHERE

! variable handler
type(adios2_variable), intent(in):: variable

! Operation index handler
integer, intent(in):: operation_index

! parameter key
character*(*), intent(in):: key

! parameter value
character*(*), intent(in):: value

! error code
integer, intent(out):: ierr
```

- **subroutine** adios2_variable_name retrieve variable name

```
subroutine adios2_variable_name(name, variable, ierr)

! WHERE

! variable name
character(len=:), allocatable, intent(out) :: name

! variable handler
type(adios2_variable), intent(in) :: variable

! error code
integer, intent(out) :: ierr
```

- **subroutine** adios2_variable_type retrieve variable datatype

```
subroutine adios2_variable_type(type, variable, ierr)

! WHERE

! variable type
integer, intent(out) :: type

! variable handler
type(adios2_variable), intent(in) :: variable

! error code
integer, intent(out) :: ierr
```

- **subroutine** adios2_variable_ndims retrieve number of dimensions for a variable

```
subroutine adios2_variable_ndims(ndims, variable, ierr)

! WHERE

! No. of dimensions
integer, intent(out) :: ndims

! variable handler
type(adios2_variable), intent(in) :: variable

! error code
integer, intent(out) :: ierr
```

- **subroutine** adios2_variable_shape retrieve the shape of a variable

```
subroutine adios2_variable_shape(shape_dims, ndims, variable, ierr)

! WHERE

! array that contains the shape
integer(kind=8), dimension(:), allocatable, intent(out) :: shape_dims

! no. of dimensions
```

(continues on next page)

(continued from previous page)

```
integer, intent(out) :: ndims

! variable handler
type(adios2_variable), intent(in) :: variable

! error code
integer, intent(out) :: ierr
```

- **subroutine** adios2_variable_steps retrieve the number of available steps

```
subroutine adios2_variable_steps(steps, variable, ierr)

! WHERE

! no. of steps
integer(kind=8), intent(out) :: steps

! variable handler
type(adios2_variable), intent(in) :: variable

! error code
integer, intent(out) :: ierr
```

- **subroutine** adios2_set_block_selection Read mode only. Required for reading local variables. For global arrays it will set the appropriate Start and Count selection for the global array coordinates.

```
subroutine adios2_set_block_selection(variable, block_id, ierr)

! WHERE

! variable handler
type(adios2_variable), intent(in) :: variable

! variable block index defined at write time. Blocks can be inspected with␣
↪`bpls -D variableName`
integer(kind=8), intent(in) :: block_id

! error code
integer, intent(out) :: ierr
```

- **subroutine** adios2_set_memory_selection set the local start (offset) point to the memory pointer passed at adios2_put and the memory local dimensions (count). Used for non-contiguous memory writes and reads (e.g. multidimensional ghost-cells). Currently Get only works for formats based on BP.

```
subroutine adios2_set_memory_selection(variable, ndims, memory_start_dims,␣
↪memory_count_dims, ierr)

! WHERE

! variable handler
type(adios2_variable), intent(in) :: variable

! no. of dimensions of the variable
```

(continues on next page)

(continued from previous page)

```
integer, intent(in) :: ndims

! memory start offsets
integer(kind=8), dimension(:), intent(in) :: memory_start_dims

! no. of elements in each dimension
integer(kind=8), dimension(:), intent(in) :: memory_count_dims

! error code
integer, intent(out) :: ierr
```

- **subroutine** adios2_set_step_selection set a step selection modifying current step_start, step_count. step_count is the number of steps from step_start

```
subroutine adios2_set_step_selection(variable, step_start, step_count, ierr)

! WHERE

! variable handler
type(adios2_variable), intent(in) :: variable

! starting step
integer(kind=8), intent(in) :: step_start

! no. of steps from start
integer(kind=8), intent(in) :: step_count

! error code
integer, intent(out) :: ierr
```

- **subroutine** adios2_remove_operations remove all current operations associated with the variable. Provides the posibility to apply operators on a block basis.

```
subroutine adios2_remove_operations(variable, ierr)

! WHERE

! variable handler
type(adios2_variable), intent(in):: variable

! error code
integer, intent(out):: ierr
```

10.2.5 Engine subroutines

- **subroutine** adios2_open opens an engine to execute IO tasks

```
! MPI version: duplicates communicator from adios2_init
! Non-MPI serial version
subroutine adios2_open(engine, io, name, adios2_mode, ierr)

! MPI version only to pass a communicator other than the one from adios_init
subroutine adios2_open(engine, io, name, adios2_mode, comm, ierr)

! WHERE:

! handler to newly opened adios2 engine
type(adios2_engine), intent(out) :: engine

! IO that spawns an engine based on its configuration
type(adios2_io), intent(in) :: io

! unique engine identifier within io, file name for default BPFile engine
character*(*), intent(in) :: name

! Optional MPI communicator, only in MPI library
integer, intent(in) :: comm

! error code
integer, intent(out) :: ierr

! open mode parameter:
!                      adios2_mode_write,
!                      adios2_mode_append,
!                      adios2_mode_read,
integer, intent(in):: adios2_mode
```

- **subroutine** adios2_begin_step begin a new step or progress to the next step. Starts from 0

```
subroutine adios2_begin_step(engine, adios2_step_mode, timeout_seconds,␣
↪status, ierr)
! Default Timeout = -1.    (block until step available)
subroutine adios2_begin_step(engine, adios2_step_mode, ierr)
! Default step_mode for read and write
subroutine adios2_begin_step(engine, ierr)

! WHERE

! engine handler
type(adios2_engine), intent(in) :: engine

! step_mode parameter:
!     adios2_step_mode_read (read mode default)
!     adios2_step_mode_append (write mode default)
integer, intent(in):: adios2_step_mode

! optional
```

(continues on next page)

(continued from previous page)

```
! engine timeout (if supported), in seconds
real, intent(in):: timeout_seconds

! status of the stream from adios2_step_status_* parameters
integer, intent(out):: status

! error code
integer, intent(out) :: ierr
```

- **subroutine** adios2_current_step extracts current step number

```
subroutine adios2_current_step(current_step, engine, ierr)

! WHERE:

! engine handler
type(adios2_engine), intent(in) :: engine

! populated with current_step value
integer(kind=8), intent(out) :: current_step

! error code
integer, intent(out) :: ierr
```

- **subroutine** adios2_steps inspect total number of available steps, use for file engines in read mode only

```
subroutine adios2_steps(steps, engine, ierr)

! WHERE:

! engine handler
type(adios2_engine), intent(in) :: engine

! populated with steps value
integer(kind=8), intent(out) :: steps

! error code
integer, intent(out) :: ierr
```

- **subroutine** adios2_end_step end current step and execute transport IO (flush or read).

```
subroutine adios2_end_step(engine, ierr)

! WHERE:

! engine handler
type(adios2_engine), intent(in) :: engine

! error code
integer, intent(out) :: ierr
```

- **subroutine** adios2_put put variable data and metadata into adios2 for IO operations. Default is deferred mode. For optional sync mode, see *Put: modes and memory contracts*. Variable and data types must match.

```
subroutine adios2_put(engine, variable, data, adios2_mode, ierr)

! Default adios2_mode_deferred
subroutine adios2_put(engine, variable, data, ierr)

! WHERE:

! engine handler
type(adios2_engine), intent(in) :: engine

! variable handler containing metadata information
type(adios2_variable), intent(in) :: variable

! Fortran bindings supports data types from adios2_type in variables,
! up to 6 dimensions
! Generic Fortran type from adios2_type
Generic Fortran types, intent(in):: data
Generic Fortran types, dimension(:), intent(in):: data
Generic Fortran types, dimension(:,:), intent(in):: data
Generic Fortran types, dimension(:,:,:), intent(in):: data
Generic Fortran types, dimension(:,:,:,:), intent(in):: data
Generic Fortran types, dimension(:,:,:,:,:), intent(in):: data
Generic Fortran types, dimension(:,:,:,:,:,:), intent(in):: data

! mode:
! adios2_mode_deferred: won't execute until adios2_end_step, adios2_perform_
↪puts or adios2_close
! adios2_mode_sync: special case, put data immediately, can be reused after␣
↪this call
integer, intent(in):: adios2_mode

! error code
integer, intent(out) :: ierr
```

- **subroutine** adios2_perform_puts execute deferred calls to adios2_put

```
subroutine adios2_perform_puts(engine, ierr)

! WHERE:

! engine handler
type(adios2_engine), intent(in) :: engine

! error code
integer, intent(out) :: ierr
```

- **subroutine** adios2_get get variable data into ADIOS2 for IO operations. Default is deferred mode. For optional sync mode, see *Get: modes and memory contracts*. Variable and data types must match, variable can be obtained from adios2_inquire_variable. Memory for data must be pre-allocated.

```
subroutine adios2_get(engine, variable, data, adios2_mode, ierr)

! Default adios2_mode_deferred
```

(continues on next page)

(continued from previous page)

```
subroutine adios2_get(engine, variable, data, ierr)

! WHERE:

! engine handler
type(adios2_engine), intent(in) :: engine

! variable handler containing metadata information
type(adios2_variable), intent(in) :: variable

! Fortran bindings supports data types from adios2_type in variables,
! up to 6 dimensions. Must be pre-allocated
! Generic Fortran type from adios2_type
Generic Fortran types, intent(out):: data
Generic Fortran types, dimension(:), intent(out):: data
Generic Fortran types, dimension(:,:), intent(out):: data
Generic Fortran types, dimension(:,:,:), intent(out):: data
Generic Fortran types, dimension(:,:,:,:), intent(out):: data
Generic Fortran types, dimension(:,:,:,:,:), intent(out):: data
Generic Fortran types, dimension(:,:,:,:,:,:), intent(out):: data

! mode:
! adios2_mode_deferred: won't execute until adios2_end_step, adios2_perform_
↪gets or adios2_close
! adios2_mode_sync: special case, get data immediately, can be reused after␣
↪this call
integer, intent(in):: adios2_mode

! error code
integer, intent(out) :: ierr
```

- **subroutine** adios2_perform_gets execute deferred calls to adios2_get

```
subroutine adios2_perform_gets(engine, ierr)

! WHERE:

! engine handler
type(adios2_engine), intent(in) :: engine

! error code
integer, intent(out) :: ierr
```

- **subroutine** adios2_close close engine. May re-open.

```
subroutine adios2_close(engine, ierr)

! WHERE:

! engine handler
type(adios2_engine), intent(in) :: engine
```

(continues on next page)

(continued from previous page)

```
! error code
integer, intent(out) :: ierr
```

- **subroutine** adios2_io_engine_type get current engine type

```
subroutine adios2_io_engine_type(type, io, ierr)

! WHERE

! engine type (BP, SST, SSC, HDF5, DataMan)
character(len=:), allocatable, intent(out) :: type

! IO handler
type(adios2_io), intent(in) :: io

! error code
integer, intent(out) :: ierr
```

- **subroutine** adios2_lock_writer_definitions promise that no more definitions or changes to defined variables will occur. Useful information if called before the first EndStep() of an output Engine, as it will know that the definitions are complete and constant for the entire lifetime of the output and may optimize metadata handling.

```
subroutine adios2_lock_writer_definitions(engine, ierr)

! WHERE

! adios2 engine handler
  type(adios2_engine), intent(in) :: engine

! error code
  integer, intent(out) :: ierr
```

- **subroutine** adios2_lock_reader_selections promise that the reader data selections of are fixed and will not change in future timesteps. This information, provided before the end_step() representing a fixed read pattern, may be utilized by the input Engine to optimize data flow.

```
subroutine adios2_lock_reader_selections(engine, ierr)

! WHERE

! adios2 engine handler
  type(adios2_engine), intent(in) :: engine

! error code
  integer, intent(out) :: ierr
```

10.2.6 Operator subroutines

- **subroutine** adios2_operator_type get current Operator type

```
subroutine adios2_operator_type(type, op, ierr)

! WHERE

! Operator type name. See list of supported operator types.
character(len=:), allocatable, intent(out) :: type

! Operator handler
type(adios2_operator), intent(in) :: op

! error code
integer, intent(out) :: ierr
```

10.2.7 Attribute subroutines

- **subroutine** adios2_define_attribute define/create a new user attribute

```
! Single value attributes
subroutine adios2_define_attribute(attribute, io, attribute_name, data,␣
↪ierr)

! 1D array attributes
subroutine adios2_define_attribute(attribute, io, attribute_name, data,␣
↪elements, ierr)

! WHERE:

! handler to newly defined attribute
type(adios2_attribute), intent(out):: attribute

! IO component owning the attribute
type(adios2_io), intent(in):: io

! unique attribute identifier within io
character*(*), intent(in):: attribute_name

! overloaded subroutine allows for multiple attribute data types
! they can be single values or 1D arrays
Generic Fortran types, intent(in):: data
Generic Fortran types, dimension(:), intent(in):: data

! number of elements if passing a 1D array in data argument
integer, intent(in):: elements

! error code
integer, intent(out) :: ierr
```

- **subroutine** adios2_inquire_attribute inquire for existing attribute by its unique name

```
subroutine adios2_inquire_attribute(attribute, io, name, ierr)

! WHERE:

! output attribute handler:
! attribute%valid = .true. points to valid found attribute
! attribute%valid = .false. attribute not found
type(adios2_attribute), intent(out) :: attribute

! IO in which search for attribute is performed
type(adios2_io), intent(in) :: io

! unique key name to search for attribute
character*(*), intent(in) :: name

! error code
integer, intent(out) :: ierr
```

Caution: Use the `adios2_remove_*` subroutines with extreme CAUTION. They create outdated dangling information in the `adios2_type` handlers. If you don't need them, don't use them.

- **subroutine** `adios2_attribute_data` retrieve attribute data

```
subroutine adios2_attribute_data(data, attribute, ierr)

! WHERE

! data handler
character*(*), intent(out):: data
real, intent(out):: data
real(kind=8), intent(out):: data
integer(kind=1), intent(out):: data
integer(kind=2), intent(out):: data
integer(kind=4), intent(out):: data
integer(kind=8), intent(out):: data
character*(*), dimension(:), intent(out):: data
real, dimension(:), intent(out):: data
real(kind=8), dimension(:), intent(out):: data
integer(kind=2), dimension(:), intent(out):: data
integer(kind=4), dimension(:), intent(out):: data
integer(kind=8), dimension(:), intent(out):: data


! attribute
type(adios2_attribute), intent(in):: attribute

! error code
integer, intent(out) :: ierr
```

- **subroutine** `adios2_attribute_name` inspect attribute name

```
subroutine adios2_attribute_name(name, attribute, ierr)

! WHERE

! name to be output
character(len=:), allocatable, intent(out) :: name

! attribute handler
type(adios2_attribute), intent(in) :: attribute

! error code
integer, intent(out) :: ierr
```

- **subroutine** adios2_inquire_variable_attribute retrieve a handler to a previously defined attribute associated to a variable

```
subroutine adios2_inquire_variable_attribute(attribute, io, attribute_name,␣
↪variable_name, separator, ierr)

! WHERE

! attribute handler
type(adios2_attribute), intent(out) :: attribute

! IO handler
type(adios2_io), intent(in) :: io

! attribute name
character*(*), intent(in) :: attribute_name

! variable name
character*(*), intent(in) :: variable_name

! hierarchy separator (e.g. "/" in variable/attribute )
character*(*), intent(in) :: separator

! error code
integer, intent(out) :: ierr
```

10.3 C bindings

The C bindings are specifically designed for C applications and those using an older C++ standard (98 and 03). If you are using a C++11 or more recent standard, please use the C++11 bindings.

The C bindings are based on opaque pointers to the components described in the *Components Overview* section.

C API handlers:

```
adios2_adios*
adios2_io*
adios2_variable*
adios2_attribute*
```

(continues on next page)

(continued from previous page)

```
adios2_engine*
adios2_operator*
```

Every ADIOS2 function that generates a new `adios2_*` unique handler returns the latter explicitly. Therefore, checks can be applied to know if the resulting handler is NULL. Other functions used to manipulate these valid handlers will return a value of type `enum adios2_error` explicitly. These possible errors are based on the C++ standardized exceptions . Each error will issue a more detailed description in the standard error output: `stderr`.

`adios2_error` possible values:

```
typedef enum {
    /** success */
    adios2_error_none = 0,

    /**
     * user input error
     */
    adios2_error_invalid_argument = 1,

    /** low-level system error, e.g. system IO error */
    adios2_error_system_error = 2,

    /** runtime errors other than system errors, e.g. memory overflow */
    adios2_error_runtime_error = 3,

    /** any other error exception */
    adios2_error_exception = 4
} adios2_error;
```

Usage:

```
adios2_variable* var = adios2_define_variable(io, ...)
if(var == NULL )
{
    // ... something went wrong with adios2_define_variable
    // ... check stderr
}
else
{
    adios2_type type;
    adios2_error errio = adios2_variable_type(&type, var)
    if(errio){
      // ... something went wrong with adios2_variable_type
      if( errio == adios2_error_invalid_argument)
      {
          // ... user input error
          // ... check stderr
      }
    }
}
```

Note: Use `#include "adios2_c.h"` for the C bindings, `adios2.h` is the C++11 header.

10.3.1 adios2_adios handler functions

Defines

adios2_init(comm)

adios2_init_config(config_file, comm)

Functions

adios2_adios ***adios2_init_mpi**(MPI_Comm comm)

Starting point for MPI apps. Creates an ADIOS handler. MPI collective and it calls MPI_Comm_dup

Parameters
comm – defines domain scope from application

Returns
success: handler, failure: NULL

adios2_adios ***adios2_init_config_mpi**(const char *config_file, MPI_Comm comm)

Starting point for MPI apps. Creates an ADIOS handler allowing a runtime config file. MPI collective and it calls MPI_Comm_dup and MPI_Bcast to pass the configFile contents

Parameters

- **config_file** – runtime configuration file in xml format
- **comm** – defines domain scope from application

Returns
success: handler, failure: NULL

adios2_adios ***adios2_init_serial**(void)

Initialize an ADIOS struct pointer handler in a serial, non-MPI application. Doesn't require a runtime config file.

Returns
success: handler, failure: NULL

adios2_adios ***adios2_init_config_serial**(const char *config_file)

Initialize an ADIOS struct pointer handler in a serial, non-MPI application. Doesn't require a runtime config file.

Returns
success: handler, failure: NULL

adios2_io ***adios2_declare_io**(adios2_adios *adios, const char *name)

Declares a new io handler

Parameters

- **adios** – owner the io handler
- **name** – unique io identifier within current adios handler

Returns
success: handler, failure: NULL

adios2_io *adios2_declare_io_order(adios2_adios *adios, const char *name, adios2_arrayordering order)

Declares a new io handler with specific array ordering

Parameters

- **adios** – owner the io handler
- **name** – unique io identifier within current adios handler
- **order** – array ordering

Returns

success: handler, failure: NULL

adios2_io *adios2_at_io(adios2_adios *adios, const char *name)

Retrieves a previously declared io handler by name

Parameters

- **adios** – owner the io handler
- **name** – unique name for the previously declared io handler

Returns

success: handler, failure: NULL

adios2_operator *adios2_define_operator(adios2_adios *adios, const char *name, const char *type)

Defines an adios2 supported operator by its type.

Parameters

- **adios** – owner the op handler
- **name** – unique operator name identifier within current ADIOS object
- **type** – supported ADIOS2 operator type: zfp, sz

Returns

success: handler, failure: NULL

adios2_operator *adios2_inquire_operator(adios2_adios *adios, const char *name)

Retrieves a previously defined operator handler

Parameters

- **adios** – owner the op handler
- **name** – unique name for the previously defined op handler

Returns

success: handler, failure: NULL

adios2_error adios2_flush_all(adios2_adios *adios)

Flushes all adios2_engine in write mode in all adios2_io handlers. If no adios2_io or adios2_engine exists it does nothing.

Parameters

adios – owner of all io and engines to be flushed

Returns

adios2_error 0: success, see enum adios2_error for errors

adios2_error **adios2_finalize**(adios2_adios *adios)

Final point for adios handler. Deallocates adios pointer. Required to avoid memory leaks. MPI collective and it calls MPI_Comm_free

Parameters
adios – handler to be deallocated, must be initialized with adios2_init or adios2_init_config

Returns
adios2_error 0: success, see enum adios2_error for errors

adios2_error **adios2_remove_io**(adios2_bool *result, adios2_adios *adios, const char *name)

DANGER ZONE: removes an io created with adios2_declare_io. Will create dangling pointers for all the handlers inside removed io. NOTE: Use result, not adios2_error to check if the IO was removed.

Parameters

- **result** – output adios2_true: io not found and not removed, adios2_false: io not found and not removed
- **adios** – owner of io to be removed
- **name** – input unique identifier for io to be removed

Returns
adios2_error 0: success, see enum adios2_error for errors

adios2_error **adios2_remove_all_ios**(adios2_adios *adios)

DANGER ZONE: removes all ios created with adios2_declare_io. Will create dangling pointers for all the handlers inside all removed io.

Parameters
adios – owner of all ios to be removed

Returns
adios2_error 0: success, see enum adios2_error for errors

adios2_error **adios2_enter_computation_block**(adios2_adios *adios)

Inform ADIOS about entering communication-free computation block in main thread. Useful when using Async IO

adios2_error **adios2_exit_computation_block**(adios2_adios *adios)

Inform ADIOS about exiting communication-free computation block in main thread. Useful when using Async IO

10.3.2 adios2_io handler functions

Functions

adios2_error **adios2_in_config_file**(adios2_bool *result, const adios2_io *io)

Check if io exists in a config file passed to the adios handler that created this io.

Parameters

- **result** – output adios2_true=1: in config file, adios2_false=0: not in config file
- **io** – handler

Returns
adios2_error 0: success, see enum adios2_error for errors

adios2_error **adios2_set_engine**(adios2_io *io, const char *engine_type)

Set the engine type for current io handler.

Parameters

- **io** – handler
- **engine_type** – predefined engine type, default is bpfile

Returns
adios2_error 0: success, see enum adios2_error for errors

adios2_error **adios2_set_parameters**(adios2_io *io, const char *parameters)

Set several parameters at once.

Parameters

- **io** – handler
- **string** – parameters in the format "param1=value1 , param2 = value2"

Returns
adios2_error 0: success, see enum adios2_error for errors

adios2_error **adios2_set_parameter**(adios2_io *io, const char *key, const char *value)

Set a single parameter. Overwrites value if key exists.

Parameters

- **io** – handler
- **key** – parameter key
- **value** – parameter value

Returns
adios2_error 0: success, see enum adios2_error for errors

adios2_error **adios2_get_parameter**(char *value, size_t *size, const adios2_io *io, const char *key)

Return IO parameter value string and length without '\0\ character For safe use, call this function first with NULL name parameter to get the size, then preallocate the buffer (with room for '\0' if desired), then call the function again with the buffer. Then '\0' terminate it if desired.

Parameters

- **value** – output
- **size** – output, value size without '\0'
- **io** – input handler
- **key** – input parameter key, if not found return size = 0 and value is untouched

Returns
adios2_error 0: success, see enum adios2_error for errors

adios2_error **adios2_clear_parameters**(adios2_io *io)

Clear all parameters.

Parameters
io – handler

Returns
adios2_error 0: success, see enum adios2_error for errors

adios2_error **adios2_add_transport**(size_t *transport_index, adios2_io *io, const char *type)

Add a transport to current io handler. Must be supported by current engine type.

Parameters

- **transport_index** – handler used for setting transport parameters or at adios2_close
- **io** – handler
- **type** – must be a supported transport type for a particular Engine. CAN'T use the keywords "Transport" or "transport"

Returns

adios2_error 0: success, see enum adios2_error for errors

adios2_error **adios2_set_transport_parameter**(adios2_io *io, const size_t transport_index, const char *key, const char *value)

Set a single parameter to an existing transport identified with a transport_index handler from adios2_add_transport. Overwrites existing parameter with the same key.

Parameters

- **io** – handler
- **transport_index** – handler from adios2_add_transport
- **key** – parameter key
- **value** – parameter value

Returns

adios2_error 0: success, see enum adios2_error for errors

adios2_variable ***adios2_define_variable**(adios2_io *io, const char *name, const adios2_type type, const size_t ndims, const size_t *shape, const size_t *start, const size_t *count, const adios2_constant_dims constant_dims)

Define a variable within io.

Parameters

- **io** – handler that owns the variable
- **name** – unique variable identifier
- **type** – primitive type from enum adios2_type in adios2_c_types.h
- **ndims** – number of dimensions
- **shape** – global dimension
- **start** – local offset
- **count** – local dimension
- **constant_dims** – adios2_constant_dims_true:: shape, start, count won't change; adios2_constant_dims_false: shape, start, count will change after definition

Returns

success: handler, failure: NULL

adios2_variable ***adios2_inquire_variable**(adios2_io *io, const char *name)

Retrieve a variable handler within current io handler.

Parameters

- **io** – handler to variable io owner

- **name** – unique variable identifier within io handler

Returns
found: handler, not found: NULL

adios2_error **adios2_inquire_all_variables**(adios2_variable ***variables, size_t *size, adios2_io *io)

Returns an array of variable handlers for all variable present in the io group

Parameters

- **variables** – output array of variable pointers (pointer to an adios2_variable**)
- **size** – output number of variables
- **io** – handler to variables io owner

Returns
adios2_error 0: success, see enum adios2_error for errors

adios2_error **adios2_inquire_group_variables**(adios2_variable ***variables, const char *full_group_name, size_t *size, adios2_io *io)

adios2_attribute ***adios2_define_attribute**(adios2_io *io, const char *name, const adios2_type type, const void *value)

Define an attribute value inside io.

Parameters

- **io** – handler that owns the attribute
- **name** – unique attribute name inside IO handler
- **type** – primitive type from enum adios2_type in adios2_c_types.h
- **value** – attribute single value

Returns
success: handler, failure: NULL

adios2_attribute ***adios2_define_attribute_array**(adios2_io *io, const char *name, const adios2_type type, const void *data, const size_t size)

Define an attribute array inside io.

Parameters

- **io** – handler that owns the attribute
- **name** – unique attribute name inside IO handler
- **type** – primitive type from enum adios2_type in adios2_c_types.h
- **data** – attribute data array
- **size** – number of elements of data array

Returns
success: handler, failure: NULL

adios2_attribute ***adios2_define_variable_attribute**(adios2_io *io, const char *name, const adios2_type type, const void *value, const char *variable_name, const char *separator)

Define an attribute single value associated to an existing variable by its name

Parameters

- **io** – handler that owns the variable and attribute
- **name** – unique attribute name inside a variable in io handler
- **type** – primitive type from enum adios2_type in adios2_c_types.h
- **value** – attribute single value
- **variable_name** – unique variable identifier in io handler. If variable doesn't exist adios2_error is adios2_error_invalid_argument.
- **separator** – hierarchy separator (e.g. "/" in variable_name/name)

Returns
success: handler, failure: NULL

adios2_attribute *adios2_define_variable_attribute_array(adios2_io *io, const char *name, const adios2_type type, const void *data, const size_t size, const char *variable_name, const char *separator)

Define an attribute array associated to an existing variable by its name

Parameters

- **io** – handler that owns the variable and attribute
- **name** – unique attribute name inside a variable in io handler
- **type** – primitive type from enum adios2_type in adios2_c_types.h
- **data** – attribute data single value or array
- **size** – number of elements of data array
- **variable_name** – unique variable identifier in io handler. If variable doesn't exist adios2_error is true.
- **separator** – hierarchy separator (e.g. "/" in variable/attribute)

Returns
success: handler, failure: NULL

adios2_attribute *adios2_inquire_attribute(adios2_io *io, const char *name)

Returns a handler to a previously defined attribute by name

Parameters

- **io** – handler to attribute io owner
- **name** – unique attribute identifier within io handler

Returns
found: handler, not found: NULL

adios2_attribute *adios2_inquire_variable_attribute(adios2_io *io, const char *name, const char *variable_name, const char *separator)

Retrieve a handler to a previously defined attribute associated to a variable

Parameters

- **io** – handler to attribute and variable io owner
- **name** – unique attribute name inside a variable in io handler
- **variable_name** – name of the variable associate with this attribute
- **separator** – hierarchy separator (e.g. "/" in variable/attribute)

Returns
found: handler, not found: NULL

adios2_error **adios2_inquire_all_attributes**(adios2_attribute ***attributes, size_t *size, adios2_io *io)

Returns an array of attribute handlers for all attribute present in the io group

Parameters

- **attributes** – output array of attribute pointers (pointer to an adios2_attribute**)
- **size** – output number of attributes
- **io** – handler to attributes io owner

Returns
adios2_error 0: success, see enum adios2_error for errors

adios2_error **adios2_inquire_group_attributes**(adios2_attribute ***attributes, const char *full_prefix, size_t *size, adios2_io *io)

adios2_error **adios2_inquire_subgroups**(char ***subGroupNames, const char *full_prefix, size_t *size, adios2_io *io)

Return a list of list sub group names

adios2_error **adios2_remove_variable**(adios2_bool *result, adios2_io *io, const char *name)

DANGEROUS! Removes a variable identified by name. Might create dangling pointers.

Parameters

- **result** – output adios2_true(1): found and removed variable, adios2_false(0): not found, nothing to remove
- **io** – handler variable io owner
- **name** – unique variable name within io handler

Returns
adios2_error 0: success, see enum adios2_error for errors

adios2_error **adios2_remove_all_variables**(adios2_io *io)

DANGEROUS! Removes all existing variables in current IO object. Might create dangling pointers.

Parameters
io – handler variables io owner

Returns
adios2_error 0: success, see enum adios2_error for errors

char ****adios2_available_variables**(adios2_io *io, size_t *size)

returns an array or c strings for names of available variables Might create dangling pointers

Parameters

- **io** – handler variables io owner
- **length** – of array of strings

Returns
names of variables as an array of strings

char ****adios2_available_attributes**(adios2_io *io, size_t *size)

returns an array or c strings for names of available attributes Might create dangling pointers

Parameters

- **io** – handler variables io owner
- **length** – of array of strings

Returns
names of variables as an array of strings

adios2_error **adios2_remove_attribute**(adios2_bool *result, adios2_io *io, const char *name)

DANGEROUS! Removes an attribute identified by name. Might create dangling pointers.

Parameters

- **result** – output adios2_true(1): found and removed attribute, adios2_false(0): not found, nothing to remove
- **io** – handler attribute io owner
- **name** – unique attribute name within io handler

Returns
adios2_error 0: success, see enum adios2_error for errors

adios2_error **adios2_remove_all_attributes**(adios2_io *io)

DANGEROUS! Removes all existing attributes in current IO object. Might create dangling pointers.

Parameters
io – handler attributes io owner

Returns
adios2_error 0: success, see enum adios2_error for errors

adios2_engine ***adios2_open**(adios2_io *io, const char *name, const adios2_mode mode)

Open an Engine to start heavy-weight input/output operations. In MPI version reuses the communicator from adios2_init or adios2_init_config MPI Collective function as it calls MPI_Comm_dup

Parameters

- **io** – engine owner
- **name** – unique engine identifier
- **mode** – adios2_mode_write, adios2_mode_read, adios2_mode_append, and adios2_mode_readRandomAccess

Returns
success: handler, failure: NULL

adios2_engine ***adios2_open_new_comm**(adios2_io *io, const char *name, const adios2_mode mode, MPI_Comm comm)

Open an Engine to start heavy-weight input/output operations. MPI Collective function as it calls MPI_Comm_dup

Parameters

- **io** – engine owner
- **name** – unique engine identifier
- **mode** – adios2_mode_write, adios2_mode_read, adios2_mode_append, and adios2_mode_readRandomAccess
- **comm** – communicator other than adios' handler comm. MPI only.

Returns
success: handler, failure: NULL

adios2_error **adios2_flush_all_engines**(adios2_io *io)

Flushes all engines created with current io handler using adios2_open

Parameters
io – handler whose engine will be flushed

Returns
adios2_error 0: success, see enum adios2_error for errors

adios2_error **adios2_engine_type**(char *engine_type, size_t *size, const adios2_io *io)

return engine type string and length without null character For safe use, call this function first with NULL name parameter to get the size, then preallocate the buffer (with room for '\0' if desired), then call the function again with the buffer. Then '\0' terminate it if desired.

Parameters

- **engine_type** – output, string without trailing '\0', NULL or preallocated buffer
- **size** – output, engine_type size without '\0'
- **io** – handler

Returns
adios2_error 0: success, see enum adios2_error for errors

adios2_engine *__adios2_get_engine__(adios2_io *io, const char *name)

10.3.3 adios2_variable handler functions

Functions

adios2_error **adios2_set_shape**(adios2_variable *variable, const size_t ndims, const size_t *shape)

Set new shape, care must be taken when reading back the variable for different steps. Only applies to Global arrays.

Parameters

- **variable** – handler for which new selection will be applied to
- **ndims** – number of dimensions for start and count
- **shape** – new shape dimensions array

Returns
adios2_error 0: success, see enum adios2_error for errors

adios2_error **adios2_set_block_selection**(adios2_variable *variable, const size_t block_id)

Read mode only. Required for reading local variables. For Global Arrays it will Set the appropriate Start and Count Selection for the global array coordinates.

Parameters

- **variable** – handler for which new selection will be applied to
- **block_id** – variable block index defined at write time. Blocks can be inspected with bpls -D variableName

Returns
adios2_error 0: success, see enum adios2_error for errors

adios2_error **adios2_set_selection**(adios2_variable *variable, const size_t ndims, const size_t *start, const size_t *count)

Set new start and count dimensions

Parameters

- **variable** – handler for which new selection will be applied to
- **ndims** – number of dimensions for start and count
- **start** – new start dimensions array
- **count** – new count dimensions array

Returns
adios2_error 0: success, see enum adios2_error for errors

adios2_error **adios2_set_memory_selection**(adios2_variable *variable, const size_t ndims, const size_t *memory_start, const size_t *memory_count)

Set the local start (offset) point to the memory pointer passed at Put and the memory local dimensions (count). Used for non-contiguous memory writes and reads (e.g. multidimensional ghost-cells). Currently not working for calls to Get.

Parameters

- **variable** – handler for which new memory selection will be applied to
- **ndims** – number of dimensions for memory_start and memory_count
- **memory_start** – relative local offset of variable.start to the contiguous memory pointer passed at Put from which data starts. e.g. if variable start = {rank*Ny,0} and there is 1 ghost cell per dimension, then memory_start = {1,1}
- **memory_count** – local dimensions for the contiguous memory pointer passed at adios2_put, e.g. if there is 1 ghost cell per dimension and variable count = {Ny,Nx}, then memory_count = {Ny+2,Nx+2}

Returns
adios2_error 0: success, see enum adios2_error for errors

adios2_error **adios2_set_step_selection**(adios2_variable *variable, const size_t step_start, const size_t step_count)

Set new step selection using step_start and step_count. Used mostly for reading from file-based engines (e.g. bpfile, hdf5)

Parameters

- **variable** – handler for which new selection will be applied to
- **step_start** – starting step for reading
- **step_count** – number of steps to read from step start

Returns
adios2_error 0: success, see enum adios2_error for errors

adios2_error **adios2_variable_name**(char *name, size_t *size, const adios2_variable *variable)

Retrieve variable name For safe use, call this function first with NULL name parameter to get the size, then preallocate the buffer (with room for '\0' if desired), then call the function again with the buffer. Then '\0' terminate it if desired.

Parameters

- **name** – output, string without trailing '\0', NULL or preallocated buffer

- **size** – output, name size without '\0'
- **variable** – handler

Returns
adios2_error 0: success, see enum adios2_error for errors

adios2_error **adios2_variable_type**(adios2_type *type, const adios2_variable *variable)

Retrieve variable type

Parameters

- **type** – output, from enum adios2_type
- **variable** – handler

Returns
adios2_error 0: success, see enum adios2_error for errors

adios2_error **adios2_variable_type_string**(char *type, size_t *size, const adios2_variable *variable)

Retrieve variable type in string form "char", "unsigned long", etc. For safe use, call this function first with NULL name parameter to get the size, then preallocate the buffer (with room for '\0' if desired), then call the function again with the buffer. Then '\0' terminate it if desired.

Parameters

- **type** – output, string without trailing '\0', NULL or preallocated buffer
- **size** – output, type size without '\0'
- **variable** – handler

Returns
adios2_error 0: success, see enum adios2_error for errors

adios2_error **adios2_variable_shapeid**(adios2_shapeid *shapeid, const adios2_variable *variable)

Retrieve variable shapeid

Parameters

- **shapeid** – output, from enum adios2_shapeid
- **variable** – handler

Returns
adios2_error 0: success, see enum adios2_error for errors

adios2_error **adios2_variable_ndims**(size_t *ndims, const adios2_variable *variable)

Retrieve current variable number of dimensions

Parameters

- **ndims** – output
- **variable** – handler

Returns
adios2_error 0: success, see enum adios2_error for errors

adios2_error **adios2_variable_shape**(size_t *shape, const adios2_variable *variable)

Retrieve current variable shape

Parameters

- **shape** – output, must be pre-allocated with ndims

- **variable** – handler

Returns
adios2_error 0: success, see enum adios2_error for errors

adios2_error **adios2_variable_start**(size_t *start, const adios2_variable *variable)

Retrieve current variable start

Parameters

- **start** – output, single value
- **variable** – handler

Returns
adios2_error 0: success, see enum adios2_error for errors

adios2_error **adios2_variable_count**(size_t *count, const adios2_variable *variable)

Retrieve current variable start

Parameters

- **count** – output, must be pre-allocated with ndims
- **variable** – handler

Returns
adios2_error 0: success, see enum adios2_error for errors

adios2_error **adios2_variable_steps_start**(size_t *steps_start, const adios2_variable *variable)

Read API, get available steps start from available steps count (e.g. in a file for a variable).

Parameters

- **steps_start** – output absolute first available step, don't use with adios2_set_step_selection as inputs are relative, use 0 instead.
- **variable** – handler

Returns
adios2_error 0: success, see enum adios2_error for errors

adios2_error **adios2_variable_steps**(size_t *steps, const adios2_variable *variable)

Read API, get available steps count from available steps count (e.g. in a file for a variable). Not necessarily contiguous.

Parameters

- **steps** – output available steps, single value
- **variable** – handler

Returns
adios2_error 0: success, see enum adios2_error for errors

adios2_error **adios2_selection_size**(size_t *size, const adios2_variable *variable)

Returns the minimum required allocation (in number of elements of a certain type, not bytes) for the current selection

Parameters

- **size** – number of elements of current type to be allocated by a pointer/vector to read current selection
- **variable** – handler for which data size will be inspected from

Returns
adios2_error 0: success, see enum adios2_error for errors

adios2_error **adios2_add_operation**(size_t *operation_index, adios2_variable *variable, adios2_operator *op, const char *key, const char *value)

Adds an operation to a variable (e.g. compression)

Parameters

- **operation_index** – output handler to be used with adios2_add_operation_param
- **variable** – handler on which operation is applied to
- **op** – handler to adios2_operator associated to current operation
- **key** – parameter key supported by the operation, empty if none
- **value** – parameter value supported by the operation, empty if none

Returns
adios2_error 0: success, see enum adios2_error for errors

adios2_error **adios2_set_operation_parameter**(adios2_variable *variable, const size_t operation_id, const char *key, const char *value)

Adds a parameter to an operation created with adios2_add_operation

Parameters

- **variable** – handler on which operation is applied to
- **operation_id** – handler returned from adios2_add_operation
- **key** – parameter key supported by the operation
- **value** – parameter value supported by the operation

Returns
adios2_error 0: success, see enum adios2_error for errors

adios2_error **adios2_remove_operations**(adios2_variable *variable)

Removes all current Operations associated with AddOperation. Provides the posibility to apply or not operators on a block basis.

Parameters
variable – handler on which operation is applied to

Returns
adios2_error 0: success, see enum adios2_error for errors

adios2_error **adios2_variable_min**(void *min, const adios2_variable *variable)

Read mode only: return the absolute minimum for current variable

Parameters

- **min** – output: variable minimum, must be of the same type as the variable handler
- **variable** – handler

Returns
adios2_error 0: success, see enum adios2_error for errors

adios2_error **adios2_variable_max**(void *max, const adios2_variable *variable)

Read mode only: return the absolute maximum for current variable

Parameters

- **max** – output: variable minimum, must be of the same type as the variable handler
- **variable** – handler

Returns
adios2_error 0: success, see enum adios2_error for errors

10.3.4 adios2_attribute handler functions

Functions

adios2_error **adios2_attribute_name**(char *name, size_t *size, const adios2_attribute *attribute)

Retrieve attribute name For safe use, call this function first with NULL name parameter to get the size, then preallocate the buffer (with room for '\0' if desired), then call the function again with the buffer. Then '\0' terminate it if desired.

Parameters

- **name** – output, string without trailing '\0', NULL or preallocated buffer
- **size** – output, name size without '\0'
- **attribute** – handler

Returns
adios2_error 0: success, see enum adios2_error for errors

adios2_error **adios2_attribute_type**(adios2_type *type, const adios2_attribute *attribute)

Retrieve attribute type

Parameters

- **type** –
- **attribute** – handler

Returns
adios2_error 0: success, see enum adios2_error for errors

adios2_error **adios2_attribute_type_string**(char *type, size_t *size, const adios2_attribute *attribute)

Retrieve attribute type in string form "char", "unsigned long", etc. For safe use, call this function first with NULL name parameter to get the size, then preallocate the buffer (with room for '\0' if desired), then call the function again with the buffer. Then '\0' terminate it if desired.

Parameters

- **type** – output, string without trailing '\0', NULL or preallocated buffer
- **size** – output, type size without '\0'
- **attribute** – handler

Returns
adios2_error 0: success, see enum adios2_error for errors

adios2_error **adios2_attribute_is_value**(adios2_bool *result, const adios2_attribute *attribute)

Checks if attribute is a single value or an array

Parameters

- **result** – output, adios2_true: single value, adios2_false: array
- **attribute** – handler

Returns
adios2_error 0: success, see enum adios2_error for errors

adios2_error **adios2_attribute_size**(size_t *size, const adios2_attribute *attribute)

Returns the number of elements (as in C++ STL size() function) if attribute is a 1D array. If single value returns 1

Parameters

- **size** – output, number of elements in attribute
- **attribute** – handler

Returns
adios2_error 0: success, see enum adios2_error for errors

adios2_error **adios2_attribute_data**(void *data, size_t *size, const adios2_attribute *attribute)

Retrieve attribute data pointer (read-only)

Parameters

- **data** – output attribute values, must be pre-allocated
- **size** – data size
- **attribute** – handler

Returns
adios2_error 0: success, see enum adios2_error for errors

10.3.5 adios2_engine handler functions

Functions

adios2_error **adios2_engine_name**(char *name, size_t *size, const adios2_engine *engine)

Return engine name string and length without '\0\ character For safe use, call this function first with NULL name parameter to get the size, then preallocate the buffer (with room for '\0' if desired), then call the function again with the buffer. Then '\0' terminate it if desired.

Parameters

- **name** – output, string without trailing '\0', NULL or preallocated buffer
- **size** – output, engine_type size without '\0'
- **engine** – handler

Returns
adios2_error 0: success, see enum adios2_error for errors

adios2_error **adios2_engine_get_type**(char *type, size_t *size, const adios2_engine *engine)

Return engine type string and length without '\0\ character For safe use, call this function first with NULL name parameter to get the size, then preallocate the buffer (with room for '\0' if desired), then call the function again with the buffer. Then '\0' terminate it if desired.

Parameters

- **type** – output, string without trailing '\0', NULL or preallocated buffer
- **size** – output, engine_type size without '\0'
- **engine** – handler

Returns
adios2_error 0: success, see enum adios2_error for errors

adios2_error **adios2_engine_openmode**(adios2_mode *mode, const adios2_engine *engine)

Return the engine's Open mode.

Parameters

- **mode** – output, adios2_mode parameter used in *adios2_open()*
- **engine** – handler

Returns
adios2_error 0: success, see enum adios2_error for errors

adios2_error **adios2_begin_step**(adios2_engine *engine, const adios2_step_mode mode, const float timeout_seconds, adios2_step_status *status)

Begin a logical adios2 step stream Check each engine documentation for MPI collective/non-collective behavior.

Parameters

- **engine** – handler
- **mode** – see enum adios2_step_mode in adios2_c_types.h for options, read is the common use case
- **timeout_seconds** – provide a time out in Engine opened in read mode
- **status** – output from enum adios2_step_status in adios2_c_types.h

Returns
adios2_error 0: success, see enum adios2_error for errors

adios2_error **adios2_current_step**(size_t *current_step, const adios2_engine *engine)

Inspect current logical step

Parameters

- **current_step** – output
- **engine** – input handler

Returns
adios2_error 0: success, see enum adios2_error for errors

adios2_error **adios2_between_step_pairs**(size_t *between_step_pairs, const adios2_engine *engine)

Inspect current between step status

Parameters

- **between_step_pairs** – output boolean
- **engine** – input handler

Returns
adios2_error 0: success, see enum adios2_error for errors

adios2_error **adios2_steps**(size_t *steps, const adios2_engine *engine)

Inspect total number of available steps, use for file engines in read mode only

Parameters

- **steps** – output available steps in engine
- **engine** – input handler

Returns
adios2_error 0: success, see enum adios2_error for errors

adios2_error **adios2_put**(adios2_engine *engine, adios2_variable *variable, const void *data, const adios2_mode launch)

Put data associated with a Variable in an engine, used for engines with adios2_mode_write at adios2_open

Parameters

- **engine** – handler for a particular engine where data will be put
- **variable** – contains variable metadata information
- **data** – user data to be associated with a variable, must be the same type passed to adios2_define_variable
- **launch** – mode launch policy

Returns
adios2_error 0: success, see enum adios2_error for errors

adios2_error **adios2_put_by_name**(adios2_engine *engine, const char *variable_name, const void *data, const adios2_mode launch)

Put data associated with a Variable in an engine, used for engines with adios2_mode_write at adios2_open. This is the name string version

Parameters

- **engine** – handler for a particular engine where data will be put
- **variable_name** – variable with this name must exists in adios2_io that opened the engine handler (1st parameter)
- **data** – user data to be associated with a variable, must be the same type passed to adios2_define_variable
- **launch** – mode launch policy

Returns
adios2_error 0: success, see enum adios2_error for errors

adios2_error **adios2_perform_puts**(adios2_engine *engine)

Performs all the adios2_put and adios2_put_by_name called with mode adios2_mode_deferred, up to this point, by copying user data into internal ADIOS buffers. User data can be reused after this point.

Parameters
engine – handler for a particular engine where data will be put

Returns
adios2_error 0: success, see enum adios2_error for errors

adios2_error **adios2_perform_data_write**(adios2_engine *engine)

Write array data to disk. This may relieve memory pressure by clearing ADIOS buffers. It is a collective call. User data can be reused after this point.

Parameters
engine – handler for a particular engine where data will be put

Returns
adios2_error 0: success, see enum adios2_error for errors

adios2_error **adios2_get**(adios2_engine *engine, adios2_variable *variable, void *data, const adios2_mode launch)

Gets data associated with a Variable from an engine, used for engines with adios2_mode_read at adios2_open. This is the name string version

Parameters

- **engine** – handler for a particular engine where data will be put
- **variable** – handler must exists in adios2_io that opened the engine handler (1st parameter). Typically from adios2_inquire_variable
- **data** – user data to be associated with a variable, must be the same type passed to adios2_define_variable. Must be pre-allocated for the required variable selection.
- **launch** – mode launch policy

Returns
adios2_error 0: success, see enum adios2_error for errors

adios2_error **adios2_get_by_name**(adios2_engine *engine, const char *variable_name, void *data, const adios2_mode launch)

Gets data associated with a Variable from an engine, used for engines with adios2_mode_read at adios2_open. This is the name string version

Parameters

- **engine** – handler for a particular engine where data will be put
- **variable_name** – variable with this name must exists in adios2_io that opened the engine handler (1st parameter).
- **data** – user data to be associated with a variable, must be the same type passed to adios2_define_variable. Must be pre-allocated for the required variable selection.
- **launch** – mode launch policy

Returns
adios2_error 0: success, see enum adios2_error for errors

adios2_error **adios2_perform_gets**(adios2_engine *engine)

Performs all the adios2_get and adios2_get_by_name called with mode adios2_mode_deferred up to this point by getting the data from the Engine. User data can be reused after this point.

Parameters
engine – handler for a particular engine where data will be obtained

Returns
adios2_error 0: success, see enum adios2_error for errors

adios2_error **adios2_end_step**(adios2_engine *engine)

Terminates interaction with current step. By default puts/gets data to/from all transports Check each engine documentation for MPI collective/non-collective behavior.

Parameters
engine – handler executing IO tasks

Returns
adios2_error 0: success, see enum adios2_error for errors

adios2_error **adios2_flush**(adios2_engine *engine)

Explicit engine buffer flush to transports

Parameters
engine – input

Returns
adios2_error 0: success, see enum adios2_error for errors

adios2_error **adios2_flush_by_index**(adios2_engine *engine, const int transport_index)

Explicit engine buffer flush to transport index

Parameters

- **engine** – input
- **transport_index** – index to be flushed

Returns
adios2_error 0: success, see enum adios2_error for errors

adios2_error **adios2_close**(adios2_engine *engine)

Close all transports in adios2_Engine. Call is required to close system resources. MPI Collective, calls MPI_Comm_free for duplicated communicator at Open

Parameters
engine – handler containing all transports to be closed. NOTE: engines NEVER become NULL after this function is called.

Returns
adios2_error 0: success, see enum adios2_error for errors

adios2_error **adios2_close_by_index**(adios2_engine *engine, const int transport_index)

Close a particular transport from the index returned by adios2_add_transport

Parameters

- **engine** – handler containing all transports to be closed. NOTE: engines NEVER become NULL due to this function.
- **transport_index** – handler from adios2_add_transport

Returns
adios2_error 0: success, see enum adios2_error for errors

adios2_error **adios2_lock_writer_definitions**(adios2_engine *engine)

Promise that no more definitions or changes to defined variables will occur. Useful information if called before the first *adios2_end_step()* of an output Engine, as it will know that the definitions are complete and constant for the entire lifetime of the output and may optimize metadata handling.

Parameters
engine – handler

adios2_error **adios2_lock_reader_selections**(adios2_engine *engine)

Promise that the reader data selections of are fixed and will not change in future timesteps. This information, provided before the EndStep() representing a fixed read pattern, may be utilized by the input Engine to optimize data flow.

Parameters
engine – handler

adios2_varinfo ***adios2_inquire_blockinfo**(adios2_engine *engine, adios2_variable *variable, const size_t step)

Get the list of blocks for a variable in a given step. In Streaming mode, step is unused, always the current step is processed.

Returns
Newly allocated adios2_varinfo structure, NULL pointer if step does not exist. The memory must be freed by the adios2_free_blockinfo function

void **adios2_free_blockinfo**(adios2_varinfo *data_blocks)

free adios2_varinfo structure

Parameters
data_blocks –

Returns
void

10.3.6 adios2_operator handler functions

Functions

adios2_error **adios2_operator_type**(char *type, size_t *size, const adios2_operator *op)

Retrieve operator type For safe use, call this function first with NULL name parameter to get the size, then preallocate the buffer (with room for '\0' if desired), then call the function again with the buffer. Then '\0' terminate it if desired.

Parameters

- **type** – output, string without trailing '\0', NULL or preallocated buffer
- **size** – output, type size without '\0'
- **op** – operator handler to be inspected

Returns
adios2_error 0: success, see enum adios2_error for errors

10.4 Python bindings

Note: Product Application Developers targeting finer-level control for their IO tasks for optimization should use the current full APIs. If you want to use ADIOS2 in simple use cases (*e.g.* reading a file for analysis, interactive Python, or saving some data for a small project) please refer to the *High-Level APIs* for a flat learning curve.

The full Python APIs follow very closely the full C++11 API interface.

10.4.1 ADIOS class

class adios2.**ADIOS**

AtIO(*self:* adios2.adios2.ADIOS, *arg0: str*) → adios2::py11::IO

returns an IO object previously defined IO object with DeclareIO, throws an exception if not found

DeclareIO(*self:* adios2.adios2.ADIOS, *arg0: str*) → adios2::py11::IO

spawn IO object component returning a IO object with a unique name, throws an exception if IO with the same name is declared twice

DefineOperator(*self:* adios2.adios2.ADIOS, *arg0: str*, *arg1: str*, *arg2: Dict[str, str]*) → adios2::py11::Operator

FlushAll(*self:* adios2.adios2.ADIOS) → None

flushes all engines in all spawned IO objects

InquireOperator(*self:* adios2.adios2.ADIOS, *arg0: str*) → adios2::py11::Operator

RemoveAllIOs(*self:* adios2.adios2.ADIOS) → None

DANGER ZONE: remove all IOs in current ADIOS object, creates dangling objects to parameters, variable, attributes, engines created with removed IO

RemoveIO(*self:* adios2.adios2.ADIOS, *arg0: str*) → bool

DANGER ZONE: remove a particular IO by name, creates dangling objects to parameters, variable, attributes, engines created with removed IO

10.4.2 IO class

class adios2.**IO**

AddTransport(*self:* adios2.adios2.IO, *type: str*, *parameters: Dict[str, str] = {}*) → int

AvailableAttributes(*self:* adios2.adios2.IO) → Dict[str, Dict[str, str]]

AvailableVariables(*self:* adios2.adios2.IO) → Dict[str, Dict[str, str]]

DefineAttribute(**args*, ***kwargs*)

Overloaded function.

1. DefineAttribute(self: adios2.adios2.IO, name: str, array: numpy.ndarray, variable_name: str = '', separator: str = '/') -> adios2::py11::Attribute
2. DefineAttribute(self: adios2.adios2.IO, name: str, stringValue: str, variable_name: str = '', separator: str = '/') -> adios2::py11::Attribute
3. DefineAttribute(self: adios2.adios2.IO, name: str, strings: List[str], variable_name: str = '', separator: str = '/') -> adios2::py11::Attribute

DefineVariable(**args*, ***kwargs*)

Overloaded function.

1. DefineVariable(self: adios2.adios2.IO, name: str, array: numpy.ndarray, shape: List[int] = [], start: List[int] = [], count: List[int] = [], isConstantDims: bool = False) -> adios2::py11::Variable
2. DefineVariable(self: adios2.adios2.IO, name: str) -> adios2::py11::Variable

EngineType(*self:* adios2.adios2.IO) → str

FlushAll(*self:* adios2.adios2.IO) → None

InquireAttribute(*self:* adios2.adios2.IO, *arg0: str*) → adios2::py11::Attribute

InquireVariable(*self:* adios2.adios2.IO, *arg0: str*) → adios2::py11::Variable

Open(*self:* adios2.adios2.IO, *arg0: str*, *arg1: int*) → adios2::py11::Engine

Parameters(*self:* adios2.adios2.IO) → Dict[str, str]

RemoveAllAttributes(*self:* adios2.adios2.IO) → None

RemoveAllVariables(*self:* adios2.adios2.IO) → None

RemoveAttribute(*self:* adios2.adios2.IO, *arg0: str*) → bool

RemoveVariable(*self:* adios2.adios2.IO, *arg0: str*) → bool

SetEngine(*self:* adios2.adios2.IO, *arg0: str*) → None

SetParameter(*self:* adios2.adios2.IO, *arg0: str*, *arg1: str*) → None

SetParameters(*self:* adios2.adios2.IO, *parameters: Dict[str, str] = {}*) → None

10.4.3 Variable class

class adios2.**Variable**

AddOperation(*self: adios2.adios2.Variable, arg0: adios2::py11::Operator, arg1: Dict[str, str]*) → int

BlockID(*self:* adios2.adios2.Variable) → int

Count(*self:* adios2.adios2.Variable) → List[int]

Name(*self:* adios2.adios2.Variable) → str

Operations(*self:* adios2.adios2.Variable) → List[adios2::py11::Operator]

SelectionSize(*self:* adios2.adios2.Variable) → int

SetBlockSelection(*self:* adios2.adios2.Variable, *arg0: int*) → None

SetSelection(*self:* adios2.adios2.Variable, *arg0: Tuple[List[int], List[int]]*) → None

SetShape(*self:* adios2.adios2.Variable, *arg0: List[int]*) → None

SetStepSelection(*self:* adios2.adios2.Variable, *arg0: Tuple[int, int]*) → None

Shape(*self:* adios2.adios2.Variable, *step: int = 18446744073709551615*) → List[int]

ShapeID(*self:* adios2.adios2.Variable) → adios2.adios2.ShapeID

Sizeof(*self:* adios2.adios2.Variable) → int

Start(*self:* adios2.adios2.Variable) → List[int]

Steps(*self:* adios2.adios2.Variable) → int

StepsStart(*self:* adios2.adios2.Variable) → int

Type(*self:* adios2.adios2.Variable) → str

10.4.4 Attribute class

class adios2.**Attribute**

Data(*self:* adios2.adios2.Attribute) → numpy.ndarray

DataString(*self:* adios2.adios2.Attribute) → List[str]

Name(*self:* adios2.adios2.Attribute) → str

Type(*self:* adios2.adios2.Attribute) → str

10.4.5 Engine class

class adios2.**Engine**

BeginStep(**args*, ***kwargs*)

Overloaded function.

1. BeginStep(self: adios2.adios2.Engine, mode: adios2.adios2.StepMode, timeoutSeconds: float = -1.0) -> adios2.adios2.StepStatus

2. BeginStep(self: adios2.adios2.Engine) -> adios2.adios2.StepStatus

BlocksInfo(*self:* adios2.adios2.Engine, *arg0: str*, *arg1: int*) → List[Dict[str, str]]

Close(*self:* adios2.adios2.Engine, *transportIndex: int = -1*) → None

CurrentStep(*self:* adios2.adios2.Engine) → int

EndStep(*self:* adios2.adios2.Engine) → None

Flush(*self:* adios2.adios2.Engine, *arg0: int*) → None

Get(**args*, ***kwargs*)

Overloaded function.

1. Get(self: adios2.adios2.Engine, variable: adios2.adios2.Variable, array: numpy.ndarray, launch: adios2.adios2.Mode = <Mode.Deferred: 6>) -> None

2. Get(self: adios2.adios2.Engine, variable: adios2.adios2.Variable, launch: adios2.adios2.Mode = <Mode.Deferred: 6>) -> str

LockReaderSelections(*self:* adios2.adios2.Engine) → None

LockWriterDefinitions(*self:* adios2.adios2.Engine) → None

Name(*self:* adios2.adios2.Engine) → str

PerformDataWrite(*self:* adios2.adios2.Engine) → None

PerformGets(*self:* adios2.adios2.Engine) → None

PerformPuts(*self:* adios2.adios2.Engine) → None

Put(**args*, ***kwargs*)

Overloaded function.

1. Put(self: adios2.adios2.Engine, variable: adios2.adios2.Variable, array: numpy.ndarray, launch: adios2.adios2.Mode = <Mode.Deferred: 6>) -> None

2. Put(self: adios2.adios2.Engine, arg0: adios2.adios2.Variable, arg1: str) -> None

Steps(*self:* adios2.adios2.Engine) → int

Type(*self:* adios2.adios2.Engine) → str

10.4.6 Operator class

class adios2.**Operator**

Parameters(*self:* adios2.adios2.Operator) → Dict[str, str]

SetParameter(*self:* adios2.adios2.Operator, *arg0: str*, *arg1: str*) → None

Type(*self:* adios2.adios2.Operator) → str

CHAPTER

ELEVEN

HIGH-LEVEL APIS

The high-level APIs are designed for simple tasks for which performance is not critical. Unlike the *Full APIs*, the high-level APIs only require a single object handler resembling a C++ fstream or a Python file I/O idiom. The high-level APIs are recommended to both first-time and advanced users; the low-level APIs being recommended only when performance testing identifies a bottleneck or when more control is needed.

Typical scenarios for using the simple high-level APIs are:

- Reading a file to perform data analysis with libraries (matplotlib, scipy, etc.)
- Interactive: few calls make interactive usage easier.
- Saving data to files is small or personal projects
- Online frameworks: *e.g.* Jupyter notebooks, see python-mpi examples running on MyBinder

The designed functionality syntax is closely related to the native language IO bindings for formatted text files *e.g.* C++ `fstream` `getline`, and Python file IO. The main function calls are: `open` (or constructor in C++), `write`, `read` and `close` (or destructor in C++). In addition, ADIOS2 borrows the corresponding language native syntax for advancing lines to advance the step in write mode, and for a "step-by-step" streaming basis in read mode. See each language section in this chapter for a write/read example.

Note: The simplified APIs are based on language native file IO interface. Hence `write` and `read` calls are always synchronized and variables data memory is ready to use immediately after these calls.

Currently ADIOS2 support bindings for the following languages and their minimum standards:

Language	Standard	Interface	Based on
C++	11/newer	`#include adios2.h`	`fstream`
Python	2.7/3	`import adios2`	Python IO
Matlab			

The following sections provide a summary of the API calls on each language and links to Write and Read examples to put it all together.

11.1 C++ High-Level API

C++11 High-Level APIs are based on a single object *adios2::fstream*

Caution: DO NOT place `use namespace adios2` in your C++ code. Use `adios2::fstream` directly to prevent conflicts with `std::fstream`.

11.1.1 C++11 Write example

```
#include <adios2.h>
...

int rank, size;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

// Nx, Ny from application, std::size_t
const adios2::Dims shape{Nx, Ny * static_cast<std::size_t>(size)};
const adios2::Dims start{0, Ny * static_cast<std::size_t>(rank)};
const adios2::Dims count{Nx, Ny};

adios2::fstream oStream("cfd.bp", adios2::fstream::out, MPI_COMM_WORLD);

// NSteps from application
for (std::size_t step = 0; step < NSteps; ++step)
{
    if(rank == 0 && step == 0) // global variable
    {
        oStream.write<int32_t>("size", size);
    }

    // physicalTime double, <double> is optional
    oStream.write<double>( "physicalTime", physicalTime );
    // T and P are std::vector<float>
    oStream.write( "temperature", T.data(), shape, start, count );
    // adios2::endl will advance the step after writing pressure
    oStream.write( "pressure", P.data(), shape, start, count, adios2::end_step );

}

// Calling close is mandatory!
oStream.close();
```

11.1.2 C++11 Read "step-by-step" example

```
#include <adios2.h>
...

int rank, size;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

// Selection Window from application, std::size_t
const adios2::Dims start{0, 0};
const adios2::Dims count{SelX, SelY};

if( rank == 0)
{
   // if only one rank is active use MPI_COMM_SELF
   adios2::fstream iStream("cfd.bp", adios2::fstream::in, MPI_COMM_SELF);

   adios2::fstep iStep;
   while (adios2::getstep(iStream, iStep))
   {
       if( iStep.currentstep() == 0 )
       {
           const std::size_t sizeOriginal = iStep.read<std::size_t>("size");
       }
       const double physicalTime = iStream.read<double>( "physicalTime");
       const std::vector<float> temperature = iStream.read<float>( "temperature", start,␣
↪count );
       const std::vector<float> pressure = iStream.read<float>( "pressure", start, count␣
↪);
   }
   // Don't forget to call close!
   iStream.close();
}
```

11.1.3 `adios2::fstream` API documentation

class **fstream**

Public Types

enum **openmode**

Available open modes for *adios2::fstream* constructor or open calls

Values:

enumerator **out**

write

enumerator **in**

read

enumerator **in_random_access**

read_random_access

enumerator **app**

append, not yet supported

Public Functions

fstream(const std::string &name, adios2::*fstream*::*openmode* mode, MPI_Comm comm, const std::string engineType = "BPFile")

High-level API MPI constructor, based on C++11 fstream. Allows for passing parameters in source code.

Parameters

- **name** – stream name
- **mode** – *fstream::in* (Read), *fstream::out* (Write), *fstream::app* (Append)
- **comm** – MPI communicator establishing domain for fstream
- **engineType** – available adios2 engine

Throws

`std::invalid_argument` – (user input error) or std::runtime_error (system error)

fstream(const std::string &name, const adios2::*fstream*::*openmode* mode, MPI_Comm comm, const std::string &configFile, const std::string ioInConfigFile)

High-level API MPI constructor, based on C++11 fstream. Allows for runtime config file.

Parameters

- **name** – stream name
- **mode** – *fstream::in* (Read), *fstream::out* (Write), *fstream::app* (Append)
- **comm** – MPI communicator establishing domain for fstream
- **configFile** – adios2 runtime configuration file
- **ioInConfigFile** – specific io name in configFile

Throws

`std::invalid_argument` – (user input error) or std::runtime_error (system error)

fstream(const std::string &name, const adios2::*fstream*::*openmode* mode, const std::string engineType = "BPFile")

High-level API non-MPI constructor, based on C++11 fstream. Allows for passing parameters in source code.

Parameters

- **name** – stream name
- **mode** – *fstream::in* (Read), *fstream::out* (Write), *fstream::app* (Append)
- **engineType** – available adios2 engine

Throws
`std::invalid_argument` – (user input error) or std::runtime_error (system error)

fstream(const std::string &name, const adios2::*fstream*::*openmode* mode, const std::string &configFile, const std::string ioInConfigFile)

High-level API MPI constructor, based on C++11 fstream. Allows for runtime config file.

Parameters

- **name** – stream name
- **mode** – *fstream::in* (Read), *fstream::out* (Write), *fstream::app* (Append)
- **configFile** – adios2 runtime configuration file
- **ioInConfigFile** – specific io name in configFile

Throws
`std::invalid_argument` – (user input error) or std::runtime_error (system error)

fstream() = default

Empty constructor, allows the use of open later in the code

~fstream() = default

Using RAII STL containers only

explicit **operator bool**() const noexcept

Checks if fstream object is valid

void **open**(const std::string &name, const *openmode* mode, MPI_Comm comm, const std::string engineType = "BPFile")

High-level API MPI open, based on C++11 fstream. Allows for passing parameters in source code. Used after empty constructor.

Parameters

- **name** – stream name
- **mode** – *adios2::fstream::in* (Read), *adios2::fstream::out* (Write), *adios2::fstream::app* (Append)
- **comm** – MPI communicator establishing domain for fstream
- **engineType** – available adios2 engine

Throws
`std::invalid_argument` – (user input error) or std::runtime_error (system error)

void **open**(const std::string &name, const *openmode* mode, MPI_Comm comm, const std::string configFile, const std::string ioInConfigFile)

High-level API MPI constructor, based on C++11 fstream. Allows for runtime config file. Used after empty constructor.

Parameters

- **name** – stream name
- **mode** – *fstream::in* (Read), *fstream::out* (Write), *fstream::app* (Append)
- **comm** – MPI communicator establishing domain for fstream
- **configFile** – adios2 runtime configuration file
- **ioInConfigFile** – specific io name in configFile

Throws
`std::invalid_argument` – (user input error) or std::runtime_error (system error)

void **open**(const std::string &name, const *openmode* mode, const std::string engineType = "BPFile")

High-level API non-MPI open, based on C++11 fstream. Allows for passing parameters in source code. Used after empty constructor.

Parameters

- **name** – stream name
- **mode** – *fstream::in* (Read), *fstream::out* (Write), *fstream::app* (Append)
- **engineType** – available adios2 engine

Throws
`std::invalid_argument` – (user input error) or std::runtime_error (system error)

void **open**(const std::string &name, const *openmode* mode, const std::string configFile, const std::string ioInConfigFile)

High-level API non-MPI constructor, based on C++11 fstream. Allows for runtime config file. Used after empty constructor.

Parameters

- **name** – stream name
- **mode** – *fstream::in* (Read), *fstream::out* (Write), *fstream::app* (Append)
- **configFile** – adios2 runtime configuration file
- **ioInConfigFile** – specific io name in configFile

Throws
`std::invalid_argument` – (user input error) or std::runtime_error (system error)

void **set_parameter**(const std::string key, const std::string value) noexcept

Set a single stream parameter based on *Engine* supported parameters. MUST be passed before the first call to write or read. See: https://adios2.readthedocs.io/en/latest/engines/engines.html

Parameters

- **key** – input parameter key
- **value** – input parameter value

void **set_parameters**(const adios2::Params &parameters) noexcept

Set stream parameters based on *Engine* supported parameters. MUST be passed before the first call to write or read. See: https://adios2.readthedocs.io/en/latest/engines/engines.html

Parameters
parameters – input key/value parameters

template<class **T**>
void **write_attribute**(const std::string &name, const *T* &value, const std::string &variableName = "", const std::string separator = "/", const bool endStep = false)

Define attribute inside fstream or for a variable after write. Single value input version.

Parameters

- **name** – unique attribute identifier *IO* object or for a *Variable* if variableName is not empty (associated to a variable)
- **value** – single data value

- **variableName** – default is empty, if not empty attributes is associated to a variable after a write
- **separator** – default is "/", hierarchy between variable name and attribute, e.g. variableName/attribute1, variableName::attribute1. Not used if variableName is empty.
- **endStep** – similar to std::endStep, end current step and flush (default). Use adios2::endStep for true.

template<class **T**>
void **write_attribute**(const std::string &name, const *T* *data, const size_t size, const std::string &variableName = "", const std::string separator = "/", const bool endStep = false)

Define attribute inside fstream or for a variable after write. Array input version.

Parameters

- **name** – unique attribute identifier *IO* object or for a *Variable* if variableName is not empty (associated to a variable)
- **data** – pointer to user data
- **size** – number of data elements
- **variableName** – default is empty, if not empty attributes is associated to a variable after a write
- **separator** – default is "/", hierarchy between variable name and attribute, e.g. variableName/attribute1, variableName::attribute1. Not used if variableName is empty.
- **endStep** – similar to std::endStep, end current step and flush (default). Use adios2::endStep for true.

template<class **T**>
void **write**(const std::string &name, const *T* *data, const adios2::Dims &shape = adios2::Dims(), const adios2::Dims &start = adios2::Dims(), const adios2::Dims &count = adios2::Dims(), const bool endStep = false)

writes a self-describing array variable

Parameters

- **name** – variable name
- **data** – variable data data
- **shape** – variable global MPI dimensions. Pass empty for local variables.
- **start** – variable offset for current MPI rank. Pass empty for local variables.
- **count** – variable dimension for current MPI rank. Local variables only have count.
- **endStep** – similar to std::endStep, end current step and flush (default). Use adios2::endStep if true.

Throws

`std::invalid_argument` – (user input error) or std::runtime_error (system error)

template<class **T**>
void **write**(const std::string &name, const *T* *data, const adios2::Dims &shape, const adios2::Dims &start, const adios2::Dims &count, const adios2::vParams &operations, const bool endStep = false)

write overload that allows passing supported operations (e.g. lossy compression "zfp", "mgard", "sz") to a self-described array variable

Parameters

- **name** – variable name
- **data** – variable data data
- **shape** – variable global MPI dimensions. Pass empty for local variables.
- **start** – variable offset for current MPI rank. Pass empty for local variables.
- **count** – variable dimension for current MPI rank. Local variables only have count.
- **operations** – vector of operations, each entry is a std::pair:
- **endStep** – similar to std::endStep, end current step and flush (default). Use adios2::endStep if true.

Throws
`std::invalid_argument` – (user input error) or std::runtime_error (system error)

template<class **T**>
void **write**(const std::string &name, const *T* &value, const bool isLocalValue = false, const bool endStep = false)

Write a self-describing single-value variable

Parameters

- **name** – variable name
- **value** – variable data value (can be r-value)
- **isLocalValue** – true: local value (returned as GlobalArray), false: global value (returned as global value)
- **endStep** – similar to std::endStep, end current step and flush (default). Use adios2::endStep for true.

Throws
`std::invalid_argument` – (user input error) or std::runtime_error (system error)

template<class **T**>
void **read**(const std::string &name, *T* *data, const size_t blockID = 0)

Reads into a pre-allocated pointer. When used with adios2::getstep reads current step

Parameters

- **name** – variable name
- **data** – pre-allocated pointer to hold read data
- **blockID** – required for local variables, specify current block to be selected

Throws
`throws` – exception if variable name, dimensions or step not found

template<class **T**>
void **read**(const std::string &name, *T* &value, const size_t blockID = 0)

Reads a value. When used with adios2::getstep reads current step value

Parameters

- **name** – variable name
- **value** – output value, if variable is not found (name and type don't match) the returned value address becomes nullptr
- **blockID** – required for local variables, specify current block to be selected

Throws
`throws` – exception if variable name, dimensions or step not found

template<class **T**>
void **read**(const std::string &name, *T* *data, const size_t stepsStart, const size_t stepsCount = 1, const size_t blockID = 0)

Read accessing steps in random access mode. Not be used with adios2::getstep as it throw an exception when reading in stepping mode.

Parameters

- **name** – variable name
- **data** – pre-allocated pointer to hold read data, if variable is not found (name and type don't match) it becomes nullptr
- **stepsStart** – variable initial step (relative to the variable first appearance, not absolute step in stream)
- **stepsCount** – variable number of steps form step_start, don't have to be contiguous, necessarily
- **blockID** – required for local variables, specify current block to be selected

Throws
`throws` – exception if variable name, dimensions or step not found

template<class **T**>
void **read**(const std::string &name, *T* &value, const size_t step, const size_t blockID = 0)

Reads into a single value for a single step. Not be used with adios2::getstep as it throws an exception when reading in stepping mode.

Parameters

- **name** – variable name
- **value** – filled with value, if variable is not found (name, type and step don't match) the returned value address becomes nullptr
- **step** – selected single step
- **blockID** – required for local variables, specify current block to be selected

Throws
`throws` – exception if variable name, dimensions or step not found

template<class **T**>
void **read**(const std::string &name, *T* *data, const adios2::Dims &start, const adios2::Dims &count, const size_t blockID = 0)

Reads into a pre-allocated pointer a selection piece in dimension. When used with adios2::getstep reads current step

Parameters

- **name** – variable name
- **data** – pre-allocated pointer to hold read data, if variable is not found (name and type don't match) it becomes nullptr
- **start** – variable local offset selection
- **count** – variable local dimension selection from start
- **blockID** – required for local variables, specify current block to be selected

Throws
throws – exception if variable name, dimensions or step not found

template<class **T**>
void **read**(const std::string &name, *T* *data, const adios2::Dims &start, const adios2::Dims &count, const size_t stepsStart, const size_t stepsCount, const size_t blockID = 0)

Reads into a pre-allocated pointer a selection piece in dimensions and steps. Not be used with adios2::getstep as it throws an exception when reading in stepping mode.

Parameters

- **name** – variable name
- **data** – pre-allocated pointer to hold read data, if variable is not found (name and type don't match) it becomes a nullptr
- **start** – variable local offset selection
- **count** – variable local dimension selection from start
- **stepsStart** – variable initial step (relative to the variable first appearance, not absolute step in stream)
- **stepsCount** – variable number of steps form step_start, don't have to be necessarily contiguous
- **blockID** – required for local variables, specify current block to be selected

Throws
throws – exception if variable name, dimensions or step not found

template<class **T**>
std::vector<*T*> **read**(const std::string &name, const size_t blockID = 0)

Reads entire variable for current step (streaming mode: step by step)

Parameters

- **name** – variable name
- **blockID** – required for local variables, specify current block to be selected

Throws
throws – exception if variable name, dimensions or step not found

Returns
data of variable name for current step. Single data will have a size=1 vector

template<class **T**>
std::vector<*T*> **read**(const std::string &name, const size_t stepsStart, const size_t stepsCount = 1, const size_t blockID = 0)

Returns a vector with full variable dimensions for the current step selection. Not be used with adios2::getstep as it throw an exception when reading in stepping mode.

Parameters

- **name** – variable name
- **stepsStart** – variable initial step (relative to the variable first appearance, not absolute step in stream)
- **stepsCount** – variable number of steps form step_start, don't have to be contiguous, necessarily
- **blockID** – required for local variables, specify current block to be selected

Throws
throws – exception if variable name, dimensions or step not found

Returns
data of variable name for current step, empty if exception is thrown

template<class **T**>
std::vector<*T*> **read**(const std::string &name, const Dims &start, const Dims &count, const size_t blockID = 0)

Reads a selection piece in dimension for current step (streaming mode: step by step)

Parameters

- **name** – variable name
- **start** – variable local offset selection
- **count** – variable local dimension selection from start
- **blockID** – required for local variables, specify current block to be selected

Throws
throws – exception if variable name, dimensions or step not found

Returns
data of variable name for current step, empty if exception is thrown

template<class **T**>
std::vector<*T*> **read**(const std::string &name, const Dims &start, const Dims &count, const size_t stepsStart, const size_t stepsCount, const size_t blockID = 0)

Reads a selection piece in dimension and a selection piece in steps (non-streaming mode). Not be used with adios2::getstep as it throw an exception when reading in stepping mode.

Parameters

- **name** – variable name
- **start** – variable local offset selection
- **count** – variable local dimension selection from start
- **stepsStart** – variable initial step (relative to the variable first appearance, not absolute step in stream)
- **stepsCount** – variable number of steps form step_start, don't have to be contiguous, necessarily
- **blockID** – required for local variables, specify current block to be selected

Throws
throws – exception if variable name, dimensions or step not found

Returns
variable data, empty if exception is thrown

template<class **T**>
std::vector<*T*> **read_attribute**(const std::string &name, const std::string &variableName = "", const std::string separator = "/")

Reads an attribute returning a vector For single data vector size = 1

Parameters

- **name** – attribute name

- **variableName** – default is empty, if not empty look for an attribute associated to a variable
- **separator** – default is "/", hierarchy between variable name and attribute, e.g. variableName/attribute1, variableName::attribute1. Not used if variableName is empty.

Returns
vector containing attribute data

void **end_step**()

At write: ends the current step At read: use it in streaming mode to inform the writer that the reader is done consuming the step. No effect for file engines.

void **close**()

close current stream becoming inaccessible

size_t **current_step**() const noexcept

Return current step when getstep is called in a loop, read mode only

Returns
current step

Friends

friend bool **getstep**(adios2::*fstream* &stream, adios2::fstep &step)

Gets step from stream Based on std::getline, enables reading on a step-by-step basis in a while or for loop. Read mode only

Parameters

- **stream** – input stream containing steps
- **step** – output object current step, adios2::fstep in an alias to *adios2::fstream* with scope narrowed to one step

Returns
true: step is valid, false: step is invalid (end of stream).

11.2 Python High-Level API

Python simple bindings follow closely Python style directives. Just like the full APIs, they rely on numpy and, optionally, on `mpi4py`, if the underlying ADIOS2 library is compiled with MPI.

For online examples on MyBinder :

- Python-MPI Notebooks
- Python-noMPI Notebooks

11.2.1 Python Write example

```
from mpi4py import MPI
import numpy as np
import adios2

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()
...
shape = [size * nx]
start = [rank * nx]
count = [nx]

# with-as will call adios2.close on fh at the end
with adios2.open("cfd.bp", "w", comm) as fh:

   # NSteps from application
   for i in range(0, NSteps):

      if(rank == 0 and i == 0):
         fh.write("size", np.array([size]))

      fh.write("physical_time", np.array([physical_time]) )
      # temperature and pressure are numpy arrays
      fh.write("temperature", temperature, shape, start, count)
      # advances to next step
      fh.write("pressure", pressure, shape, start, count, end_step=True)
```

11.2.2 Python Read "step-by-step" example

```
from mpi4py import MPI
import numpy as np
import adios2

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()
...
shape = [size * nx]
start = [rank * nx]
count = [nx]

if( rank == 0 ):
   # with-as will call adios2.close on fh at the end
   # if only one rank is active pass MPI.COMM_SELF
   with adios2.open("cfd.bp", "r", MPI.COMM_SELF) as fh:

      for fstep in fh:

         # inspect variables in current step
```

(continues on next page)

(continued from previous page)

```
        step_vars = fstep.available_variables()

        # print variables information
        for name, info in step_vars.items():
            print("variable_name: " + name)
            for key, value in info.items():
               print("\t" + key + ": " + value)
            print("\n")

        # track current step
        step = fstep.current_step()
        if( step == 0 ):
           size_in = fstep.read("size")

        # read variables return a numpy array with corresponding selection
        physical_time = fstep.read("physical_time")
        temperature = fstep.read("temperature", start, count)
        pressure = fstep.read("pressure", start, count)
```

Caution: When reading in stepping mode with the for-in directive, as in the example above, use the step handler (`fstep`) inside the loop rather than the global handler (`fh`)

11.2.3 File class API

adios2.**open**(**args*, ***kwargs*)

Overloaded function.

1. open(name: str, mode: str, engine_type: str = 'BPFile') -> adios2::py11::File

High-level API, file object open

2. open(name: str, mode: str, config_file: str, io_in_config_file: str) -> adios2::py11::File

High-level API, file object open with a runtime config file

class adios2.**File**

add_transport(*self:* adios2.adios2.File, *type: str*, *parameters: Dict[str, str] = {}*) → int

Adds a transport and its parameters to current IO. Must be supported by current engine type.

Parameters

type
: must be a supported transport type for current engine.

parameters
: acceptable parameters for a particular transport CAN'T use the keywords "Transport" or "transport" in key

Returns

transport_index
: handler to added transport

available_attributes(*self:* adios2.adios2.File) → Dict[str, Dict[str, str]]

Returns a 2-level dictionary with attribute information. Read mode only.

Returns

attributes dictionary

key
attribute name

value
attribute information dictionary

available_variables(*self:* adios2.adios2.File, *keys: List[str] = []*) → Dict[str, Dict[str, str]]

Returns a 2-level dictionary with variable information. Read mode only.

Parameters

keys
list of variable information keys to be extracted (case insensitive) keys=['AvailableStepsCount','Type','Max','Min','SingleValue','Shape'] keys=['Name'] returns only the variable names as 1st-level keys leave empty to return all possible keys

Returns

variables dictionary

key
variable name

value
variable information dictionary

close(*self:* adios2.adios2.File) → None

Closes file, thus becoming unreachable. Not required if using open in a with-as statement. Required in all other cases per-open to avoid resource leaks.

current_step(*self:* adios2.adios2.File) → int

Inspect current step when using for-in loops, read mode only

Returns
current step

end_step(*self:* adios2.adios2.File) → None

Write mode: advances to the next step. Convenient when declaring variable attributes as advancing to the next step is not attached to any variable.

Read mode: in streaming mode releases the current step (no effect in file based engines)

read(**args*, ***kwargs*)

Overloaded function.

1. read(self: adios2.adios2.File, name: str, block_id: int = 0) -> numpy.ndarray

   Reads entire variable for current step (streaming mode step by step)

   Parameters

   name
   variable name

   block_id
   required for local array variables

Returns

array
values of variable name for current step. Single values will have a shape={1} numpy array

2. read(self: adios2.adios2.File, name: str, start: List[int] = [], count: List[int] = [], block_id: int = 0) -> numpy.ndarray

   Reads a selection piece in dimension for current step (streaming mode step by step)

   Parameters

   name
   variable name

   start
   variable local offset selection (defaults to (0, 0, ...)

   count
   variable local dimension selection from start defaults to whole array for GlobalArrays, or selected Block size for LocalArrays

   block_id
   required for local array variables

   Returns

   array
   values of variable name for current step empty if exception is thrown

3. read(self: adios2.adios2.File, name: str, start: List[int], count: List[int], step_start: int, step_count: int, block_id: int = 0) -> numpy.ndarray

   Random access read allowed to select steps, only valid with File Engines

   Parameters

   name
   variable to be read

   start
   variable offset dimensions

   count
   variable local dimensions from offset

   step_start
   variable step start

   step_count
   variable number of steps to read from step_start

   block_id
   required for local array variables

   Returns

   array
   resulting array from selection

read_attribute(*self:* adios2.adios2.File, *name: str*, *variable_name: str = ''*, *separator: str = '/'*) → numpy.ndarray

Reads a numpy based attribute

Parameters

name
attribute name

variable_name
if attribute is associated with a variable

separator
concatenation string between variable_name and attribute e.g. variable_name + separator + name (var/attr) Not used if variable_name is empty

Returns

array
resulting array attribute data

`read_attribute_string`(*self:* adios2.adios2.File, *name: str*, *variable_name: str = ''*, *separator: str = '/'*) → List[str]

Read a string attribute

Parameters

name
attribute name

variable_name
if attribute is associated with a variable

separator
concatenation string between variable_name and attribute e.g. variable_name + separator + name (var/attr) Not used if variable_name is empty

Returns

list
resulting string list attribute data

`read_string`(**args*, ***kwargs*)

Overloaded function.

1. read_string(self: adios2.adios2.File, name: str, block_id: int = 0) -> List[str]

   Reads string value for current step (use for streaming mode step by step)

   Parameters

   name
   string variable name

   block_id
   required for local variables

   Returns

   list
   data string values. For global values: returns 1 element For local values: returns n-block elements

2. read_string(self: adios2.adios2.File, name: str, step_start: int, step_count: int, block_id: int = 0) -> List[str]

   Reads string value for a certain step (random access mode)

Parameters

name
string variable name

step_start
variable step start

step_count
variable number of steps to read from step_start

block_id
required for local variables

Returns

string
data string values for a certain step range.

`set_parameter`(*self:* adios2.adios2.File, *key: str*, *value: str*) → None

Sets a single parameter. Overwrites value if key exists.

Parameters

key
input parameter key

value
parameter value

`set_parameters`(*self:* adios2.adios2.File, *parameters: Dict[str, str]*) → None

Sets parameters using a dictionary. Removes any previous parameter.

Parameters

parameters
input key/value parameters

value
parameter value

`steps`(*self:* adios2.adios2.File) → int

Inspect available number of steps, for file engines, read mode only

Returns
steps

`write`(**args*, ***kwargs*)

Overloaded function.

1. write(self: adios2.adios2.File, name: str, array: numpy.ndarray, shape: List[int] = [], start: List[int] = [], count: List[int] = [], end_step: bool = False) -> None

   writes a self-describing array (numpy) variable

   Parameters

   name
   variable name

   array
   variable data values

   shape
   variable global MPI dimensions. Pass empty numpy array for local variables.

start
variable offset for current MPI rank. Pass empty numpy array for local variables.

count
variable dimension for current MPI rank. Pass a numpy array for local variables.

end_step
end current step, begin next step and flush (default = false).

2. write(self: adios2.adios2.File, name: str, array: numpy.ndarray, shape: List[int], start: List[int], count: List[int], operations: List[Tuple[str, Dict[str, str]]], end_step: bool = False) -> None

   writes a self-describing array (numpy) variable with operations e.g. compression: 'zfp', 'mgard', 'sz'

   Parameters

   name
   variable name

   array
   variable data values

   shape
   variable global MPI dimensions. Pass empty numpy array for local variables.

   start
   variable offset for current MPI rank. Pass empty numpy array for local variables.

   count
   variable dimension for current MPI rank. Pass a numpy array for local variables.

   end_step
   end current step, begin next step and flush (default = false).

3. write(self: adios2.adios2.File, name: str, array: numpy.ndarray, local_value: bool = False, end_step: bool = False) -> None

   writes a self-describing single value array (numpy) variable

   Parameters

   name
   variable name

   array
   variable data single value

   local_value
   true: local value, false: global value

   end_step
   end current step, begin next step and flush (default = false).

4. write(self: adios2.adios2.File, name: str, string: str, local_value: bool = False, end_step: bool = False) -> None

   writes a self-describing single value string variable

   Parameters

   name
   variable name

string
variable data single value

local_value
true: local value, false: global value

end_step
end current step, begin next step and flush (default = false).

`write_attribute`(**args*, ***kwargs*)

Overloaded function.

1. write_attribute(self: adios2.adios2.File, name: str, array: numpy.ndarray, variable_name: str = ‘’, separator: str = ‘/’, end_step: bool = False) -> None

   writes a self-describing single value array (numpy) variable

   Parameters

   name
   attribute name

   array
   attribute numpy array data

   variable_name
   if attribute is associated with a variable

   separator
   concatenation string between variable_name and attribute e.g. variable_name + separator + name (“var/attr”) Not used if variable_name is empty

   end_step
   end current step, begin next step and flush (default = false).

2. write_attribute(self: adios2.adios2.File, name: str, string_value: str, variable_name: str = ‘’, separator: str = ‘/’, end_step: bool = False) -> None

   writes a self-describing single value array (numpy) variable

   Parameters

   name
   attribute name

   string_value
   attribute single string

   variable_name
   if attribute is associated with a variable

   separator
   concatenation string between variable_name and attribute e.g. variable_name + separator + name (“var/attr”) Not used if variable_name is empty

   end_step
   end current step, begin next step and flush (default = false).

3. write_attribute(self: adios2.adios2.File, name: str, string_array: List[str], variable_name: str = ‘’, separator: str = ‘/’, end_step: bool = False) -> None

   writes a self-describing single value array (numpy) variable

   Parameters

name
attribute name

string_array
attribute string array

variable_name
if attribute is associated with a variable

separator
concatenation string between variable_name and attribute e.g. variable_name + separator + name ("var/attr") Not used if variable_name is empty

end_step
end current step, begin next step and flush (default = false).

11.3 Matlab simple bindings

The ADIOS Matlab API supports reading data from ADIOS BP files with a simplified API that consists of three functions:

- `ADIOSOPEN` returns a structure with information on an ADIOS BP File (variables and attributes).
- `ADIOSREAD` reads in a variable from the file. It expects the info structure returned by `ADIOSOPEN`.
- `ADIOSCLOSE` closes the file.

11.3.1 Organization of an ADIOS BP file

An ADIOS BP file contains a set of variables and attributes. Each variable in the group has a path, which defines a logical hierarchy of the variables within the file.

11.3.2 Time dimension of a variable

Variables can be written several times from a program, if they have a time dimension. The reader exposes the variables with an extra dimension, i.e. a 2D variable written over time is seen as a 3D variable. In MATLAB, the extra dimension is the last dimension (the slowest changing dimension). Since the reader allows reading an arbitrary slice of a variable, data for one timestep can be read in with slicing.

11.3.3 Min/max of arrays

The ADIOS BP format stores the min/max values in each variable. The info structure therefore contains these min/max values. There is practically no overhead to provide this information (along with the values of all attributes) even for file sizes of several terabytes.

In the Matlab console use help for these functions

```
>>> help adiosopen
>>> help adiosread
>>> help adiosclose
```

11.3.4 ADIOSOPEN

FILE = adiosopen(PATH)
Open a file for reading pointed by PATH and return an information structure (FILE).

The returned FILE structure contains the following information

```
Name              File path

Handlers          Object handlers to pass on to ADIOS functions
  FileHandler        uint64 file handler
  GroupHandler       uint64 IO group object handler
  ADIOSHandler       uint64 ADIOS object handler

Variables         Structure array of variables
     Name            Path of variable
     Type            Matlab type class of data
     Dims            Array of dimensions
     StepsStart      First step's index for this variable in file, always at␣
↪least 1
     StepsCount      Number of steps for this variable in file, always at␣
↪least 1
     GlobalMin       Global minimum  of the variable (1-by-1 mxArray)
     GlobalMax       Global maximum of the variable

Attribute         Structure array of attributes
     Name            Path of attribute
     Type            Matlab type class of data
     Value           Attribute value
```

11.3.5 ADIOSREAD

Read data from a BP file opened with adiosopen. Provide the structure returned by adiosopen as the first input argument, and the path to a variable. Inspect file.Variables and file.Attributes for the list of variables and attributes available in a file.

data = adiosread(file, VARPATH)

Read the entire variable VARPATH from a BP file. file is the output of ADIOSOPEN. VARPATH is a string to a variable or attribute. If an N-dimensional array variable has multiple steps in the file this function reads all steps and returns an N+1 dimensional array where the last dimension equals the number of steps.

data = adiosread(file, INDEX)

Read the entire variable from a BP file. INDEX points to a variable in the file.Variables array.

data = adiosread(..., START, COUNT, STEPSTART, STEPCOUNT)

Read a portion of a variable.

```
START and COUNT:
A slice is defined as two arrays of N integers, where N is the
number of dimensions of the variable, describing the
"start" and "count" values. The "start" values start from 1.
    E.g. [1 5], [10 2] reads the first 10 values in the first dimension
```

(continues on next page)

(continued from previous page)

```
and 2 values from the 5th position in the second dimension resulting in
a 10-by-2 array.
    You can use negative numbers to index from the end of the array
as in python. -1 refers to the last element of the array, -2 the one
before and so on.
    E.g. [-1], [1] reads in the last value of a 1D array.
         [1], [-1] reads in the complete 1D array.

STEPSTART and STEPCOUNT:
Similarly, the number of steps from a specific step can be read instead
of all data. Steps start from 1. Negative index can be used as well.
    E.g. -1, 1  will read in the last step from the file
         n, -1  will read all steps from 'n' to the last one
```

11.3.6 ADIOSCLOSE

adiosclose(file)

Close file and free internal data structures. `file` is the structure returned by `adiosopen`.

CHAPTER

TWELVE

AGGREGATION

The basic problem of large-scale I/O is that the N-to-1 and N-to-N (process-to-file) patterns do not scale and one must set the number of files in an output to the capability of the file system, not the size of the application. Hence, N processes need to write to M files to

1) utilize the bandwidth of the file system and to
2) minimize the cost of multiple process writing to a single file, while
3) not overwhelming the file system with too many files.

12.1 Aggregation in BP5

There are two implementations of aggregation in BP5, none of them is the same as the one in BP4. The aggregation setup in ADIOS2 consist of: a) *NumAggregators*, which processes do write to disk (others will send data to them), and b) *NumSubFiles*, how many files they will write.

EveryoneWritesSerial is a simple aggregation strategy. Every process is writing its own data to disk, to one particular file only, and the processes are serialized over each particular file. In this aggregator, *NumAggregators* = *NumSubFiles* (= M). This approach should scale well with application size. On Summit's GPFS though we observe that a single writer per compute node is better than multiple process writing to the file system, hence this aggregation method performs poorly there.

EveryoneWrites is the same strategy as the previous except that every process immediately writes its own data to its designated file. Since it basically implements an N-to-N write pattern, this method does not scale, so only use it up to a moderate number of processes (1-4 process * number of file system servers). At small scale, as long as the file system can deal with the on-rush of the write requests, this method can provide the fastest I/O.

TwoLevelShm has a subset of processes that actually write to disk (*NumAggregators*). There must be at least one process per compute node, which creates a shared-memory segment for other processes on the node to send their data. The aggregator process basically serializes the writing of data from this subset of processes (itself and the processes that send data to it). TwoLevelShm performs similarly to EveryoneWritesSerial on Lustre, and is the only good option on Summit's GPFS.

The number of files (*NumSubFiles*) can be smaller than *NumAggregators*, and then multiple aggregators will write to one file concurrently. Such a setup becomes useful when the number of nodes is many times more than the number of file servers.

TwoLevelShm works best if each process's output data fits into the shared-memory segment, which holds two pages. Since POSIX writes are limited to about 2GB, the best setup is to use 4GB shared-memory size by each aggregator. This is the default size, but you can use the *MaxShmSize* parameter to set this lower if necessary. At runtime, BP5 will only allocate twice the maximum size of the largest data size any process has, but up to MaxShmSize. If the data from two processes does not fit into the shared-memory segment, BP5 will need to perfom multiple iterations of copy and disk-write, which is generally slower than writing large data blocks at once.

The **default setup** is *TwoLevelShm*, where *NumAggregators* is the number of compute nodes the application is running on, and the number of files is the same. This setup is good for Summit's GPFS and good for Lustre at large scale. However, the default setup leaves potential performance on the table when running applications at smaller scale, where the one process per node setup cannot utilize the full bandwidth of a large parallel file system.

CHAPTER

THIRTEEN

MEMORY MANAGEMENT

13.1 BP4 buffering

BP4 has a simple buffering mechanism to provide ultimate performance at the cost of high memory usage: all user data (passed in *Put()* calls) is buffered in one contiguous memory allocation and writing/aggregation is done with this large buffer in *EndStep()*. Aggregation in BP4 uses MPI to send this buffer to the aggregator and hence maintaining two such large buffers. Basically, if an application writes *N* bytes of data in a step, then BP4 needs approximately *2xN* bytes extra memory for buffering.

A potential performance problem is that BP4 needs to extend the buffer occasionally to fit more incoming data (more *Put()* calls). At large sizes the reallocation may need to move the buffer into a different place in memory, which requires copying the entire existing buffer. When there are GBs of data already buffered, this copy will noticably decrease the overall observed write performance. This situation can be avoided if one can guess a usable upper limit to how much data each process is going to write, and telling this to the BP4 engine through the **InitialBufferSize** parameter before *Open()*.

Another potential problem is that reallocation may fail at some point, well before the limits of memory, since it needs a single contiguous allocation be available.

13.2 BP5 buffering

BP5 is designed to use less memory than BP4. The buffer it manages is a list of large chunks. The advantages of the list of chunks is that no reallocation of existing buffer is needed, and that BP5 can potentially allocate more buffer than BP4 since it requests many smaller chunks instead of a large contiguous buffer. In general, chunks should be as big as the system/application can afford, up to **2147381248** bytes (almost but less than 2GB, the actual size limit POSIX write() calls have). Each chunk will result in a separate write call, hence minimizing the number of chunks is preferred. The current default is set to 128MB, so please increase this on large computers if you can and if you write more than that amount of data per process, using the parameter **BufferChunkSize**.

Second, BP5 can add a large user variable as a chunk to this list without copying it at all and use it directly to write (or send to aggregator). *Put(..., adios2::Mode::Deferred)* will handle the user data directly, unless its size is below a threshold (see parameter **MinDeferredSize**).

Note: Do not call *PerformPuts()* when using BP5, because this call forces copying all user data into the internal buffer before writing, eliminating all benefits of zero-copy that BP5 provides when operating with large buffers. Instead, consider using Put() with the Sync option if you want to force ADIOS to copy data immediately. Alternatively, BP5 offers PerformDataWrite(), an collective operation that actually moves data to storage, potentially freeing up buffer and application memory.

Third, BP5 is using a shared memory segment on each compute node for aggregation, instead of MPI. The best settings for the shared memory is 4GB (see parameter **MaxShmSize**), enough place for two chunks with the POSIX write limit. More is useless but can be smaller if a system/application cannot allow this much space for aggregation (but there will be more write calls to disk as a result).

13.3 Span object in internal buffer

Another option to decrease memory consumption is to pre-allocate space in the BP4/BP5 buffer and then prepare output variables directly in that space. This will avoid a copy and the need for doubling memory for temporary variables that are only created for output purposes. This Span feature is only available in C++. See the *Span()* function in *Engine class* ../api_full/api_full.html#engine-class

CHAPTER

FOURTEEN

GPU-AWARE I/O

The `Put` and `Get` functions in the BP4 and BP5 engines can receive user buffers allocated on the host or the device in both Sync and Deferred modes.

Note: Currently only CUDA and HIP allocated buffers are supported for device data.

If ADIOS2 is built without GPU support, only buffers allocated on the host are supported. If ADIOS2 is built with any GPU support, by default, the library will automatically detect where does the buffer memory physically resides.

Users can also provide information about where the buffer was allocated by using the `SetMemorySpace` function within each variable.

```
enum class MemorySpace
{
    Detect, ///< Detect the memory space automatically
    Host,   ///< Host memory space (default)
    GPU     ///< GPU memory spaces
};
```

ADIOS2 can use a CUDA or Kokkos backend for enabling GPU support. Only one backend can be active at a given time based on how ADIOS2 is build.

14.1 Building ADIOS2 with a GPU backend

14.1.1 Building with CUDA enabled

If there is no CUDA toolkit installed, cmake will turn CUDA off automatically. ADIOS2 default behavior for `ADIOS2_USE_CUDA` is to enable CUDA if it can find a CUDA toolkit on the system. In case the system has a CUDA toolkit installed, but it is desired to build ADIOS2 without CUDA enabled `-DADIOS2_USE_CUDA=OFF` must be used.

When building ADIOS2 with CUDA enabled, the user is responsible with setting the correct `CMAKE_CUDA_ARCHITECTURES` (e.g. for Summit the `CMAKE_CUDA_ARCHITECTURES` needs to be set to 70 to match the NVIDIA Volta V100).

14.1.2 Building with Kokkos enabled

The Kokkos library can be used to enable GPU within ADIOS2. Based on how Kokkos is build, either the CUDA or HIP backend will be enabled. Building with Kokkos requires `-DADIOS2_USE_Kokkos=ON`. The user is responsible to set the `CMAKE_CUDA_ARCHITECTURES` to the same architecture used when configuring the Kokkos library it links against.

Note: Kokkos version >= 3.7 is required to enable the GPU backend in ADIOS2

14.2 Writing GPU code

The following is a simple example of writing data to storage directly from a GPU buffer allocated with CUDA relying on the automatic detection of device pointers in ADIOS2. The ADIOS2 API is identical to codes using Host buffers for both the read and write logic.

```
float *gpuSimData;
cudaMalloc(&gpuSimData, N * sizeof(float));
cudaMemset(gpuSimData, 0, N);
auto data = io.DefineVariable<float>("data", shape, start, count);

io.SetEngine("BP5"); // or BPFile
adios2::Engine bpWriter = io.Open(fname, adios2::Mode::Write);
// Simulation steps
for (size_t step = 0; step < nSteps; ++step)
{
    bpWriter.BeginStep();
    bpWriter.Put(data, gpuSimData, adios2::Mode::Deferred); // or Sync
    bpWriter.EndStep();
}
```

If the `SetMemorySpace` function is used, the ADIOS2 library will not detect automatically where the buffer was allocated and will use the information provided by the user for all subsequent Puts or Gets. Example:

```
variable.SetMemorySpace(adios2::MemorySpace::CUDA);
for (size_t step = 0; step < nSteps; ++step)
{
    bpWriter.BeginStep();
    bpWriter.Put(data, gpuSimData, adios2::Mode::Deferred); // or Sync
    bpWriter.EndStep();
}
```

Underneath, ADIOS2 uses the backend used at build time to transfer the data. If ADIOS2 was build with CUDA, only CUDA buffers can be provided. If ADIOS2 was build with Kokkos (with CUDA enabled) only CUDA buffers can be provided. If ADIOS2 was build with Kokkos (with HIP enabled) only HIP buffers can be provided.

14.2.1 Using Kokkos buffers

ADIOS2 supports GPU buffers provided in the form of `Kokkos::View` directly in the Put/Get calls. The memory space can be automatically detected or provided by the user, in the same way as in the CUDA example.

```
Kokkos::View<float *, Kokkos::CudaSpace> gpuSimData("data", N);
bpWriter.Put(data, gpuSimData);
```

If the CUDA backend is being used (and not Kokkos) to enable GPU support in ADIOS2, Kokkos applications can still directly pass `Kokkos::View` as long as the correct external header is included: `#include <adios2/cxx11/KokkosView.h>`.

CHAPTER

FIFTEEN

PLUGINS

ADIOS now has the ability for users to load their own engines and operators through the plugin interface. The basic steps for doing this are:

1. Write your plugin class, which needs to inherit from the appropriate `Plugin*Interface` class.
2. Build as a shared library and add the path to your shared library to the `ADIOS2_PLUGIN_PATH` environment variable.
3. Start using your plugin in your application.

These steps are discussed in further detail below.

15.1 Writing Your Plugin Class

15.1.1 Engine Plugin

Your engine plugin class needs to inherit from the `PluginEngineInterface` class in the `adios2/engine/plugin/PluginEngineInterface.h` header. Depending on the type of engine you want to implement, you'll need to override a number of methods that are inherited from the `adios2::core::Engine` class. These are briefly described in the following table. More detailed documentation can be found in `adios2/core/Engine.h`.

Method	Engine Type	Description
`BeginStep()`	Read/Write	Indicates the beginning of a step
`EndStep()`	Read/Write	Indicates the end of a step
`CurrentStep()`	Read/Write	Returns current step info
`DoClose()`	Read/Write	Close a particular transport
`Init()`	Read/Write	Engine initialization
`InitParameters()`	Read/Write	Initialize parameters
`InitTransports()`	Read/Write	Initialize transports
`PerformPuts()`	Write	Execute all deferred mode `Put`
`Flush()`	Write	Flushes data and metadata to a transport
`DoPut()`	Write	Implementation for `Put`
`DoPutSync()`	Write	Implementation for `Put` (Sync mode)
`DoPutDeferred()`	Write	Implementation for `Put` (Deferred Mode)
`PerformGets()`	Read	Execute all deferred mode `Get`
`DoGetSync()`	Read	Implementation for `Get` (Sync mode)
`DoGetDeferred()`	Read	Implementation for `Get` (Deferred Mode)

Examples showing how to implement an engine plugin can be found in `examples/plugins/engine`. An example write engine is `ExampleWritePlugin.h`, while an example read engine is in `ExampleReadPlugin.h`. The writer is a

simple file writing engine that creates a directory (called `ExamplePlugin` by default) and writes variable information to vars.txt and actual data to data.txt. The reader example reads the files output by the writer example.

In addition to implementing the methods above, you'll need to implement `EngineCreate()` and `EngineDestroy()` functions so ADIOS can create/destroy the engine object. Because of C++ name mangling, you'll need to use `extern "C"`. Looking at `ExampleWritePlugin.h`, this looks like:

```
extern "C" {

adios2::plugin::ExampleWritePlugin *
EngineCreate(adios2::core::IO &io, const std::string &name,
             const adios2::Mode mode, adios2::helper::Comm comm)
{
    return new adios2::plugin::ExampleWritePlugin(io, name, mode,
                                                  comm.Duplicate());
}

void EngineDestroy(adios2::plugin::ExampleWritePlugin * obj)
{
    delete obj;
}

}
```

15.1.2 Operator Plugin

Your operator plugin class needs to inherit from the `PluginOperatorInterface` class in the `adios2/operator/plugin/PluginOperatorInterface.h` header. There's three methods that you'll need to override from the `adios2::core::Operator` class, which are described below.

Method	Description
`Operate()`	Performs the operation, e.g., compress data
`InverseOperate()`	Performs the inverse operation, e.g., decompress data
`IsDataTypeValid()`	Checks that a given data type can be processed

An example showing how to implement an operator plugin can be found at `plugins/EncryptionOperator.h` and `plugins/EncryptionOperator.cpp`. This operator uses libsodium for encrypting and decrypting data.

In addition to implementing the methods above, you'll need to implement `OperatorCreate()` and `OperatorDestroy()` functions so ADIOS can create/destroy the operator object. Because of C++ name mangling, you'll need to use `extern "C"`. Looking at `EncryptionOperator`, this looks like:

```
extern "C" {

adios2::plugin::EncryptionOperator *
OperatorCreate(const adios2::Params &parameters)
{
    return new adios2::plugin::EncryptionOperator(parameters);
}

void OperatorDestroy(adios2::plugin::EncryptionOperator * obj)
{
    delete obj;
```

(continues on next page)

(continued from previous page)

```
}

}
```

15.2 Build Shared Library

To build your plugin, your CMake should look something like the following (using the plugin engine example described above):

```
find_package(ADIOS2 REQUIRED)
set(BUILD_SHARED_LIBS ON)
add_library(PluginEngineWrite
  ExampleWritePlugin.cpp
)
target_link_libraries(PluginEngineWrite adios2::cxx11 adios2::core)
```

When using the Plugin Engine, ADIOS will check for your plugin at the path specified in the `ADIOS2_PLUGIN_PATH` environment variable. If `ADIOS2_PLUGIN_PATH` is not set, and a path is not specified when loading your plugin (see below steps for using a plugin in your application), then the usual `dlopen` search is performed (see the dlopen man page).

Note: The `ADIOS2_PLUGIN_PATH` environment variable can contain multiple paths, which must be separated with a `:`.

When building on Windows, you will likely need to explicitly export the Create and Destroy symbols for your plugin, as symbols are invisible by default on Windows. To do this in a portable way across platforms, you can add something similar to the following lines to your CMakeLists.txt:

```
include(GenerateExportHeader)
generate_export_header(PluginEngineWrite BASE_NAME plugin_engine_write)
target_include_directories(PluginEngineWrite PUBLIC
    $<BUILD_INTERFACE:${CMAKE_CURRENT_BINARY_DIR}>
    $<INSTALL_INTERFACE:include>)
```

Then in your plugin header, you'll need to `#include "plugin_engine_write_export.h"`. Then edit your function defintions as follows:

```
extern "C" {

PLUGIN_ENGINE_WRITE_EXPORT adios2::plugin::ExampleWritePlugin *
    EngineCreate(adios2::core::IO &io, const std::string &name,
                 const adios2::Mode mode, adios2::helper::Comm comm);

PLUGIN_ENGINE_WRITE_EXPORT void
    EngineDestroy(adios2::plugin::ExampleWritePlugin * obj);

}
```

15.3 Using Your Plugin in an Application

For both types of plugins, loading the plugin is done by setting the `PluginName` and `PluginLibrary` parameters in an `adios2::Params` object or `<parameter>` XML tag.

15.3.1 Engine Plugins

For engine plugins, this looks like:

```
adios2::ADIOS adios;
adios2::IO io = adios.DeclareIO("writer");
io.SetEngine("Plugin");
adios2::Params params;
params["PluginName"] = "WritePlugin";
params["PluginLibrary"] = "PluginEngineWrite";
// If the engine plugin has any other parameters, these can be added to
// the same params object and they will be forwarded to the engine
io.SetParameters(params);
```

Where "WritePlugin" is the name that ADIOS will use to keep track of the plugin, and "PluginEngineWrite" is the shared library name. At this point you can open the engine and use it as you would any other ADIOS engine. You also shouldn't need to make any changes to your CMake files for your application.

The second option is using an ADIOS XML config file. If you'd like to load your plugins through an XML config file, the following shows an example XML when using Engine Plugins:

```
<adios-config>
    <io name="writer">
        <engine type="Plugin">
            <parameter key="PluginName" value="WritePlugin" />
            <parameter key="PluginLibrary" value="PluginEngineWrite" />
            <!-- any parameters needed for your plugin can be added here in the␣
↪parameter tag -->
        </engine>
    </io>
    <io name="reader">
        <engine type="Plugin">
            <parameter key="PluginName" value="ReadPlugin" />
            <parameter key="PluginLibrary" value="PluginEngineRead" />
            <!-- any parameters needed for your plugin can be added here in the␣
↪parameter tag -->
        </engine>
    </io>
</adios-config>
```

The examples `examples/plugins/engine/examplePluginEngine_write.cpp` and `examples/plugins/engine/examplePluginEngine_read.cpp` are an example of how to use the engine plugins described above.

15.3.2 Operator Plugins

For operator plugins, the code to use your plugin looks like:

```
// for an adios2::Variable<T> var
adios2::Params params;
params["PluginName"] = "MyOperator";
params["PluginLibrary"] = "EncryptionOperator";
// example param required for the EncryptionOperator
params["SecretKeyFile"] = "test-key";
var.AddOperation("plugin", params);
```

If you'd like to load your operator plugin through an XML config file, the following shows an example:

```
<adios-config>
    <io name="writer">
        <variable name="data">
            <operation type="plugin">
                <parameter key="PluginName" value="OperatorPlugin"/>
                <parameter key="PluginLibrary" value="EncryptionOperator" />
                <parameter key="SecretKeyFile" value="test-key" />
            </operation>
        </variable>
        <engine type="BP5">
        </engine>
    </io>
</adios-config>
```

The examples `examples/plugins/operator/examplePluginOperator_write.cpp` and `examples/plugins/engine/examplePluginOperator_read.cpp` show an example of how to use the `EncryptionOperator` plugin described above.

Note: You don't need to add the `lib` prefix or the shared library ending (e.g., `.so`, `.dll`, etc.) when setting `PluginLibrary`. ADIOS will add these when searching for your plugin library. If you do add the prefix/suffix, ADIOS will still be able to find your plugin. It's also possible to put the full path to the shared library here, instead of using `ADIOS2_PLUGIN_PATH`.

CHAPTER

SIXTEEN

ADIOS2 IN ECP HARDWARE

ADIOS2 is widely used in ECP (Exascale Computing Project) HPC (high performance computing) systems, some particular ADIOS2 features needs from specifics workarounds to run successfully.

16.1 OLCF CRUSHER

16.1.1 SST MPI Data Transport

MPI Data Transport relies on client-server features of MPI which are currently supported in Cray-MPI implementations with some caveats. Here are some of the observed issues and what its workaround (if any) are:

MPI_Finalize will block the system process in the "Writer/Producer" ADIOS2 instance. The reason is that the Producer ADIOS instance internally calls *MPI_Open_port* which somehow even after calling *MPI_Close_port* *MPI_Finalize* still consider its port to be in used, hence blocking the process. The workaround is to use a *MPI_Barrier(MPI_COMM_WORLD)* instead of *MPI_Finalize()* call.

srun does not understand mpmd instructions Simply disable them with the flag -*DADIOS2_RUN_MPI_MPMD_TESTS=OFF*

Tests timeout Since we launch every tests with srun the scheduling times can exceed the test default timeout. Use a large timeout (5mins) for running your tests.

Examples of launching ADIOS2 SST unit tests using MPI DP:

Alternatively, you can configure your CMake build to use srun directly:

CHAPTER

SEVENTEEN

HDF5 API SUPPORT THROUGH VOL

We have developed a HDF5 VOL in order to comply with the ECP request to support HDF5 API. Through this VOL the HDF5 clients can read and write general ADIOS files.

17.1 Disclaimer

The Virtual Object Layer (VOL) is a feature introduced in recent release of HDF5 1.12 (https://hdf5.wiki/index.php/New_Features_in_HDF5_Release_1.12).

So please do make sure your HDF5 version supports the latest VOL.

Once the ADIOS VOL is compiled, There are two ways to apply it:

- externally (through dynamic library, no code change)
- internally (through client code).

17.2 External

- Set the following environment parameters:

```
HDF5_VOL_CONNECTOR=ADIOS2_VOL
HDF5_PLUGIN_PATH=/replace/with/your/adios2_vol/lib/path/
```

Without code change, ADIOS2 VOL will be loaded at runtime by HDF5, to access ADIOS files without changing user code.

17.3 Internal

- include adios header
- call the function to set VOL when H5F is initiated
- call the function to unset VOL when H5F is closed

```
// other includes
#include <adios2/h5vol/H5Vol.h> // ADD THIS LINE TO INCLUDE ADIOS VOL

hid_t  pid = H5Pcreate(H5P_FILE_ACCESS);
// other declarations
```

(continues on next page)

(continued from previous page)

```
hid_t fid = H5Fopen(filename, mode, pid);

H5VL_ADIOS2_set(pid); // ADD THIS LINE TO USE ADIOS VOL

H5Fclose(fid);

H5VL_ADIOS2_unset();  // ADD THIS LINE TO EXIT ADIOS VOL
```

Note: The following features are not supported in this VOL:

- hyperslab support
- HDF5 parameters are not passed down. e.g. compression/decompression
- ADIOS2 parameters is not setup.
- user defined types
- change of variable extent is not supported in ADIOS2.

CHAPTER

EIGHTEEN

COMMAND LINE UTILITIES

ADIOS 2 provides a set of command line utilities for quick data exploration and manipulation that builds on top of the library. The are located in the inside the `adios2-install-location/bin` directory after a `make install`

Tip: Optionally the `adios2-install-location/bin` location can be added to your PATH to avoid absolute paths when using adios2 command line utilities

Currently supported tools are:

- `bpls` : exploration of bp/hdf5 files data and metadata in human readable formats
- `adios_reorganize`
- `adios2-config`
- `sst_conn_tool` : SST staging engine connectivity diagnostic tool

18.1 bpls : Inspecting Data

The `bpls` utility is for examining and pretty-printing the content of ADIOS output files (BP and HDF5 files). By default, it lists the variables in the file including the type, name, and dimensionality.

Let's assume we run the Heat Transfer tutorial example and produce the output by

```
$ mpirun -n 12 ./heatSimulation  sim.bp  3 4   5 4   3 1
Process decomposition  : 3 x 4
Array size per process : 5 x 4
Number of output steps : 3
Iterations per step    : 1

$ mpirun -n 3 ./heatAnalysis sim.bp a.bp 3 1

$ bpls a.bp
  double   T     3*{15, 16}
  double   dT    3*{15, 16}
```

In our example, we have two arrays, `T` and `dT`. Both are 2-dimensional `double` arrays, their global size is `15x16` and the file contains `3 output steps` of these arrays.

Note: bpls is written in C++ and therefore sees the order of the dimensions in *row major*. If the data was written from Fortran in column-major order, you will see the dimension order flipped when listing with bpls, just as a code written

in C++ or python would see the data.

Here is the description of the most used options (use `bpls -h` to print help on all options for this utility).

- `-l`

  Print the min/max of the arrays and the values of scalar values

```
$ bpls -l a.bp
  double   T     3*{15, 16} = 0 / 200
  double   dT    3*{15, 16} = -53.1922 / 49.7861
```

- `-a -A`

  List the attributes along with the variables. `-A` will print the attributes only.

```
$ bpls a.bp -la
  double   T               3*{15, 16} = 0 / 200
  string   T/description   attr   = "Temperature from simulation"
  string   T/unit          attr   = "C"
  double   dT              3*{15, 16} = -53.1922 / 49.7861
  string   dT/description  attr   = "Temperature difference between two steps␣
↪calculated in analysis"
```

- `pattern`, `-e`

  Select which variables/attributes to list or dump. By default the pattern(s) are interpreted as shell file patterns.

```
$ bpls a.bp -la T*
  double   T               3*{15, 16} = 0 / 200
```

  Multiple patterns can be defined in the command line.

```
$ bpls a.bp -la T/* dT/*
  string   T/description   attr   = "Temperature from simulation"
  string   T/unit          attr   = "C"
  string   dT/description  attr   = "Temperature difference between two steps␣
↪calculated in analysis"
```

  If the -e option is given (all) the pattern(s) will be interpreted as regular expressions.

```
$ bpls a.bp -la T.* -e
  double   T               3*{15, 16} = 0 / 200
  string   T/description   attr   = "Temperature from simulation"
  string   T/unit          attr   = "C"
```

- `-D`

  Print the decomposition of a variable. In the BP file, the data blocks written by different writers are stored separately and have their own size info and min/max statistics. This option is useful at code development to check if the output file is written the way intended.

```
$ bpls a.bp -l T -D
  double   T               3*{15, 16} = 0 / 200
    step 0:
      block 0: [ 0: 4,  0:15] = 3.54199e-14 / 200
```

(continues on next page)

(continued from previous page)

```
    block 1: [ 5: 9,  0:15] = 58.3642 / 200
    block 2: [10:14,  0:15] = 0 / 200
  step 1:
    block 0: [ 0: 4,  0:15] = 31.4891 / 153.432
    block 1: [ 5: 9,  0:15] = 68.2107 / 180.184
    block 2: [10:14,  0:15] = 31.4891 / 161.699
  step 2:
    block 0: [ 0: 4,  0:15] = 48.0431 / 135.225
    block 1: [ 5: 9,  0:15] = 74.064 / 170.002
    block 2: [10:14,  0:15] = 48.0431 / 147.87
```

In this case we find 3 blocks per output step and 3 output steps. We can see that the variable `T` was decomposed in the first (slow) dimension. In the above example, the `T` variable in the simulation output (`sim.bp`) had 12 blocks per step, but the analysis code was running on 3 processes, effectively reorganizing the data into fewer larger blocks.

- `-d`

  Dump the data content of a variable. For pretty-printing, one should use the additional `-n` and `-f` options. For selecting only a subset of a variable, one should use the `-s` and `-c` options.

  By default, six values are printed per line and using C style `-g` prints for floating point values.

```
$ bpls a.bp -d T
  double   T     3*{15, 16}
    (0, 0, 0)    124.925 124.296 139.024 95.2078 144.864 191.485
    (0, 0, 6)    139.024 140.814 124.925 109.035 110.825 58.3642
    (0, 0,12)    104.985 154.641 110.825 125.553 66.5603 65.9316
    ...
    (2,14, 4)    105.918 116.842 111.249 102.044 93.3121 84.5802
    (2,14,10)    75.3746 69.782 80.706 93.5492 94.7595 95.0709
```

  For pretty-printing, use the additional `-n` and `-f` options.

```
$ bpls a.bp -d T -n 16 -f "%3.0f"
  double   T     3*{15, 16}
    (0, 0, 0)    125 124 139  95 145 191 139 141 125 109 111  58 105 155 111 126
    (0, 1, 0)     67  66  81  37  86 133  81  82  67  51  52   0  47  96  52  67
    (0, 2, 0)    133 133 148 104 153 200 148 149 133 118 119  67 114 163 119 134
    ...
    (2,13, 0)     98  98  96  96 115 132 124 109  97  86  71  63  79  98  97  95
    (2,14, 0)     96  96  93  93 106 117 111 102  93  85  75  70  81  94  95  95
```

  For selecting a subset of a variable, use the `-s` and `-c` options. These options are N+1 dimensional for N-dimensional arrays with more than one steps. The first element of the options are used to select the starting step and the number of steps to print.

  The following example dumps a `4x4` small subset from the center of the array, one step from the second (middle) step:

```
$ bpls a.bp -d T -s "1,6,7" -c "1,4,4" -n 4
  double   T     3*{15, 16}
    slice (1:1, 6:9, 7:10)
    (1,6, 7)    144.09 131.737 119.383 106.787
    (1,7, 7)    145.794 133.44 121.086 108.49
```

(continues on next page)

(continued from previous page)

```
    (1,8, 7)    145.794 133.44 121.086 108.49
    (1,9, 7)    144.09 131.737 119.383 106.787
```

- `-y --noindex`

  Data can be dumped in a format that is easier to import later into other tools, like Excel. The leading array indexes can be omitted by using this option. Non-data lines, like the variable and slice info, are printed with a starting `;`.

```
$ bpls a.bp -d T -s "1,6,7" -c "1,4,4" -n 4 --noindex
  ; double   T     3*{15, 16}
  ;   slice (1:1, 6:9, 7:10)
  144.09 131.737 119.383 106.787
  145.794 133.44 121.086 108.49
  145.794 133.44 121.086 108.49
  144.09 131.737 119.383 106.787
```

Note: HDF5 files can also be dumped with bpls if ADIOS was built with HDF5 support. Note that the HDF5 files do not contain min/max information for the arrays and therefore bpls always prints 0 for them:

```
$ bpls -l a.h5
  double   T     3*{15, 16} = 0 / 0
  double   dT    3*{15, 16} = 0 / 0
```

18.2 adios_reorganize

`adios_reorganize` and `adios_reorganize_mpi` are generic ADIOS tools to read in ADIOS streams and output the same data into another ADIOS stream. The two tools are for serial and MPI environments, respectively. They can be used for

- converting files between ADIOS BP and HDF5 format
- using separate compute nodes to stage I/O from/to disk to/from a large scale application
- reorganizing the data blocks for a different number of blocks

Let's assume we run the Heat Transfer tutorial example and produce the output by

```
$ mpirun -n 12 ./heatSimulation  sim.bp  3 4   5 4   3 1
Process decomposition  : 3 x 4
Array size per process : 5 x 4
Number of output steps : 3
Iterations per step    : 1

$ bpls sim.bp
  double   T     3*{15, 16}
```

In our example, we have an array, `T`. It is a 2-dimensional `double` array, its global size is `15x16` and the file contains `3 output steps` of this array. The array is composed of 12 separate blocks coming from the 12 producers in the application.

- Convert BP file to HDF5 file

If ADIOS is built with HDF5 support, this tool can be used to convert between the two file formats.

```
$ mpirun -n 2 adios_reorganize_mpi sim.bp sim.h5 BPFile "" HDF5 "" 2 1

$ bpls sim.h5
  double   T     3*{15, 16}

$ h5ls -r sim.h5
/                        Group
/Step0                   Group
/Step0/T                 Dataset {15, 16}
/Step1                   Group
/Step1/T                 Dataset {15, 16}
/Step2                   Group
/Step2/T                 Dataset {15, 16}
```

- Stage I/O through extra compute nodes

  If writing data to disk is a bottleneck to the application, it may be worth to use extra nodes for receiving the data quickly from the application and then write to disk while the application continues computing. Similarly, data can be staged in from disk into extra nodes and make it available for fast read-in for an application. One can use one of the staging engines in ADIOS to perform this data staging (SST, SSC, DataMan).

  Assuming that the heatSimulation is using SST instead of file I/O in a run (set in its `adios2.xml` configuration file), staging to disk can be done this way:

```
Make sure adios2.xml sets SST for the simulation:
    <io name="SimulationOutput">
        <engine type="SST">
        </engine>
    </io>


$ mpirun -n 12 ./heatSimulation  sim.bp  3 4   5 4   3 1 : \
         -n 2 adios_reorganize_mpi sim.bp staged.bp SST "" BPFile "" 2 1

$ bpls staged.bp
  double   T     3*{15, 16}
```

  Data is staged to the extra 2 cores and those will write the data to disk while the heatSimulation calculates the next step. Note, that this staging can only be useful if the tool can write all data to disk before the application produces the next output step. Otherwise, it will still block the application for I/O.

- Reorganizing the data blocks in file for a different number of blocks

  In the above example, the application writes the array from 12 processes, but then `adios_reorganize_mpi` reads the global arrays on 2 processes. The output file on disk will therefore contain the array in 2 blocks. This reorganization of the array may be useful if reading is too slow for a dataset created by many-many processes. One may want to reorganize a file written by tens or hundreds of thousands of processes if one wants to read the content more than one time and the read time proves to be a bottleneck in one's work flow.

```
$ mpirun -n 12 ./heatSimulation  sim.bp  3 4   5 4   3 1
$ bpls sim.bp -D
```

(continues on next page)

(continued from previous page)

```
  double   T     3*{15, 16}
      step 0:
        block  0: [ 0: 4,  0: 3]
        block  1: [ 5: 9,  0: 3]
        block  2: [10:14,  0: 3]
        block  3: [ 0: 4,  4: 7]
        block  4: [ 5: 9,  4: 7]
        block  5: [10:14,  4: 7]
        block  6: [ 0: 4,  8:11]
        block  7: [ 5: 9,  8:11]
        block  8: [10:14,  8:11]
        block  9: [ 0: 4, 12:15]
        block 10: [ 5: 9, 12:15]
        block 11: [10:14, 12:15]
      step 1:
        block  0: [ 0: 4,  0: 3]
        block  1: [ 5: 9,  0: 3]
        block  2: [10:14,  0: 3]
        block  3: [ 0: 4,  4: 7]
        block  4: [ 5: 9,  4: 7]
        block  5: [10:14,  4: 7]
        block  6: [ 0: 4,  8:11]
        block  7: [ 5: 9,  8:11]
        block  8: [10:14,  8:11]
        block  9: [ 0: 4, 12:15]
        block 10: [ 5: 9, 12:15]
        block 11: [10:14, 12:15]
      step 2:
        block  0: [ 0: 4,  0: 3]
        block  1: [ 5: 9,  0: 3]
        block  2: [10:14,  0: 3]
        block  3: [ 0: 4,  4: 7]
        block  4: [ 5: 9,  4: 7]
        block  5: [10:14,  4: 7]
        block  6: [ 0: 4,  8:11]
        block  7: [ 5: 9,  8:11]
        block  8: [10:14,  8:11]
        block  9: [ 0: 4, 12:15]
        block 10: [ 5: 9, 12:15]
        block 11: [10:14, 12:15]


$ mpirun -n 2 adios_reorganize_mpi sim.bp reorg.bp BPFile "" BPFile "" 2 1
$ bpls reorg.bp -D
  double   T     3*{15, 16}
      step 0:
        block 0: [ 0: 6,  0:15]
        block 1: [ 7:14,  0:15]
      step 1:
        block 0: [ 0: 6,  0:15]
        block 1: [ 7:14,  0:15]
      step 2:
```

(continues on next page)

(continued from previous page)

```
        block 0: [ 0: 6,  0:15]
        block 1: [ 7:14,  0:15]
```

18.3 adios2-config

adios2-config is provided to aid with non-CMake builds (*e.g.* manually generated Makefile). Running the *adios2-config* command under *adios2-install-dir/bin/adios2-config* will generate the following usage information:

```
./adios2-config --help
adios2-config (--help | [--c-flags] [--c-libs] [--cxx-flags] [--cxx-libs] [-fortran-
↪flags] [--fortran-libs])
  --help           Display help information
  -c               Both compile and link flags for the C bindings
  --c-flags        Preprocessor and compile flags for the C bindings
  --c-libs         Linker flags for the C bindings
  -x               Both compile and link flags for the C++ bindings
  --cxx-flags      Preprocessor and compile flags for the C++ bindings
  --cxx-libs       Linker flags for the C++ bindings
  -f               Both compile and link flags for the F90 bindings
  --fortran-flags  Preprocessor and compile flags for the F90 bindings
  --fortran-libs   Linker flags for the F90 bindings
```

Please refer to the *From non-CMake build systems* for more information on how to use this command.

18.4 sst_conn_tool : SST network connectivity tool

The `sst_conn_tool` utility exposes some aspects of SST network connectivity parameters and activity in order to allow debugging of SST connections.

In its simplest usage, it just lets you test an SST connection (between two runs of the program) and tells you the network information its trying, I.E. what IP address and port it determined to use for listening, and if it's connecting somewhere what those parameters are. For example, you'd first run sst_conn_tool in one window and its output would look like this:

```
bash-3.2$ bin/sst_conn_tool

        Sst writer is listening for TCP/IP connection at IP 192.168.1.17, port 26051

        Sst connection tool waiting for connection...
```

To try to connect from another window, you run sst_conn_tool with the -c or —connect option:

```
bash-3.2$ bin/sst_conn_tool -c

        Sst reader at IP 192.168.1.17, listening UDP port 26050


        Attempting TCP/IP connection to writer at IP 192.168.1.17, port 26051

```

(continues on next page)

(continued from previous page)

```
Connection success, all is well!
bash-3.2$
```

Here, it has found the contact information file, tried and succeeded in making the connection and has indicated that all is well. In the first window, we get a similar message about the success of the connection.

In the event that there is trouble with the connection, there is a "-i" or "—info" option that will provide additional information about the network configuration options. For example:

```
bash-3.2$ bin/sst_conn_tool -i

        ADIOS2_IP_CONFIG best guess hostname is "sandy.local"
        ADIOS2_IP_CONFIG Possible interface lo0 : IPV4 127.0.0.1
        ADIOS2_IP_CONFIG Possible interface en0 : IPV4 192.168.1.56
        ADIOS2_IP_CONFIG Possible interface en5 : IPV4 192.168.1.17
        ADIOS2_IP_CONFIG best guess IP is "192.168.1.17"
        ADIOS2_IP_CONFIG default port range is "any"

        The following environment variables can impact ADIOS2_IP_CONFIG operation:
                ADIOS2_IP               -  Publish the specified IP address for contact
                ADIOS2_HOSTNAME         -  Publish the specified hostname for contact
                ADIOS2_USE_HOSTNAME     -  Publish a hostname preferentially over IP␣
↪address
                ADIOS2_INTERFACE        -  Use the IP address associated with the␣
↪specified network interface
                ADIOS2_PORT_RANGE       -  Use a port within the specified range
↪"low:high",
                                           or specify "any" to let the OS choose

        Sst writer is listening for TCP/IP connection at IP 192.168.1.17, port 26048

        Sst connection tool waiting for connection...
```

Full options for sst_conn_tool:

Operational Modes:

- **-l --listen**
  Display connection parameters and wait for an SST connection (default)
- -c --connect Attempt a connection to an already-running instance of sst_conn_tool

Additional Options:

- **-i --info**
  Display additional networking information for this host
- **-f --file**
  Use file-based contact info sharing (default). The SST contact file is created in the current directory
- **-s --screen**
  Use screen-based contact info sharing, SST contact info is displayed/entered via terminal
- **-h --help**
  Display this message usage and options

CHAPTER

NINETEEN

VISUALIZING DATA

Certain ADIOS2 bp files can be recognized by third party visualization tools. This section describes how to create an ADIOS2 bp file to accomodate the visualization product requirements.

19.1 Using VTK and Paraview

ADIOS BP files can now be seamlessly integrated into the Visualization Toolkit and Paraview. Datasets can be described with an additional attribute that conforms to the VTK XML data model formats as high-level descriptors that will allow interpretation of ADIOS2 variables into a hierarchy understood by the VTK infrastructure. This XML data format is saved in ADIOS2 as either an attribute or as an additional `vtk.xml` file inside the `file.bp.dir` directory.

There are currently a number of limitations:

- It only works with MPI builds of VTK and Paraview
- Support only one block per ADIOS dataset
- Only supports BP Files, streams are not supported
- Currently working up to 3D (and linearized 1D) variables for scalars and vectors.
- Image Data, vti, is supported with ADIOS2 Global Array Variables only
- Unstructured Grid, vtu, is supported with ADIOS2 Local Arrays Variables only

Two VTK file types are supported:

1. Image data (.vti)
2. Unstructured Grid (.vtu)

The main idea is to populate the above XML format contents describing the extent and the data arrays with ADIOS variables in the BP data set. The result is a more-than-compact typical VTK data file since extra information about the variable, such as dimensions and type, as they already exist in the BP data set.

A typical VTK image data XML descriptor (.vti):

```
<?xml version="1.0"?>
<VTKFile type="ImageData">
  <ImageData WholeExtent="x1 x2 y1 y2 z1 z2" Origin="x0 y0 z0" Spacing="dx dy dz">
    <Piece Extent="x1 x2 y1 y2 z1 z2">
      <PointData>
        <DataArray Name="p1"/>
        <DataArray Name="p2"/>
      </PointData>
      <CellData>
```

(continues on next page)

(continued from previous page)

```
        <DataArray Name="c1"/>
        <DataArray Name="c2"/>
      </CellData>
    </Piece>
  </ImageData>
</VTKFile>
```

A typical VTK unstructured grid XML descriptor (.vtu):

```
<?xml version="1.0"?>
<VTKFile type="ImageData">
  <ImageData WholeExtent="x1 x2 y1 y2 z1 z2" Origin="x0 y0 z0" Spacing="dx dy dz">
    <Piece Extent="x1 x2 y1 y2 z1 z2">
      <PointData>
        <DataArray Name="p1"/>
        <DataArray Name="p2"/>
      </PointData>
      <CellData>
        <DataArray Name="c1"/>
        <DataArray Name="c2"/>
      </CellData>
    </Piece>
  </ImageData>
</VTKFile>
```

In addition, VTK can interpret physical-time or output-step varying data stored with ADIOS by reusing the special "TIME" tag. This is better illustrated in the following section.

19.1.1 Saving the vtk.xml data model

For the full source code of the following illustration example see the gray-scott adios2 tutorial.

To incorporate the data model in a BP data file, the application has two options:

1) Adding a string attribute called "vtk.xml" in code. "TIME" is a special tag for adding physical time variables.

```
const std::string imageData = R"(
     <?xml version="1.0"?>
     <VTKFile type="ImageData" version="0.1" byte_order="LittleEndian">
       <ImageData WholeExtent=")" + extent + R"(" Origin="0 0 0" Spacing="1 1 1">
         <Piece Extent=")" + extent + R"(">
           <CellData Scalars="U">
               <DataArray Name="U" />
               <DataArray Name="V" />
               <DataArray Name="TIME">
                 step
               </DataArray>
           </CellData>
         </Piece>
       </ImageData>
     </VTKFile>)";

io.DefineAttribute<std::string>("vtk.xml", imageData);
```

Tip: C++11 users should take advantage C++11 string literals (`R"( xml_here )"`) to simplify escaping characters in the XML.

The resulting bpls output should contain the "vtk.xml" attribute and the variables in the model:

```
> bpls gs.bp -lav
File info:
  of variables:  3
  of attributes: 7
  statistics:    Min / Max

  double   Du       attr   = 0.2
  double   Dv       attr   = 0.1
  double   F        attr   = 0.02
  double   U        24*{48, 48, 48} = 0.107439 / 1.04324
  double   V        24*{48, 48, 48} = 0 / 0.672232
  double   dt       attr   = 1
  double   k        attr   = 0.048
  double   noise    attr   = 0.01
  int32_t  step     24*scalar = 0 / 575
  string   vtk.xml  attr   =
<VTKFile type="ImageData" version="0.1" byte_order="LittleEndian">
  <ImageData WholeExtent="0 49 0 49 0 49" Origin="0 0 0" Spacing="1 1 1">
    <Piece Extent="0 49 0 49 0 49">
      <CellData Scalars="U">
        <DataArray Name="U" />
        <DataArray Name="V" />
        <DataArray Name="TIME">
          step
        </DataArray>
      </CellData>
    </Piece>
  </ImageData>
</VTKFile>
```

2) Saving a "vtk.xml" file inside the file.bp.dir to describe the data after is created

```
> cat gs.bp.dir/vtk.xml

  <?xml version="1.0"?>
  <VTKFile type="ImageData" version="0.1" byte_order="LittleEndian">
    <ImageData WholeExtent=")" + extent + R"(" Origin="0 0 0" Spacing="1 1 1">
      <Piece Extent=")" + extent + R("">
        <CellData Scalars="U">
          <DataArray Name="U" />
          <DataArray Name="V" />
          <DataArray Name="TIME">
            step
          </DataArray>
        </CellData>
      </Piece>
    </ImageData>
```

(continues on next page)

(continued from previous page)

```
  </VTKFile>
```

This BP file should be recognize by Paraview:

Similarly, unstructured grid (.vtu) support can be added with the limitations of using specific labels for the variable names setting the "connectivity", "vertices", and cell "types".

The following example is taken from example 2 of the MFEM product examples website using ADIOS2:

The resulting *bpls* output for unstructured grid data types:

```
File info:
  of variables:  6
  of attributes: 4
  statistics:    Min / Max

  uint32_t  NumOfElements       {4} = 1024 / 1024
  uint32_t  NumOfVertices       {4} = 1377 / 1377
  string    app                 attr   = "MFEM"
  uint64_t  connectivity        [4]*{1024, 9} = 0 / 1376
  uint32_t  dimension           attr   = 3
  string    glvis_mesh_version  attr   = "1.0"
  double    sol                 [4]*{1377, 3} = -0.201717 / 1.19304
  uint32_t  types               scalar = 11
  double    vertices            [4]*{1377, 3} = -1.19304 / 8.20172
  string    vtk.xml             attr   =
<VTKFile type="UnstructuredGrid" version="0.1" byte_order="LittleEndian">
  <UnstructuredGrid>
    <Piece NumberOfPoints="NumOfVertices" NumberOfCells="NumOfElements">
      <Points>
        <DataArray Name="vertices" />
      </Points>
      <Cells>
        <DataArray Name="connectivity" />
        <DataArray Name="types" />
      </Cells>
      <PointData>
        <DataArray Name="sol" />
      </PointData>
    </Piece>
  </UnstructuredGrid>
</VTKFile>
```

and resulting visualization in Paraview for different "cell" types:

CHAPTER

TWENTY

FAQ

20.1 MPI vs Non-MPI

1. *Can I use the same library for MPI and non-MPI code?*

20.2 APIs

1. *Can I use ADIOS 2 C++11 library with C++98 codes?*
2. *Why are C and Fortran APIs missing functionality?*
3. *C++11: Why are std::string arguments passed sometimes by value and sometimes by reference?*
4. *C++11: Should I pass adios2:: objects by value or by reference?*
5. *Fortran: Can I pass slices and temporary arrays to adios2_put?*

20.3 Building on Titan

1. *My application uses PGI compilers on Titan, can I link ADIOS 2?*
2. *How do I enable the Python bindings on Titan?*

20.4 Building and Running on Fujitsu FX100

1. *How do I build ADIOS 2 on Fujitsu FX100?*
2. *SST engine hangs on Fujitsu FX100. Why?*

20.5 FAQs Answered

20.5.1 Can I use the same library for MPI and non-MPI code?

Short answer: Yes, since version 2.6.0.

Long answer: One build of ADIOS can be used by both serial and parallel applications. Use the `-s` and `-m` flags in the `adios2-config` command. By default, or with the `-m` flag, the command gives the flags for a parallel build, which add `-DADIOS2_USE_MPI` to the compilation flags and include extra libaries containing the MPI implementations into the linker flags. The `-s` flag will omit these flags. For example, if ADIOS is installed into /opt/adios2, the flags for a Fortran application will look like these:

```
$ /opt/adios2/bin/adios2-config --fortran-flags
-DADIOS2_USE_MPI -I/opt/adios2/include/adios2/fortran
$ /opt/adios2/bin/adios2-config --fortran-flags -m
-DADIOS2_USE_MPI -I/opt/adios2/include/adios2/fortran
$ /opt/adios2/bin/adios2-config --fortran-flags -s
-I/opt/adios2/include/adios2/fortran

$ /opt/adios2/bin/adios2-config --fortran-libs
-Wl,-rpath,/opt/adios2/lib /opt/adios2/lib/libadios2_fortran_mpi.so.2.6.0 /opt/
↪adios2/lib/libadios2_fortran.so.2.6.0 -Wl,-rpath-link,/opt/adios2/lib
$ /opt/adios2/bin/adios2-config --fortran-libs -s
-Wl,-rpath,/opt/adios2/lib /opt/adios2/lib/libadios2_fortran.so.2.6.0 -Wl,-
↪rpath-link,/opt/adios2/lib
```

If using cmake, there are different targets to build parallel

```
find_package(MPI REQUIRED)
find_package(ADIOS2 REQUIRED)
#...
add_library(my_library src1.cxx src2.cxx)
target_link_libraries(my_library PRIVATE adios2::cxx11_mpi MPI::MPI_C)
#...
add_library(my_f_library src1.F90 src2.F90)
target_link_libraries(my_f_library PRIVATE adios2::fortran_mpi adios2::fortran␣
↪MPI::MPI_Fortran)
```

and serial applications:

```
find_package(ADIOS2 REQUIRED)
#...
add_library(my_library src1.cxx src2.cxx)
target_link_libraries(my_library PRIVATE adios2::cxx11)
#...
add_library(my_f_library src1.F90 src2.F90)
target_link_libraries(my_f_library PRIVATE adios2::fortran)
```

20.5.2 Can I use ADIOS 2 C++11 library with C++98 codes?

Use the *C bindings*. C++11 is a brand new language standard and many new (and old, *e.g.* `std::string`) might cause ABI conflicts.

20.5.3 Why are C and Fortran APIs missing functionality?

Because language intrinsics are NOT THE SAME. For example, C++ and Python support key/value pair structures natively, *e.g.* `std::map` and dictionaries, respectively. Fortran and C only support arrays natively. Use the right language (tool) for the right task.

20.5.4 C++11: Why are std::string arguments passed sometimes by value and sometimes by reference?

C++11, provides mechanisms to optimize copying small objects, rather than passing by reference. The latter was always the rule for C++98. When a string is passed by value, it's assumed that the name will be short, <= 15 characters, most of the time. While passing by reference indicates that the string can be of any size. Check the isocpp guidelines on this topic for more information.

20.5.5 C++11: Should I pass adios2:: objects by value or by reference?

`adios2::ADIOS`: always pass by reference this is the only "large memory" object; all others: pass by reference or value depending on your coding standards and requirements, they are small objects that wrap around a pointer to an internal object inside `adios2::ADIOS`.

20.5.6 Fortran: Can I pass slices and temporary arrays to adios2_put?

By definition the lifetime of a temporary is the scope of the function is passed to. Therefore, you must use sync mode with `adios2_put`. Deferred mode will save garbage data since the memory location of a temporary is undefined after `adios2_put`, not able to reach `adios2_end_step`, `adios2_close` or `adios2_perform_puts` where the memory is actually used.

20.5.7 My application uses PGI compilers on Titan, can I link ADIOS 2?

Follow directions at *Building on HPC Systems* to setup support for PGI on Titan. PGI compilers depend on GNU headers, but they must point to a version greater than gcc 4.8.1 to support C++11 features. The gcc module doesn't need to be loaded, though. Example:

```
$ module load gcc/7.2.0
$ makelocalrc $(dirname $(which pgc++)) -gcc $(which gcc) -gpp $(which g++) -
↪g77 $(which gfortran) -o -net 1>${HOME}/.mypgirc 2>/dev/null
$ module unload gcc/7.2.0
```

20.5.8 How do I enable the Python bindings on Titan?

The default ADIOS2 configuration on Titan builds a static library. Python bindings require enabling the dynamic libraries and the Cray dynamic environment variable. See *Building on HPC Systems* and *Enabling the Python bindings*. For example:

```
[atkins3@titan-ext4 code]$ mkdir adios
[atkins3@titan-ext4 code]$ cd adios
[atkins3@titan-ext4 adios]$ git clone https://github.com/ornladios/adios2.git␣
↪source
[atkins3@titan-ext4 adios]$ module swap PrgEnv-pgi PrgEnv-gnu
[atkins3@titan-ext4 adios]$ module load cmake3/3.11.3
[atkins3@titan-ext4 adios]$ module load python python_numpy python_mpi4py
[atkins3@titan-ext4 adios]$ export CRAYPE_LINK_TYPE=dynamic CC=cc CXX=CC FC=ftn
[atkins3@titan-ext4 adios]$ mkdir build
[atkins3@titan-ext4 build]$ cd build
[atkins3@titan-ext4 build]$ cmake ../source
-- The C compiler identification is GNU 6.3.0
-- The CXX compiler identification is GNU 6.3.0
-- Cray Programming Environment 2.5.13 C
-- Check for working C compiler: /opt/cray/craype/2.5.13/bin/cc
-- Check for working C compiler: /opt/cray/craype/2.5.13/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Cray Programming Environment 2.5.13 CXX
-- Check for working CXX compiler: /opt/cray/craype/2.5.13/bin/CC
-- Check for working CXX compiler: /opt/cray/craype/2.5.13/bin/CC -- works
...
-- Found PythonInterp: /sw/titan/.swci/0-login/opt/spack/20180315/linux-suse_
↪linux11-x86_64/gcc-4.3.4/python-2.7.9-v6ctjewwdx6k2qs7ublexz7gnx457jo5/bin/
↪python2.7 (found version "2.7.9")
-- Found PythonLibs: /sw/titan/.swci/0-login/opt/spack/20180315/linux-suse_
↪linux11-x86_64/gcc-4.3.4/python-2.7.9-v6ctjewwdx6k2qs7ublexz7gnx457jo5/lib/
↪libpython2.7.so (found version "2.7.9")
-- Found PythonModule_numpy: /sw/xk6/python_numpy/1.7.1/python2.7.9_craylibsci_
↪gnu4.9.0/lib64/python2.7/site-packages/numpy
-- Found PythonModule_mpi4py: /lustre/atlas/sw/xk7/python_mpi4py/2.0.0/cle5.
↪2up04_python2.7.9/lib64/python2.7/site-packages/mpi4py
-- Found PythonFull: /sw/titan/.swci/0-login/opt/spack/20180315/linux-suse_
↪linux11-x86_64/gcc-4.3.4/python-2.7.9-v6ctjewwdx6k2qs7ublexz7gnx457jo5/bin/
↪python2.7  found components:  Interp Libs numpy mpi4py
...
ADIOS2 build configuration:
  ADIOS Version: 2.4.0
  C++ Compiler : GNU 6.3.0 CrayPrgEnv
    /opt/cray/craype/2.5.13/bin/CC

  Fortran Compiler : GNU 6.3.0 CrayPrgEnv
    /opt/cray/craype/2.5.13/bin/ftn

  Installation prefix: /usr/local
```

(continues on next page)

(continued from previous page)

```
        bin: bin
        lib: lib
    include: include
      cmake: lib/cmake/adios2
     python: lib/python2.7/site-packages

  Features:
    Library Type: shared
    Build Type:   Release
    Testing: ON
    Build Options:
      BZip2    : ON
      ZFP      : OFF
      SZ       : OFF
      MGARD    : OFF
      MPI      : ON
      DataMan  : ON
      SST      : ON
      ZeroMQ   : OFF
      HDF5     : OFF
      Python   : ON
      Fortran  : ON
      SysVShMem: ON
      Endian_Reverse: OFF

-- Configuring done
-- Generating done
-- Build files have been written to: /ccs/home/atkins3/code/adios/build
```

20.5.9 How do I build ADIOS 2 on Fujitsu FX100?

- Cross-compilation (building on the login node) is not recommended. Submit an interactive job and build on the compute nodes.
- Make sure CMake >= 3.6 is installed on the compute nodes. If not, you need to build and install it from source since CMake does not provide SPARC V9 binaries.
- Use gcc instead of the Fujitsu compiler. We tested with gcc 6.3.0
- CMake fails to automatically find the correct MPI library on FX100. As a workaround, set CC, CXX, and FC to the corresponding MPI compiler wrappers:

```
$ CC=mpigcc CXX=mpig++ FC=mpigfortran cmake  ..
```

20.5.10 SST engine hangs on Fujitsu FX100. Why?

The communication thread of SST might have failed to start. FX100 requires users to set the maximum stack size manually when launching POSIX threads. One way to do this is through ulimit (*e.g.* `ulimit -s 1024`). You can also set the stack size when submitting the job. Please contact your system administrator for details.

CHAPTER

TWENTYONE

ADVICE

This list is similar to the Advice sections for each chapter in The C++ Programming Language, Fourth Edition by Bjarne Stroustrup The goal is to provide specific advice and good practices about the use of ADIOS2 in other projects.

1. Use `MPI_COMM_SELF` to run MPI compiled versions of ADIOS 2 in "serial" mode
2. Use a runtime configuration file in the `ADIOS` constructor or `adios2_init` when targeting multiple engines
3. Check object validity when developing (similar to `fstream`):
   - C++: *operator bool*

     ```
     if(var) engine.Put(var, data);
     ```

   - C: NULL pointer

     ```
     if(var) adios2_put(engine, var, data, adios2_mode_deferred);
     ```

   - **Python: v2 `__nonzero__` v3 `__bool__`. Note: do not check for None object**

     ```
     if(var) engine.Put(var, data);
     ```

   - Fortran: `type%valid`

     ```
     if( adios%valid .eqv. .true. ) then
        adios2_declare_io(adios, io, "foo")
     end if
     ```

4. C++11 and Python: use `try-catch` (`try-except` in Python) blocks to handle exceptions from ADIOS 2
5. C++11: use fixed-width types (`int8_t`, `uint32_t`) for portability
6. Define your data structure: set of variables and attributes before developing. Data hierarchies/models can be built on top of ADIOS 2.
7. Read the documentation for *Supported Engines* before targeting development for a particular engine
8. MPI development: treat `ADIOS` constructor/destructor (`adios2_init`/`adios2_finalize`) and Engine `Open` and `Close` always as collective functions. For the most part, ADIOS 2 API functionality is local, but other Engine functions might follow other rules, *Supported Engines*.
9. Use *Remove* functions carefully. They create dangling objects/pointers.
10. Thread-safety: treat ADIOS 2 as NOT thread-safe. Either use a mutex or only handle I/O from a master thread. ADIOS 2 is about performance, adding I/O serial algorithm operations into a parallel execution block may reduce parallel portions from Amdahl's Law.

11. Prefer the high-level Python and C++ APIs for simple tasks that do not require performance. The more familiar Write/Read overloads for File I/O return native data constructs (`std::vector` and `numpy arrays`) immediately for a requested selection. `open` only explores the metadata index.
12. C++: prefer templates to `void*` to increase compile-time safety. Use `IO::InquireVariableType("variableName")` and `adios2::GetType<T>()` to cast upfront to a `Variable<T>`. C++17 has `std::any` as an alternative to `void*`. ADIOS 2 follows closely the STL model.
13. Understand `Put` and `Get` memory contracts from *Engine*
14. Prefer Put/Get `Deferred` mode, treat `Sync` as a special mode
15. `Put Span`: create all spans in a step before populating them. Spans follow the same iterator invalidation rules as `std::vector`, so use `span.data()` to always keep the span pointer up-to-date
16. Always populate data before calling `Put` in deferred mode, and do not change it between `Put` and `EndStep`, or `Close`
17. Never call `PerformPuts` right before `EndStep`. This was a code pattern that had no adverse effects with the BP3/4 file engines and is present in some older code, but was never beneficial.
18. Use `BeginStep` and `EndStep` to write code that is portable across all ADIOS 2 Engine types: file and streaming.
19. Always use `Close` for every call to `Open`.
20. C, Fortran: always call `adios2_finalize` for every call to `adios2_init` to avoid memory leaks.
21. Reminder: C++, C, Python: Row-Major, while Fortran: Column-Major. ADIOS 2 will handle interoperability between ordering. Remember that *bpls : Inspecting Data* is always a Row-Major reader so Fortran reader need to swap dimensions seen in bpls. bpls: (slow, ..., fast) -> Fortran(fast,...,slow).
22. Fortran API: use the type members (`var%valid`, `var%name`, etc.) to get extra type information.
23. Fortran C interoperability: Fortran bindings support the majority of applications using Fortran 90. We currently don't support the `ISO_C_BINDING` interoperability module in Fortran 2003.
24. Always keep the `IO` object self-contained keeping its own set of `Variables`, `Attributes` and `Engines`. Do not combine Variables with multiple Engines or multiple modes, unless it's 100% guaranteed to be safe in your program avoiding Variable access conflicts.
25. Developers: explore the testing infrastructure `ADIOS2/testing` in ADIOS 2 as a starting point for using ADIOS 2 in your own testing environment.
26. Become a super-user of *bpls : Inspecting Data* to analyze datasets generated by ADIOS 2.

- search

PYTHON MODULE INDEX

INDEX

A

B

C

D

E

F

G

I

L

M

N

O

P

R

S

T

V

W