ADIOS2-Examples

Release 2.7.0

Oak Ridge National Laboratory

Apr 16, 2021

INTRODUCTION

CHAPTER

ONE

INTRODUCTION

This documentation contains examples for using the ADIOS2 I/O framework. Install ADIOS2 first on your system. In this documentation we will assume it is installed in `/opt/adios2` and that the binaries are in the path. You need to modify the paths to your commands if it is installed somewhere else.

If you are interested in the Fortran examples, make sure ADIOS2 is configured and built with a Fortran compiler. Similarly, if you are interested in the Python examples, make sure `Python 3` is available (with `Numpy` and `MPI4py` included) when configuring and building ADIOS2, then add the installed adios2 python library to the `PYTHONPATH`.

1.1 Downloading the examples

The examples are on GitHub. You can download a release matching the version of your ADIOS2 version, or clone the repository:

```
git clone https://github.com/ornladios/ADIOS2-Examples
cd ADIOS2-Examples
```

1.2 Building the examples

As explained in the README, you can build all the examples at once using CMake or Meson builders. In this case, all the binaries will be located in the build directory. This approach is mostly useful if you want to see how to incorporate ADIOS2 into your own CMake/Meson project or if you just want to quickly build all examples and run them, or if you are on Windows.

A more manual approach is preferred if you are interested in particular examples. In this documentation we describe and use the examples in this manual approach but you can find the same binaries in the full build, just in a different place (in the build directory).

1.3 Where to start?

First we suggest to read the Components of ADIOS2 in the ADIOS2 documentation. Next, familiarize yourself with the tools of ADIOS2, mainly `bpls` and `adios2-config`.

Then start with the *Manual build* example for your choice of language. Then you will be able to build any of the examples individually.

Please read the CONCEPTS section carefully. It is important to understand them to get the ADIOS IO right.

CHAPTER

TWO

MANUAL BUILD

Location of the example(s): `source/manual-build`

The notes below assume, ADIOS2 is in the path, and the MPI compilers are (mpic++, mpicc, mpif90) and serial compilers are (g++, gcc, gfortran). You need to modify the commands below for your own environment and choice of compilers.

2.1 MPI vs serial codes

A single installation of the ADIOS2 library can be used for both serial and MPI applications. You need to use the correct compile flags and link libraries to build your code. Basically, the MPI functions are incorporated in separate libraries (*libadios2_core_mpi.so*, *libadios2_cxx_mpi.so*, *libadios2_fortran_mpi.so*, *libadios2_c_mpi.so*), which you need to add to the linking phase.

2.2 adios2-config

The *adios2-config* utility is created during the installation of ADIOS2 (not during build!). It is located in the *bin/* directory of the installation. It provides the compiler flags and linker libraries to your manual/Makefile builds.

```
$ adios2-config -h
adios2-config [OPTION]
  -h, --help       Display help information
  -v, --version    Display version 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, --cxx        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, --fortran    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
  -s, --serial     Select flags for serial applications
  -m, --mpi        Select flags for mpi applications
```

2.3 Compiling and linking a C++/MPI program

Compile and link example-mpi.cpp -> example-mpi-cpp

```
CXXFLAGS=`adios2-config --cxx-flags`
LDFLAGS=`adios2-config --cxx-libs`
mpic++ -o example-mpi-cpp ${CXXFLAGS} example-mpi.cpp ${LDFLAGS}
```

2.4 Compiling and linking a C++ serial program

Compile and link example-serial.cpp -> example-serial-cpp

```
CXXFLAGS=`adios2-config --cxx-flags -s`
LDFLAGS=`adios2-config --cxx-libs -s`
g++ -o example-serial-cpp ${CXXFLAGS} example-serial.cpp ${LDFLAGS}
```

2.5 Compiling and linking a C/MPI program

Compile and link example-mpi.c -> example-mpi-c

```
CFLAGS=`adios2-config --c-flags`
LDFLAGS=`adios2-config --c-libs`
mpicc -o example-mpi-c ${CFLAGS} example-mpi.c ${LDFLAGS}
```

2.6 Compiling and linking a C serial program

Compile and link example-serial.c -> example-serial-c

```
CFLAGS=`adios2-config --c-flags -s`
LDFLAGS=`adios2-config --c-libs -s`
gcc -o example-serial-c ${CXXFLAGS} example-serial.c ${LDFLAGS}
```

2.7 Compiling and linking a Fortran/MPI program

Compile and link example-mpi.F90 -> example-mpi-f

```
FCFLAGS=`adios2-config --fortran-flags`
LDFLAGS=`adios2-config --fortran-libs`
mpif90 -o example-mpi-f ${FCFLAGS} example-mpi.F90 ${LDFLAGS}
```

2.8 Compiling and linking a Fortran serial program

Compile and link example-serial.F90 -> example-serial-f

```
FCFLAGS=`adios2-config --fortran-flags -s`
LDFLAGS=`adios2-config --fortran-libs -s`
gfortran -o example-serial-f ${FCFLAGS} example-serial.F90 ${LDFLAGS}
```

2.9 Running the examples in *source/manual-build*

Each of these examples create an output file <*example*>.bp, which contains a single string variable *Greeting*. *bpls* is a tool from the ADIOS2 installation.

```
$ mpirun -n 4 example-mpi-f
Hello World from ADIOS2 Fortran/MPI exam
Hello World from ADIOS2 Fortran/MPI exam
Hello World from ADIOS2 Fortran/MPI exam
Hello World from ADIOS2 Fortran/MPI exam

$ bpls -la example-mpi-f.bp/
string   Greeting  scalar = "Hello World from ADIOS2 Fortran/MPI example"
```

2.10 Running other examples

Other examples in the ADIOS2-Examples repository are more complex than just a single source file. You can still create the compile and link command line yourself but we provide a simple local Makefile for each example. Each Makefile includes the `make.settings` file from the root directory of the repository. You need to edit this settings file to set the ADIOS2 installation directory and the names of the serial and MPI compilers.

CHAPTER

THREE

STREAM VS FILE

When we start thinking of I/O we first focus on a single application writing an output file to disk, or read an input file from disk. In general, however, that output is going to be read by another application or applications, once or more. That input had been written by another (or the same) application. Thinking further, we may want to read portions of the data written by another application while the first application is still producing more data. And we certainly don't want the file system to be the bottleneck when we do so.

The file system always is, or is going to be at some point of scaling an application, a bottleneck. It forces us to separate data meant for immediate consumption from data meant for preservation. Why do we store temporary data on the permanent (and slow) storage? Because we don't have better solutions or because it is too complicated to do otherwise.

3.1 Step

The ADIOS I/O abstraction is designed around the concept of producers and consumers that exchange data, consumed immediately or much later, and around the fact that small pieces of data exchange is inefficient. It forces a producer to output a big chunk of data (called **Step**). Consumers get access to a complete Step and are guaranteed that it is available for consumption until the consumer releases it. A step can consist of anything, composed of n-dimensional distributed arrays, single values, strings, and a new step can be completely different in structure than the previous one. But usually, a repetitive application, like most simulations, produce the same structure of data with updated content. So usually, the content of the Step, the list of variables and attributes, need to be defined once and then publish new content every now and then. But remember, variables can be added, modified (e.g. array sizes), removed in new steps if necessary.

3.2 Checkpoint

So what about the checkpoint file which is written once and read never (usually)? It fits the ADIOS abstraction as a single step, written to a file. And later the application may read it from the file as a single step.

3.3 Code coupling at every iteration

Two separate applications (e.g. for multiphysics multiscale problems solved partially by different code bases) may want to exchange data very frequently (once or multiple times in every iteration). Exhange = two separate input-output streams between two applications. Minimizing the time for data movement is key. The ADIOS interface allows to write this data exchange with two streams. Exchanging data is still possible using files on disk but the cost of I/O will be overwhelming. ADIOS provides other solutions (`Engines`) that can use the local network to move data much faster then through the file system. Using a file-based engine, however, is the best thing one can have while developing these applications and to debug the data content being exchanged.

3.4 But I just want a file

That's all right. There are two extra capabilities in ADIOS that only applies to files on disk. These are random access to steps from a file, and reading multiple steps at once from the file. This is provided by the `Variable.SetStepSelection` (C++ method), `adios2_set_step_selection` (Fortran/C function). This is a per-variable function, so one can read different variables from different steps from a file. Don't use `BeginStep`/`EndStep` functions in the reading code after opening a file and you are good to go. Just remember, if you do so, your code can never connect to a running application to process data on the fly.

Note: `bpls` is a tool that reads files and you can dump multiple steps at once. Alternatively you can use the `-t` parameter to force bpls read step by step (still from a file).

CHAPTER

FOUR

WHY MPMD MODE?

Location of examples: `source/fortran/adios-module/mpivars.F90`

MPMD (Multiple Program Multiple Data) mode is when we compose one MPI World from multiple executables. Usually one can launch two executables separately by two MPI launch commands (mpirun, jsrun, aprun, etc.) and each application will have its own world and own `MPI_COMM_WORLD` communicator. Sharing the world is beneficial if the two codes know about each other and can use MPI for communicating with each other. The cost of preparing your code for MPMD mode is simply to expect other codes be present and avoid using MPI_COMM_WORLD for in-application operations.

ADIOS has specialized solutions to use MPI for the data movement in a stream but it requires MPMD mode. If the steps in a stream is "fixed", i.e. the list of variables, their definition and the writing/reading patterns are the same in every step, the SSC engine of ADIOS can use one-sided MPI functions to push the data to the consumer quickly. SSC still works if the content of the steps change but needs to spend considerable amount of time to figure out which process of the consumer gets which portion of the data every step.

You don't need to use MPMD mode at all to get faster communication. RDMA and TCP based solutions are available to communicate between two or more separate concurrent applications (SST and DataMan engines). However, the effort to prepare your code for MPMD mode is very small, and it may come handy some day on some system to use MPI for ADIOS I/O.

4.1 Split communicator

The only thing required for MPMD mode is that each application has its own communicator that only includes the processes of that application. *MPI_COMM_WORLD* includes all processes created in one launch command. Therefore, the application must first create its own "app" communicator and then completely avoid using MPI_COMM_WORLD in the application code.

MPI has an operation for this: `MPI_Comm_split`. Each application must pick a unique ID ("color") to distinguish themselves from other applications. Additionally, every process can pick its new rank in the "app" communicator but the simplest way is to keep the original order of ranks.

```
integer:: app_comm, rank, nproc
integer:: wrank, wnproc
integer:: ierr
integer, parameter:: color = 7215

call MPI_Init(ierr)
! World comm spans all applications started with the same mpirun command
call MPI_Comm_rank(MPI_COMM_WORLD, wrank, ierr)
call MPI_Comm_size(MPI_COMM_WORLD, wnproc, ierr)

! Have to split and create a 'world' communicator for this app only
```

(continues on next page)

(continued from previous page)

```
! color must be unique for each application
call MPI_Comm_split (MPI_COMM_WORLD, color, wrank, app_comm, ierr)
call MPI_Comm_rank (app_comm, rank, ierr)
call MPI_Comm_size (app_comm, nproc , ierr)
```

```
MPI_Init(&argc, &argv);

int wrank, wnproc;
MPI_Comm_rank(MPI_COMM_WORLD, &wrank);
MPI_Comm_size(MPI_COMM_WORLD, &wnproc);

const unsigned int color = 18325;
MPI_Comm app_comm;
MPI_Comm_split(MPI_COMM_WORLD, color, wrank, &app_comm);

int rank, nproc;
MPI_Comm_rank(app_comm, &rank);
MPI_Comm_size(app_comm, &nproc);
```

```
from mpi4py import MPI

color = 3231
wrank = MPI.COMM_WORLD.Get_rank()
wnproc = MPI.COMM_WORLD.Get_size()
app_comm = MPI.COMM_WORLD.Split(color, wrank)
nproc = app_comm.Get_size()
rank = app_comm.Get_rank()
```

CHAPTER

FIVE

ADIOS OBJECTS

ADIOS2 has a few objects that need to be created and maintained for performing I/O. Since you have already read the Components of ADIOS2 in the ADIOS2 documentation, there is nothing to be added here about this topic ;-)

CHAPTER

SIX

ANATOMY OF AN ADIOS OUTPUT

```
ADIOS adios("config.xml", MPI_COMM_WORLD);
|
|   IO io = adios.DeclareIO(...);
|   |
|   |   Variable<...> var = io.DefineVariable<...>(...)
|   |   Attribute<...> attr = io.DefineAttribute<...>(...)
|   |   Engine e = io.Open(...);
|   |   |
|   |   |   e.BeginStep()
|   |   |   |
|   |   |   |   e.Put(var, datapointer);
|   |   |   |
|   |   |   e.EndStep()
|   |   |
|   |   e.Close();
|   |
|   |--> IO goes out of scope
|
|--> ADIOS goes out of scope or adios2_finalize()
```

The pseudo code above depicts the basic structure of performing output. The `ADIOS` object is necessary to hold all other objects. It is initialized with an MPI communicator in a parallel program or without in a serial program. Additionally, a config file (XML or YAML format) can be specified here to load runtime configuration. Only one ADIOS object is needed throughout the entire application but you can create as many as you want (e.g. if you need to separate IO objects using the same name in a program that reads similar input from an ensemble of multiple applications).

The `IO` object is required to hold the variable and attribute definitions, and runtime options for a particular input or output stream. The IO object has a name, which is used only to refer to runtime options in the configuration file. One IO object can only be used in one output or input stream. The only exception where an IO object can be used twice is one input stream plus one output stream where the output is reusing the variable definitions loaded during input.

`Variable` and `Attribute` definitions belong to one IO object, which means, they can only be used in one output. You need to define new ones for other outputs. Just because a Variable is defined, it will not appear in the output unless an associated Put() call provides the content.

A stream is opened and closed once. The `Engine` object implements the data movement for the stream. It depends on the runtime options of the IO object that what type of an engine is created in the Open() call. One output step is denoted by a pair of BeginStep..EndStep block.

An output step consist of variables and attributes. Variables are just definitions without content, so one must call a Put() function to provide the application data pointer that contains the data content one wants to write out. Attributes have their content in their definitions so there is no need for an extra call.

Some rules:

- Variables can be defined any time, before the corresponding Put() call
- Attributes can be defined any time before EndStep
- The following functions must be treated as Collective operations
- ADIOS
- Open
- BeginStep
- EndStep
- Close

Note: If there is only one output step, and we only want to write it to a file on disk, never stream it to other application, then BeginStep and EndStep are not required but it does not make any difference if they are called.

CHAPTER

SEVEN

ANATOMY OF AN ADIOS INPUT

```
ADIOS adios("config.xml", MPI_COMM_WORLD);
|
|   IO io = adios.DeclareIO(...);
|   |
|   |   Engine e = io.Open(...);
|   |   |
|   |   |   e.BeginStep()
|   |   |   |
|   |   |   |   varlist = io.AvailableVariables(...)
|   |   |   |   Variable var = io.InquireVariable(...)
|   |   |   |   Attribute attr = io.InquireAttribute(...)
|   |   |   |   |
|   |   |   |   |   e.Get(var, datapointer);
|   |   |   |   |
|   |   |   |
|   |   |   e.EndStep()
|   |   |
|   |   e.Close();
|   |
|   |--> IO goes out of scope
|
|--> ADIOS goes out of scope or adios2_finalize()
```

The difference between input and output is that while we have to define the variables and attributes for an output, we have to retrieve the available variables in an input first as definitions (Variable and Attribute objects).

If we know the particular variable (name and type) in the input stream, we can get the definition using InquireVariable(). Generic tools that process any input must use other functions to retrieve the list of variable names and their types first and then get the individual Variable objects. The same is true for Attributes.

7.1 File-only Input

This only works for files on permanent storage for post-mortem processing. Do not use BeginStep/Endstep, so the file-based Engine knows that you want to see all steps. Then use var.SetStepSelection() to select the (range of contigous) steps you want to read in at once. Allocate enough memory to hold multiple steps of the variable content.

```
ADIOS adios("config.xml", MPI_COMM_WORLD);
|
|   IO io = adios.DeclareIO(...);
|   |
|   |   Engine e = io.Open(...);
```

(continues on next page)

(continued from previous page)

```
|   |   |
|   |   |   Variable var = io.InquireVariable(...)
|   |   |   |   var.SetStepSelection()
|   |   |   |   e.Get(var, datapointer);
|   |   |   |
|   |   |
|   |   e.Close();
|   |
|   |--> IO goes out of scope
|
|--> ADIOS goes out of scope or adios2_finalize()
```

CHAPTER

EIGHT

DEFERRED VS SYNC

Warning: This is the most dangerous and controversial concept of ADIOS and you must understand what is going on to avoid writing or reading garbage instead of your precious data.

8.1 Default mode: Deferred

The Put() and Get() functions simply associate an application data pointer to a variable definition. They do not perform actual IO on their own, unless when explicitly asked.

By default, the list of Put() and Get() calls in a step enumerate the IO requests that need to be performed in the step. The actions are performed in EndStep() and the results are available after EndStep().

For an output stream this means, that the data pointer passed in the Put() call must hold the actual data until the end of EndStep(). If you pass temporary data here (because you reuse the same memory to prepare multiple output variables in consecutive Put calls, or pass a Fortran subselection on an array), there will be garbage in the memory by the time EndStep takes care of the output.

For an input stream this means, that after the Get() call, you must not attempt to use the data pointer yet because the data is not there yet. You can only use the data pointer after the EndStep call.

So remember, keep the data pointers valid until EndStep and use their content only after EndStep.

8.2 Sync mode

Of course, Deferred mode is limiting. Sometimes we want to pass temporary pointers to Put() that will go out of scope right after the Put() call. And sometimes, we need to read in some data to know how to read the rest. `Sync` mode flag can be passed to both Put() and Get() to force the engine to take care of that piece of data right away.

```
C++: engine.Get<float>(varF, data, adios2::Mode::Sync);
```

```
F90: call adios2_get(engine, varF, data, adios2_mode_sync, ierr)
```

```
Python: engine.Get(varF, data, adios2.Mode.Sync)
```

8.3 Why deferred?

We wanted to design an API that enables specialized engines to perform at their best. We did not want to add a single function for multiple purposes and then be stuck with its semantics. An engine may perform better if it knows all the outputs or inputs to perform in a batch instead of performing them one by one.

A particularly good example is the Inline engine that is used to run an analysis code inside the application. All application data is exposed directly (zero-copy) to the reading code when EndStep is called by the application. If Sync mode is used in a Put(), the engine has no choice but to copy the data into a private buffer to preserve it for the reader.

8.4 PerformPuts/PerformGets

The ADIOS2 API has PerformGets and PerformPuts functions for the situation when one can Put/Get multiple variables in one batch, then another set of variables, within one step. These function enfore Sync mode for all Puts and Gets called before it. In practice, one really does not need to bother with these functions.

Technically, EndStep always calls PerformGets/PerformPuts to do this part of IO and then publishes the entire step for any readers, but this topic belongs to Engine development, not for users.

CHAPTER

NINE

VARIABLES

Start with reading about Variables here https://adios2.readthedocs.io/en/latest/components/components.html#variable

Location of examples:

- C++: `source/cpp/basics/variables-shapes.cpp`
- Fortran: `source/fortran/shapes`

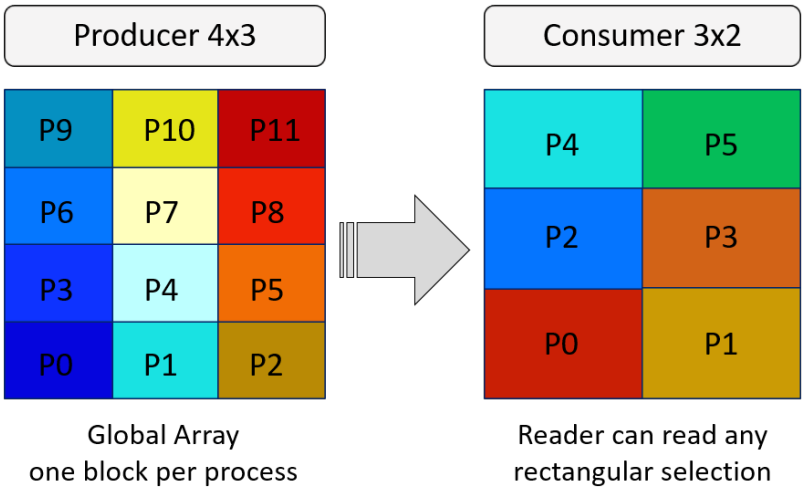
CHAPTER

TEN

GLOBAL ARRAY

ADIOS2 is focusing on writing and reading N-dimensional, distributed, global arrays of primitive types (floats, doubles, integers, complex numbers, chars, etc.). The basic idea is that, usually, a simulation has such a data structure in memory (distributed across multiple processes) and wants to dump its content regularly as it progresses. ADIOS2 was designed

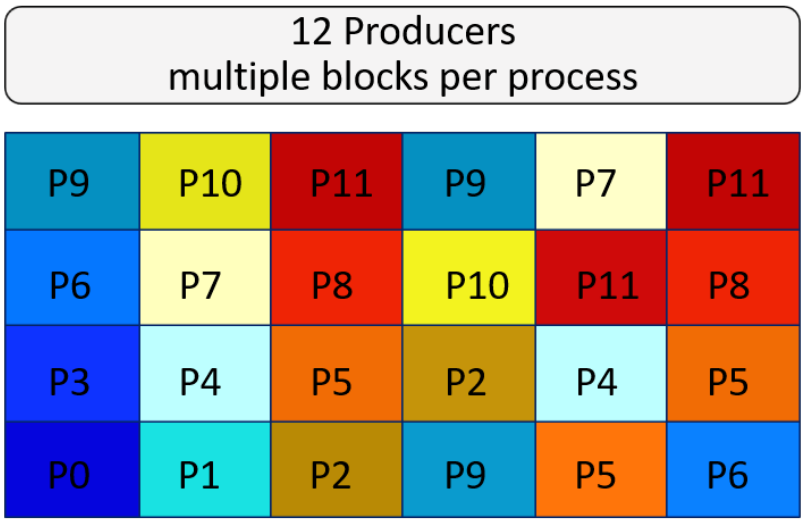
- to do this writing and reading as fast as possible
- to enable reading any subsection of the array



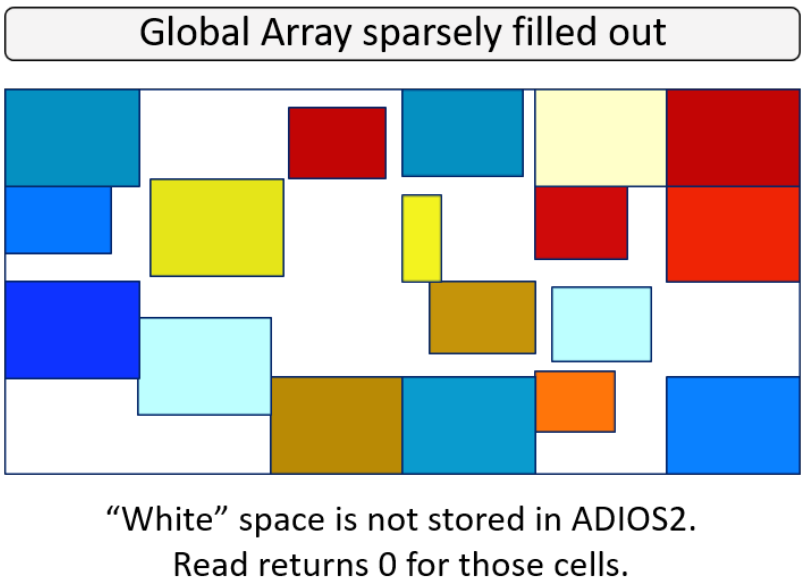
The figure above shows a parallel application of 12 processes producing a 2D array. Each process has a 2D array locally and the output is created by placing them into a 4x3 pattern. A reading application's individual process then can read any subsection of the entire global array. In the figure, a 6 process application decomposes the array in a 3x2 pattern and each process reads a 2D array whose content comes from multiple producer processes.

The figure hopefully helps to understand the basic concept but it can be also misleading if it suggests limitations that are not there. Global Array is simply a boundary in N-dimensional space where processes can place their blocks of data. In the global space

- one process can place multiple blocks

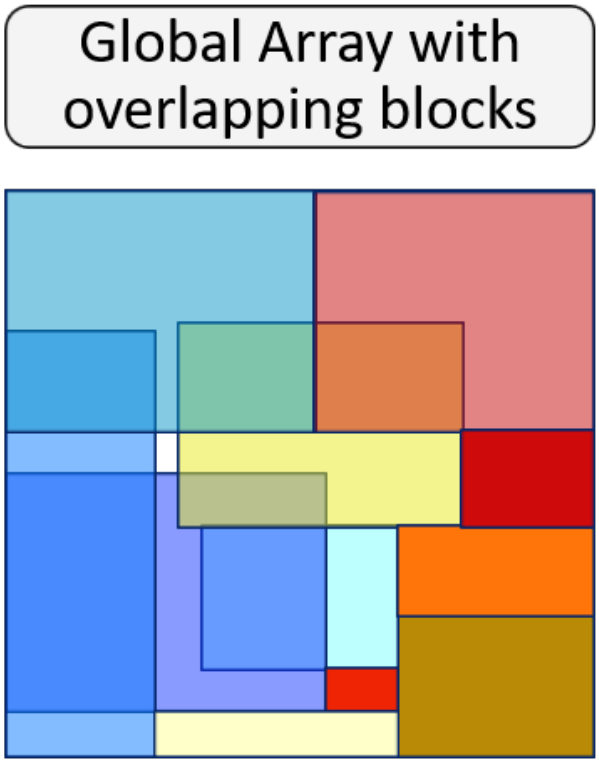


- does NOT need to be fully covered by the blocks



  - at reading, unfilled positions will not change the allocated memory

- blocks can overlap

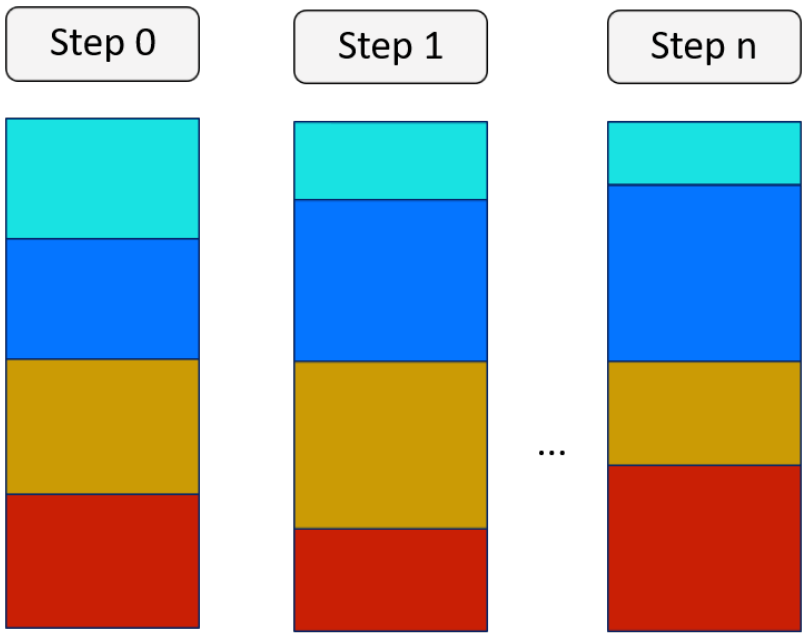


  - the reader will get values in an overlapping position from one of the block but there is no control over from which block

- each process can put a different size of block, or put multiple blocks of different sizes
- some process may not contribute anything to the global array

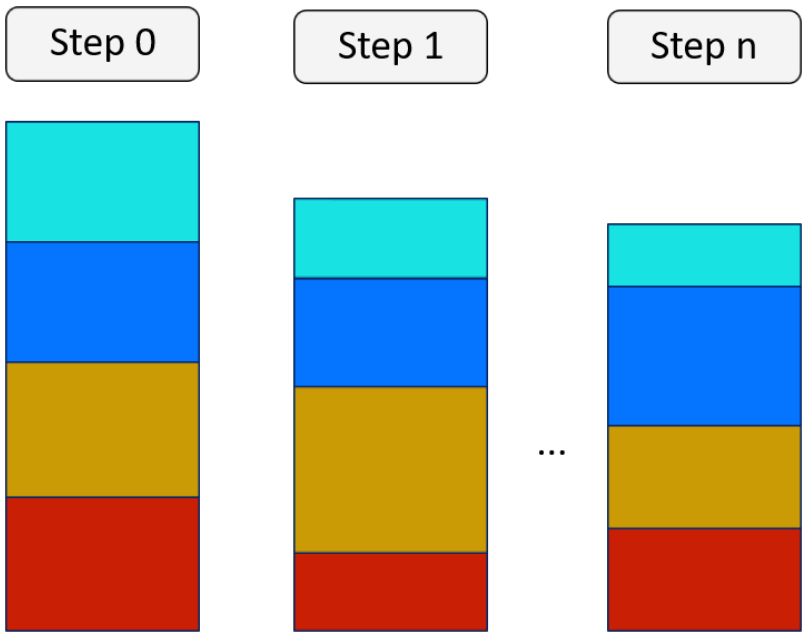
Over multiple output steps

- the processes CAN change the size (and number) of blocks in the array
  - E.g. atom table: global size is fixed but atoms wander around processes, so their block size is changing



2D table where
the number of rows per producer is changing

- the global dimensions CAN change over output steps
  - but then you cannot read multiple steps at once
  - E.g. particle table size changes due to particles disappearing or appearing



2D table where
the number of rows per producer as well as
the total global size are changing

Limitations of the ADIOS global array concept

- Indexing starts from 0
- Cyclic data patterns are not supported; only blocks can be written or read
- Some blocks's may fully or partially fall outside of the global boundary. The reader will not be able to read those parts

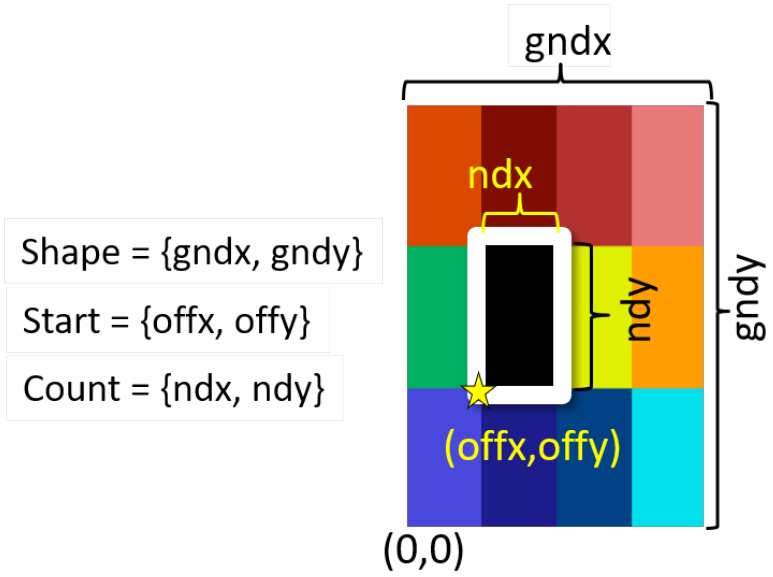
---

Note: Technically, the content of the individual blocks is kept in the BP format (but not in HDF5 format) and in staging. If you really, really want to retrieve all the blocks, you need to handle this array as a Local Array and read the blocks one by one.

10.1 Defining a global array

When defining a global array, all processes, that want to contribute, must define the global array with

- *Name* of the variable as the readers will see it
- *Shape*: a shape (the global boundary, i.e. global size)
- *Count*: the size of the local block and its
- *Start*: position of the local block in the global space



The only fixed information at the point of definition is the name and the number of dimensions. The *Start* and *Count* can be modified later with *SetSelection* before writing a block to the output. This actually needs to be done if one process wants to produce multiple blocks in the array. The *Shape* of the array can be modified up to the point of the last *Put* call in the output step but the final shape must be the same on every producer.

CHAPTER

ELEVEN

FORTRAN EXAMPLES

The ADIOS2 Fortran bindings are documented in https://adios2.readthedocs.io/en/latest/api_full/api_full.html#fortran-bindings.

When writing Fortran code, it comes handy to look at the source code of the Fortran bindings directly to find the possible signatures of each function, and especially at the ADIOS2 constants.

11.1 Including ADIOS2 in a Fortran application

ADIOS2 is using a Fortran 90 interface, so it cannot be directly used in an F77 code base. Files that include adios calls, should be named *.F90 (capital F) so that macros are pre-processed by the Fortran compiler.

Since ADIOS2 is object-oriented and is implemented in C++, there is a need to use global variables in Fortran to initialize an `adios` object and to keep it alive ("in scope") as long as it is needed. Usually, the adios object is created after MPI is initialized and destroyed before MPI is finalized. An `io` object is used to define all variables, attributes and runtime parameters for an input or output. Finally, an `engine` object needed to call the actual operations on the input or output (open, close and read/write-related calls).

The easiest way to manage the objects is to put them into a new module for ADIOS I/O. The example is in `source/fortran/adios-module/adiosio.F90`

Assuming you have the MPI communicator in another module name *mpivars*, and the communicator is named *app_comm*:

```
module adiosio
    use adios2
    implicit none

    type(adios2_adios) :: adios2obj
    type(adios2_io) :: io_diag
    type(adios2_engine) :: engine_diag

contains

subroutine init_adiosio()
    use mpivars
    use adios2
    implicit none
    integer*4 :: err
    call adios2_init(adios2obj, app_comm, err)
    if (.not.adios2obj%valid) then
        write(*,*) "Error when creating ADIOS2 object"
    endif
```

(continues on next page)

(continued from previous page)

```
end subroutine init_adiosio

subroutine finalize_adiosio()
    use adios2
    implicit none
    integer*4 :: err
    ! close the engine now if it was not closed yet
    if (engine_diag%valid) then
        call adios2_close(engine_diag, err)
    endif
    if (adios2obj%valid) then
        call adios2_finalize(adios2obj, err)
    endif
end subroutine finalize_adiosio

end module adiosio
```

Note: The example above adds an IO and an Engine object to the module, which is not required if you write an output or read an input in a single subroutine and do so only once. However, if you want to refer to them in multiple subroutines, or you want to read/write multiple steps, you need to keep these objects (technically, object references) alive to avoid premature destruction of the C++ objects behind.

Note: The example above also closes the engine object in the finalization. This is an example to follow if you want to open an output somewhere else in the code, and output new data (new steps) regularly until the end. This usage is not encouraged here. It is cleaner if you manage the (single) open and (single) close of your outputs and the (multiple) output steps yourself. However, in existing large applications with many optional modules it may be difficult to find the exact spot where the output stream actually ends. This subroutine is the last point where the output must be closed.

In the main program, include these snippets of code:

```
program adios2_module_example
    use mpivars
#ifdef ADIOS2
    use adiosio
#endif
    implicit none

...

    ! After call MPI_Init()
#ifdef ADIOS2
    call init_adiosio()
#endif

...

    ! Before call MPI_Finalize()
#ifdef ADIOS2
    call finalize_adiosio()
#endif
```

Note: We are using the *ADIOS2* macro everywhere to allow for building the application with or without ADIOS2. If

you make ADIOS2 a required dependency, there is no need for this macro.

CHAPTER

TWELVE

DEFINING, WRITING AND READING VARIABLES

12.1 Global Array

12.1.1 Global array of fixed shape

This is how we define a 1D array of type *real4* and of size *nproc* x *NX* where every process has an array of size *NX*, there are *nproc* processes, and their *rank* runs from 0..*nproc*-1:

```
use adios2
implicit none
type(adios2_io) :: io
type(adios2_variable) :: var_g
...
integer*4 :: ndim
integer*8, dimension(1) :: fixed_shape, fixed_start, fixed_count

ndim = 1
fixed_count = NX
fixed_shape = nproc * NX
fixed_start = rank * NX

call adios2_define_variable(var_g, io, "GlobalArray", &
                            adios2_type_real4, ndim, &
                            fixed_shape, fixed_start, fixed_count, &
                            adios2_constant_dims, ierr)
```

Note: The flag `adios2_constant_dims` indicates that this definition is fixed, i.e. the block selection and the global array shape cannot be changed. Use `adios2_variable_dims` if you intend to change any of these later.

The example in `source/fortran/shapes/global-array-fixed-write.F90` and `global-array-fixed-read.F90` is similar but each process holds a (random) size of array between 2 and 5 and the global shape has to be calculated that involves an MPI Allgather operation.

Note: For an N-dimensional arrays, the function is the same, just the value of *ndim* is N, and the shape, start and count arrays must have N elements.

12.1.2 Global array that changes

If you don't know the size of the local arrays at the point of the definition, you can set this information later (before calling Put).

```
! Change the shape and decomposition information
call adios2_set_shape(var_g, 1, changing_shape, ierr)
call adios2_set_selection(var_g, 1, changing_start, changing_count, ierr);
```

Note: The global shape of the array is a single global piece of information about the array (in one output step), which can only change between steps. Technically, one can call adios2_set_shape multiple times but only the last call before the last adios2_put() call will count. The final shape must be set to the same shape on every process.

The example in `source/fortran/shapes/global-array-changing-shape-write.F90` shows how to write an array whose global size changes over time.

12.1.3 Multiblock

By multiblock we mean that one process contributes more than one block to the global array. Since only one definition is allowed for a variable, one needs to call `adios2_set_selection` and then `adios2_put()` for each block once.

CHAPTER

THIRTEEN

AGGREGATION

There is one basic knob to turn for improving output performance as we scale up an application. Traditional approaches to write a single file from arbitrary number of processes, or to write one file per process, are not scalable. The ADIOS BP format allows for writing the global arrays from many processes into a few files. The number of files should be tuned to the capability of the underlying file system, not to the number of processes.

You can see that the *.bp* output is actually a directory. So when we say ADIOS file, we actually mean a directory, which contains one or more data files (data. *NNN*), a metadata file (md.0) and an index table (md.idx).

The BP engine parameter `NumAggregators` is an absolute number to choose the desired number of files (if the number of processes is smaller then this number, then the number of processes will apply). The parameter `AggregatorRatio` is an integer value to divide the number of processes. The default ratio is one file per shared-memory-domain (i.e. per compute node), which is a good setting for most large scale supercomputers but may be underperforming at small scale if the file system could deal with more files at once.

```
! Fortran
call adios2_set_parameter(io, "NumAggregators", "100", ierr)
```

```
// C
adios2_set_parameter(io,  "NumAggregators", "100");
```

```
// C++
io.SetParameter("NumAggregators", "100");
```

```
<!-- XML file to be used in ADIOS2 constructor/init -->

<?xml version="1.0"?>
<adios-config>
    <io name="output">
        <engine type="FileStream">
            <parameter key="NumAggregators" value="100"/>
        </engine>
    </io>
</adios-config>
```

CHAPTER

FOURTEEN

INDICES AND TABLES

- genindex
- modindex
- search