Conduit Documentation

Release 0.8.1

LLNS

Jan 25, 2022

Contents

Conduit: Simplified Data Exchange for HPC Simulations

CHAPTER 1

Introduction

Conduit is an open source project from Lawrence Livermore National Laboratory that provides an intuitive model for describing hierarchical scientific data in C++, C, Fortran, and Python. It is used for data coupling between packages in-core, serialization, and I/O tasks.

Conduit's Core API provides:

- A flexible way to describe hierarchal data:

  A JSON-inspired data model for describing hierarchical in-core scientific data.

- A sane API to access hierarchal data:

  A dynamic API for rapid construction and consumption of hierarchical objects.

Conduit is under active development and targets Linux, OSX, and Windows platforms. The C++ API underpins the other language APIs and currently has the most features. We are still filling out the C, Fortran, and Python APIs.

Describing and sharing computational simulation meshes are very important use cases of Conduit. The `Mesh Blueprint` facilitates this. For more details, please see the *Mesh Blueprint Docs and Examples*.

For more background on Conduit, please see *Presentations and Publications*.

CHAPTER 2

Getting Started

To get started building and using Conduit, see the *Quick Start Guide* and the Conduit Tutorials for *C++* and *Python*. For more details about building Conduit see the *Building documentation*.

CHAPTER 3

Unique Features

Conduit was built around the concept that an intuitive in-core data description capability simplifies many other common tasks in the HPC simulation eco-system. To this aim, Conduit's Core API:

- Provides a runtime focused in-core data description API that does not require repacking or code generation.
- Supports a mix of externally owned and Conduit allocated memory semantics.

CHAPTER 4

Projects Using Conduit

Conduit is used in VisIt, ALPINE Ascent, MFEM, and Axom.

CHAPTER 5

Conduit Project Resources

Online Documentation

http://software.llnl.gov/conduit/

Github Source Repo

https://github.com/llnl/conduit

Issue Tracker

https://github.com/llnl/conduit/issues

CHAPTER 6

Conduit Libraries

The *conduit* library provides Conduit's core data API. The *relay* and *blueprint* libraries provide higher-level services built on top of the core API.

6.1 conduit

- Provides Conduit's Core API in C++ and subsets of Core API in Python, C, and Fortran.
- *Optionally depends on Fortran and Python with NumPy*

6.2 relay

- Provides:
  - I/O functionally beyond simple binary, memory mapped, and json-based text file I/O.
  - A light-weight web server for REST and WebSocket clients.
  - Interfaces for MPI communication using conduit::Node instances as payloads.
- *Optionally depends on silo, hdf5, szip, adios, and mpi*

6.3 blueprint

- Provides interfaces for common higher-level conventions and data exchange protocols (eg. describing a "mesh") using Conduit.
- *No optional dependencies*

See the *User Documentation* for more details on these libraries.

CHAPTER 7

Contributors

- Cyrus Harrison (LLNL)
- Brian Ryujin (LLNL)
- Adam Kunen (LLNL)
- Joe Ciurej (LLNL)
- Kathleen Biagas (LLNL)
- Eric Brugger (LLNL)
- Aaron Black (LLNL)
- George Zagaris (LLNL)
- Kenny Weiss (LLNL)
- Matt Larsen (LLNL)
- Markus Salasoo (LLNL)
- Rebecca Haluska (LLNL)
- Arlie Capps (LLNL)
- Mark Miller (LLNL)
- Todd Gamblin (LLNL)
- Kevin Huynh (LLNL)
- Brad Whitlock (Intelligent Light)
- Chris Laganella (Intelligent Light)
- George Aspesi (Harvey Mudd)
- Justin Bai (Harvey Mudd)
- Rupert Deese (Harvey Mudd)
- Linnea Shin (Harvey Mudd)

In 2014 and 2015 LLNL sponsored a Harvey Mudd Computer Science Clinic project focused on using Conduit in HPC Proxy apps. You can read about more details about the clinic project from this LLNL article: http://computation.llnl.gov/newsroom/hpc-partnership-harvey-mudd-college-and-livermore

CHAPTER 8

Conduit Documentation

8.1 Quick Start

8.1.1 Installing Conduit and Third Party Dependencies

The quickest path to install conduit and its dependencies is via *uberenv*:

```
git clone --recursive https://github.com/llnl/conduit.git
cd conduit
python scripts/uberenv/uberenv.py --install --prefix="build"
```

After this completes, `build/conduit-install` will contain a Conduit install.

For more details about building and installing Conduit see *Building*. This page provides detailed info about Conduit's CMake options, *uberenv* and *Spack* support. We also provide info about *building for known HPC clusters using uberenv* and a *Docker example* that leverages Spack.

8.1.2 Installing Conduit with pip

If you want to use Conduit primarily in Python, another option is to build Conduit with pip. This assumes you have CMake in your path.

Basic Install:

```
git clone --recursive https://github.com/llnl/conduit.git
cd conduit
pip install . --user
```

If you have a system MPI and an existing HDF5 install you can add those to the build using environment variables.

Install with HDF5 and MPI:

```
git clone --recursive https://github.com/llnl/conduit.git
cd conduit
env ENABLE_MPI=ON HDF5_DIR={path/to/hdf5_dir} pip install . --user
```

See *Pip Install Docs* for more details.

8.1.3 Using Conduit in Your Project

The install includes examples that demonstrate how to use Conduit in a CMake-based build system and via a Makefile.

CMake-based build system example (see: `examples/conduit/using-with-cmake`):

```
###############################################################################
#
# Example that shows how to use an installed instance of Conduit in another
# CMake-based build system.
#
# To build:
#  mkdir build
#  cd build
#  cmake -DCONDUIT_DIR={conduit install path} ../
#  make
#  ./conduit_example
#
#
# If run in sub directory of a conduit install,
# CONDUIT_DIR will default to ../../..
#
#  mkdir build
#  cd build
#  cmake ..
#  make
#  ./conduit_example
#
###############################################################################

cmake_minimum_required(VERSION 3.0)

project(using_with_cmake)


############################################################
# Option 1: import conduit using an explicit path (recommended)
############################################################

#
# Provide default CONDUIT_DIR that works in a conduit install
#
if(NOT CONDUIT_DIR)
    set(CONDUIT_DIR "../../..")
endif()

#
# Check for valid conduit install
#
```

(continues on next page)

(continued from previous page)

```
# if given relative path, resolve
get_filename_component(CONDUIT_DIR ${CONDUIT_DIR} ABSOLUTE)
# check for expected cmake exported target files
if(NOT EXISTS ${CONDUIT_DIR}/lib/cmake/conduit/ConduitConfig.cmake)
    message(FATAL_ERROR "Could not find Conduit CMake include file (${CONDUIT_DIR}/
↪lib/cmake/conduit/ConduitConfig.cmake)")
endif()

#
# Use CMake's find_package to import conduit's targets
# using explicit path
#
find_package(Conduit REQUIRED
             NO_DEFAULT_PATH
             PATHS ${CONDUIT_DIR}/lib/cmake/conduit)


##############################################################
# Option 2: import conduit using find_package search
##############################################################
##
## Add Conduit's install path to CMAKE_PREFIX_PATH
## and use following find_package call to import conduit's targets
##
#
# find_package(Conduit REQUIRED)
#


######
# If Conduit was built with c++11 support, make sure we enable it
######
if(CONDUIT_USE_CXX11)
    set(CMAKE_CXX_STANDARD 11)
    set(CMAKE_CXX_STANDARD_REQUIRED ON)
endif()

#####################################################
# create our example
#####################################################
add_executable(conduit_example conduit_example.cpp)

# link to conduit targets
target_link_libraries(conduit_example conduit::conduit)

# if you are using conduit's python CAPI capsule interface
# target_link_libraries(conduit_example conduit::conduit_python)


# if you are using conduit's mpi features
# if(NOT MPI_FOUND)
#    find_package(MPI COMPONENTS CXX)
# endif()
# target_link_libraries(conduit_example conduit::conduit_mpi)
```

Makefile-based build system example (see: `examples/conduit/using-with-make`):

```
##############################################################################
#
# Example that shows how to use an installed instance of Conduit in Makefile
# based build system.
#
# To build:
#  make CONDUIT_DIR={conduit install path}
#  ./conduit_example
#
# From within a conduit install:
#  make
#  ./conduit_example
#
# Which corresponds to:
#
#  make CONDUIT_DIR=../../../
#  ./conduit_example
#
##############################################################################

CONDUIT_DIR ?= ../../..

# See $(CONDUIT_DIR)/share/conduit/conduit_config.mk for detailed linking info
include $(CONDUIT_DIR)/share/conduit/conduit_config.mk

# If Conduit was built with c++11 support, make sure we enable it
CXX_FLAGS = $(if $(CONDUIT_USE_CXX11),-std=c++11)
INC_FLAGS = $(CONDUIT_INCLUDE_FLAGS)
LNK_FLAGS = $(CONDUIT_LINK_RPATH) $(CONDUIT_LIB_FLAGS)

main:
        $(CXX) $(CXX_FLAGS) $(INC_FLAGS) conduit_example.cpp $(LNK_FLAGS) -o conduit_
↪example

clean:
        rm -f conduit_example
```

8.1.4 Learning Conduit

To get starting learning the core Conduit API, see the Conduit Tutorials for *C++* and *Python*.

8.2 User Documentation

8.2.1 Conduit

C++ Tutorial

This short tutorial provides C++ examples that demonstrate the Conduit's Core API. Conduit's unit tests (*src/tests/{library_name}/*) also provide a rich set of examples for Conduit's Core API and additional libraries. Ascent's Tutorial also provides a brief intro to Conduit basics with C++ and Python examples.

Basic Concepts

Node basics

The *Node* class is the primary object in conduit.

Think of it as a hierarchical variant object.

```
Node n;
n["my"] = "data";
n.print();
```

```
my: "data"
```

The *Node* class supports hierarchical construction.

```
Node n;
n["my"] = "data";
n["a/b/c"] = "d";
n["a"]["b"]["e"] = 64.0;
n.print();

std::cout << "total bytes: " << n.total_strided_bytes() << std::endl;
```

```
my: "data"
a:
  b:
    c: "d"
    e: 64.0

total bytes: 15
```

Borrowing form JSON (and other similar notations), collections of named nodes are called *Objects* and collections of unnamed nodes are called *Lists*, all other types are leaves that represent concrete data.

```
Node n;
n["object_example/val1"] = "data";
n["object_example/val2"] = 10u;
n["object_example/val3"] = 3.1415;

for(int i = 0; i < 5 ; i++ )
{
    Node &list_entry = n["list_example"].append();
    list_entry.set(i);
}

n.print();
```

```
object_example:
  val1: "data"
  val2: 10
  val3: 3.1415
```

(continues on next page)

(continued from previous page)

```
list_example:
  - 0
  - 1
  - 2
  - 3
  - 4
```

You can use a *NodeIterator* (or a *NodeConstIterator*) to iterate through a Node's children.

```
Node n;
n["object_example/val1"] = "data";
n["object_example/val2"] = 10u;
n["object_example/val3"] = 3.1415;

for(int i = 0; i < 5 ; i++ )
{
    Node &list_entry = n["list_example"].append();
    list_entry.set(i);
}

n.print();

NodeIterator itr = n["object_example"].children();
while(itr.has_next())
{
    Node &cld = itr.next();
    std::string cld_name = itr.name();
    std::cout << cld_name << ": " << cld.to_string() << std::endl;
}

std::cout << std::endl;

itr = n["list_example"].children();
while(itr.has_next())
{
    Node &cld = itr.next();
    std::cout << cld.to_string() << std::endl;
}
```

```
object_example:
  val1: "data"
  val2: 10
  val3: 3.1415
list_example:
  - 0
  - 1
  - 2
  - 3
  - 4

val1: "data"
val2: 10
val3: 3.1415
```

(continues on next page)

(continued from previous page)

```
0
1
2
3
4
```

Behind the scenes, *Node* instances manage a collection of memory spaces.

```
Node n;
n["my"] = "data";
n["a/b/c"] = "d";
n["a"]["b"]["e"] = 64.0;

Node ninfo;
n.info(ninfo);
ninfo.print();
```

```
mem_spaces:
  0x7fcad7c04c00:
    path: "my"
    type: "allocated"
    bytes: 5
  0x7fcad7c04440:
    path: "a/b/c"
    type: "allocated"
    bytes: 2
  0x7fcad7c04430:
    path: "a/b/e"
    type: "allocated"
    bytes: 8
total_bytes_allocated: 15
total_bytes_mmaped: 0
total_bytes_compact: 15
total_strided_bytes: 15
```

There is no absolute path construct, all paths are fetched relative to the current node (a leading `/` is ignored when fetching). Empty paths names are also ignored, fetching `a///b` is equalvalent to fetching `a/b`.

Bitwidth Style Types

When sharing data in scientific codes, knowing the precision of the underlining types is very important.

Conduit uses well defined bitwidth style types (inspired by NumPy) for leaf values.

```
Node n;
uint32 val = 100;
n["test"] = val;
n.print();
n.print_detailed();
```

```
test: 100
```

(continues on next page)

(continued from previous page)

```
{
  "test":
{
"dtype":"uint32",
"number_of_elements": 1,
"offset": 0,
"stride": 4,
"element_bytes": 4,
"endianness": "little"
, "value": 100}
}
```

Standard C++ numeric types will be mapped by the compiler to bitwidth style types.

```
Node n;
int val = 100;
n["test"] = val;
n.print_detailed();
```

```
{
  "test":
{
"dtype":"int32",
"number_of_elements": 1,
"offset": 0,
"stride": 4,
"element_bytes": 4,
"endianness": "little"
, "value": 100}
}
```

Supported Bitwidth Style Types:

- signed integers: int8,int16,int32,int64
- unsigned integers: uint8,uint16,uint32,uint64
- floating point numbers: float32,float64

Conduit provides these types by constructing a mapping for the current platform the from the following types:

- char, short, int, long, long long, float, double, long double

When C++11 support is enabled, Conduit's bitwidth style types will match the C++11 standard bitwidth types defined in `<cstdint>`.

When a **set** method is called on a leaf Node, if the data passed to the **set** is compatible with the Node's Schema the data is simply copied.

Compatible Schemas

When passed a compatible Node, Node methods `update` and `update_compatible` allow you to copy data into Node or extend a Node with new data without changing existing allocations.

Schemas do not need to be identical to be compatible.

You can check if a Schema is compatible with another Schema using the **Schema::compatible(Schema &test)** method. Here is the criteria for checking if two Schemas are compatible:

- **If the calling Schema describes an Object** : The passed test Schema must describe an Object and the test Schema's children must be compatible with the calling Schema's children that have the same name.
- **If the calling Schema describes a List**: The passed test Schema must describe a List, the calling Schema must have at least as many children as the test Schema, and when compared in list order each of the test Schema's children must be compatible with the calling Schema's children.
- **If the calling Schema describes a leaf data type**: The calling Schema's and test Schema's **dtype().id()** and **dtype().element_bytes()** must match, and the calling Schema **dtype().number_of_elements()** must be greater than or equal than the test Schema's.

Node References

For most uses cases in C++, Node references are the best solution to build and manipulate trees. They allow you to avoid expensive copies and pass around sub-trees without worrying about valid pointers.

```
//////////////////////////////////////////////////////////
// In C++, use Node references!
//////////////////////////////////////////////////////////
// Using Node references is common (good) pattern!

// setup a node
Node root;
// set data in hierarchy
root["my/nested/path"] = 0.0;
// display the contents
root.print();

// Get a ref to the node in the tree
Node &data = root["my/nested/path"];
// change the value
data = 42.0;
// display the contents
root.print();
```

```
my:
  nested:
    path: 0.0


my:
  nested:
    path: 42.0
```

In C++ the Node assignment operator that takes a Node input is really an alias to set. That is, if follows set (deep copy) semantics.

```
//////////////////////////////////////////////////////////
// C++ anti-pattern to avoid: copy instead of reference
//////////////////////////////////////////////////////////
```

(continues on next page)

(continued from previous page)

```
// setup a node
Node root;
// set data in hierarchy
root["my/nested/path"] = 0.0;

// display the contents
root.print();

// In this case, notice we aren't using a reference.
// This creates a copy, disconnected from the orignal tree!
// This is probably not what you are looking for ...
Node data = root["my/nested/path"];
// change the value
data = 42.0;

// display the contents
root.print();
```

```
my:
  nested:
    path: 0.0


my:
  nested:
    path: 0.0
```

const Nodes

If you aren't careful, the ability to easily create dynamic trees can also undermine your process to consume them. For example, asking for an expected but non-existent path will return a reference to an empty Node. Surprise!

Methods like *fetch_existing* allow you to be more explicit when asking for expected data. In C++, `const Node` references are also common way to process trees in an read-only fashion. `const` methods will not modify the tree structure, so if you ask for a non-existent path, you will receive an error instead of reference to an empty Node.

```
// with non-const references, you can modify the node,
// leading to surprises in cases were read-only
// validation and processing is intended
void important_suprise(Node &data)
{
    // if this doesn't exist, we will get a new empty node
    // Note: we could also ask if the path exists via Node:has_path()
    int val = data["my/important/data"].to_int();
    std::cout << "\n==> important: " << val << std::endl;
}

// with const references,  the api provides checks
// that help
void important(const Node &data)
{
    // if this doesn't exist, const access will trigger exception here
```

(continues on next page)

(continued from previous page)

```
    // Note: we could also ask if the path exists via Node:has_path()
    int val = data["my/important/data"].to_int();
    std::cout << "\n==> important: " << val << std::endl;
}
```

```
//////////////////////////////////////////////////////////
// In C++, leverage const refs for processing existing nodes
//////////////////////////////////////////////////////////

// setup a node
Node n1;
n1["my/important/but/mistyped/path/to/data"] = 42.0;

std::cout << "== n1 == " << std::endl;
n1.print();

// method with non-const arg drives on ...
try
{
    important_suprise(n1);
}
catch(conduit::Error &e)
{
    e.print();
}

// check n1, was it was modified ( yes ... )
std::cout << "n1 after calling `important_suprise`" << std::endl;
n1.print();

Node n2;
n2["my/important/but/mistyped/path/to/data"] = 42.0;

std::cout << "== n2 == " << std::endl;
n2.print();

// method with const arg lets us know, and also makes sure
// the node structure isn't modified
try
{
    important(n2);
}
catch(conduit::Error &e)
{
    e.print();
}

// check n2, was it was modified ( no ... )
std::cout << "n2 after calling `important`" << std::endl;
n2.print();
```

```
== n1 ==

my:
  important:
    but:
```

(continues on next page)

(continued from previous page)

```
      mistyped:
        path:
          to:
            data: 42.0


==> important: 0
n1 after calling `important_suprise`

my:
  important:
    but:
      mistyped:
        path:
          to:
            data: 42.0
    data:

== n2 ==

my:
  important:
    but:
      mistyped:
        path:
          to:
            data: 42.0


file: /Users/harrison37/Work/github/llnl/conduit/src/libs/conduit/conduit_node.cpp
line: 13050
message:
Cannot fetch non-existent child "data" from Node(my/important)

n2 after calling `important`

my:
  important:
    but:
      mistyped:
        path:
          to:
            data: 42.0
```

Accessing Numeric Data

Accessing Scalars and Arrays

You can access leaf types (numeric scalars or arrays) using Node's *as_{type}* methods.

```
Node n;
int64 val = 100;
n = val;
std::cout << n.as_int64() << std::endl;
```

```
100
```

Or you can use Node::value(), which can infer the correct return type via a cast.

```
Node n;
int64 val = 100;
n = val;
int64 my_val = n.value();
std::cout << my_val << std::endl;
```

```
100
```

Accessing array data via pointers works the same way, using Node's *as_{type}* methods.

```
int64 vals[4] = {100,200,300,400};

Node n;
n.set(vals,4);

int64 *my_vals = n.as_int64_ptr();

for(index_t i=0; i < 4; i++)
{
    std::cout << "my_vals[" << i << "] = " << my_vals[i] << std::endl;
}
```

```
my_vals[0] = 100
my_vals[1] = 200
my_vals[2] = 300
my_vals[3] = 400
```

Or using Node::value():

```
int64 vals[4] = {100,200,300,400};

Node n;
n.set(vals,4);

int64 *my_vals = n.value();

for(index_t i=0; i < 4; i++)
{
    std::cout << "my_vals[" << i << "] = " << my_vals[i] << std::endl;
}
```

```
my_vals[0] = 100
my_vals[1] = 200
my_vals[2] = 300
my_vals[3] = 400
```

For non-contiguous arrays, direct pointer access is complex due to the indexing required. Conduit provides a simple DataArray class that handles per-element indexing for all types of arrays.

```
int64 vals[4] = {100,200,300,400};

Node n;
```

(continues on next page)

(continued from previous page)

```
n.set(vals,2, // # of elements
           0, // offset in bytes
           sizeof(int64)*2); // stride in bytes

int64_array  my_vals = n.value();

for(index_t i=0; i < 2; i++)
{
    std::cout << "my_vals[" << i << "] = " << my_vals[i] << std::endl;
}

my_vals.print();
```

```
my_vals[0] = 100
my_vals[1] = 300
[100, 300]
```

C++11 Initializer Lists

When C++11 support is enabled you can set Node values using initializer lists with numeric literals.

```
Node n;

// set with integer c++11 initializer list
n.set({100,200,300});
n.print();

// assign with integer c++11 initializer list
n = {100,200,300};
n.print();

// set with floating point c++11 initializer list
n.set({1.0,2.0,3.0});
n.print();

// assign with floating point c++11 initializer list
n = {1.0,2.0,3.0};
n.print();
```

```
[100, 200, 300]
[100, 200, 300]
[1.0, 2.0, 3.0]
[1.0, 2.0, 3.0]
```

Using Introspection and Conversion

In this example, we have an array in a node that we are interested in processing using an existing function that only handles doubles. We ensure the node is compatible with the function, or transform it to a contiguous double array.

```
//-----------------------------------------------------------------------------
void must_have_doubles_function(double *vals,index_t num_vals)
{
```

(continues on next page)

(continued from previous page)

```
    for(int i = 0; i < num_vals; i++)
    {
        std::cout << "vals[" << i << "] = " <<  vals[i] << std::endl;
    }
}

//-----------------------------------------------------------------------------
void process_doubles(Node & n)
{
    Node res;
    // We have a node that we are interested in processing with
    // and existing function that only handles doubles.

    if( n.dtype().is_double() && n.dtype().is_compact() )
    {
        std::cout << " using existing buffer" << std::endl;

        // we already have a contiguous double array
        res.set_external(n);
    }
    else
    {
        std::cout << " converting to temporary double array " << std::endl;

        // Create a compact double array with the values of the input.
        // Standard casts are used to convert each source element to
        // a double in the new array.
        n.to_double_array(res);
    }

    res.print();

    double *dbl_vals = res.value();
    index_t num_vals = res.dtype().number_of_elements();
    must_have_doubles_function(dbl_vals,num_vals);
}

//-----------------------------------------------------------------------------
TEST(conduit_tutorial, numeric_double_conversion)
{

    float32 f32_vals[4] = {100.0,200.0,300.0,400.0};
    double  d_vals[4]   = {1000.0,2000.0,3000.0,4000.0};

    Node n;
    n["float32_vals"].set(f32_vals,4);
    n["double_vals"].set(d_vals,4);

    std::cout << "float32 case: " << std::endl;

    process_doubles(n["float32_vals"]);

    std::cout << "double case: " << std::endl;

    process_doubles(n["double_vals"]);
}
```

```
[       OK ] conduit_tutorial.numeric_double_conversion_start (0 ms)
[ RUN      ] conduit_tutorial.numeric_double_conversion
float32 case:
 converting to temporary double array
[100.0, 200.0, 300.0, 400.0]
vals[0] = 100
vals[1] = 200
vals[2] = 300
vals[3] = 400
double case:
 using existing buffer
[1000.0, 2000.0, 3000.0, 4000.0]
vals[0] = 1000
vals[1] = 2000
vals[2] = 3000
vals[3] = 4000
[       OK ] conduit_tutorial.numeric_double_conversion (0 ms)
[ RUN      ] conduit_tutorial.numeric_double_conversion_end
```

Reading YAML and JSON Strings

Parsing text with *Node::parse()*

Node::parse() parses YAML and JSON strings into a *Node* tree.

```
std::string yaml_txt("mykey: 42.0");

Node n;
n.parse(yaml_txt,"yaml");

std::cout << n["mykey"].as_float64() <<std::endl;

n.print_detailed();
```

```
42

{
  "mykey":
{
"dtype":"float64",
"number_of_elements": 1,
"offset": 0,
"stride": 8,
"element_bytes": 8,
"endianness": "little"
, "value": 42.0}
}
```

```
std::string json_txt("{\"mykey\": 42.0}");

Node n;
n.parse(json_txt,"json");

std::cout << n["mykey"].as_float64() <<std::endl;
```

(continues on next page)

(continued from previous page)

```
n.print_detailed();
```

```
42

{
  "mykey":
{
"dtype":"float64",
"number_of_elements": 1,
"offset": 0,
"stride": 8,
"element_bytes": 8,
"endianness": "little"
, "value": 42.0}
}
```

The first argument is the string to parse and the second argument selects the protocol to use when parsing.

Valid Protocols: `json`, `conduit_json`, `conduit_base64_json`, `yaml`.

- `json` and `yaml` protocols parse pure JSON or YAML strings. For leaf nodes wide types such as *int64*, *uint64*, and *float64* are inferred.

Homogeneous numeric lists are parsed as Conduit arrays.

```
std::string yaml_txt("myarray: [0.0, 10.0, 20.0, 30.0]");

Node n;
n.parse(yaml_txt,"yaml");

n["myarray"].print();

n.print_detailed();
```

```
[0.0, 10.0, 20.0, 30.0]

{
  "myarray":
{
"dtype":"float64",
"number_of_elements": 4,
"offset": 0,
"stride": 8,
"element_bytes": 8,
"endianness": "little"
, "value": [0.0, 10.0, 20.0, 30.0]}
}
```

- `conduit_json` parses JSON with conduit data type information, allowing you to specify bitwidth style types, strides, etc.
- `conduit_base64_json` combines the *conduit_json* protocol with an embedded base64-encoded data block

Generators

Using *Generator* instances

Node::parse() is sufficient for most use cases, but you can also use a *Generator* instance to parse JSON and YAML. Additionally, Generators can parse a conduit JSON schema and bind it to in-core data.

```
Generator g("{test: {dtype: float64, value: 100.0}}","conduit_json");

Node n;
g.walk(n);

std::cout << n["test"].as_float64() <<std::endl;
n.print();
n.print_detailed();
```

```
100

test: 100.0


{
  "test":
{
"dtype":"float64",
"number_of_elements": 1,
"offset": 0,
"stride": 8,
"element_bytes": 8,
"endianness": "little"
, "value": 100.0}
}
```

Like *Node::parse()*, *Generators* can also parse pure JSON or YAML. For leaf nodes: wide types such as *int64*, *uint64*, and *float64* are inferred.

```
Generator g("{test: 100.0}","json");

Node n;
g.walk(n);

std::cout << n["test"].as_float64() <<std::endl;
n.print_detailed();
n.print();
```

```
100

{
  "test":
{
"dtype":"float64",
"number_of_elements": 1,
"offset": 0,
"stride": 8,
"element_bytes": 8,
"endianness": "little"
, "value": 100.0}
}
```

(continues on next page)

(continued from previous page)

```
test: 100.0
```

```
Generator g("test: 100.0","yaml");

Node n;
g.walk(n);

std::cout << n["test"].as_float64() <<std::endl;
n.print_detailed();
n.print();
```

```
100

{
  "test":
{
"dtype":"float64",
"number_of_elements": 1,
"offset": 0,
"stride": 8,
"element_bytes": 8,
"endianness": "little"
, "value": 100.0}
}

test: 100.0
```

Schemas can be bound to in-core data.

```
float64 vals[2];
Generator g("{a: {dtype: float64, value: 100.0}, b: {dtype: float64, value: 200.0} }",
            "conduit_json",
            vals);

Node n;
g.walk_external(n);

std::cout << n["a"].as_float64() << " vs " << vals[0] << std::endl;
std::cout << n["b"].as_float64() << " vs " << vals[1] << std::endl;

n.print();

Node ninfo;
n.info(ninfo);
ninfo.print();
```

```
100 vs 100
200 vs 200

a: 100.0
b: 200.0
```

(continues on next page)

(continued from previous page)

```
mem_spaces:
  0x7ffeebcfa130:
    path: "a"
    type: "external"
total_bytes_allocated: 0
total_bytes_mmaped: 0
total_bytes_compact: 16
total_strided_bytes: 16
```

Compacting Nodes

Nodes can be compacted to transform sparse data.

```
float64 vals[] = { 100.0,-100.0,
                   200.0,-200.0,
                   300.0,-300.0,
                   400.0,-400.0,
                   500.0,-500.0};

// stride though the data with two different views.
Generator g1("{dtype: float64, length: 5, stride: 16}",
             "conduit_json",
             vals);
Generator g2("{dtype: float64, length: 5, stride: 16, offset:8}",
             "conduit_json",
              vals);

Node n1;
g1.walk_external(n1);
n1.print();

Node n2;
g2.walk_external(n2);
n2.print();

// look at the memory space info for our two views
Node ninfo;
n1.info(ninfo);
ninfo.print();

n2.info(ninfo);
ninfo.print();

// compact data from n1 to a new node
Node n1c;
n1.compact_to(n1c);

// look at the resulting compact data
n1c.print();
n1c.schema().print();
n1c.info(ninfo);
ninfo.print();

// compact data from n2 to a new node
```

(continues on next page)

(continued from previous page)

```
Node n2c;
n2.compact_to(n2c);

// look at the resulting compact data
n2c.print();
n2c.info(ninfo);
ninfo.print();
```

```
[100.0, 200.0, 300.0, 400.0, 500.0]
[-100.0, -200.0, -300.0, -400.0, -500.0]

mem_spaces:
  0x7ffeebcfa0f0:
    path: ""
    type: "external"
total_bytes_allocated: 0
total_bytes_mmaped: 0
total_bytes_compact: 40
total_strided_bytes: 72


mem_spaces:
  0x7ffeebcfa0f0:
    path: ""
    type: "external"
total_bytes_allocated: 0
total_bytes_mmaped: 0
total_bytes_compact: 40
total_strided_bytes: 72

[100.0, 200.0, 300.0, 400.0, 500.0]
{"dtype":"float64","number_of_elements": 5,"offset": 0,"stride": 8,"element_bytes": 8,
↪"endianness": "little"}

mem_spaces:
  0x7f9510404140:
    path: ""
    type: "allocated"
    bytes: 40
total_bytes_allocated: 40
total_bytes_mmaped: 0
total_bytes_compact: 40
total_strided_bytes: 40

[-100.0, -200.0, -300.0, -400.0, -500.0]

mem_spaces:
  0x7f95104040c0:
    path: ""
    type: "allocated"
    bytes: 40
total_bytes_allocated: 40
total_bytes_mmaped: 0
total_bytes_compact: 40
total_strided_bytes: 40
```

Data Ownership

The *Node* class provides two ways to hold data, the data is either **owned** or **externally described**:

- If a *Node* **owns** data, the *Node* allocated the memory holding the data and is responsible or deallocating it.
- If a *Node* **externally describes** data, the *Node* holds a pointer to the memory where the data resides and is not responsible for deallocating it.

set vs *set_external*

The **Node::set** methods support creating **owned** data and copying data values in both the **owned** and **externally described** cases. The **Node::set_external** methods allow you to create **externally described** data:

- **set(...)**: Makes a copy of the data passed into the *Node*. This will trigger an allocation if the current data type of the *Node* is incompatible with what was passed. The *Node* assignment operators use their respective **set** variants, so they follow the same copy semantics.
- **set_external(...)**: Sets up the *Node* to describe data passed and access the data externally. Does not copy the data.

```
int vsize = 5;
std::vector<float64> vals(vsize,0.0);
for(int i=0;i<vsize;i++)
{
    vals[i] = 3.1415 * i;
}

Node n;
n["v_owned"] = vals;
n["v_external"].set_external(vals);

n.info().print();

n.print();

vals[1] = -1 * vals[1];
n.print();
```

```
mem_spaces:
  0x7f9779501180:
    path: "v_owned"
    type: "allocated"
    bytes: 40
  0x7f9779500dc0:
    path: "v_external"
    type: "external"
total_bytes_allocated: 40
total_bytes_mmaped: 0
total_bytes_compact: 80
total_strided_bytes: 80


v_owned: [0.0, 3.1415, 6.283, 9.4245, 12.566]
v_external: [0.0, 3.1415, 6.283, 9.4245, 12.566]
```

(continues on next page)

(continued from previous page)

```
v_owned: [0.0, 3.1415, 6.283, 9.4245, 12.566]
v_external: [0.0, -3.1415, 6.283, 9.4245, 12.566]
```

Node Update Methods

The *Node* class provides three **update** methods which allow you to easily copy data or the description of data from a source node.

- **Node::update(Node &source)**:

  This method behaves similar to a python dictionary update. Entires from the source Node are copied into the calling Node, here are more concrete details:

  - **If the source describes an Object**:
    - Update copies the children of the source Node into the calling Node. Normal set semantics apply: if a compatible child with the same name already exists in the calling Node, the data will be copied. If not, the calling Node will dynamically construct children to hold copies of each child of the source Node.
  - **If the source describes a List**:
    - Update copies the children of the source Node into the calling Node. Normal set semantics apply: if a compatible child already exists in the same list order in the calling Node, the data will be copied. If not, the calling Node will dynamically construct children to hold copies of each child of the source Node.
  - **If the source Node describes a leaf data type**:
    - Update works exactly like a **set** (not true yet).

- **Node::update_compatible(Node &source)**:

  This method copies data from the children in the source Node that are compatible with children in the calling node. No changes are made where children are incompatible.

- **Node::update_external(Node &source)**:

  This method creates children in the calling Node that externally describe the children in the source node. It differs from **Node::set_external(Node &source)** in that **set_external()** will clear the calling Node so it exactly match an external description of the source Node, whereas **update_external()** will only change the children in the calling Node that correspond to children in the source Node.

String Formatting Helpers

fmt

For C++ users, conduit includes a built-in version of the `fmt` library (https://fmt.dev/). Since other projects also bundle `fmt`, the conduit version is modified to place everything in the `conduit_fmt` namespace instead of the default `fmt` namespace. This is a safe approach to avoid potential confusion and static linking consequences.

When using conduit in C++, you can use its built-in `fmt` as follows:

```
// conduit_fmt is installed along with conduit
#include "conduit_fmt/conduit_fmt.h"

// fmt features are in the conduit_fmt namespace
std::string res = conduit_fmt::format("The answer is {}.", 42);
std::cout << res << std::endl;

res = conduit_fmt::format("The answer is {answer:0.4f}.",
                          conduit_fmt::arg("answer",3.1415));
std::cout << res << std::endl;
```

```
The answer is 42.
The answer is 3.1415.
```

conduit::utils::format

In addition to direct `fmt` support, conduit utils provides `conduit::utils::format` methods that enable `fmt` style string formatting with the arguments are passed as a `conduit::Node` tree. These simplify use cases such as generating path string, allowing the pattern string and arguments to be stored as part of a conduit hierarchy (and in HDF5, YAML, etc files). This feature is also available in Conduit's Python API (`conduit.utils.format`).

conduit::utils::format(string, args)

The `args` case allows named arguments (args passed as object) or ordered args (args passed as list).

conduit::utils::format(string, args) – object case:

```
// conduit::utils::format w/ args + object
// processes named args passed via a conduit Node
Node args;
args["answer"] = 42;
std::string res = conduit::utils::format("The answer is {answer:04}.",
                                         args);
std::cout << res << std::endl;

args.reset();
args["adjective"] =  "other";
args["answer"] = 3.1415;

res = conduit::utils::format("The {adjective} answer is {answer:0.4f}.",
                             args);

std::cout << res << std::endl;
```

```
The answer is 0042.
The other answer is 3.1415.
```

conduit::utils::format(string, args) – list case:

```
// conduit::utils::format w/ args + list
// processes ordered args passed via a conduit Node
Node args;
args.append() = 42;
```

(continues on next page)

(continued from previous page)

```
std::string res = conduit::utils::format("The answer is {}.", args);
std::cout << res << std::endl;

args.reset();
args.append() = "other";
args.append() = 3.1415;

res = conduit::utils::format("The {} answer is {:0.4f}.", args);

std::cout << res << std::endl;
```

```
The answer is 42.
The other answer is 3.1415.
```

conduit::utils::format(string, maps, map_index)

The `maps` case also supports named or ordered args and works in conjunction with a `map_index`. The `map_index` is used to fetch a value from an array, or list of strings, which is then passed to fmt. The `maps` style of indexed indirection supports generating path strings for non-trivial domain partition mappings in Blueprint.

conduit::utils::format(string, maps, map_index) – object case:

```
// conduit::utils::format w/ maps + object
// processing named args passed via a conduit Node, indexed by map_index
Node maps;
maps["answer"].set({ 42.0, 3.1415});

std::string res = conduit::utils::format("The answer is {answer:04}.",
                                         maps, 0);
std::cout << res << std::endl;

res = conduit::utils::format("The answer is {answer:04}.", maps, 1);
std::cout << res << std::endl << std::endl;


maps.reset();
maps["answer"].set({ 42.0, 3.1415});
Node &slist = maps["position"];
slist.append() = "first";
slist.append() = "second";


res = conduit::utils::format("The {position} answer is {answer:0.4f}.",
                             maps, 0);

std::cout << res << std::endl;

res = conduit::utils::format("The {position} answer is {answer:0.4f}.",
                             maps, 1);

std::cout << res << std::endl;
```

```
The answer is 0042.
The answer is 3.1415.
```

(continues on next page)

(continued from previous page)

```
The first answer is 42.0000.
The second answer is 3.1415.
```

conduit::utils::format(string, maps, map_index) – list case:

```
// conduit::utils::format w/ maps + list
// processing ordered args passed via a conduit Node, indexed by map_index
Node maps;
maps.append() = { 42.0, 3.1415};
std::string res = conduit::utils::format("The answer is {}.",
                                         maps, 0);
std::cout << res << std::endl;

res = conduit::utils::format("The answer is {}.", maps, 1);
std::cout << res << std::endl << std::endl;

maps.reset();

// first arg
Node &slist = maps.append();
slist.append() = "first";
slist.append() = "second";

// second arg
maps.append() = { 42.0, 3.1415};

res = conduit::utils::format("The {} answer is {:0.4f}.", maps, 0);
std::cout << res << std::endl;

res = conduit::utils::format("The {} answer is {:0.4f}.", maps, 1);
std::cout << res << std::endl;
```

```
The answer is 42.
The answer is 3.1415.

The first answer is 42.0000.
The second answer is 3.1415.
```

Error Handling

Conduit's APIs emit three types of messages for logging and error handling:

Message Type	Description
Info	General Information
Warning	Recoverable Error
Error	Fatal Error

Default Error Handlers

Conduit provides a default handler for each message type:

Message Type	Default Action
Info	Prints the message to standard out
Warning	Throws a C++ Exception (conduit::Error instance)
Error	Throws a C++ Exception (conduit::Error instance)

Using Custom Error Handlers

The conduit::utils namespace provides functions to override each of the three default handlers with a method that provides the following signature:

```
void my_handler(const std::string &msg,
                const std::string &file,
                int line)
{
  // your handling code here ...
}

conduit::utils::set_error_handler(my_handler);
```

Here is an example that re-wires all three error handlers to print to standard out:

```
//-----------------------------------------------------------------------------
void my_info_handler(const std::string &msg,
                     const std::string &file,
                     int line)
{
    std::cout << "[INFO] " << msg << std::endl;
}

void my_warning_handler(const std::string &msg,
                        const std::string &file,
                        int line)
{
    std::cout << "[WARNING!] " << msg << std::endl;
}

void my_error_handler(const std::string &msg,
                      const std::string &file,
                      int line)
{
    std::cout << "[ERROR!] " << msg << std::endl;
    // errors are considered fatal, aborting or unwinding the
    // call stack with an exception are the only viable options
    throw conduit::Error(msg,file,line);
}
```

```
// rewire error handlers
conduit::utils::set_info_handler(my_info_handler);
conduit::utils::set_warning_handler(my_warning_handler);
conduit::utils::set_error_handler(my_error_handler);

// emit an example info message
CONDUIT_INFO("An info message");

Node n;
```

(continues on next page)

(continued from previous page)

```
n["my_value"].set_float64(42.0);

// emit an example warning message

// using "as" for wrong type emits a warning, returns a default value (0.0)
float32 v = n["my_value"].as_float32();

// emit an example error message

try
{
    // fetching a non-existant path from a const Node emits an error
    const Node &n_my_value = n["my_value"];
    n_my_value["bad"];
}
catch(conduit::Error &e)
{
    // pass
}
```

```
[INFO] An info message
[WARNING!] Node::as_float32() const -- DataType float64 at path my_value does not␣
↪equal expected DataType float32
[ERROR!] Cannot fetch_existing, Node(my_value) is not an object
```

Using Restoring Default Handlers

The default handlers are part of the conduit::utils interface, so you can restore them using:

```
// restore default handlers
conduit::utils::set_info_handler(conduit::utils::default_info_handler);
conduit::utils::set_warning_handler(conduit::utils::default_warning_handler);
conduit::utils::set_error_handler(conduit::utils::default_error_handler);
```

Accessing Current Handlers

You can access the currently active handlers using the *conduit::utils::info_handler()*, *conduit::utils::warning_handler()*, and *conduit::utils::error_handler()* methods. Here is an example that shows how to save the current handlers, temporarily restore the default handlers, execute an operation, and finally restore the saved handlers:

```
// store current handlers
conduit::utils::conduit_info_handler    on_info  = conduit::utils::info_handler();
conduit::utils::conduit_warning_handler on_warn  = conduit::utils::warning_handler();
conduit::utils::conduit_error_handler   on_error = conduit::utils::error_handler();

// temporarily restore default handlers
conduit::utils::set_info_handler(conduit::utils::default_info_handler);
conduit::utils::set_warning_handler(conduit::utils::default_warning_handler);
conduit::utils::set_error_handler(conduit::utils::default_error_handler);

// do something exciting ...
```

(continues on next page)

(continued from previous page)

```
// done with excitement, reset to previously saved handlers
conduit::utils::set_info_handler(on_info);
conduit::utils::set_warning_handler(on_warn);
conduit::utils::set_error_handler(on_error);
```

Python Tutorial

This short tutorial provides Python examples that demonstrate the Conduit's Core API. Conduit's unit tests (*src/tests/{library_name}/python*) also provide a rich set of examples for Conduit's Core API and additional libraries. Ascent's Tutorial also provides a brief intro to Conduit basics with C++ and Python examples.

Basic Concepts

Node basics

The *Node* class is the primary object in conduit.

Think of it as a hierarchical variant object.

```
import conduit
n = conduit.Node()
n["my"] = "data"
print(n)
```

```
my: "data"
```

The *Node* class supports hierarchical construction.

```
n = conduit.Node()
n["my"] = "data";
n["a/b/c"] = "d";
n["a"]["b"]["e"] = 64.0;
print(n)
print("total bytes: {}\n".format(n.total_strided_bytes()))
```

```
my: "data"
a:
  b:
    c: "d"
    e: 64.0

total bytes: 15
```

Borrowing from JSON (and other similar notations), collections of named nodes are called *Objects* and collections of unnamed nodes are called *Lists*, all other types are leaves that represent concrete data.

```
n = conduit.Node()
n["object_example/val1"] = "data"
n["object_example/val2"] = 10
n["object_example/val3"] = 3.1415

for i in range(5):
    l_entry = n["list_example"].append()
    l_entry.set(i)
print(n)
```

```
 object_example:
   val1: "data"
   val2: 10
   val3: 3.1415
 list_example:
   - 0
   - 1
   - 2
   - 3
   - 4
```

You can iterate through a Node's children.

```
n = conduit.Node()
n["object_example/val1"] = "data"
n["object_example/val2"] = 10
n["object_example/val3"] = 3.1415

for i in range(5):
    l_entry = n["list_example"].append()
    l_entry.set(i)
print(n)

for v in n["object_example"].children():
    print("{}: {}".format(v.name(),str(v.node())))
print()
for v in n["list_example"].children():
    print(v.node())
```

```
 object_example:
   val1: "data"
   val2: 10
   val3: 3.1415
 list_example:
   - 0
   - 1
   - 2
   - 3
   - 4

 val1: "data"
 val2: 10
 val3: 3.1415
```

(continues on next page)

(continued from previous page)

```
0
1
2
3
4
```

Behind the scenes, *Node* instances manage a collection of memory spaces.

```
n = conduit.Node()
n["my"] = "data"
n["a/b/c"] = "d"
n["a"]["b"]["e"] = 64.0
print(n.info())
```

```
mem_spaces:
  0x7fa45d800830:
    path: "my"
    type: "allocated"
    bytes: 5
  0x7fa45d82caa0:
    path: "a/b/c"
    type: "allocated"
    bytes: 2
  0x7fa45d816b10:
    path: "a/b/e"
    type: "allocated"
    bytes: 8
total_bytes_allocated: 15
total_bytes_mmaped: 0
total_bytes_compact: 15
total_strided_bytes: 15
```

There is no absolute path construct, all paths are fetched relative to the current node (a leading `/` is ignored when fetching). Empty paths names are also ignored, fetching `a///b` is equalvalent to fetching `a/b`.

Bitwidth Style Types

When sharing data in scientific codes, knowing the precision of the underlining types is very important.

Conduit uses well defined bitwidth style types (inspired by NumPy) for leaf values. In Python, leaves are provided as NumPy ndarrays.

```
n = conduit.Node()
n["test"] = numpy.uint32(100)
print(n)
```

```
test: 100
```

Standard Python numeric types will be mapped to bitwidth style types.

```
n = conduit.Node()
n["test"] = 10
print(n.schema())
```

```
 {
   "test": {"dtype":"int64","number_of_elements": 1,"offset": 0,"stride": 8,"element_
↪bytes": 8,"endianness": "little"}
 }
```

Supported Bitwidth Style Types:

- signed integers: int8,int16,int32,int64
- unsigned integers: uint8,uint16,uint32,uint64
- floating point numbers: float32,float64

Conduit provides these types by constructing a mapping for the current platform the from the following C++ types:

- char, short, int, long, long long, float, double, long double

When a **set** method is called on a leaf Node, if the data passed to the **set** is compatible with the Node's Schema the data is simply copied.

Compatible Schemas

When passed a compatible Node, Node methods `update` and `update_compatible` allow you to copy data into Node or extend a Node with new data without changing existing allocations.

Schemas do not need to be identical to be compatible.

You can check if a Schema is compatible with another Schema using the **Schema::compatible(Schema &test)** method. Here is the criteria for checking if two Schemas are compatible:

- **If the calling Schema describes an Object** : The passed test Schema must describe an Object and the test Schema's children must be compatible with the calling Schema's children that have the same name.
- **If the calling Schema describes a List**: The passed test Schema must describe a List, the calling Schema must have at least as many children as the test Schema, and when compared in list order each of the test Schema's children must be compatible with the calling Schema's children.
- **If the calling Schema describes a leaf data type**: The calling Schema's and test Schema's **dtype().id()** and **dtype().element_bytes()** must match, and the calling Schema **dtype().number_of_elements()** must be greater than or equal than the test Schema's.

Differences between C++ and Python APIs

In Python, Node objects are reference-counted containers that hold C++ Node pointers. This provides a Python API similar to using references in C++. However, you should be aware of some key differences.

This provides similar API in Python to using references in C++, however there are a few key differences to be aware of.

The [] operator is different in that it will return not only Nodes, but numpy arrays depending on the context:

```
# setup a node with a leaf array
n = conduit.Node()
data = numpy.zeros((5,),dtype=numpy.float64)
n["my/path/to/data"] = data

# this will be an ndarray
my_data = n["my/path/to/data"]
print("== this will be an ndarray == ")
print("data: ", my_data)
print("repr: ",repr(my_data))
print()

# this will be a node
n_my_path = n["my/path"]
print("== this will be a node == ")
print("{node}\n", n_my_path)
print("{schema}\n",n_my_path.schema().to_yaml())
```

```
 == this will be an ndarray ==
 data:  [0. 0. 0. 0. 0.]
 repr:  array([0., 0., 0., 0., 0.])

 == this will be a node ==
 {node}

 to:
   data: [0.0, 0.0, 0.0, 0.0, 0.0]

 {schema}

 to:

   data:
       dtype: "float64"
       number_of_elements: 5
       offset: 0
       stride: 8
       element_bytes: 8
       endianness: "little"
```

If you are expecting a Node, the best way to access a subpath is using `fetch()` or `fetch_existing()`:

```
# setup a node with a leaf array
n = conduit.Node()
data = numpy.zeros((5,),dtype=numpy.float64)
n["my/path/to/data"] = data

# this will be an ndarray
my_data = n["my/path/to/data"]
print("== this will be an ndarray == ")
print("data: ", my_data)
print("repr: ",repr(my_data))
print()

# equiv access via fetch (or fetch_existing)
# first fetch the node and then the array
```

(continues on next page)

(continued from previous page)

```
my_data = n.fetch("my/path/to/data").value()
print("== this will be an ndarray == ")
print("data: ",my_data)
print("repr: ",repr(my_data))
print()
```

```
 == this will be an ndarray ==
 data:  [0. 0. 0. 0. 0.]
 repr:  array([0., 0., 0., 0., 0.])

 == this will be an ndarray ==
 data:  [0. 0. 0. 0. 0.]
 repr:  array([0., 0., 0., 0., 0.])
```

We use the const construct in C++ to provide additional seat belts for read-only style access to Nodes. We don't provide a similar overall construct in Python, but the standard methods like `fetch_existing()` do support these types of use cases:

```
# setup a node with a leaf array
n = conduit.Node()
data = numpy.zeros((5,),dtype=numpy.float64)
n["my/path/to/data"] = data

# access via fetch existing
# first fetch the node
n_data = n.fetch_existing("my/path/to/data")
# then the value
my_data = n_data.value()
print("== this will be an ndarray == ")
print("data: ",my_data)
print("repr: ",repr(my_data))
print()

# using fetch_existing,
# if the path doesn't exist - we will get an Exception
try:
    n_data = n.fetch_existing("my/path/TYPO/data")
except Exception as e:
    print("Here is what went wrong:")
    print(e)
```

```
 == this will be an ndarray ==
 data:  [0. 0. 0. 0. 0.]
 repr:  array([0., 0., 0., 0., 0.])

 Here is what went wrong:

 file: /Users/harrison37/Work/github/llnl/conduit/src/libs/conduit/conduit_node.cpp
 line: 13050
 message:
 Cannot fetch non-existent child "TYPO" from Node(my/path)
```

Reading YAML and JSON Strings

Parsing text with *Node::parse()*

Node.parse() parses YAML and JSON strings into a *Node* tree.

```
yaml_txt = "mykey: 42.0"

n = conduit.Node()
n.parse(yaml_txt,"yaml")

print(n["mykey"])
print(n.schema())
```

```
 42.0

 {
   "mykey": {"dtype":"float64","number_of_elements": 1,"offset": 0,"stride": 8,
↪"element_bytes": 8,"endianness": "little"}
 }
```

```
json_txt = '{"mykey": 42.0}'

n = conduit.Node()
n.parse(json_txt,"json")

print(n["mykey"])
print(n.schema())
```

```
 42.0

 {
   "mykey": {"dtype":"float64","number_of_elements": 1,"offset": 0,"stride": 8,
↪"element_bytes": 8,"endianness": "little"}
 }
```

The first argument is the string to parse and the second argument selects the protocol to use when parsing.

Valid Protocols: `json`, `conduit_json`, `conduit_base64_json`, `yaml`.

- `json` and `yaml` protocols parse pure JSON or YAML strings. For leaf nodes wide types such as *int64*, *uint64*, and *float64* are inferred.

Homogeneous numeric lists are parsed as Conduit arrays.

```
yaml_txt = "myarray: [0.0, 10.0, 20.0, 30.0]"

n = conduit.Node()
n.parse(yaml_txt,"yaml")

print(n["myarray"])

print(n.fetch("myarray").schema())
```

```
 [ 0. 10. 20. 30.]
 {"dtype":"float64","number_of_elements": 4,"offset": 0,"stride": 8,"element_bytes":␣
↪8,"endianness": "little"}
```

- `conduit_json` parses JSON with conduit data type information, allowing you to specify bitwidth style types, strides, etc.
- `conduit_base64_json` combines the *conduit_json* protocol with an embedded base64-encoded data block

Generators

Using *Generator* instances

Node.parse() is sufficient for most use cases, but you can also use a *Generator* instance to parse JSON and YAML. Additionally, Generators can parse a conduit JSON schema and bind it to in-core data.

```
g = conduit.Generator("{test: {dtype: float64, value: 100.0}}",
                      "conduit_json")
n = conduit.Node()
g.walk(n)
print(n["test"])
print(n)
```

```
 100.0

 test: 100.0
```

Like *Node::parse()*, *Generators* can also parse pure JSON or YAML. For leaf nodes: wide types such as *int64*, *uint64*, and *float64* are inferred.

```
g = conduit.Generator("{test: 100.0}",
                      "json")
n = conduit.Node()
g.walk(n)
print(n["test"])
print(n)
```

```
 100.0

 test: 100.0
```

```
g = conduit.Generator("test: 100.0",
                      "yaml")
n = conduit.Node()
g.walk(n)
print(n["test"])
print(n)
```

```
 100.0

 test: 100.0
```

String Formatting Helpers

conduit.utils.format

String formatting in Python land has always been much more pleasant than in C++ land. In C++, we bundle `fmt`, but Python's out-of-the box support for string formatting is fantastic. Since users may encode format string arguments in conduit Nodes (and in HDF5, YAML, etc files) we still provide access the `fmt` based `conduit.utils.format` functionality in Python.

conduit.utils.format(string, args)

The `args` case allows named arguments (args passed as object) or ordered args (args passed as list).

conduit.utils.format(string, args) – object case:

```
import conduit
import conduit.utils

args = conduit.Node()
args["answer"] = 42

print(conduit.utils.format("The answer is {answer:04}.", args = args))

args.reset()
args["adjective"] =  "other";
args["answer"] = 3.1415;

print(conduit.utils.format("The {adjective} answer is {answer:0.4f}.",
                           args = args))
```

```
 The answer is 0042.
 The other answer is 3.1415.
```

conduit.utils.format(string, args) – list case:

```
import conduit
import conduit.utils

args = conduit.Node()
args.append().set(42)

print(conduit.utils.format("The answer is {:04}.",args = args))

args.reset()
args.append().set("other")
args.append().set(3.1415)

print(conduit.utils.format("The {} answer is {:0.4f}.", args =args))
```

```
 The answer is 0042.
 The other answer is 3.1415.
```

conduit.utils.format(string, maps, map_index)

The `maps` case also supports named or ordered args and works in conjunction with a `map_index`. The `map_index` is used to fetch a value from an array, or list of strings, which is then passed to fmt. The `maps` style of indexed

indirection supports generating path strings for non-trivial domain partition mappings in Blueprint.

conduit.utils.format(string, maps, map_index) – object case:

```
import conduit
import conduit.utils
import numpy as np

maps = conduit.Node()
maps["answer"].set(np.array([42.0, 3.1415]))

print(conduit.utils.format("The answer is {answer:04}.",
                           maps = maps, map_index = 0))

print(conduit.utils.format("The answer is {answer:04}.",
                           maps = maps, map_index = 1))
print()

maps.reset()
maps["answer"].set(np.array([42.0, 3.1415]));
slist = maps["position"];
slist.append().set("first")
slist.append().set("second")

print(conduit.utils.format("The {position} answer is {answer:0.4f}.",
                           maps = maps, map_index = 0))

print(conduit.utils.format("The {position} answer is {answer:0.4f}.",
                           maps = maps, map_index = 1))
```

```
 The answer is 0042.
 The answer is 3.1415.

 The first answer is 42.0000.
 The second answer is 3.1415.
```

conduit.utils.format(string, maps, map_index) – list case:

```
import conduit
import conduit.utils
import numpy as np

maps = conduit.Node()
vals = np.array([42.0, 3.1415])
maps.append().set(vals)

print(conduit.utils.format("The answer is {}.",
                           maps = maps, map_index = 0))

print(conduit.utils.format("The answer is {}.",
                           maps = maps, map_index = 1))
print()

maps.reset()
# first arg
slist = maps.append();
slist.append().set("first")
slist.append().set("second")
```

(continues on next page)

(continued from previous page)

```
# second arg
maps.append().set(vals)

print(conduit.utils.format("The {} answer is {:0.4f}.",
                           maps = maps, map_index = 0))

print(conduit.utils.format("The {} answer is {:0.4f}.",
                           maps = maps, map_index = 1))
```

```
 The answer is 42.
 The answer is 3.1415.

 The first answer is 42.0000.
 The second answer is 3.1415.
```

Passing Conduit Nodes between C++, Fortran, and Python

The `cpp_fort_and_py` example demonstrates how to pass Conduit Nodes between C++, Fortran, and Python. It is a standalone example that you can build with CMake against your Conduit install.

You can find this example under `src/examples/cpp_fort_and_py` in Conduit's source tree, or under `examples/conduit/cpp_fort_and_py` in a Conduit install.

It includes source for an embedded python interpreter and also shows how to create a Fortran module that binds Conduit Nodes via Conduit's C-API.

It creates two executables:

conduit_cpp_and_py_ex	Demo of C++ to Python and vice versa
conduit_fort_and_py_ex	Demo of Fortran to Python and vice versa

This demos wrapping Conduit Nodes, effectively creating referenced data across languages. You can also use *set_external* to directly access and change zero-copied data.

Please see the main CMakeList.txt file for details on building and running:

`cpp_fort_and_py/CMakeLists.txt` excerpt:

```
Example that shows how to use Conduit across C++, Fortran, and an
embedded Python interpreter.


Building:

Note: The python instance must have the conduit python module installed
or it must be in your PYTHONPATH.

> mkdir build
> cd build

# if conduit python module is not installed in your python instance
# > export PYTHONPATH=/path/to/conduit-install/python-modules

> cmake  \
```

(continues on next page)

(continued from previous page)

```
   -DCONDUIT_DIR=/path/to/conduit/install
   -DPYTHON_EXECUTABLE=/path/to/python/bin/python
   ../
> make


Running:
> ./conduit_cpp_and_py_ex
> ./conduit_fort_and_py_ex

# if conduit python module is not installed in your python instance
> env PYTHONPATH=/path/to/conduit-install/python-modules  ./conduit_cpp_and_py_ex
> env PYTHONPATH=/path/to/conduit-install/python-modules  ./conduit_fort_and_py_ex
```

8.2.2 Relay

Note: The **relay** APIs and docs are work in progress.

Conduit Relay is an umbrella project for I/O and communication functionality built on top of Conduit's Core API. It includes four components:

- **io** - I/O functionally beyond binary, memory mapped, and json-based text file I/O. Includes optional Silo, HDF5, and ADIOS I/O support.
- **web** - An embedded web server (built using CivetWeb) that can host files and supports developing custom REST and WebSocket backends that use conduit::Node instances as payloads.
- **mpi** - Interfaces for MPI communication using conduit::Node instances as payloads.
- **mpi::io** - I/O functionality as with io library but with some notion of collective writing to a shared file that can include multiple time steps and domains.

The **io** and **web** features are built into the *conduit_relay* library. The MPI functionality exists in a separate library *conduit_relay_mpi* to avoid include and linking issues for serial codes that want to use relay. Likewise, the parallel versions of the I/O functions are built into the *conduit_relay_mpi_io* library so it can be linked to parallel codes.

Relay I/O

Conduit Relay I/O provides optional Silo, HDF5, and ADIOS I/O interfaces.

These interfaces can be accessed through a generic path-based API, generic handle class, or through APIs specific to each underlying I/O interface. The specific APIs provide lower level control and allow reuse of handles, which is more efficient for most non-trivial use cases. The generic handle class strikes a balance between usability and efficiency.

Relay I/O Path-based Interface

The path-based Relay I/O interface allows you to read and write conduit::Nodes using any enabled I/O interface through a simple path-based (string) API. The underlying I/O interface is selected using the extension of the destination path or an explicit protocol argument.

The `conduit_relay` library provides the following methods in the `conduit::relay::io` namespace:

- `relay::io::save`

  - Saves the contents of the passed Node to a file. Works like a `Node::set` to the file: if the file exists, it is overwritten to reflect contents of the passed Node.

- `relay::io::save_merged`
  - Merges the contents of the passed Node to a file. Works like a `Node::update` to the file: if the file exists, new data paths are appended, common paths are overwritten, and other existing paths are not changed.
- `relay::io::load`
  - Loads the contents of a file into the passed Node. Works like a `Node::set` from the contents of the file: if the Node has existing data, it is overwritten to reflect contents of the file.
- `relay::io::load_merged`
  - Merges the contents of a file into the passed Node. Works like a `Node::update` rom the contents of the file: if the Node has existing data, new data paths are appended, common paths are overwritten, and other existing paths are not changed.

The `conduit_relay_mpi_io` library provides the `conduit::relay::mpi::io` namespace which includes variants of these methods which take a MPI Communicator. These variants pass the communicator to the underlying I/O interface to enable collective I/O. Relay currently only supports collective I/O for ADIOS.

Relay I/O Path-based Interface Examples

Save and Load

- **C++ Example:**

```
// setup node to save
Node n;
n["a/my_data"] = 1.0;
n["a/b/my_string"] = "value";
std::cout << "\nNode to write:" << std::endl;
n.print();

//save to json using save
conduit::relay::io::save(n,"my_output.json");

//load back from json using load
Node n_load;
conduit::relay::io::load("my_output.json",n_load);
std::cout << "\nLoad result:" << std::endl;
n_load.print();
```

- **Output:**

```

Node to write:

a:
  my_data: 1.0
  b:
    my_string: "value"


Load result:

```

(continues on next page)

(continued from previous page)

```
a:
  my_data: 1.0
  b:
    my_string: "value"
```

Save Merged

- **C++ Example:**

```
// setup node to save
Node n;
n["a/my_data"] = 1.0;
n["a/b/my_string"] = "value";
std::cout << "\nNode to write:" << std::endl;
n.print();

//save to hdf5 using save
conduit::relay::io::save(n,"my_output.hdf5");

// append a new path to the hdf5 file using save_merged
Node n2;
n2["a/b/new_data"] = 42.0;
std::cout << "\nNode to append:" << std::endl;
n2.print();
conduit::relay::io::save_merged(n2,"my_output.hdf5");

Node n_load;
// load back from hdf5 using load:
conduit::relay::io::load("my_output.hdf5",n_load);
std::cout << "\nLoad result:" << std::endl;
n_load.print();
```

- **Output:**

```

Node to write:

a:
  my_data: 1.0
  b:
    my_string: "value"


Node to append:

a:
  b:
    new_data: 42.0


Load result:

a:
  my_data: 1.0
```

(continues on next page)

(continued from previous page)

```
  b:
    my_string: "value"
    new_data: 42.0
```

Load Merged

- **C++ Example:**

```
// setup node to save
Node n;
n["a/my_data"] = 1.0;
n["a/b/my_string"] = "value";
std::cout << "\nNode to write:" << std::endl;
n.print();

//save to hdf5 using generic i/o save
conduit::relay::io::save(n,"my_output.hdf5");

// append to existing node with data from hdf5 file using load_merged
Node n_load;
n_load["a/b/new_data"] = 42.0;
std::cout << "\nNode to load into:" << std::endl;
n_load.print();
conduit::relay::io::load_merged("my_output.hdf5",n_load);
std::cout << "\nLoad result:" << std::endl;
n_load.print();
```

- **Output:**

```

Node to write:

a:
  my_data: 1.0
  b:
    my_string: "value"


Node to load into:

a:
  b:
    new_data: 42.0


Load result:

a:
  b:
    new_data: 42.0
    my_string: "value"
  my_data: 1.0
```

Load from Subpath

- **C++ Example:**

```
// setup node to save
Node n;
n["path/to/my_data"] = 1.0;
std::cout << "\nNode to write:" << std::endl;
n.print();

//save to hdf5 using generic i/o save
conduit::relay::io::save(n,"my_output.hdf5");

// load only a subset of the tree
Node n_load;
conduit::relay::io::load("my_output.hdf5:path/to",n_load);
std::cout << "\nLoad result from 'path/to'" << std::endl;
n_load.print();
```

- **Output:**

```
Node to write:

path:
  to:
    my_data: 1.0


Load result from 'path/to'

my_data: 1.0
```

Save to Subpath

- **C++ Example:**

```
// setup node to save
Node n;
n["my_data"] = 1.0;
std::cout << "\nNode to write to 'path/to':" << std::endl;
n.print();

//save to hdf5 using generic i/o save
conduit::relay::io::save(n,"my_output.hdf5:path/to");

// load only a subset of the tree
Node n_load;
conduit::relay::io::load("my_output.hdf5",n_load);
std::cout << "\nLoad result:" << std::endl;
n_load.print();
```

- **Output:**

```
Node to write to 'path/to':

my_data: 1.0


Load result:

path:
  to:
    my_data: 1.0
```

Relay I/O Handle Interface

The `relay::io::IOHandle` class provides a high level interface to query, read, and modify files.

It provides a generic interface that is more efficient than the path-based interface for protocols like HDF5 which support partial I/O and querying without reading the entire contents of a file. It also supports simpler built-in protocols (conduit_bin, json, etc) that do not support partial I/O for convenience. Its basic contract is that changes to backing (file on disk, etc) are not guaranteed to be reflected until the handle is closed. Relay I/O Handle supports reading AXOM Sidre DataStore Style files. Relay I/O Handle does not yet support Silo or ADIOS.

IOHandle has the following instance methods:

- `open`
  - Opens a handle. The underlying I/O interface is selected using the extension of the destination path or an explicit protocol argument. Supports reading and writing by default. Select a different mode by passing an options node that contains a `mode` child with one of the following strings:

`rw` read + write (default mode)	Supports both read and write operations. Creates file if it does not exist.
`r` read only	Only supports read operations. Throws an Error if you open a non-existing file or on any attempt to write.
`w` write only	Only supports write operations. Throws an Error on any attempt to read.

> **Danger:** Note: While you can read from and write to subpaths using a handle, IOHandle *does not* support opening a file with a subpath (e.g. `myhandle.open("file.hdf5:path/data")`).

- `read`
  - Merges the contents from the handle or contents from a subpath of the handle into the passed Node. Works like a `Node::update` from the handle: if the Node has existing data, new data paths are appended, common paths are overwritten, and other existing paths are not changed.
- `write`
  - Writes the contents of the passed Node to the handle or to a subpath of the handle. Works like a `Node::update` to the handle: if the handle has existing data, new data paths are appended, common paths are overwritten, and other existing paths are not changed.
- `has_path`

  - Checks if the handle contains a given path.
- `list_child_names`
  - Returns a list of the child names at a given path, or an empty list if the path does not exist.
- `remove`
  - Removes any data at and below a given path. With HDF5 the space may not be fully reclaimed.
- `close`
  - Closes a handle. This is when changes are realized to the backing (file on disc, etc).

Relay I/O Handle Examples

- **C++ Example:**

```
// setup node with example data to save
Node n;
n["a/data"]   = 1.0;
n["a/more_data"] = 2.0;
n["a/b/my_string"] = "value";
std::cout << "\nNode to write:" << std::endl;
n.print();

// save to hdf5 file using the path-based api
conduit::relay::io::save(n,"my_output.hdf5");

// inspect and modify with an IOHandle
conduit::relay::io::IOHandle h;
h.open("my_output.hdf5");

// check for and read a path we are interested in
if( h.has_path("a/data") )
{
    Node nread;
    h.read("a/data",nread);
    std::cout << "\nValue at \"a/data\" = "
              << nread.to_float64()
              << std::endl;
}

// check for and remove a path we don't want
if( h.has_path("a/more_data") )
{
    h.remove("a/more_data");
    std::cout << "\nRemoved \"a/more_data\""
              << std::endl;
}

// verify the data was removed
if( !h.has_path("a/more_data") )
{
    std::cout << "\nPath \"a/more_data\" is no more"
              << std::endl;
}

std::cout << "\nWriting to \"a/c\""
```

(continues on next page)

(continued from previous page)

```
          << std::endl;
// write some new data
n = 42.0;
h.write(n,"a/c");

// find the names of the children of "a"
std::vector<std::string> cld_names;
h.list_child_names("a",cld_names);

// print the names
std::cout << "\nChildren of \"a\": ";
std::vector<std::string>::const_iterator itr;
for (itr = cld_names.begin();
     itr < cld_names.end();
     ++itr)
{
    std::cout << "\"" << *itr << "\" ";
}

std::cout << std::endl;

Node nread;
// read the entire contents
h.read(nread);

std::cout << "\nRead Result:" << std::endl;
nread.print();
```

- **Output:**

```
Node to write:

a:
  data: 1.0
  more_data: 2.0
  b:
    my_string: "value"


Value at "a/data" = 1

Removed "a/more_data"

Path "a/more_data" is no more

Writing to "a/c"

Children of "a": "data" "b" "c"

Read Result:

a:
  data: 1.0
  b:
    my_string: "value"
  c: 42.0
```

(continues on next page)

(continued from previous page)

- **Python Example:**

```
import conduit
import conduit.relay.io

n = conduit.Node()
n["a/data"]   = 1.0
n["a/more_data"] = 2.0
n["a/b/my_string"] = "value"
print("\nNode to write:")
print(n)

# save to hdf5 file using the path-based api
conduit.relay.io.save(n,"my_output.hdf5")

# inspect and modify with an IOHandle
h = conduit.relay.io.IOHandle()
h.open("my_output.hdf5")

# check for and read a path we are interested in
if h.has_path("a/data"):
     nread = conduit.Node()
     h.read(nread,"a/data")
     print('\nValue at "a/data" = {0}'.format(nread.value()))

# check for and remove a path we don't want
if h.has_path("a/more_data"):
    h.remove("a/more_data")
    print('\nRemoved "a/more_data"')

# verify the data was removed
if not h.has_path("a/more_data"):
    print('\nPath "a/more_data" is no more')

# write some new data
print('\nWriting to "a/c"')
n.set(42.0)
h.write(n,"a/c")

# find the names of the children of "a"
cnames = h.list_child_names("a")
print('\nChildren of "a": {0}'.format(cnames))

nread = conduit.Node()
# read the entire contents
h.read(nread)

print("\nRead Result:")
print(nread)
```

- **Output:**

```
 Node to write:
```

(continues on next page)

(continued from previous page)

```
a:
  data: 1.0
  more_data: 2.0
  b:
    my_string: "value"


Value at "a/data" = 1.0

Removed "a/more_data"

Path "a/more_data" is no more

Writing to "a/c"

Children of "a": ['data', 'b', 'c']

Read Result:

a:
  data: 1.0
  b:
    my_string: "value"
  c: 42.0
```

- **C++ Sidre Basic Example:**

```
// this example reads a sample hdf5 sidre style file

std::string input_fname = relay_test_data_path(
                                  "texample_sidre_basic_ds_demo.sidre_hdf5");

// open our sidre file for read with an IOHandle
conduit::relay::io::IOHandle h;
h.open(input_fname,"sidre_hdf5");

// find the names of the children at the root
std::vector<std::string> cld_names;
h.list_child_names(cld_names);

// print the names
std::cout << "\nChildren at root: ";
std::vector<std::string>::const_iterator itr;
for (itr = cld_names.begin();
     itr < cld_names.end();
     ++itr)
{
    std::cout << "\"" << *itr << "\" ";
}

Node nread;
// read the entire contents
h.read(nread);

std::cout << "\nRead Result:" << std::endl;
```

(continues on next page)

(continued from previous page)

```
nread.print();
```

- **Output:**

```
Children at root: "my_scalars" "my_strings" "my_arrays"
Read Result:

my_scalars:
  i64: 1
  f64: 10.0
my_strings:
  s0: "s0 string"
  s1: "s1 string"
my_arrays:
  a5_i64: [0, 1, 2, 3, 4]
  a5_i64_ext: [0, 1, 2, 3, -5]
  b_v1: [1.0, 1.0, 1.0]
  b_v2: [2.0, 2.0, 2.0]
```

- **Python Sidre Basic Example:**

```
import conduit
import conduit.relay.io

# this example reads a sample hdf5 sidre style file
input_fname = relay_test_data_path("texample_sidre_basic_ds_demo.sidre_hdf5")

# open our sidre file for read with an IOHandle
h = conduit.relay.io.IOHandle()
h.open(input_fname,"sidre_hdf5")

# find the names of the children at the root
cnames = h.list_child_names()
print('\nChildren at root {0}'.format(cnames))

nread = conduit.Node()
# read the entire contents
h.read(nread);

print("Read Result:")
print(nread)
```

- **Output:**

```
 Children at root ['my_scalars', 'my_strings', 'my_arrays']
 Read Result:

 my_scalars:
   i64: 1
   f64: 10.0
 my_strings:
   s0: "s0 string"
   s1: "s1 string"
```

(continues on next page)

(continued from previous page)

```
 my_arrays:
   a5_i64: [0, 1, 2, 3, 4]
   a5_i64_ext: [0, 1, 2, 3, -5]
   b_v1: [1.0, 1.0, 1.0]
   b_v2: [2.0, 2.0, 2.0]
```

- **C++ Sidre with Root File Example:**

```
// this example reads a sample hdf5 sidre datastore, grouped by a root file
std::string input_fname = relay_test_data_path(
                                "out_spio_blueprint_example.root");

// read using the root file
conduit::relay::io::IOHandle h;
h.open(input_fname,"sidre_hdf5");

// find the names of the children at the root
std::vector<std::string> cld_names;
h.list_child_names(cld_names);

// the "root" (/) of the Sidre-based IOHandle to the datastore provides
// access to the root file itself, and all of the data groups

// print the names
std::cout << "\nChildren at root: ";
std::vector<std::string>::const_iterator itr;
for (itr = cld_names.begin();
     itr < cld_names.end();
     ++itr)
{
    std::cout << "\"" << *itr << "\" ";
}

Node nroot;
// read the entire root file contents
h.read("root",nroot);

std::cout << "\nRead \"root\" Result:" << std::endl;
nroot.print();

Node nread;
// read all of data group 0
h.read("0",nread);

std::cout << "\nRead \"0\" Result:" << std::endl;
nread.print();

// reset, or trees will blend in this case
nread.reset();

// read a subpath of data group 1
h.read("1/mesh",nread);

std::cout << "\nRead \"1/mesh\" Result:" << std::endl;
nread.print();
```

- **Output:**

```
Children at root: "root" "0" "1" "2" "3"
Read "root" Result:

blueprint_index:
  mesh:
    state:
      number_of_domains: 4
    coordsets:
      coords:
        type: "uniform"
        coord_system:
          axes:
            x:
            y:
          type: "cartesian"
        path: "mesh/coordsets/coords"
    topologies:
      mesh:
        type: "uniform"
        coordset: "coords"
        path: "mesh/topologies/mesh"
    fields:
      field:
        number_of_components: 1
        topology: "mesh"
        association: "element"
        path: "mesh/fields/field"
      rank:
        number_of_components: 1
        topology: "mesh"
        association: "element"
        path: "mesh/fields/rank"
file_pattern: "out_spio_blueprint_example/out_spio_blueprint_example_%07d.hdf5"
number_of_files: 4
number_of_trees: 4
protocol:
  name: "sidre_hdf5"
  version: "0.0"
tree_pattern: "datagroup_%07d"


Read "0" Result:

mesh:
  coordsets:
    coords:
      dims:
        i: 3
        j: 3
      origin:
        x: 0.0
        y: -10.0
      spacing:
        dx: 10.0
        dy: 10.0
```

(continues on next page)

(continued from previous page)

```
      type: "uniform"
  topologies:
    mesh:
      type: "uniform"
      coordset: "coords"
  fields:
    field:
      association: "element"
      topology: "mesh"
      volume_dependent: "false"
      values: [0.0, 1.0, 2.0, 3.0]
    rank:
      association: "element"
      topology: "mesh"
      values: [0, 0, 0, 0]


Read "1/mesh" Result:

coordsets:
  coords:
    dims:
      i: 3
      j: 3
    origin:
      x: 20.0
      y: -10.0
    spacing:
      dx: 10.0
      dy: 10.0
    type: "uniform"
topologies:
  mesh:
    type: "uniform"
    coordset: "coords"
fields:
  field:
    association: "element"
    topology: "mesh"
    volume_dependent: "false"
    values: [0.0, 1.0, 2.0, 3.0]
  rank:
    association: "element"
    topology: "mesh"
    values: [1, 1, 1, 1]
```

- **Python Sidre with Root File Example:**

```
import conduit
import conduit.relay.io

# this example reads a sample hdf5 sidre datastore,
# grouped by a root file
input_fname = relay_test_data_path("out_spio_blueprint_example.root")

# open our sidre datastore for read via root file with an IOHandle
```

(continues on next page)

(continued from previous page)

```
h = conduit.relay.io.IOHandle()
h.open(input_fname,"sidre_hdf5")

# find the names of the children at the root
# the "root" (/) of the Sidre-based IOHandle to the datastore provides
# access to the root file itself, and all of the data groups
cnames = h.list_child_names()
print('\nChildren at root {0}'.format(cnames))

nroot = conduit.Node();
# read the entire root file contents
h.read(path="root",node=nroot);

print("Read 'root' Result:")
print(nroot)

nread = conduit.Node();
# read all of data group 0
h.read(path="0",node=nread);

print("Read '0' Result:")
print(nread)

#reset, or trees will blend in this case
nread.reset();

# read a subpath of data group 1
h.read(path="1/mesh",node=nread);

print("Read '1/mesh' Result:")
print(nread)
```

- **Output:**

```
Children at root ['root', '0', '1', '2', '3']
Read 'root' Result:

blueprint_index:
  mesh:
    state:
      number_of_domains: 4
    coordsets:
      coords:
        type: "uniform"
        coord_system:
          axes:
            x:
            y:
          type: "cartesian"
        path: "mesh/coordsets/coords"
    topologies:
      mesh:
        type: "uniform"
        coordset: "coords"
        path: "mesh/topologies/mesh"
```

(continues on next page)

(continued from previous page)

```
    fields:
      field:
        number_of_components: 1
        topology: "mesh"
        association: "element"
        path: "mesh/fields/field"
      rank:
        number_of_components: 1
        topology: "mesh"
        association: "element"
        path: "mesh/fields/rank"
file_pattern: "out_spio_blueprint_example/out_spio_blueprint_example_%07d.hdf5"
number_of_files: 4
number_of_trees: 4
protocol:
  name: "sidre_hdf5"
  version: "0.0"
tree_pattern: "datagroup_%07d"

Read '0' Result:

mesh:
  coordsets:
    coords:
      dims:
        i: 3
        j: 3
      origin:
        x: 0.0
        y: -10.0
      spacing:
        dx: 10.0
        dy: 10.0
      type: "uniform"
  topologies:
    mesh:
      type: "uniform"
      coordset: "coords"
  fields:
    field:
      association: "element"
      topology: "mesh"
      volume_dependent: "false"
      values: [0.0, 1.0, 2.0, 3.0]
    rank:
      association: "element"
      topology: "mesh"
      values: [0, 0, 0, 0]

Read '1/mesh' Result:

coordsets:
  coords:
    dims:
      i: 3
      j: 3
    origin:
```

(continues on next page)

(continued from previous page)

```
      x: 20.0
      y: -10.0
    spacing:
      dx: 10.0
      dy: 10.0
    type: "uniform"
topologies:
  mesh:
    type: "uniform"
    coordset: "coords"
fields:
  field:
    association: "element"
    topology: "mesh"
    volume_dependent: "false"
    values: [0.0, 1.0, 2.0, 3.0]
  rank:
    association: "element"
    topology: "mesh"
    values: [1, 1, 1, 1]
```

Relay I/O HDF5 Interface

The Relay I/O HDF5 interface provides methods to read and write Nodes using HDF5 handles. It is also the interface used to implement the path-based and handle I/O interfaces for HDF5. This interface provides more control and allows more efficient reuse of I/O handles. It is only available in C++.

Relay I/O HDF5 Interface Examples

Here is a example exercising the basic parts of Relay I/O's HDF5 interface, for more detailed documentation see the `conduit_relay_io_hdf5_api.hpp` header file.

HDF5 I/O Interface Basics

- **C++ Example:**

```
// setup node to save
Node n;
n["a/my_data"] = 1.0;
n["a/b/my_string"] = "value";
std::cout << "\nNode to write:" << std::endl;
n.print();

// open hdf5 file and obtain a handle
hid_t h5_id = conduit::relay::io::hdf5_create_file("myoutput.hdf5");

// write data
conduit::relay::io::hdf5_write(n,h5_id);

// close our file
```

(continues on next page)

(continued from previous page)

```
conduit::relay::io::hdf5_close_file(h5_id);

// open our file to read
h5_id = conduit::relay::io::hdf5_open_file_for_read_write("myoutput.hdf5");

// check if a subpath exists
if(conduit::relay::io::hdf5_has_path(h5_id,"a/my_data"))
    std::cout << "\nPath 'myoutput.hdf5:a/my_data' exists" << std::endl;

Node n_read;
// read a subpath (Note: read works like `load_merged`)
conduit::relay::io::hdf5_read(h5_id,"a/my_data",n_read);
std::cout << "\nData loaded:" << std::endl;
n_read.print();

// write more data to the file
n.reset();
// write data (appends data, works like `save_merged`)
// the Node tree needs to be compatible with the existing
// hdf5 state, adding new paths is always fine.
n["a/my_data"] = 3.1415;
n["a/b/c"] = 144;
// lists are also supported
n["a/my_list"].append() = 42.0;
n["a/my_list"].append() = 42;

conduit::relay::io::hdf5_write(n,h5_id);

// check if a subpath of a list exists
if(conduit::relay::io::hdf5_has_path(h5_id,"a/my_list/0"))
    std::cout << "\nPath 'myoutput.hdf5:a/my_list/0' exists" << std::endl;

// Read the entire tree:
n_read.reset();
conduit::relay::io::hdf5_read(h5_id,n_read);
std::cout << "\nData loaded:" << std::endl;
n_read.print();

// other helpers:

// check if a path is a hdf5 file:
if(conduit::relay::io::is_hdf5_file("myoutput.hdf5"))
    std::cout << "\nFile 'myoutput.hdf5' is a hdf5 file" << std::endl;
```

- **Output:**

```
Node to write:

a:
  my_data: 1.0
  b:
    my_string: "value"


Path 'myoutput.hdf5:a/my_data' exists
```

(continues on next page)

(continued from previous page)

```
Data loaded:
1.0

Path 'myoutput.hdf5:a/my_list/0' exists

Data loaded:

a:
  my_data: 3.1415
  b:
    my_string: "value"
    c: 144
  my_list:
    - 42.0
    - 42


File 'myoutput.hdf5' is a hdf5 file
```

HDF5 I/O Options

- **C++ Example:**

```
Node io_about;
conduit::relay::io::about(io_about);
std::cout << "\nRelay I/O Info and Default Options:" << std::endl;
io_about.print();

Node &hdf5_opts = io_about["options/hdf5"];
// change the default chunking threshold to
// a smaller number to enable compression for
// a small array
hdf5_opts["chunking/threshold"]  = 2000;
hdf5_opts["chunking/chunk_size"] = 2000;

std::cout << "\nNew HDF5 I/O Options:" << std::endl;
hdf5_opts.print();
// set options
conduit::relay::io::hdf5_set_options(hdf5_opts);

int num_vals = 5000;
Node n;
n["my_values"].set(DataType::float64(num_vals));

float64 *v_ptr = n["my_values"].value();
for(int i=0; i< num_vals; i++)
{
    v_ptr[i] = float64(i);
}

// save using options
std::cout << "\nsaving data to 'myoutput_chunked.hdf5' " << std::endl;

conduit::relay::io::hdf5_save(n,"myoutput_chunked.hdf5");
```

- **Output:**

```
Relay I/O Info and Default Options:

protocols:
  json: "enabled"
  conduit_json: "enabled"
  conduit_base64_json: "enabled"
  yaml: "enabled"
  conduit_bin: "enabled"
  hdf5: "enabled"
  sidre_hdf5: "enabled"
  conduit_silo: "disabled"
  conduit_silo_mesh: "disabled"
  adios: "disabled"
options:
  hdf5:
    compact_storage:
      enabled: "true"
      threshold: 1024
    chunking:
      enabled: "true"
      threshold: 2000000
      chunk_size: 1000000
      compression:
        method: "gzip"
        level: 5


New HDF5 I/O Options:

compact_storage:
  enabled: "true"
  threshold: 1024
chunking:
  enabled: "true"
  threshold: 2000
  chunk_size: 2000
  compression:
    method: "gzip"
    level: 5


saving data to 'myoutput_chunked.hdf5'
```

You can verify using `h5stat` that the data set was written to the hdf5 file using chunking and compression.

Relay MPI

The Conduit Relay MPI library enables MPI communication using conduit::Node instances as payloads. It provides two categories of functionality: *Known Schema Methods* and *Generic Methods*. These categories balance flexibility and performance tradeoffs. In all cases the implementation tries to avoid unnecessary reallocation, subject to the constraints of MPI's API input requirements.

Known Schema Methods

Methods that transfer a Node's data, assuming the schema is known. They assume that Nodes used for output are implicitly **compatible** with their sources.

Supported MPI Primitives:

- send/recv
- isend/irecv
- reduce/all_reduce
- broadcast
- gather/all_gather

For both point to point and collectives, here is the basic logic for how input Nodes are treated by these methods:

- For Nodes holding data to be sent:
- If the Node is compact and contiguously allocated, the Node's pointers are passed directly to MPI.
- If the Node is not compact or not contiguously allocated, the data is compacted to temporary contiguous buffers that are passed to MPI.
- For Nodes used to hold output data:
- If the output Node is compact and contiguously allocated, the Node's pointers are passed directly to MPI.
- If the output Node is not compact or not contiguously allocated, a Node with a temporary contiguous buffer is created and that buffer is passed to MPI. An **update** call is used to copy out the data from the temporary buffer to the output Node. This avoids re-allocation and modifying the schema of the output Node.

Generic Methods

Methods that transfer both a Node's data and schema. These are useful for generic messaging, since the schema does not need to be known by receiving tasks. The semantics of MPI place constraints on what can be supported in this category.

Supported MPI Primitives:

- send/recv
- gather/all_gather
- broadcast

Unsupported MPI Primitives:

- isend/irecv
- reduce/all_reduce

For both point to point and collectives, here is the basic logic for how input Nodes are treated by these methods:

- For Nodes holding data to be sent:
  - If the Node is compact and contiguously allocated:
  - The Node's schema is sent as JSON
  - The Node's pointers are passed directly to MPI
  - If the Node is not compact or not contiguously allocated:

  - The Node is compacted to temporary Node
  - The temporary Node's schema is sent as JSON
  - The temporary Nodes's pointers are passed to MPI
- For Nodes used to hold output data:
- If the output Node is not compatible with the received schema, it is reset using the received schema.
- If the output Node is compact and contiguously allocated, its pointers are passed directly to MPI.
- If the output Node is not compact or not contiguously allocated, a Node with a temporary contiguous buffer is created and that buffer is passed to MPI. An **update** call is used to copy out the data from the temporary buffer to the output Node. This avoids re-allocation and modifying the schema of the output Node.

Python Relay MPI Module

Relay MPI is supported in Python via the conduit.relay.mpi module. Methods take Fortran-style MPI communicator handles which are effectively integers. (We hope to also support direct use of *mpi4py* communicator objects in the future.)

Use the following to get a handle from the *mpi4py* world communicator:

```
from mpi4py import MPI
comm_id   = MPI.COMM_WORLD.py2f()
```

Python Relay MPI Module Examples

Send and Receive Using Schema

- **Python Source:**

```
import conduit
import conduit.relay as relay
import conduit.relay.mpi
from mpi4py import MPI

# Note: example expects 2 mpi tasks

# get a comm id from mpi4py world comm
comm_id   = MPI.COMM_WORLD.py2f()
# get our rank and the comm's size
comm_rank = relay.mpi.rank(comm_id)
comm_size = relay.mpi.size(comm_id)

# send a node and its schema from rank 0 to rank 1
n = conduit.Node()
if comm_rank == 0:
    # setup node to send on rank 0
    n["a/data"]   = 1.0
    n["a/more_data"] = 2.0
    n["a/b/my_string"] = "value"

# show node data on rank 0
if comm_rank == 0:
```

(continues on next page)

(continued from previous page)

```
    print("[rank: {}] sending: {}".format(comm_rank,n.to_yaml()))

if comm_rank == 0:
    relay.mpi.send_using_schema(n,dest=1,tag=0,comm=comm_id)
else:
    relay.mpi.recv_using_schema(n,source=0,tag=0,comm=comm_id)

# show received node data on rank 1
if comm_rank == 1:
    print("[rank: {}] received: {}".format(comm_rank,n.to_yaml()))
```

- **Output:**

```
 [rank: 0] sending:
 a:
   data: 1.0
   more_data: 2.0
   b:
     my_string: "value"

 [rank: 1] received:
 a:
   data: 1.0
   more_data: 2.0
   b:
     my_string: "value"
```

Send and Receive

- **Python Source:**

```
import conduit
import conduit.relay as relay
import conduit.relay.mpi
from mpi4py import MPI

# Note: example expects 2 mpi tasks

# get a comm id from mpi4py world comm
comm_id   = MPI.COMM_WORLD.py2f()
# get our rank and the comm's size
comm_rank = relay.mpi.rank(comm_id)
comm_size = relay.mpi.size(comm_id)

# send data from a node on rank 0 to rank 1
# (both ranks have nodes with compatible schemas)
n = conduit.Node(conduit.DataType.int64(4))
if comm_rank == 0:
    # setup node to send on rank 0
    vals = n.value()
    for i in range(4):
        vals[i] = i * i
```

(continues on next page)

(continued from previous page)

```
# show node data on rank 0
if comm_rank == 0:
    print("[rank: {}] sending: {}".format(comm_rank,n.to_yaml()))

if comm_rank == 0:
    relay.mpi.send(n,dest=1,tag=0,comm=comm_id)
else:
    relay.mpi.recv(n,source=0,tag=0,comm=comm_id)

# show received node data on rank 1
if comm_rank == 1:
    print("[rank: {}] received: {}".format(comm_rank,n.to_yaml()))
```

- **Output:**

```
 [rank: 0] sending: [0, 1, 4, 9]
 [rank: 1] received: [0, 1, 4, 9]
```

Send and Receive

- **Python Source:**

```
import conduit
import conduit.relay as relay
import conduit.relay.mpi
from mpi4py import MPI

# Note: example expects 2 mpi tasks

# get a comm id from mpi4py world comm
comm_id   = MPI.COMM_WORLD.py2f()
# get our rank and the comm's size
comm_rank = relay.mpi.rank(comm_id)
comm_size = relay.mpi.size(comm_id)

# send data from a node on rank 0 to rank 1
# (both ranks have nodes with compatible schemas)
n = conduit.Node(conduit.DataType.int64(4))
if comm_rank == 0:
    # setup node to send on rank 0
    vals = n.value()
    for i in range(4):
        vals[i] = i * i

# show node data on rank 0
if comm_rank == 0:
    print("[rank: {}] sending: {}".format(comm_rank,n.to_yaml()))

if comm_rank == 0:
    relay.mpi.send(n,dest=1,tag=0,comm=comm_id)
else:
    relay.mpi.recv(n,source=0,tag=0,comm=comm_id)
```

(continues on next page)

(continued from previous page)

```
# show received node data on rank 1
if comm_rank == 1:
    print("[rank: {}] received: {}".format(comm_rank,n.to_yaml()))
```

- **Output:**

```
 [rank: 0] sending: [0, 1, 4, 9]
 [rank: 1] received: [0, 1, 4, 9]
```

Sum All Reduce

- **Python Source:**

```
import conduit
import conduit.relay as relay
import conduit.relay.mpi
from mpi4py import MPI

# get a comm id from mpi4py world comm
comm_id   = MPI.COMM_WORLD.py2f()
# get our rank and the comm's size
comm_rank = relay.mpi.rank(comm_id)
comm_size = relay.mpi.size(comm_id)

# gather data all ranks
# (ranks have nodes with compatible schemas)
n = conduit.Node(conduit.DataType.int64(4))
n_res = conduit.Node(conduit.DataType.int64(4))
# data to reduce
vals = n.value()
for i in range(4):
    vals[i] = 1

relay.mpi.sum_all_reduce(n,n_res,comm=comm_id)
# answer should be an array with each value == comm_size
# show result on rank 0
if comm_rank == 0:
    print("[rank: {}] sum reduce result: {}".format(comm_rank,n_res.to_yaml()))
```

- **Output:**

```
 [rank: 0] sum reduce result: [2, 2, 2, 2]
```

Broadcast Using Schema

- **Python Source:**

```
import conduit
import conduit.relay as relay
import conduit.relay.mpi
from mpi4py import MPI

# Note: example expects 2 mpi tasks

# get a comm id from mpi4py world comm
comm_id   = MPI.COMM_WORLD.py2f()
# get our rank and the comm's size
comm_rank = relay.mpi.rank(comm_id)
comm_size = relay.mpi.size(comm_id)

# send a node and its schema from rank 0 to rank 1
n = conduit.Node()
if comm_rank == 0:
    # setup node to broadcast on rank 0
    n["a/data"]   = 1.0
    n["a/more_data"] = 2.0
    n["a/b/my_string"] = "value"

# show node data on rank 0
if comm_rank == 0:
    print("[rank: {}] broadcasting: {}".format(comm_rank,n.to_yaml()))

relay.mpi.broadcast_using_schema(n,root=0,comm=comm_id)

# show received node data on rank 1
if comm_rank == 1:
    print("[rank: {}] received: {}".format(comm_rank,n.to_yaml()))
```

- **Output:**

```
 [rank: 0] broadcasting:
 a:
   data: 1.0
   more_data: 2.0
   b:
     my_string: "value"

 [rank: 1] received:
 a:
   data: 1.0
   more_data: 2.0
   b:
     my_string: "value"
```

Broadcast

- **Python Source:**

```
import conduit
import conduit.relay as relay
```

(continues on next page)

(continued from previous page)

```
import conduit.relay.mpi
from mpi4py import MPI

# Note: example expects 2 mpi tasks

# get a comm id from mpi4py world comm
comm_id   = MPI.COMM_WORLD.py2f()
# get our rank and the comm's size
comm_rank = relay.mpi.rank(comm_id)
comm_size = relay.mpi.size(comm_id)

# send data from a node on rank 0 to rank 1
# (both ranks have nodes with compatible schemas)
n = conduit.Node(conduit.DataType.int64(4))
if comm_rank == 0:
    # setup node to send on rank 0
    vals = n.value()
    for i in range(4):
        vals[i] = i * i

# show node data on rank 0
if comm_rank == 0:
    print("[rank: {}] broadcasting: {}".format(comm_rank,n.to_yaml()))

relay.mpi.broadcast_using_schema(n,root=0,comm=comm_id)

# show received node data on rank 1
if comm_rank == 1:
    print("[rank: {}] received: {}".format(comm_rank,n.to_yaml()))
```

- **Output:**

```
 [rank: 0] broadcasting: [0, 1, 4, 9]
```

All Gather Using Schema

- **Python Source:**

```
import conduit
import conduit.relay as relay
import conduit.relay.mpi
from mpi4py import MPI

# get a comm id from mpi4py world comm
comm_id   = MPI.COMM_WORLD.py2f()
# get our rank and the comm's size
comm_rank = relay.mpi.rank(comm_id)
comm_size = relay.mpi.size(comm_id)

n = conduit.Node(conduit.DataType.int64(4))
n_res = conduit.Node()
# data to gather
vals = n.value()
```

(continues on next page)

(continued from previous page)

```
for i in range(4):
    vals[i] = comm_rank

relay.mpi.all_gather_using_schema(n,n_res,comm=comm_id)
# show result on rank 0
if comm_rank == 0:
    print("[rank: {}] all gather using schema result: {}".format(comm_rank,n_res.to_
↪yaml()))
```

- **Output:**

```
[rank: 0] all gather using schema result:
- [0, 0, 0, 0]
- [1, 1, 1, 1]
```

8.2.3 Blueprint

The flexibility of the Conduit Node allows it to be used to represent a wide range of scientific data. Unconstrained, this flexibly can lead to many application specific choices for common types of data that could potentially be shared between applications.

The goal of Blueprint is to help facilite a set of shared higher-level conventions for using Conduit Nodes to hold common simulation data structures. The Blueprint library in Conduit provides methods to verify if a Conduit Node instance conforms to known conventions, which we call **protocols**. It also provides property and transform methods that can be used on conforming Nodes.

For now, Blueprint is focused on conventions for two important types of data:

- Computational Meshes (protocol: `mesh`)

  Many taxonomies and concrete mesh data models have been developed to allow computational meshes to be used in software. Blueprint's conventions for representing mesh data were formed by negotiating with simulation application teams at LLNL and from a survey of existing projects that provide scientific mesh-related APIs including: ADIOS, Damaris, EAVL, MFEM, Silo, VTK, VTKm, and Xdmf. Blueprint's mesh conventions are not a replacement for existing mesh data models or APIs. Our explicit goal is to outline a comprehensive, but small set of options for describing meshes in-core that simplifies the process of adapting data to several existing mesh-aware APIs.

- One-to-Many Relations (protocol: `o2mrelation`)

  A one-to-many relation is a collection of arbitrarily grouped values that encode element associations from a source ("one"s) to a destination ("many"s) space. These constructs are used in computational meshes to represent sparse material data, polygonal/polyhedral topologies, and other non-uniform mappings.

- Multi-Component Arrays (protocol: `mcarray`)

  A multi-component array is a collection of fixed-sized numeric tuples. They are used in the context computational meshes to represent coordinate data or field data, such as the three directional components of a 3D velocity field. There are a few common in-core data layouts used by several APIs to accept multi-component array data, these include: row-major vs column-major layouts, or the use of arrays of struct vs struct of arrays in C-style languages. Blueprint provides transforms that convert any multi-component array to these common data layouts.

- Tabular Data (protocol: `table`)

  A collection of data represented as columns with the same number of rows. Generally used to serialize data in a flattened form, specifically to and from CSV files.

Mesh Blueprint

The Mesh Blueprint is a set of hierarchical conventions to describe mesh-based simulation data both in-memory and via files. This section provides details about the Mesh Blueprint. Lots of them.

These docs provide the main reference for all of the components of the Mesh Blueprint protocol and details about *Mesh Blueprint Examples* that are included in the Conduit Blueprint Library.

Conduit docs don't have a Mesh Blueprint tutorial yet, if you are looking to wrap your mind around the basic mechanics of describing a mesh:

- The Ascent tutorial includes section on creating Meshes using Conduit. This is the best reference for getting started and includes C++ and Python code examples.
- The *Complete Uniform Example* at the end of this section shows you how to create and save a uniform grid to a file which VisIt and Ascent's Replay utility can read.
- The *Mesh Blueprint Examples* section details functions that to generate several flavors of exemplar meshes.

Protocol

The Blueprint protocol defines a single-domain computational mesh using one or more Coordinate Sets (via child `coordsets`), one or more Topologies (via child `topologies`), zero or more Materials Sets (via child `matsets`), zero or more Fields (via child `fields`), optional Adjacency Set information (via child `adjsets`), and optional State information (via child `state`). The protocol defines multi-domain meshes as *Objects* that contain zero or more single-domain mesh entries.

Note: Since the multi-domain protocol accepts zero or more single-domain mesh entries, an empty Conduit Node is considered a valid multi-domain mesh. The change to accept an empty Node was introduced in Conduit 0.8.0. To check if you have a mesh with data, you can screen with *dtype().is_empty()*, or by using mesh blueprint property methods (i.e. *number_of_domains()*).

For simplicity, the descriptions below are structured relative to a single-domain mesh *Object* that contains one Coordinate Set named `coords`, one Topology named `topo`, and one Material Set named `matset`.

Coordinate Sets

To define a computational mesh, the first required entry is a set of spatial coordinate tuples that can underpin a mesh topology.

The mesh blueprint protocol supports sets of spatial coordinates from three coordinate systems:

- Cartesian: {x,y,z}
- Cylindrical: {r,z}
- Spherical: {r,theta,phi}

The mesh blueprint protocol supports three types of Coordinate Sets: `uniform`, `rectilinear`, and `explicit`. To conform to the protocol, each entry under `coordsets` must be an *Object* with entries from one of the cases outlined below:

- **uniform**

  An implicit coordinate set defined as the cartesian product of i,j,k dimensions starting at an `origin` (ex: {x,y,z}) using a given `spacing` (ex: {dx,dy,dz}).

  - Cartesian
    - coordsets/coords/type: "uniform"
    - coordsets/coords/dims/{i,j,k}
    - coordsets/coords/origin/{x,y,z} (optional, default = {0.0, 0.0, 0.0})
    - coordsets/coords/spacing/{dx,dy,dz} (optional, default = {1.0, 1.0, 1.0})
  - Cylindrical
    - coordsets/coords/type: "uniform"
    - coordsets/coords/dims/{i,j}
    - coordsets/coords/origin/{r,z} (optional, default = {0.0, 0.0})
    - coordsets/coords/spacing/{dr,dz} (optional, default = {1.0, 1.0})
  - Spherical
    - coordsets/coords/type: "uniform"
    - coordsets/coords/dims/{i,j}
    - coordsets/coords/origin/{r,theta,phi} (optional, default = {0.0, 0.0, 0.0})
    - coordsets/coords/spacing/{dr,dtheta, dphi} (optional, default = {1.0, 1.0, 1.0})
- **rectilinear**

  An implicit coordinate set defined as the cartesian product of passed coordinate arrays.

  - Cartesian
    - coordsets/coords/type: "rectilinear"
    - coordsets/coords/values/{x,y,z}
  - Cylindrical:
    - coordsets/coords/type: "rectilinear"
    - coordsets/coords/values/{r,z}
  - Spherical
    - coordsets/coords/type: "rectilinear"
    - coordsets/coords/values/{r,theta,phi}
- **explicit**

  An explicit set of coordinates, which includes `values` that conforms to the **mcarray** blueprint protocol.

  - Cartesian
    - coordsets/coords/type: "explicit"
    - coordsets/coords/values/{x,y,z}
  - Cylindrical
    - coordsets/coords/type: "explicit"

    * coordsets/coords/values/{r,z}
  - Spherical
    * coordsets/coords/type: "explicit"
    * coordsets/coords/values/{r,theta,phi}

Note: In all of the coordinate space definitions outlined above, spherical coordinates adhere to the definitions of `theta`/`phi` used in the physics and engineering domains. Specifically, this means that `theta` refers to the polar angle of the coordinate (i.e. the angle from the +Z cartesian axis) and `phi` refers to the azimuthal angle of the coordinate (i.e. the angle from the +X cartesian axis). The figure below most succinctly describes these conventions:

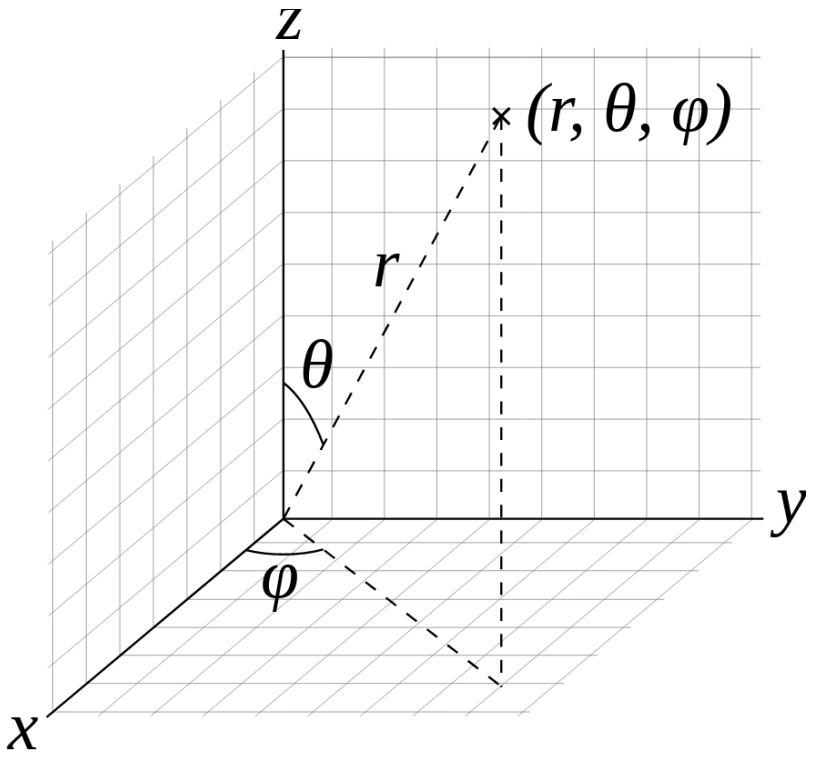

Fig. 1: Figure of `spherical` coordinate conventions (courtesy of Wikipedia)

Toplogies

The next entry required to describe a computational mesh is its topology. To conform to the protocol, each entry under *topologies* must be an *Object* that contains one of the topology descriptions outlined below.

Topology Nomenclature

The mesh blueprint protocol describes meshes in terms of `vertices`, `edges`, `faces`, and `elements`.

The following element shape names are supported:

Name	Geometric Type	Specified By
point	point	an index to a single coordinate tuple
line	line	indices to 2 coordinate tuples
tri	triangle	indices to 3 coordinate tuples
quad	quadrilateral	indices to 4 coordinate tuples
tet	tetrahedron	indices to 4 coordinate tuples
hex	hexahedron	indices to 8 coordinate tuples
polygonal	polygon	indices to N end-to-end coordinate tuples
polyhedral	polyhedron	indices to M polygonal faces

Association with a Coordinate Set

Each topology entry must have a child `coordset` with a string that references a valid coordinate set by name.

- topologies/topo/coordset: "coords"

Optional association with a Grid Function

Topologies can optionally include a child `grid_function` with a string that references a valid field by name.

- topologies/topo/grid_function: "gf"

Implicit Topology

The mesh blueprint protocol accepts four implicit ways to define a topology on a coordinate set. The first simply uses all the points in a given coordinate set and the rest define grids of elements on top of a coordinate set. For the grid cases with a coordinate set with 1D coordinate tuples, *line* elements are used, for sets with 2D coordinate tuples *quad* elements are used, and for 3D coordinate tuples *hex* elements are used.

- **points**: An implicit topology using all of the points in a coordinate set.
  - topologies/topo/coordset: "coords"
  - topologies/topo/type: "points"
- **uniform**: An implicit topology that defines a grid of elements on top of a *uniform* coordinate set.
  - topologies/topo/coordset: "coords"
  - topologies/topo/type: "uniform"
  - topologies/topo/elements/origin/{i,j,k} (optional, default = {0,0,0})
- **rectilinear**: An implicit topology that defines a grid of elements on top of a *rectilinear* coordinate set.
  - topologies/topo/coordset: "coords"
  - topologies/topo/type: "rectilinear"
  - topologies/topo/elements/origin/{i,j,k} (optional, default = {0,0,0})
- **structured**: An implicit topology that defines a grid of elements on top of an *explicit* coordinate set.
  - topologies/topo/coordset: "coords"
  - topologies/topo/type = "structured"
  - topologies/topo/elements/dims/{i,j,k}

- topologies/topo/elements/origin/{i0,j0,k0} (optional, default = {0,0,0})

Explicit (Unstructured) Topology

Single Shape Topologies

For topologies using a homogenous collection of element shapes (eg: all hexs), the topology can be specified by a connectivity array and a shape name.

- topologies/topo/coordset: "coords"
- topologies/topo/type: "unstructured"
- topologies/topo/elements/shape: (shape name)
- topologies/topo/elements/connectivity: (index array)

Mixed Shape Toplogies

For topologies using a non-homogenous collections of element shapes (eg: hexs and tets), the topology can specified using a single shape topology for each element shape.

- **list** - A Node in the *List* role, that contains a children that conform to the *Single Shape Topology* case.
- **object** - A Node in the *Object* role, that contains a children that conform to the *Single Shape Topology* case.

Note: Future version of the mesh blueprint will expand support to include mixed elements types in a single array with related index arrays.

Element Windings

The mesh blueprint does yet not have a prescribed winding convention (a way to order the association of vertices to elements) or more generally to outline a topology's *dimensional cascade* (how elements are related to faces, faces are related to edges, and edges are related to vertices.)

This is a gap we are working to solve in future versions of the mesh blueprint, with a goal of providing transforms to help convert between different winding or cascade schemes.

That said VTK (and VTK-m) winding conventions are assumed by MFEM, VisIt, or Ascent when using Blueprint data.

Polygonal/Polyhedral Topologies

The **polygonal** and **polyhedral** topology shape types are structurally identical to the other explicit topology shape types (see the *Single Shape Topologies* section above), but the contents of their `elements` sections look slightly different. In particular, these sections are structured as **o2mrelation** objects that map elements (the *ones*) to their subelement constituents (the *many*). For **polyhedral** topologies, these constituents reside in an additional `subelements` section that specifies the polyhedral faces in a format identical to `elements` in a **polygonal** schema.

Polygonal Topologies

The schema for a **polygonal** shape topology is as follows:

- topologies/topo/coordset: "coords"
- topologies/topo/type: "unstructured"
- topologies/topo/elements: (o2mrelation object)
- topologies/topo/elements/shape: "polygonal"
- topologies/topo/elements/connectivity: (index array)

It's important to note that the `elements/connectivity` path defines the vertex index sequences (relative to `coordset`) for each element in the topology. These vertex sequences must be arranged end-to-end (i.e. such that `(v[i], v[i+1])` defines an edge) relative to their container polygonal elements.

The following diagram illustrates a simple **polygonal** topology:

```
#
#     4--------5
#     |`--      |
# e1  |    `.    | e0
#     |      --.|
#     7--------6
#

topologies:
  topology:
    coordset: coords
    type: unstructured
    elements:
      shape: polygonal
      connectivity: [4, 6, 5, 7, 6, 4]
      sizes: [3, 3]
      offsets: [0, 3]
```

Polyhedral Topologies

The schema for a **polyhedral** shape topology is as follows:

- topologies/topo/coordset: "coords"
- topologies/topo/type: "unstructured"
- topologies/topo/elements: (o2mrelation object)
- topologies/topo/elements/shape: "polyhedral"
- topologies/topo/elements/connectivity: (index array)
- topologies/topo/subelements: (o2mrelation object)
- topologies/topo/subelements/shape: (shape name)
- topologies/topo/subelements/connectivity: (index array)

An important nuance to the structure of a **polyhedral** shape topology is that the `elements/connectivity` path indexes into the `subelements` object to list the *many* faces associated with each *one* polyhedron. Similarly, the `subelements/connectivity` path indexes into the `coordset` path to list the *many* vertices associated with

each *one* polyhedral face. There is no assumed ordering for constituent polyhedral faces relative to their source polyhedra.

The following diagram illustrates a simple **polyhedral** topology:

```
#
#          0
#         /|\
#        / | \ <- e0
#       /  |  \
#      /_.-3-._\
#    1.,  |  ,.4
#      \ `'2'` /
#       \  |  /
# e1 -> \ | /
#         \|/
#          5
#|

topologies:
  topology:
    coordset: coords
    type: unstructured
    elements:
      shape: polyhedral
      connectivity: [0, 1, 2, 3, 4, 0, 5, 6, 7, 8]
      sizes: [5, 5]
      offsets: [0, 5]
    subelements:
      shape: polygonal
      connectivity: [1, 2, 4, 3, 1, 2, 0, 2, 4, 0, 4, 3, 0, 3, 1, 0, 1, 2, 5,
↪ 2, 4, 5, 4, 3, 5, 3, 1, 5]
      sizes: [4, 3, 3, 3, 3, 3, 3, 3, 3]
      offsets: [0, 4, 7, 10, 13, 16, 19, 22, 25]
```

Material Sets

Materials Sets contain material name and volume fraction information defined over a specified mesh topology.

A material set is a type of **o2mrelation** that houses per-material, per-element volume fractions that are defined over a referenced source topology. Each material set conforms to a schema variant based on:

- The layout of its per-material buffers.
- The indexing scheme used to associate volume fractions with topological elements.

The options for each of these variants are detailed in the following sections.

Material Set Buffer Variants

Each material set follows one of two variants based on the presented structure of its volume fractions. These variants cover volume fractions presented in a single, unified buffer (called **uni-buffer** presentation) and in multiple, per-material buffers (called **multi-buffer** presentation). Both of these variants and their corresponding schemas are outlined in the subsections below.

Uni-Buffer Material Sets

A **uni-buffer** material set is one that presents all of its volume fraction data in a single data buffer. In this case, the material set schema must include this volume fraction data buffer, a parallel buffer associating each volume with a material identifier, and an *Object* that maps human-readable material names to unique integer material identifiers. Additionally, the top-level of this schema is an **o2mrelation** that sources from the volume fraction/material identifier buffers and targets the material topology. To conform to protocol, each `matsets` child of this type must be an *Object* that contains the following information:

- matsets/matset/topology: "topo"
- matsets/matset/material_map: (object with integer leaves)
- matsets/matset/material_ids: (integer array)
- matsets/matset/volume_fractions: (floating-point array)

The following diagram illustrates a simple **uni-buffer** material set example:

```
#     z0        z1        z2
# +--------+--------+--------+
# | a0     | a1 ___/|        |
# |___-----|----    |   b2   |
# |     b0 |     b1 |        |
# +--------+--------+--------+
#

matsets:
  matset:
    topology: topology
    material_map:
      a: 1
      b: 2
      c: 0
    material_ids: [0, 1, 2, 2, 2, 0, 1, 0]
    volume_fractions: [0, a0, b2, b1, b0, 0, a1, 0]
    sizes: [2, 2, 1]
    offsets: [0, 2, 4]
    indices: [1, 4, 6, 3, 2]
```

Multi-Buffer Material Sets

A **multi-buffer** material set is a material set variant wherein the volume fraction data is split such that one buffer exists per material. The schema for this variant dictates that each material be presented as an *Object* entry of the `volume_fractions` field with the material name as the entry key and the material volume fractions as the entry value. **Multi-buffer** material sets also support an optional `material_map`, which is an *Object* that maps human-readable material names to unique integer material identifiers. If omitted, the map from material names to ids is inferred from the order of the material names in the `volume_fractions` node.

Optionally, the value for each such entry can be specified as an **o2mrelation** instead of a flat array to enable greater specification flexibility. To conform to protocol, each `matsets` child of this type must be an *Object* that contains the following information:

- matsets/matset/topology: "topo"
- matsets/matset/volume_fractions: (object)
- matsets/matset/material_map: (optional, object with integer leaves)

The following diagram illustrates a simple **multi-buffer** material set example:

```
#     z0       z1       z2
# +--------+--------+--------+
# | a0     | a1 ___/|        |
# |___-----|----    |   b2   |
# |     b0 |     b1 |        |
# +--------+--------+--------+
#

matsets:
  matset:
    topology: topology
    volume_fractions:
      a:
        values: [0, 0, 0, a1, 0, a0]
        indices: [5, 3]
      b:
        values: [0, b0, b2, b1, 0]
        indices: [1, 3, 2]
    material_map: # (optional)
      a: 0
      b: 1
```

Material Set Indexing Variants

Material sets can also vary in how volume fractions are associated with topological elements. This associative variance leads to two additional schema variants: **element-dominant** (elements/volumes have the same ordering) and **material-dominant** (elements/volumes have independent orderings). Both of these variants and their corresponding schemas are outlined in the subsections below.

Element-Dominant Material Sets

In an **element-dominant** material set, the volume fraction data order matches the topological element order. In other words, the volume fraction group at `i` (e.g. `matset/volume_fractions/mat[i]`) contains the volume fraction data for topological element `i`. This variant is assumed in all material sets that don't have an `element_ids` child.

The following diagram illustrates a simple **element-dominant** material set example:

```
#     z0       z1       z2
# +--------+--------+--------+
# | a0     | a1 ___/|\___ c2 |
# |___-----|----    |    ----|
# |     b0 |     b1 | b2     |
# +--------+--------+--------+
#

matsets:
  matset:
    topology: topology
    volume_fractions:
      a: [a0, a1, 0]
      b: [b0, b1, b2]
      c: [0, 0, c2]
    material_map: # (optional)
```

(continues on next page)

(continued from previous page)

```
      a: 0
      b: 1
      c: 2
```

Material-Dominant Material Sets

In a **material-dominant** material set, the orders for the volume fractions and topological elements are mismatched and need to be bridged via indirection arrays. For these schemas, the `element_ids` field hosts these indirection arrays per material (with just one indirection array for uni-buffer material sets). In explicit terms, the **material-dominant** volume fraction group at `i` (e.g. `matset/volume_fractions/mat[i]`) contains the volume fraction data for the indirected topological element `i` (e.g. `matset/element_ids/mat[i]`). Complementary to the **element-dominant** variant, the **material-dominant** variant applies to all material sets that have an `element_ids` child.

The following diagram illustrates a simple **material-dominant** material set example:

```
#     z0        z1        z2
# +--------+--------+--------+
# | a0     | a1 ___/|\___ c2 |
# |___-----|----    |    ----|
# |     b0 |     b1 | b2     |
# +--------+--------+--------+
#

matsets:
  matset:
    topology: topology
    volume_fractions:
      a: [a0, a1]
      b: [b0, b1, b2]
      c: [c2]
    element_ids:
      a: [0, 1]
      b: [0, 1, 2]
      c: [2]
    material_map: # (optional)
      a: 0
      b: 1
      c: 2
```

Fields

Fields are used to hold simulation state arrays associated with a mesh topology and (optionally) a mesh material set.

Each field entry can define an **mcarray** of material-independent values and/or an **mcarray** of per-material values. These data arrays must be specified alongside a source space, which specifies the space over which the field values are defined (i.e. a topology for material-independent values and a material set for material-dependent values). Minimally, each field entry must specify one of these data sets, the source space for the data set, an association type (e.g. per-vertex, per-element, or per-grid-function-entity), and a volume scaling type (e.g. volume-dependent, volume-independent). Thus, to conform to protocol, each entry under the `fields` section must be an *Object* that adheres to one of the following descriptions:

- Material-Independent Fields:
  - fields/field/association: “vertex” | “element”

  - fields/field/grid_function: (mfem-style finite element collection name) (replaces "association")
  - fields/field/volume_dependent: "true" | "false"
  - fields/field/topology: "topo"
  - fields/field/values: (mcarray)
- Material-Dependent Fields:
  - fields/field/association: "vertex" | "element"
  - fields/field/grid_function: (mfem-style finite element collection name) (replaces "association")
  - fields/field/volume_dependent: "true" | "false"
  - fields/field/matset: "matset"
  - fields/field/matset_values: (mcarray)
- Mixed Fields:
  - fields/field/association: "vertex" | "element"
  - fields/field/grid_function: (mfem-style finite element collection name) (replaces "association")
  - fields/field/volume_dependent: "true" | "false"
  - fields/field/topology: "topo"
  - fields/field/values: (mcarray)
  - fields/field/matset: "matset"
  - fields/field/matset_values: (mcarray)

Topology Association for Field Values

For implicit topologies, the field values are associated with the topology by fast varying logical dimensions starting with `i`, then `j`, then `k`.

For explicit topologies, the field values are associated with the topology by assuming the order of the field values matches the order the elements are defined in the topology.

Species Sets

Species Sets are a means of representing multi-dimensional per-material quantities, most commonly per-material substance fractions.

Individual Species Sets are entries in the `specsets` section of the Blueprint hierarchy, and these entries are formatted in much the same way as `fields` entries that describe per-material, multi-dimensional fields. Just as with this class of `fields` entries, each `specsets` entry must specify the material set over which it is defined and enumerate its values within an **mcarray** that's organized first by materials (shallower level of nesting) and then by species components (deeper level of nesting). Additionally, like `field` entries, each `specsets` item must indicate a volumetric scaling type (e.g. volume-dependent, volume-independent). To put it in short, each entry in the `specsets` section of the Blueprint hierarchy must be an *Object* that follows this template:

- specsets/specset/volume_dependent: "true" | "false"
- specsets/specset/matset: "matset"
- specsets/specset/matset_values: (mcarray)

Nesting Sets

Nesting Sets are used to represent the nesting relationships between different domains in multi-domain mesh environments. Most commonly, this subset of the Blueprint specification is used for AMR (adaptive mesh refinement) meshes.

Each entry in the Nesting Sets section contains an independent set of nesting relationships between domains in the described mesh. On an individual basis, a nesting set contains a source topology, an element association, and a list of nesting windows. The windows for a particular nesting set describe the topological nesting pattern for a paired set of domains, which includes the ID of the partnered domain, the type of the partnered domain (parent or child), the per-dimension zone ratios of this domain relative to the partnered domain, and the self-relative dimensions and origin (provided in terms of local domain coordinates) of the nesting relationship. The Blueprint schema for each entry in the `nestsets` section matches the following template:

- nestsets/nestset/association: "vertex" | "element"
- nestsets/nestset/topology: "topo"
- nestsets/nestset/windows/window/domain_id: (integer)
- nestsets/nestset/windows/window/domain_type: "parent" | "child"
- nestsets/nestset/windows/window/ratio/{i, j, k}
- nestsets/nestset/windows/window/origin/{i, j, k}
- nestsets/nestset/windows/window/dims/{i, j, k}

Note: Many structured AMR codes use global coordinate identifiers when specifying each window's `origin`. Such coordinates must be transformed to domain-local coordinates to be Blueprint-compliant. Given the global structured origin of a window's associated topology `topo_origin` (which isn't in the Blueprint, but is likely stored somewhere in the client code), the global origin can be transformed into a local origin like so:

```
// 'window_origin': starts out as a global index, but is transformed into
// a domain-local index through this procedure
conduit::Node &window_origin = // path to nestset/windows/window/origin
conduit::Node &topo_origin = // loaded from client code; {i, j, k} structure

conduit::NodeIterator origin_it = window_origin.children();
while(origin_it.has_next())
{
    conduit::Node &window_dim = origin_it.next();
    conduit::Node &topo_dim = topo_origin[origin_it.name()];

    conduit::int64 new_dim_val = window_dim.to_int64() - topo_dim.to_int64();
    conduit::Node &new_dim(conduit::DataType::int64(1), &new_dim_val, true);
    new_dim.to_data_type(window_dim.dtype().id(), window_dim);
}
```

Each domain that contains a Nesting Sets section must also update its State section to include the domain's global nesting level. This additional requirement adds the follow constraint to the `state` section:

- state/level_id: (integer)

Note: The Nesting Sets section currently only supports nesting specifications for structured topologies. There are plans to extend this feature to support unstructured topologies in future versions of Conduit.

Adjacency Sets

Adjacency Sets are used to outline the shared geometry between subsets of domains in multi-domain meshes.

Each entry in the Adjacency Sets section is meant to encapsulate a set of adjacency information shared between domains. Each individual adjacency set contains a source topology, an element association, and a list of adjacency groups. An adjacency set's contained groups describe adjacency information shared between subsets of domains, which is represented by a subset of adjacent neighbor domains IDs and a list of shared element IDs. The fully-defined Blueprint schema for the `adjsets` entries looks like the following:

- adjsets/adjset/association: "vertex" | "element"
- adjsets/adjset/topology: "topo"
- adjsets/adjset/groups/group/neighbors: (integer array)
- adjsets/adjset/groups/group/values: (integer array)

It's important to note that the groups in an Adjacency Set associate across domains based on their names (e.g. `domain0/adjsets/adjset/groups/1` will be associated with `domain*/adjsets/adjset/groups/1`). For data publishers that are agnostic about group names, the `conduit::blueprint::mesh::utils::adjset::canonicalize` utility method can be used to assign cross-domain matching names:

```
conduit::Node &unidomain_mesh = // loaded from the client code
conduit::Node &unidomain_adjset = unidomain_mesh["adjsets"].child(0);
conduit::Node &unidomain_domid = unidomain_mesh["state/domain_id"];

unidomain_domid.print();
// > 0
unidomain_adjset["groups"].print();
// > a:
// >   neighbors: [1, 2, 3]
// >   values: [...]
// > b:
// >   neighbors: [1]
// >   values: [...]
// > c:
// >   neighbors: [2]
// >   values: [...]

conduit::bleuprint::mesh::utils::adjset::canonicalize(unidomain_adjset);

unidomain_adjset["groups"].print();
// > group_0_1_2_3:
// >   neighbors: [1, 2, 3]
// >   values: [...]
// > group_0_1:
// >   neighbors: [1]
// >   values: [...]
// > group_0_2:
// >   neighbors: [2]
// >   values: [...]
```

Adjacency Set Variants

There's a great deal of flexibility in how the adjacency groups of an Adjacency Set can be constructed. Blueprint Mesh contains detection and transformation functions for the most commonly targeted formats. The two variants currently

supported are **pairwise** and **max-share**.

Pairwise Adjacency Sets

A **pairwise** adjacency set is one that contains groups that represent the relationship between the host domain and a single neighboring domain (i.e. domain "pairs").

The following diagram illustrates a simple **pairwise** material set example:

```
#  domain0    domain1
# +--------++--------+
# |      v01||v11     |
# |         ||        |
# |      v00||v10     |
# +--------++--------+
# +--------+
# |      v20|
# |         |
# |      v21|
# +--------+
#  domain2

domain0:
  state:
    domain_id: 0
  adjsets:
    adjset:
      association: vertex
      topology: topology
      groups:
        domain_0_1:
          neighbors: [1]
          values: [v00, v01]
        domain_0_2:
          neighbors: [2]
          values: [v00]
```

Max-Share Adjacency Sets

A **max-share** adjacency set is one with groups that "maximally share" index data. In other words, these adjacency sets present index data so that it isn't duplicated between groups.

The following diagram illustrates a simple **pairwise** material set example:

```
#  domain0    domain1
# +--------++--------+
# |      v01||v11     |
# |         ||        |
# |      v00||v10     |
# +--------++--------+
# +--------+
# |      v20|
# |         |
# |      v21|
# +--------+
```

(continues on next page)

(continued from previous page)

```
#  domain2

domain0:
  state:
    domain_id: 0
  adjsets:
    adjset:
      association: vertex
      topology: topology
      groups:
        domain_0_1_2:
          neighbors: [1, 2]
          values: [v00]
        domain_0_1:
          neighbors: [1]
          values: [v01]
```

State

Optional state information is used to provide metadata about the mesh. While the mesh blueprint is focused on describing a single domain of a domain decomposed mesh, the state info can be used to identify a specific mesh domain in the context of a domain decomposed mesh.

To conform, the `state` entry must be an *Object* and can have the following optional entries:

- state/time: (number)
- state/cycle: (number)
- state/domain_id: (integer)

Mesh Blueprint Examples

The C++ `conduit::blueprint::mesh::examples` namespace and the Python `conduit.blueprint.mesh.examples` module provide functions that generate example Mesh Blueprint data. For details on how to write these data sets to files, see the unit tests that exercise these examples in `src/tests/blueprint/t_blueprint_mesh_examples.cpp` and the *mesh output* example below. This section outlines the examples that demonstrate the most commonly used mesh schemas.

basic

The simplest of the mesh examples, `basic()`, generates an homogenous example mesh with a configurable element representation/type (see the `mesh_type` table below) spanned by a single scalar field that contains a unique identifier for each mesh element. The function that needs to be called to generate an example of this type has the following signature:

```
conduit::blueprint::mesh::examples::basic(const std::string &mesh_type, // element␣
↪type/dimensionality
                                          index_t nx,                    // number of␣
↪grid points along x
                                          index_t ny,                    // number of␣
↪grid points along y
```

(continues on next page)

(continued from previous page)

```
                                        index_t nz,                    // number of␣
↪grid points along z (3d only)
                                        Node &res);                    // result␣
↪container
```

The element representation, type, and dimensionality are all configured through the `mesh_type` argument. The supported values for this parameter and their corresponding effects are outlined in the table below:

Mesh Type	Dimensionality	Coordset Type	Topology Type	Element Type
uniform	2d/3d	implicit	implicit	quad/hex
rectilinear	2d/3d	implicit	implicit	quad/hex
structured	2d/3d	explicit	implicit	quad/hex
tris	2d	explicit	explicit	tri
quads	2d	explicit	explicit	quad
polygons	2d	explicit	explicit	polygon
tets	3d	explicit	explicit	tet
hexs	3d	explicit	explicit	hex
polyhedra	3d	explicit	explicit	polyhedron

The remainder of this section demonstrates each of the different `basic()` mesh types, outlining each type with a simple example that (1) presents the generating call, (2) shows the results of the call in Blueprint schema form, and (3) displays the corresponding graphical rendering of this schema.

Uniform

- **Usage Example**

```
// create container node
Node mesh;
// generate simple uniform 2d 'basic' mesh
conduit::blueprint::mesh::examples::basic("uniform", 3, 3, 0, mesh);
// print out results
mesh.print();
```

- **Result**

```
coordsets:
  coords:
    type: "uniform"
    dims:
      i: 3
      j: 3
    origin:
      x: -10.0
      y: -10.0
    spacing:
      dx: 10.0
      dy: 10.0
topologies:
  mesh:
    type: "uniform"
    coordset: "coords"
fields:
```

(continues on next page)

(continued from previous page)

```
  field:
    association: "element"
    topology: "mesh"
    volume_dependent: "false"
    values: [0.0, 1.0, 2.0, 3.0]
```

- **Visual**

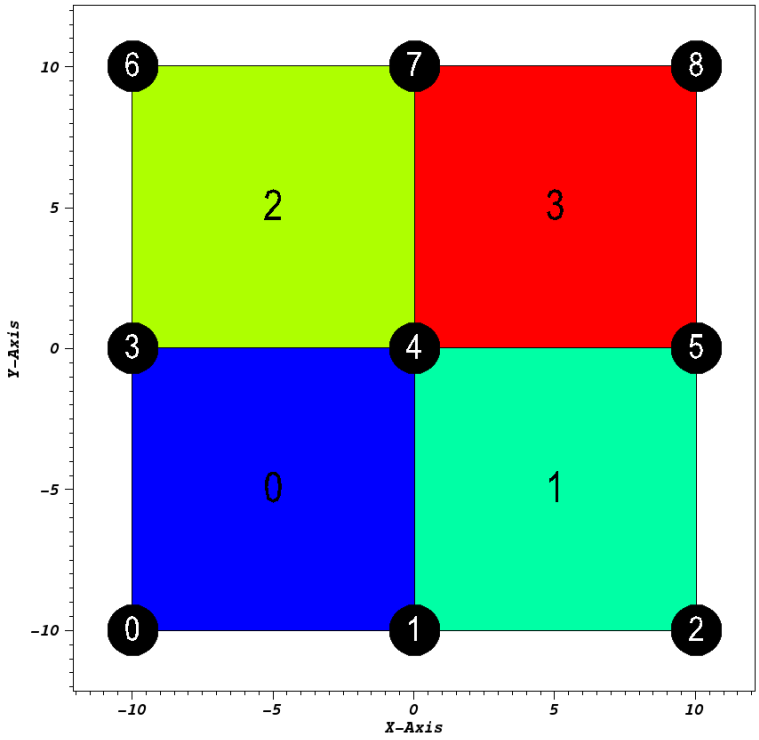

Fig. 2: Pseudocolor plot of `basic` (mesh type 'uniform')

Rectilinear

- **Usage Example**

```
// create container node
Node mesh;
// generate simple rectilinear 2d 'basic' mesh
conduit::blueprint::mesh::examples::basic("rectilinear", 3, 3, 0, mesh);
// print out results
mesh.print();
```

- **Result**

```
coordsets:
  coords:
    type: "rectilinear"
    values:
```

(continues on next page)

(continued from previous page)

```
      x: [-10.0, 0.0, 10.0]
      y: [-10.0, 0.0, 10.0]
topologies:
  mesh:
    type: "rectilinear"
    coordset: "coords"
fields:
  field:
    association: "element"
    topology: "mesh"
    volume_dependent: "false"
    values: [0.0, 1.0, 2.0, 3.0]
```

- **Visual**

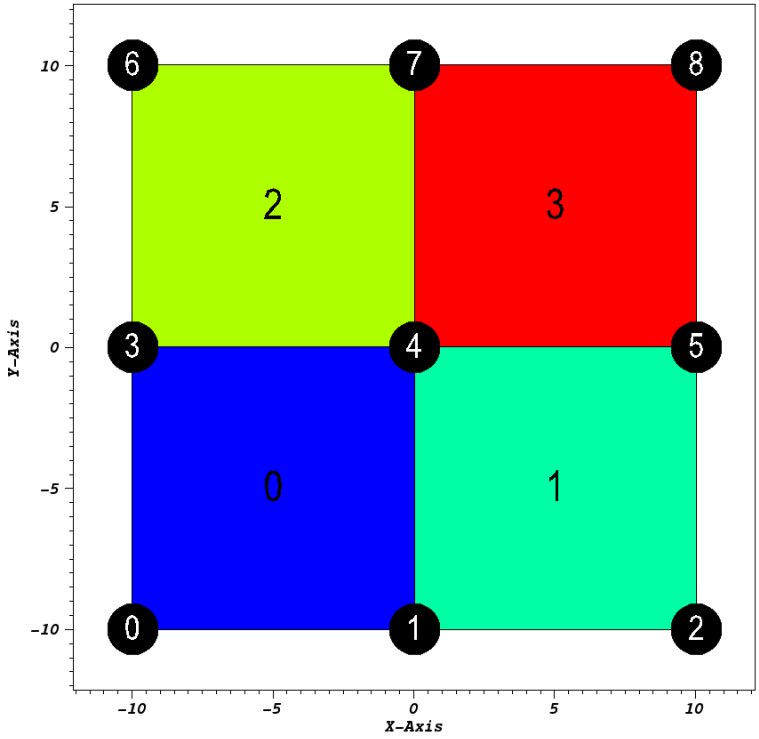

Fig. 3: Pseudocolor plot of `basic` (mesh type 'rectilinear')

Structured

- **Usage Example**

```
// create container node
Node mesh;
// generate simple structured 2d 'basic' mesh
conduit::blueprint::mesh::examples::basic("structured", 3, 3, 1, mesh);
// print out results
mesh.print();
```

- **Result**

```
coordsets:
  coords:
    type: "explicit"
    values:
      x: [-10.0, 0.0, 10.0, -10.0, 0.0, 10.0, -10.0, 0.0, 10.0]
      y: [-10.0, -10.0, -10.0, 0.0, 0.0, 0.0, 10.0, 10.0, 10.0]
topologies:
  mesh:
    type: "structured"
    coordset: "coords"
    elements:
      dims:
        i: 2
        j: 2
fields:
  field:
    association: "element"
    topology: "mesh"
    volume_dependent: "false"
    values: [0.0, 1.0, 2.0, 3.0]
```

- **Visual**

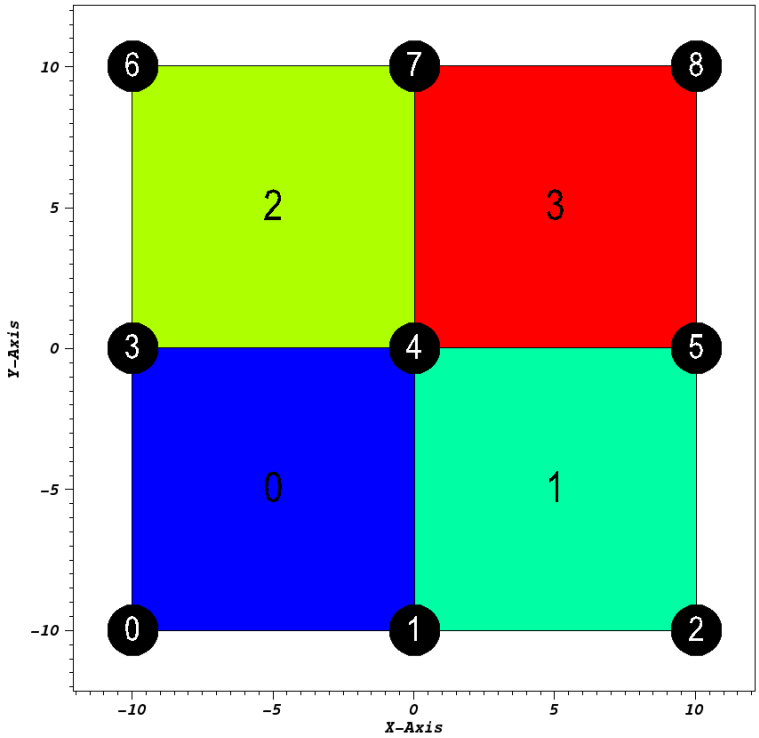

Fig. 4: Pseudocolor plot of `basic` (mesh type 'structured')

Tris

- **Usage Example**

```
// create container node
Node mesh;
// generate simple explicit tri-based 2d 'basic' mesh
conduit::blueprint::mesh::examples::basic("tris", 3, 3, 0, mesh);
// print out results
mesh.print();
```

- **Result**

```
coordsets:
  coords:
    type: "explicit"
    values:
      x: [-10.0, 0.0, 10.0, -10.0, 0.0, 10.0, -10.0, 0.0, 10.0]
      y: [-10.0, -10.0, -10.0, 0.0, 0.0, 0.0, 10.0, 10.0, 10.0]
topologies:
  mesh:
    type: "unstructured"
    coordset: "coords"
    elements:
      shape: "tri"
      connectivity: [0, 3, 4, 0, 1, 4, 1, 4, 5, 1, 2, 5, 3, 6, 7, 3, 4, 7, 4, 7, 8, 4,
↪ 5, 8]
fields:
  field:
    association: "element"
    topology: "mesh"
    volume_dependent: "false"
    values: [0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0]
```

- **Visual**

Quads

- **Usage Example**

```
// create container node
Node mesh;
// generate simple explicit quad-based 2d 'basic' mesh
conduit::blueprint::mesh::examples::basic("quads", 3, 3, 0, mesh);
// print out results
mesh.print();
```

- **Result**

```
coordsets:
  coords:
    type: "explicit"
    values:
      x: [-10.0, 0.0, 10.0, -10.0, 0.0, 10.0, -10.0, 0.0, 10.0]
      y: [-10.0, -10.0, -10.0, 0.0, 0.0, 0.0, 10.0, 10.0, 10.0]
topologies:
```

(continues on next page)

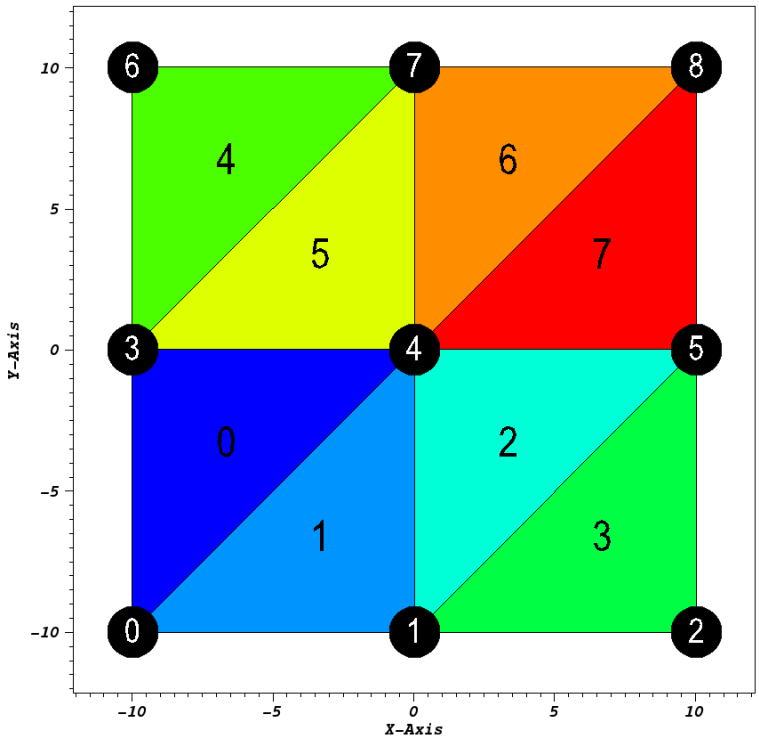

Fig. 5: Pseudocolor plot of `basic` (mesh type 'tris')

(continued from previous page)

```
  mesh:
    type: "unstructured"
    coordset: "coords"
    elements:
      shape: "quad"
      connectivity: [0, 3, 4, 1, 1, 4, 5, 2, 3, 6, 7, 4, 4, 7, 8, 5]
fields:
  field:
    association: "element"
    topology: "mesh"
    volume_dependent: "false"
    values: [0.0, 1.0, 2.0, 3.0]
```

- **Visual**

Polygons

- **Usage Example**

```
// create container node
Node mesh;
// generate simple explicit poly-based 2d 'basic' mesh
conduit::blueprint::mesh::examples::basic("polygons", 3, 3, 0, mesh);
// print out results
mesh.print();
```

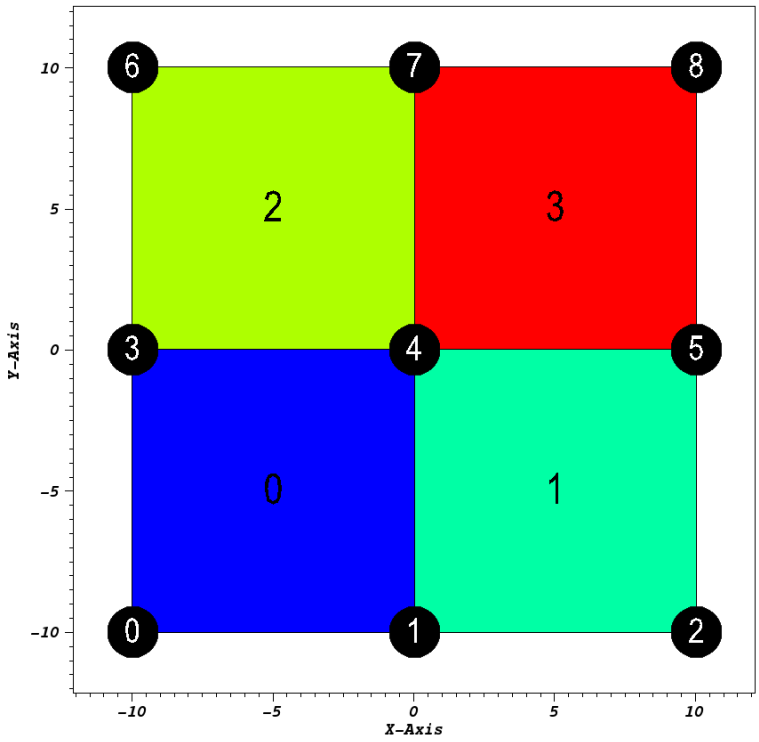

Fig. 6: Pseudocolor plot of `basic` (mesh type 'quads')

- **Result**

```
coordsets:
  coords:
    type: "explicit"
    values:
      x: [-10.0, 0.0, 10.0, -10.0, 0.0, 10.0, -10.0, 0.0, 10.0]
      y: [-10.0, -10.0, -10.0, 0.0, 0.0, 0.0, 10.0, 10.0, 10.0]
topologies:
  mesh:
    type: "unstructured"
    coordset: "coords"
    elements:
      shape: "polygonal"
      sizes: [4, 4, 4, 4]
      connectivity: [0, 3, 4, 1, 1, 4, 5, 2, 3, 6, 7, 4, 4, 7, 8, 5]
      offsets: [0, 4, 8, 12]
fields:
  field:
    association: "element"
    topology: "mesh"
    volume_dependent: "false"
    values: [0.0, 1.0, 2.0, 3.0]
```

- **Visual**

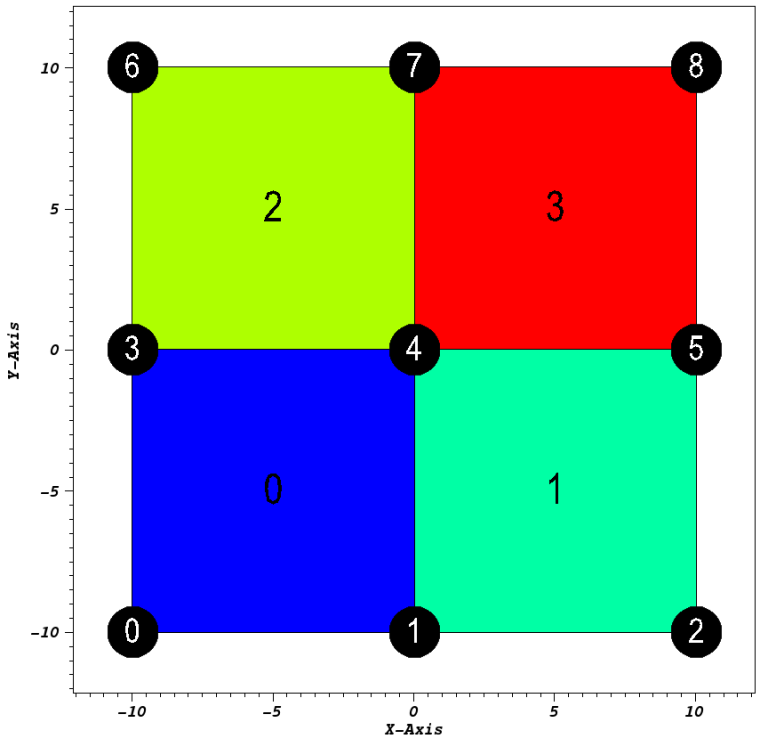

Fig. 7: Pseudocolor plot of `basic` (mesh type 'polygons')

Tets

- **Usage Example**

```
// create container node
Node mesh;
// generate simple explicit tri-based 3d 'basic' mesh
conduit::blueprint::mesh::examples::basic("tets", 3, 3, 3, mesh);
// print out results
mesh.print();
```

- **Result**

```
coordsets:
  coords:
    type: "explicit"
    values:
      x: [-10.0, 0.0, 10.0, -10.0, 0.0, 10.0, -10.0, 0.0, 10.0, -10.0, 0.0, 10.0, -10.
↪0, 0.0, 10.0, -10.0, 0.0, 10.0, -10.0, 0.0, 10.0, -10.0, 0.0, 10.0, -10.0, 0.0, 10.
↪0]
      y: [-10.0, -10.0, -10.0, 0.0, 0.0, 0.0, 10.0, 10.0, 10.0, -10.0, -10.0, -10.0,␣
↪0.0, 0.0, 0.0, 10.0, 10.0, 10.0, -10.0, -10.0, -10.0, 0.0, 0.0, 0.0, 10.0, 10.0, 10.
↪0]
      z: [-10.0, -10.0, -10.0, -10.0, -10.0, -10.0, -10.0, -10.0, -10.0, 0.0, 0.0, 0.
↪0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 10.0, 10.0, 10.0, 10.0, 10.0, 10.0, 10.0, 10.
↪0]
topologies:
```

(continues on next page)

(continued from previous page)

```
  mesh:
    type: "unstructured"
    coordset: "coords"
    elements:
      shape: "tet"
      connectivity: [0, 4, 1, 13, 0, 3, 4, 13, 0, 12, 3, 13, 0, 9, 12, 13, 0, 10, 9,␣
↪13, 0, 1, 10, 13, 1, 5, 2, 14, 1, 4, 5, 14, 1, 13, 4, 14, 1, 10, 13, 14, 1, 11, 10,␣
↪14, 1, 2, 11, 14, 3, 7, 4, 16, 3, 6, 7, 16, 3, 15, 6, 16, 3, 12, 15, 16, 3, 13, 12,␣
↪16, 3, 4, 13, 16, 4, 8, 5, 17, 4, 7, 8, 17, 4, 16, 7, 17, 4, 13, 16, 17, 4, 14, 13,␣
↪17, 4, 5, 14, 17, 9, 13, 10, 22, 9, 12, 13, 22, 9, 21, 12, 22, 9, 18, 21, 22, 9, 19,
↪ 18, 22, 9, 10, 19, 22, 10, 14, 11, 23, 10, 13, 14, 23, 10, 22, 13, 23, 10, 19, 22,␣
↪23, 10, 20, 19, 23, 10, 11, 20, 23, 12, 16, 13, 25, 12, 15, 16, 25, 12, 24, 15, 25,␣
↪12, 21, 24, 25, 12, 22, 21, 25, 12, 13, 22, 25, 13, 17, 14, 26, 13, 16, 17, 26, 13,␣
↪25, 16, 26, 13, 22, 25, 26, 13, 23, 22, 26, 13, 14, 23, 26]
fields:
  field:
    association: "element"
    topology: "mesh"
    volume_dependent: "false"
    values: [0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 11.0, 12.0, 13.0,
↪ 14.0, 15.0, 16.0, 17.0, 18.0, 19.0, 20.0, 21.0, 22.0, 23.0, 24.0, 25.0, 26.0, 27.0,
↪ 28.0, 29.0, 30.0, 31.0, 32.0, 33.0, 34.0, 35.0, 36.0, 37.0, 38.0, 39.0, 40.0, 41.0,
↪ 42.0, 43.0, 44.0, 45.0, 46.0, 47.0]
```

- **Visual**

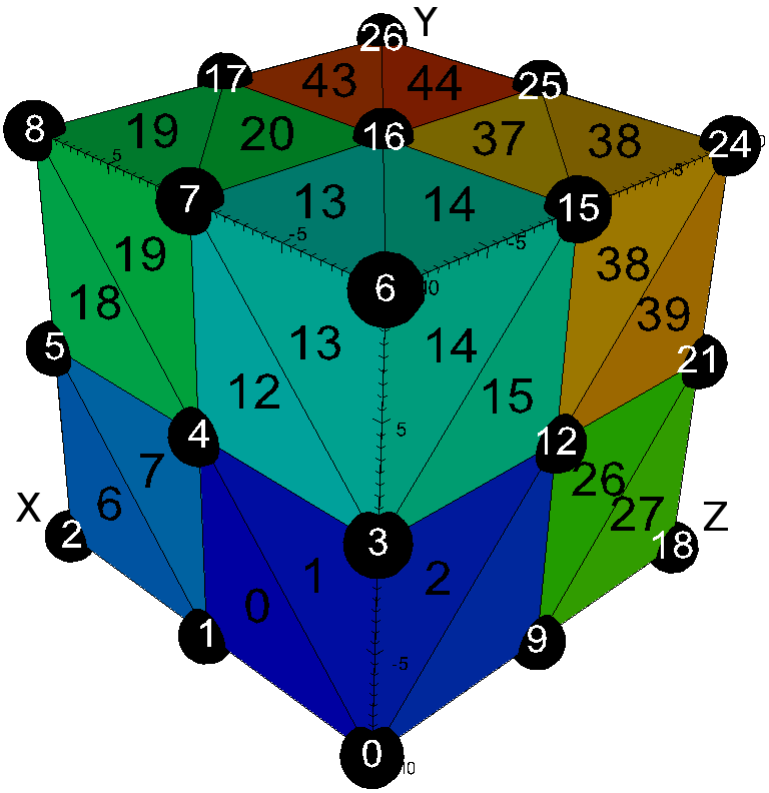

Fig. 8: Pseudocolor plot of `basic` (mesh type 'tets')

Hexs

- **Usage Example**

```
// create container node
Node mesh;
// generate simple explicit quad-based 3d 'basic' mesh
conduit::blueprint::mesh::examples::basic("hexs", 3, 3, 3, mesh);
// print out results
mesh.print();
```

- **Result**

```
coordsets:
  coords:
    type: "explicit"
    values:
      x: [-10.0, 0.0, 10.0, -10.0, 0.0, 10.0, -10.0, 0.0, 10.0, -10.0, 0.0, 10.0, -10.
↪0, 0.0, 10.0, -10.0, 0.0, 10.0, -10.0, 0.0, 10.0, -10.0, 0.0, 10.0, -10.0, 0.0, 10.
↪0]
      y: [-10.0, -10.0, -10.0, 0.0, 0.0, 0.0, 10.0, 10.0, 10.0, -10.0, -10.0, -10.0,␣
↪0.0, 0.0, 0.0, 10.0, 10.0, 10.0, -10.0, -10.0, -10.0, 0.0, 0.0, 0.0, 10.0, 10.0, 10.
↪0]
      z: [-10.0, -10.0, -10.0, -10.0, -10.0, -10.0, -10.0, -10.0, -10.0, 0.0, 0.0, 0.
↪0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 10.0, 10.0, 10.0, 10.0, 10.0, 10.0, 10.0, 10.0, 10.
↪0]
topologies:
  mesh:
    type: "unstructured"
    coordset: "coords"
    elements:
      shape: "hex"
      connectivity: [0, 1, 4, 3, 9, 10, 13, 12, 1, 2, 5, 4, 10, 11, 14, 13, 3, 4, 7,␣
↪6, 12, 13, 16, 15, 4, 5, 8, 7, 13, 14, 17, 16, 9, 10, 13, 12, 18, 19, 22, 21, 10,␣
↪11, 14, 13, 19, 20, 23, 22, 12, 13, 16, 15, 21, 22, 25, 24, 13, 14, 17, 16, 22, 23,␣
↪26, 25]
fields:
  field:
    association: "element"
    topology: "mesh"
    volume_dependent: "false"
    values: [0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0]
```

- **Visual**

Polyhedra

- **Usage Example**

```
// create container node
Node mesh;
// generate simple explicit poly-based 3d 'basic' mesh
conduit::blueprint::mesh::examples::basic("polyhedra", 3, 3, 3, mesh);
// print out results
mesh.print();
```

- **Result**

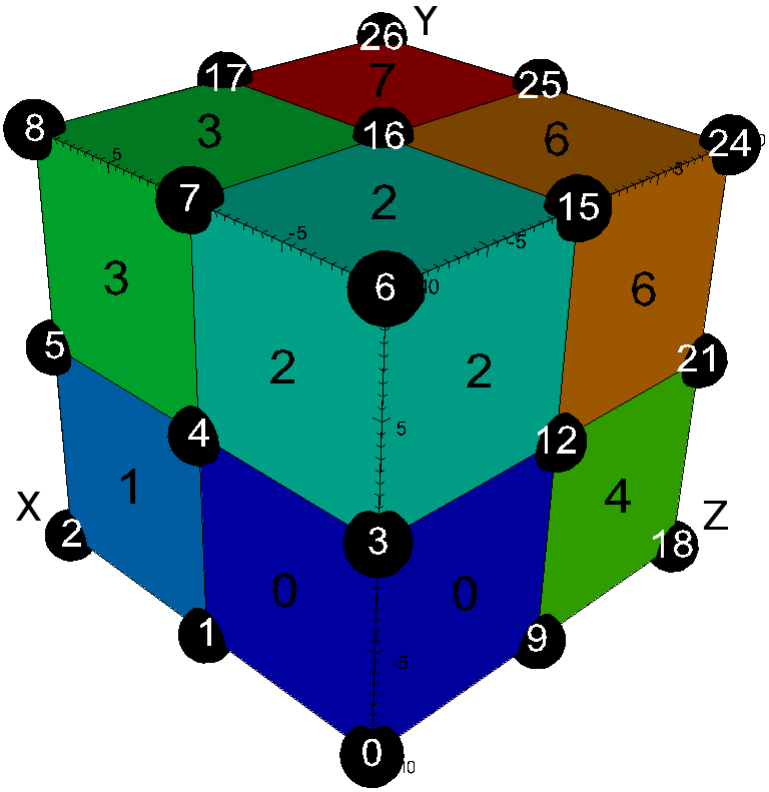

Fig. 9: Pseudocolor plot of `basic` (mesh type 'hexs')

```
coordsets:
  coords:
    type: "explicit"
    values:
      x: [-10.0, 0.0, 10.0, -10.0, 0.0, 10.0, -10.0, 0.0, 10.0, -10.0, 0.0, 10.0, -10.
↪0, 0.0, 10.0, -10.0, 0.0, 10.0, -10.0, 0.0, 10.0, -10.0, 0.0, 10.0, -10.0, 0.0, 10.
↪0]
      y: [-10.0, -10.0, -10.0, 0.0, 0.0, 0.0, 10.0, 10.0, 10.0, -10.0, -10.0, -10.0,␣
↪0.0, 0.0, 0.0, 10.0, 10.0, 10.0, -10.0, -10.0, -10.0, 0.0, 0.0, 0.0, 10.0, 10.0, 10.
↪0]
      z: [-10.0, -10.0, -10.0, -10.0, -10.0, -10.0, -10.0, -10.0, -10.0, 0.0, 0.0, 0.
↪0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 10.0, 10.0, 10.0, 10.0, 10.0, 10.0, 10.0, 10.0, 10.
↪0]
topologies:
  mesh:
    type: "unstructured"
    coordset: "coords"
    elements:
      shape: "polyhedral"
      connectivity: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 2, 10, 11, 3, 12, 13, 14, 15, 16,␣
↪9, 17, 18, 12, 19, 5, 20, 21, 22, 23, 24, 10, 25, 26, 27, 21, 28, 15, 22, 29, 30,␣
↪31, 32, 19, 27, 33, 34, 29, 35]
      sizes: [6, 6, 6, 6, 6, 6, 6, 6]
      offsets: [0, 6, 12, 18, 24, 30, 36, 42]
    subelements:
      shape: "polygonal"
      connectivity: [0, 3, 4, 1, 0, 1, 10, 9, 1, 4, 13, 10, 4, 3, 12, 13, 3, 0, 9, 12,
↪ 9, 10, 13, 12, 1, 4, 5, 2, 1, 2, 11, 10, 2, 5, 14, 11, 5, 4, 13, 14, 10, 11, 14,␣
↪13, 3, 6, 7, 4, 4, 7, 16, 13, 7, 6, 15, 16, 6, 3, 12, 15, 12, 13, 16, 15, 4, 7, 8,␣
↪5, 5, 8, 17, 14, 8, 7, 16, 17, 13, 14, 17, 16, 9, 10, 19, 18, 10, 13, 22, 19, 13,␣
↪12, 21, 22, 12, 9, 18, 21, 18, 19, 22, 21, 10, 11, 20, 19, 11, 14, 23, 20, 14, 13,␣
↪22, 23, 19, 20, 23, 22, 13, 16, 25, 22, 16, 15, 24, 25, 15, 12, 21, 24, 21, 22, 25,␣
↪24, 14, 17, 26, 23, 17, 16, 25, 26, 22, 23, 26, 25]
      sizes: [4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,␣
↪4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4]
      offsets: [0, 4, 8, 12, 16, 20, 24, 28, 32, 36, 40, 44, 48, 52, 56, 60, 64, 68,␣
↪72, 76, 80, 84, 88, 92, 96, 100, 104, 108, 112, 116, 120, 124, 128, 132, 136, 140]
fields:
  field:
    association: "element"
    topology: "mesh"
    volume_dependent: "false"
    values: [0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0]
```

- **Visual**

braid

The `braid()` generates example meshes that cover the range of coordinate sets and topologies supported by the Mesh Blueprint.

The example datasets include a vertex-centered scalar field `braid`, an element-centered scalar field `radial` and a vertex-centered vector field `vel`.

```
conduit::blueprint::mesh::examples::braid(const std::string &mesh_type,
                                          index_t nx,
```

(continues on next page)

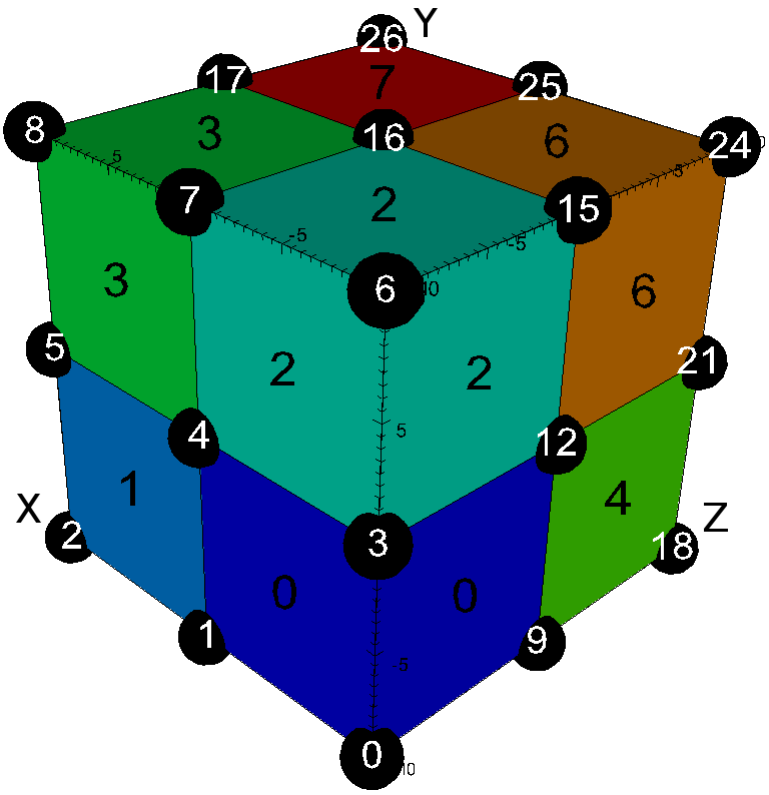

Fig. 10: Pseudocolor plot of `basic` (mesh type 'polyhedra')

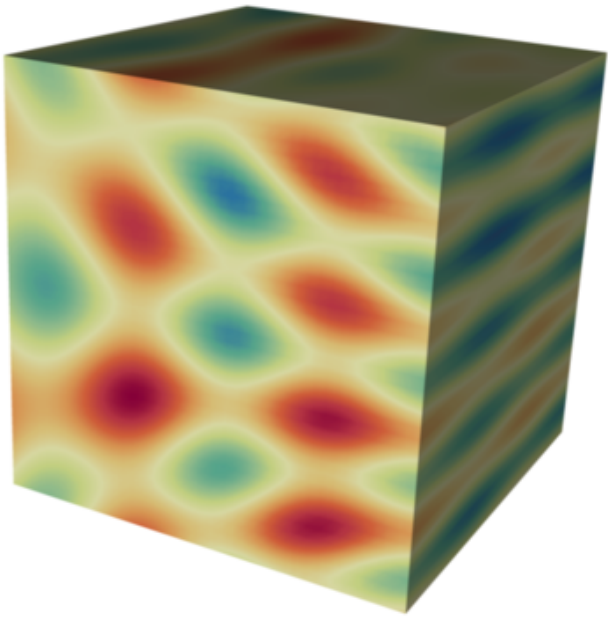

Fig. 11: Pseudocolor plot of a 3D braid example `braid` field

(continued from previous page)

```
                                                index_t ny,
                                                index_t nz,
                                                Node &res);
```

Here is a list of valid strings for the `mesh_type` argument:

Mesh Type	Description
uniform	2d or 3d uniform grid (implicit coords, implicit topology)
rectilinear	2d or 3d rectilinear grid (implicit coords, implicit topology)
structured	2d or 3d structured grid (explicit coords, implicit topology)
point	2d or 3d unstructured mesh of point elements (explicit coords, explicit topology)
lines	2d or 3d unstructured mesh of line elements (explicit coords, explicit topology)
tris	2d unstructured mesh of triangle elements (explicit coords, explicit topology)
quads	2d unstructured mesh of quadrilateral elements (explicit coords, explicit topology)
tets	3d unstructured mesh of tetrahedral elements (explicit coords, explicit topology)
hexs	3d unstructured mesh of hexahedral elements (explicit coords, explicit topology)

`nx`, `ny`, `nz` specify the number of elements in the x, y, and z directions.

`nz` is ignored for 2d-only examples.

The resulting data is placed the Node `res`, which is passed in via reference.

spiral

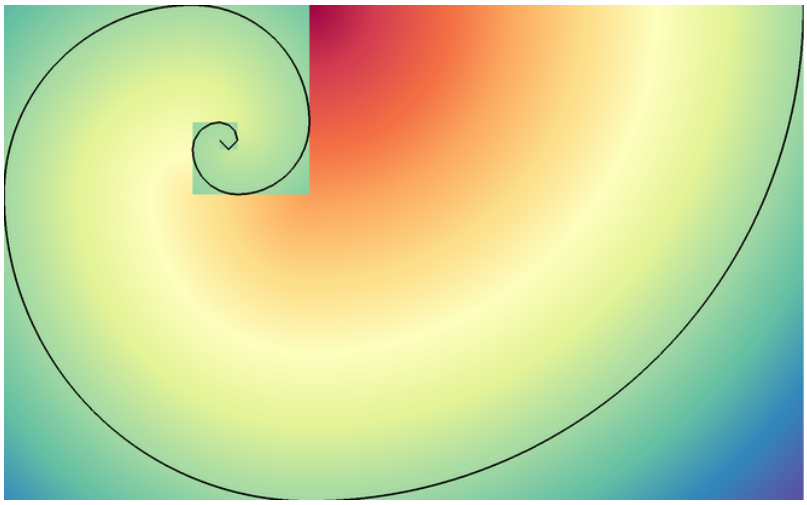

Fig. 12: Pseudocolor and Contour plots of the spiral example `dist` field.

The `sprial()` function generates a multi-domain mesh composed of 2D square domains with the area of successive fibonacci numbers. The result estimates the Golden spiral.

The example dataset provides a vertex-centered scalar field `dist` that estimates the distance from each vertex to the Golden spiral.

```
conduit::blueprint::mesh::examples::spiral(conduit::index_t ndomains,
                                           Node &res);
```

`ndomains` specifies the number of domains to generate, which is also the number of entries from fibonacci sequence used.

The resulting data is placed the Node `res`, which is passed in via reference.

julia

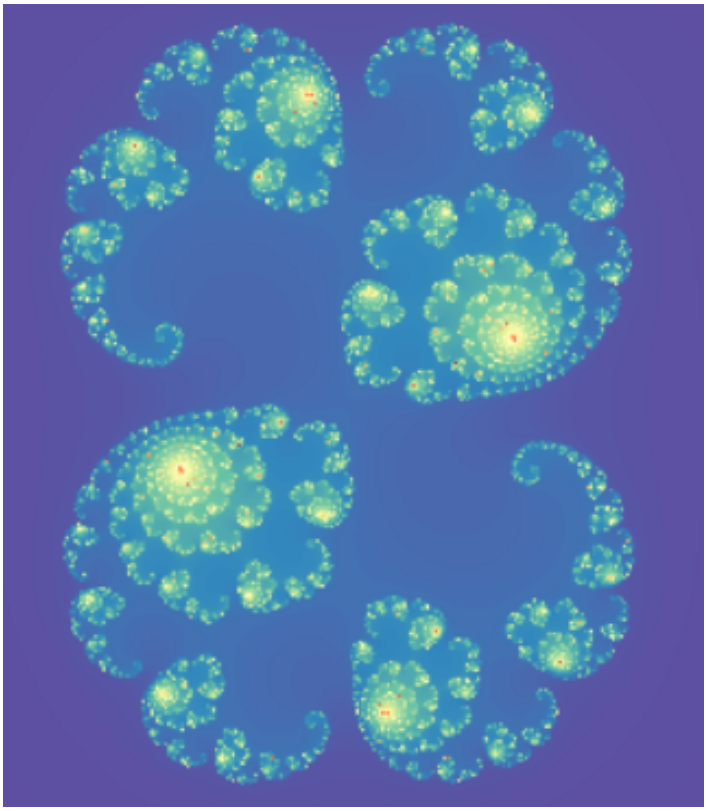

Fig. 13: Pseudocolor plot of the julia example `iter` field

The `julia()` function creates a uniform grid that visualizes Julia set fractals.

The example dataset provides an element-centered scalar field `iter` that represents the number of iterations for each point tested or zero if not found in the set.

```
conduit::blueprint::mesh::examples::julia(index_t nx,
                                          index_t ny,
                                          float64 x_min,
                                          float64 x_max,
                                          float64 y_min,
                                          float64 y_max,
                                          float64 c_re,
                                          float64 c_im,
                                          Node &res);
```

`nx`, `ny` specify the number of elements in the x and y directions.

`x_min`, `x_max`, `y_min`, `y_max` specify the x and y extents.

`c_re`, `c_im` specify real and complex parts of the constant used.

The resulting data is placed the Node `res`, which is passed in via reference.

julia amr examples

We also provide examples that represent the julia set using AMR meshes. These functions provide concrete examples of the Mesh Blueprint *nestset* protocol for patch-based AMR meshes.

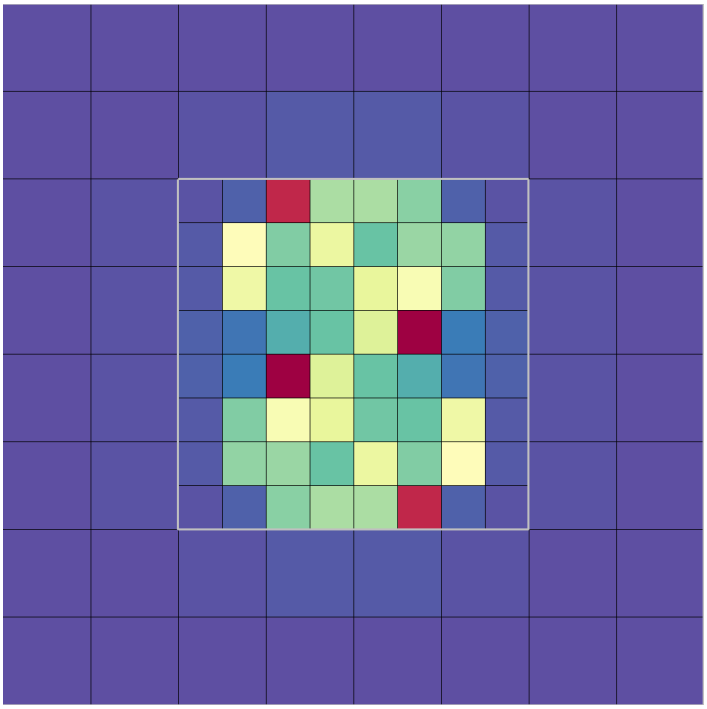

Fig. 14: Pseudocolor, Mesh, and Domain Boundary plots of the julia_nestsets_simple example.

```
conduit::blueprint::mesh::examples::julia_nestsets_simple(float64 x_min,
                                                          float64 x_max,
                                                          float64 y_min,
                                                          float64 y_max,
                                                          float64 c_re,
                                                          float64 c_im,
                                                          Node &res);
```

julia_nestsets_simple provides a basic AMR example with two levels and one parent/child nesting relationship.

`x_min`, `x_max`, `y_min`, `y_max` specify the x and y extents.

`c_re`, `c_im` specify real and complex parts of the constant used.

The resulting data is placed the Node `res`, which is passed in via reference.

```
conduit::blueprint::mesh::examples::julia_nestsets_complex(index_t nx,
                                                           index_t ny,
                                                           float64 x_min,
                                                           float64 x_max,
                                                           float64 y_min,
                                                           float64 y_max,
                                                           float64 c_re,
```

(continues on next page)

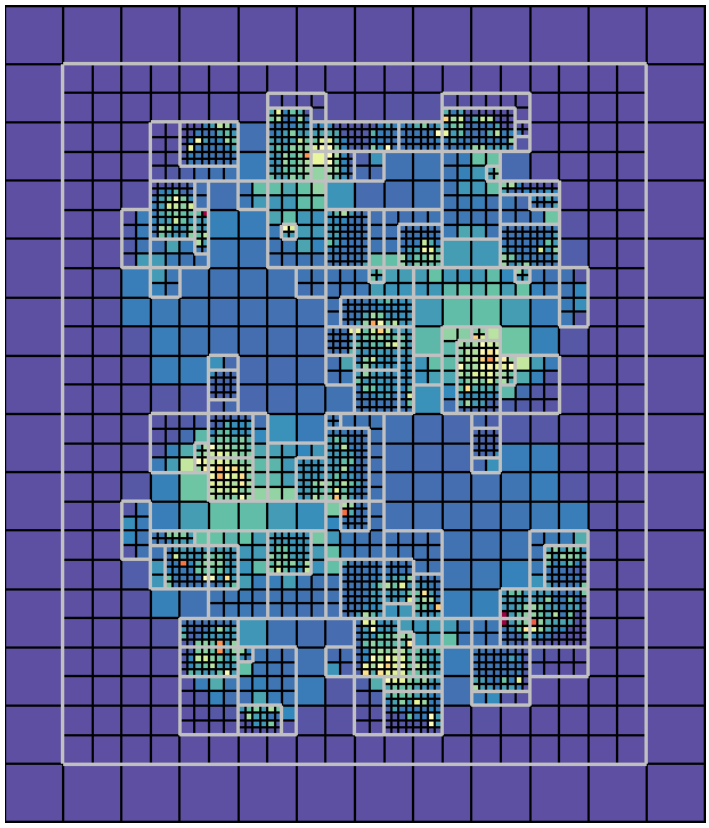

Fig. 15: Pseudocolor, Mesh, and Domain Boundary plots of the julia_nestsets_complex example.

(continued from previous page)

```
                                                    float64 c_im,
                                                    index_t levels,
                                                    Node &res);
```

julia_nestsets_complex provides an AMR example that refines the mesh using more resolution in complex areas.

`nx`, `ny` specify the number of elements in the x and y directions.

`x_min`, `x_max`, `y_min`, `y_max` specify the x and y extents.

`c_re`, `c_im` specify real and complex parts of the constant used.

`levels` specifies the number of refinement levels to use.

The resulting data is placed the Node `res`, which is passed in via reference.

venn

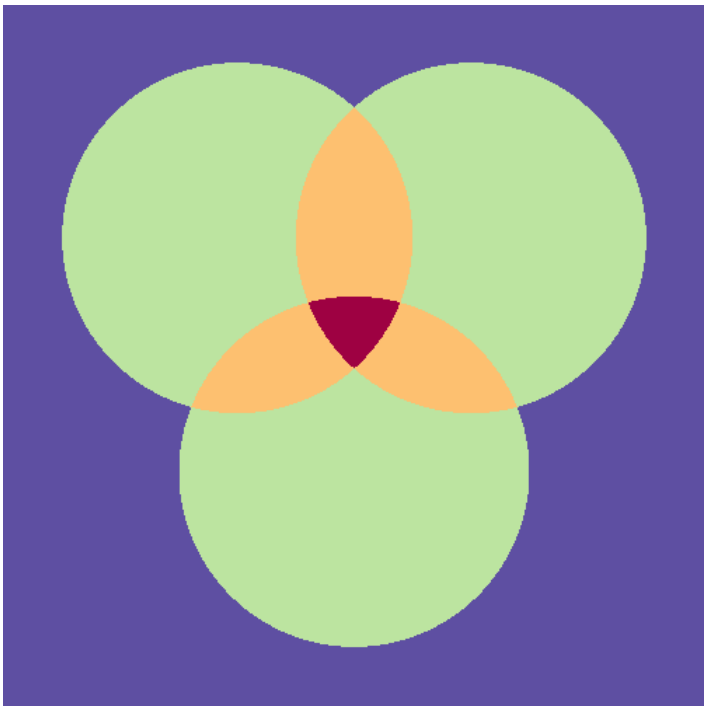

Fig. 16: Pseudocolor plot of the venn example `overlap` field

The `venn()` function creates meshes that use three overlapping circle regions, demonstrating different ways to encode volume fraction based multi-material fields. The volume fractions are provided as both standard fields and using the Material sets (`matsets`) Blueprint. It also creates other fields related to overlap pattern.

```
conduit::blueprint::mesh::examples::venn(const std::string &matset_type,
                                         index_t nx,
                                         index_t ny,
                                         float64 radius,
                                         Node &res);
```

`matset_type` specifies the style of matset generated by the example.

Here is a list of valid strings for the `matset_type` argument:

Matset Type	Description
full	non-sparse volume fractions and matset values
sparse_by_material	sparse (material dominant) volume fractions and matset values
sparse_by_element	sparse (element dominant) volume fractions and matset values

`nx`, `ny` specify the number of elements in the x and y directions.

`radius` specifies the radius of the three circles.

The resulting data is placed the Node `res`, which is passed in via reference.

polytess

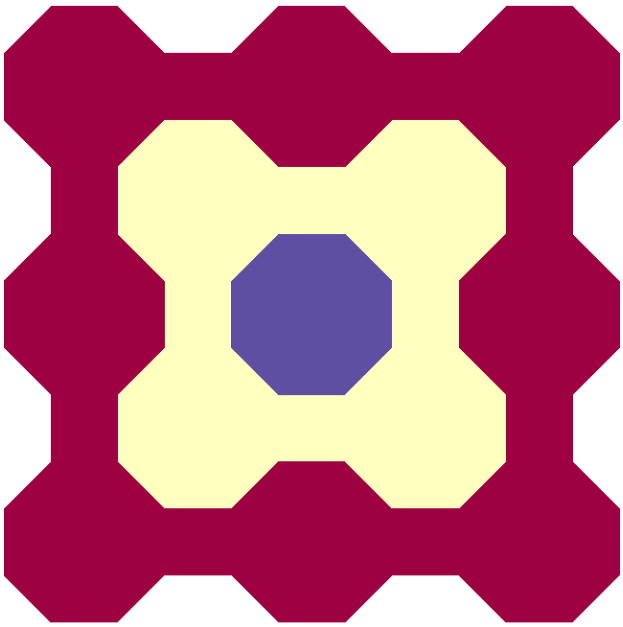

Fig. 17: Pseudocolor plot of the polytess example `level` field, with `nz` = 1.

The `polytess()` function generates a polygonal tessellation in the 2D plane comprised of octagons and squares (known formally as a two-color truncated square tiling). This can be extended into 3D using the `nz` parameter, which, if greater than 1, will stack polytessalations on top of one another as follows: first, a polytess is placed into 3D space, and then a copy of it is placed into a plane parallel to the original. Then "walls" are added, and finally polyhedra are specified that use faces from the original polytess, the reflected copy, and the walls. An `nz` value of 3 or more will simply add layers to this setup, essentially stacking "sheets" of polytess on top of one another.

The scalar element-centered field `level` defined in the result mesh associates each element with its topological distance from the center of the tessellation.

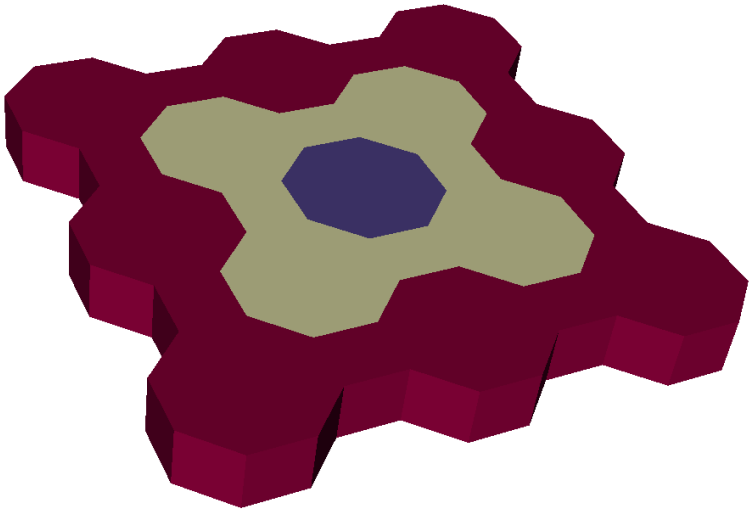

Fig. 18: Pseudocolor plot of the polytess example `level` field, with `nz` = 2.

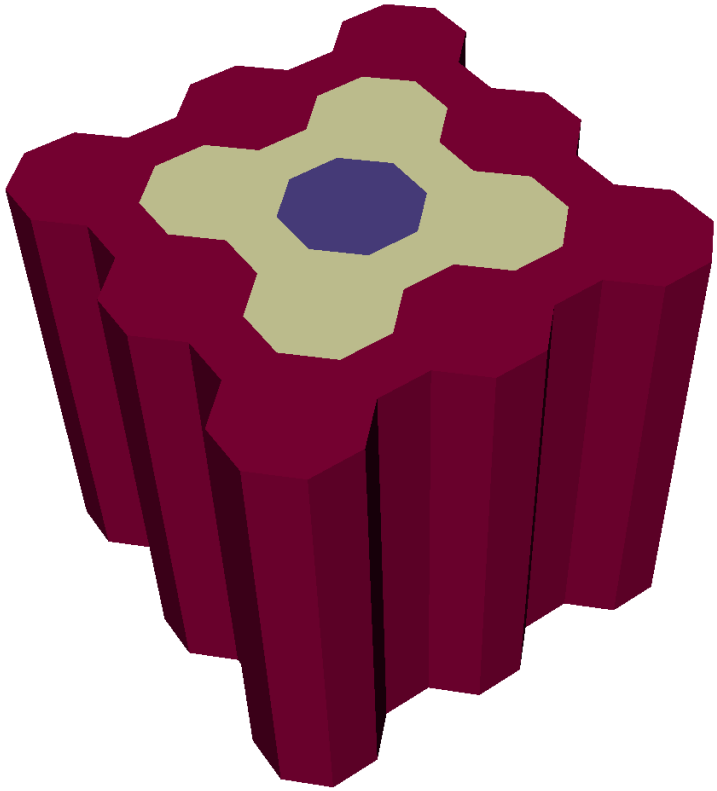

Fig. 19: Pseudocolor plot of the polytess example `level` field, with `nz` = 10.

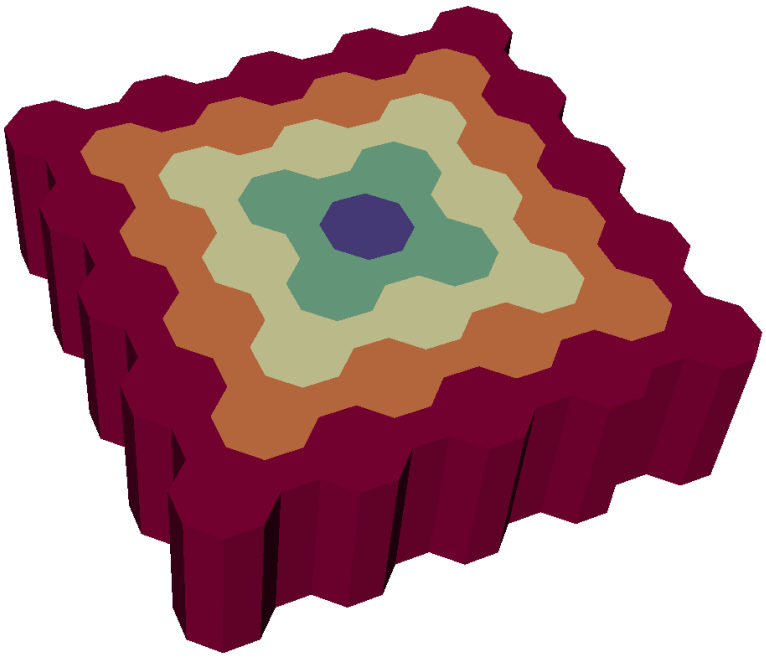

Fig. 20: Pseudocolor plot of the polytess example `level` field, with `nz` = 6.

```
conduit::blueprint::mesh::examples::polytess(index_t nlevels,
                                             index_t nz,
                                             Node &res);
```

`nlevels` specifies the number of tessellation levels/layers to generate. If this value is specified as 1 or less, only the central tessellation level (i.e. the octagon in the center of the geometry) will be generated in the result.

The resulting data is placed the Node `res`, which is passed in via reference.

polychain

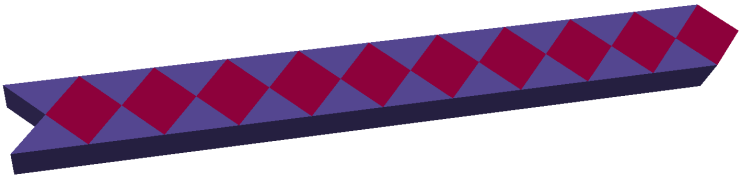

Fig. 21: Pseudocolor plot of the polyhedral chain example `chain` field.

The `polychain()` function generates a chain of cubes and triangular prisms that extends diagonally.

The scalar element-centered field `chain` defined in the result mesh associates with each cube the value 0 and with each triangular prism the value 1.

```
conduit::blueprint::mesh::examples::polychain(const index_t length,
                                              Node &res);
```

`length` specifies how long the chain ought to be. The length is equal to the number of cubes and equal to half the number of prisms.

The resulting data is placed the Node `res`, which is passed in via reference.

miscellaneous

This section doesn't overview any specific example in the `conduit::blueprint::mesh::examples` namespace, but rather provides a few additional code samples to help with various common tasks. Each subsection covers a specific task and presents how it can be accomplished using a function or set of functions in Conduit and/or the Mesh Blueprint library.

Outputting Meshes for Visualization

Suppose that you have an arbitrary Blueprint mesh that you want to output from a running code and subsequently visualize using a visualization tool (e.g. VisIt). You can save your mesh to a set of files, using one of the following `conduit::relay::io::blueprint` library functions:

Save a mesh to disk:

```
conduit::relay::io::blueprint::write_mesh(const conduit::Node &mesh,
                                          const std::string &path);
```

Save a mesh to disk using a specific protocol:

```
conduit::relay::io::blueprint::write_mesh(const conduit::Node &mesh,
                                          const std::string &protocol,
                                          const std::string &path);
```

Save a mesh to disk using a specific protocol and options:

```
/// Options accepted via the `opts` Node argument:
///
///      file_style: "default", "root_only", "multi_file"
///            when # of domains == 1,  "default"   ==> "root_only"
///            else,                    "default"   ==> "multi_file"
///
///      suffix: "default", "cycle", "none"
///            when # of domains == 1,  "default"   ==> "none"
///            else,                    "default"   ==> "cycle"
///
///      mesh_name:  (used if present, default ==> "mesh")
///
///      number_of_files:  {# of files}
///            when "multi_file":
///                 <= 0, use # of files == # of domains
///                  > 0, # of files == number_of_files
///
conduit::relay::io::blueprint::write_mesh(const conduit::Node &mesh,
                                          const std::string &path,
                                          const std::string &protocol,
                                          const conduit::Node &opts);
```

Loading Meshes from Files

If you have a mesh written to a set of blueprint files, you can load them by passing the root file path to the following `conduit::relay::io::blueprint` library functions:

Load a mesh given a root file:

```
conduit::relay::io::blueprint::read_mesh(const std::string &root_file_path,
                                         conduit::Node &mesh);
```

Load a mesh given a root file and options:

```
/// Options accepted via the `opts` Node argument:
///
///      mesh_name: "{name}"
///          provide explicit mesh name, for cases where bp data includes
///           more than one mesh.
///
conduit::relay::io::blueprint::read_mesh(const std::string &root_file_path,
                                         const conduit::Node &opts,
                                         conduit::Node &mesh);
```

Complete Uniform Example

This snippet provides a complete C++ example that demonstrates:

- Describing a single-domain uniform mesh in a Conduit tree
- Verifying the tree conforms to the Mesh Blueprint
- Saving the result to a file that VisIt and Ascent Replay can open

```
// create a Conduit node to hold our mesh data
Node mesh;

// create the coordinate set
mesh["coordsets/coords/type"] = "uniform";
mesh["coordsets/coords/dims/i"] = 3;
mesh["coordsets/coords/dims/j"] = 3;
// add origin and spacing to the coordset (optional)
mesh["coordsets/coords/origin/x"] = -10.0;
mesh["coordsets/coords/origin/y"] = -10.0;
mesh["coordsets/coords/spacing/dx"] = 10.0;
mesh["coordsets/coords/spacing/dy"] = 10.0;

// add the topology
// this case is simple b/c it's implicitly derived from the coordinate set
mesh["topologies/topo/type"] = "uniform";
// reference the coordinate set by name
mesh["topologies/topo/coordset"] = "coords";

// add a simple element-associated field
mesh["fields/ele_example/association"] =  "element";
// reference the topology this field is defined on by name
mesh["fields/ele_example/topology"] =  "topo";
// set the field values, for this case we have 4 elements
mesh["fields/ele_example/values"].set(DataType::float64(4));
```

(continues on next page)

(continued from previous page)

```
float64 *ele_vals_ptr = mesh["fields/ele_example/values"].value();

for(int i=0;i<4;i++)
{
    ele_vals_ptr[i] = float64(i);
}

// add a simple vertex-associated field
mesh["fields/vert_example/association"] =  "vertex";
// reference the topology this field is defined on by name
mesh["fields/vert_example/topology"] =  "topo";
// set the field values, for this case we have 9 vertices
mesh["fields/vert_example/values"].set(DataType::float64(9));

float64 *vert_vals_ptr = mesh["fields/vert_example/values"].value();

for(int i=0;i<9;i++)
{
    vert_vals_ptr[i] = float64(i);
}

// make sure we conform:
Node verify_info;
if(!blueprint::mesh::verify(mesh, verify_info))
{
    std::cout << "Verify failed!" << std::endl;
    verify_info.print();
}

// print out results
mesh.print();

// save our mesh to a file that can be read by VisIt
//
// this will create the file: complete_uniform_mesh_example.root
// which includes the mesh blueprint index and the mesh data
conduit::relay::io::blueprint::save_mesh(mesh,
                                         "complete_uniform_mesh_example",
                                         "json");
```

Expressions (Derived Fields)

An *expression* is a mathematical formula which defines a new field in terms of other fields and/or other expressions. Expressions are specified in the `expressions` section of the Blueprint protocol. The `expressions` section is optional. When it exists, it is a peer to the `fields` section. It is a list of *Objects* of the form:

- expressions/expression/number_of_components
- expressions/expression/topology
- expressions/expression/definition

The `number_of_components` and `topology` entries are identical to their meaning as entries in the `fields` section.

The `definition` entry is string valued and holds the expression (e.g. *mathematical formula*) defining how the new

field is computed. Blueprint does not interpret this string. It simply passes it along for downstream consumers that have the ability to interpret the string and perform the associated operations to compute the expression.

If the expected consumer is VisIt, data producers may wish to consult the Expressions chapter of the VisIt user's manual. In addition, data producers should *escape* all names of fields or expressions by bracketing them in < and > characters. An example expressions entry in the index is

```
"fields":
{
  "braid":
  {
    ...
  },
  "radial":
  {
    ...
  },
"expressions":
{
  "scalar_expr":
  {
    "number_of_components": 1,
    "topology": "mesh",
    "definition": "<vector_expr>[1]"
  },
  "vector_expr":
  {
    "number_of_components": 2,
    "topology": "mesh",
    "definition": "{<braid>,recenter(<radial>,\"nodal\")}"
  }
}
```

O2MRelation Blueprint

Protocol

To conform to the **o2mrelation** protocol, a *Node* must have the following characteristics:

- Be of an *Object* type (*List* types are not allowed)
- Contain at least one child that is a numeric leaf

The numeric leaf/leaves of the *Node* must not be under any of the following "meta component" paths, which all have special meanings and particular requirements when specified as part of an **o2mrelation**:

- `sizes`: An integer leaf that specifies the number "many" items associated with each "one" in the relationship.
- `offsets`: An integer leaf that denotes the start index of the "many" sequence for each "one" in the relationship.
- `indices`: An integer leaf that indicates the index values of items in the values array(s).

All of the above paths are optional and will resolve to simple defaults if left unspecified. These defaults are outlined below:

- `sizes`: An array of ones (i.e. `[1, 1, 1, ...]`) to indicate that the values have one-to-one correspondance.
- `offsets`: An array of monotonically increasing index values (i.e. `[0, 1, 2, ...]`) to indicate that the values are compacted.

- `indices`: An array of monotonically increasing index values (i.e. `[0, 1, 2, ...]`) to indicate that the values are ordered sequentially.

Taken in sum, the consituents of the **o2mrelation** schema describe how data (contained in numeric leaves and indexed through `indices`) maps in grouped clusters (defined by `sizes` and `offsets`) from a source space (the "one" space) to a destination space (the "many" space).

Note: While the `sizes`, `offsets`, and `indices` meta components of the **o2mrelation** definition are independently defined, they interplay in ways that aren't immediately obvious. The most commonly missed of these "gotcha" behaviors are defined below:

- Every **o2mrelation** must define both or neither of `sizes` and `offsets`.
- If none of the meta component paths are specified, their defaults set the **o2mrelation** to be a compacted, one-to-one relationship.
- The `sizes` and `offsets` values always refer to entries in `indices`. If `indices` isn't present, it defaults to a "pass through" index, so in this case `sizes` and `offsets` can be thought of as indexing directly into the numeric leaves.

Properties, Queries, and Transforms

- **conduit::blueprint::o2mrelation::data_paths(const Node &o2mrelation)**

  Returns a `std::vector<std::string>` object containing all of the data paths in the given **o2mrelation** node.

  – Example:

```
// Input //
{
  "values": [int64],
  "sizes": [int64],
  "offsets": [int32],
  "other": [char8]
}

// Output //
["values"]
```

- **conduit::blueprint::o2mrelation::compact_to(const Node &o2mrelation, Node &res)**

  Generates a data-compacted version of the given **o2mrelation** (first parameter) and stores it in the given output node (second parameter).

  – Example:

```
// Input //
{
  "values": [-1, 2, 3, -1, 0, 1, -1],
  "sizes": [2, 2],
  "offsets": [4, 1]
}

// Output //
{
  "values": [0, 1, 2, 3],
```

(continues on next page)

(continued from previous page)

```
  "sizes": [2, 2],
  "offsets": [0, 2]
}
```

- **conduit::blueprint::o2mrelation::generate_offsets(Node &n, Node &info)**

  Updates the contents of the given node's `offsets` child so that it refers to a compacted sequence of one-to-many relationships.

  – Example:

```
// Input //
{
  "values": [0, 1, 2, 3],
  "sizes": [2, 2]
}

// Output //
{
  "values": [0, 1, 2, 3],
  "sizes": [2, 2],
  "offsets": [0, 2]
}
```

O2MRelation Examples

The **o2mrelation** blueprint namespace includes a function *uniform()*, which generates example hierarchies that cover a range of **o2mrelation** use cases.

```
conduit::blueprint::o2mrelation::examples::uniform(conduit::Node &res,
                                                   conduit::index_t nones,
                                                   conduit::index_t nmany = 0,
                                                   conduit::index_t noffset = 0,
                                                   const std::string &index_type =
↪"unspecified");
```

This function's arguments have the following precise meanings:

- `nones`: The number of "one"s in the one-to-many relationship.
- `nmany`: The number of "many"s associated with each of the "one"s.
- `noffset`: The stride between each "many" sequence (must be at least `nmany`).
- `index_type`: The style of element indirection, which must be one of the following:
  - `"unspecified"`: Index indirection will be omitted from the output.
  - `"default"`: The default value for index indirection will be supplied in the output.
  - `"reversed"`: The index indirection will be specified such that the data is reversed relative to its default order.

The `nmany` and `noffset` parameters can both be set to zero to omit the `sizes` and `offsets` meta components from the output. Similarly, the `index_type` parameter can be omitted or set to `"unspecified"` in order to remove the `indices` section from the output.

For more details, see the unit tests that exercise these examples in `src/tests/blueprint/t_blueprint_o2mrelation_examples.cpp`.

MCArray Blueprint

Protocol

To conform to the mcarray blueprint protocol, a Node must have at least one child and:

- All children must be numeric leaves
- All children must have the same number of elements

Properties and Transforms

- **conduit::Node::is_contiguous()** conduit::Node contains a general is_contiguous() instance method that is useful in the context of an mcarray. It can be used to detect if an mcarray has a contiguous memory layout for tuple components (eg: struct of arrays style)
  - Example: {x0, x1, . . . , xN, y0, y1, . . . , yN , z0, z1, . . . , xN}
- **conduit::blueprint::mcarray::is_interleaved(const Node &mcarray)**

  Checks if an mcarray has an interleaved memory layout for tuple components (eg: array of structs style)
  - Example: {x0, y0, z0, x1, y1, z1, . . . , xN, yN, zN}
- **conduit::blueprint::mcarray::to_contiguous(const Node &mcarray, Node &out)**

  Copies the data from an mcarray into a new mcarray with a contiguous memory layout for tuple components
  - Example: {x0, x1, . . . , xN, y0, y1, . . . , yN , z0, z1, . . . , xN}
- **conduit::blueprint::mcarray::to_interleaved(const Node &mcarray, Node &out)**

  Copies the data from an mcarray into a new mcarray with interleaved tuple values
  - Example: {x0, y0, z0, x1, y1, z1, . . . , xN, yN, zN}

MCArray Examples

The mcarray blueprint namespace includes a function *xyz()*, that generates examples that cover a range of mcarray memory layout use cases.

```
conduit::blueprint::mcarray::examples::xyz(const std::string &mcarray_type,
                                           index_t npts,
                                           Node &out);
```

Here is a list of valid strings for the *mcarray_type* argument:

MCArray Type	Description
interleaved	One allocation, using interleaved memory layout with float64 components (array of structs style)
separate	Three allocations, separate float64 components arrays for {x,y,z}
contiguous	One allocation, using a contiguous memory layout with float64 components (struct of arrays style)
interleaved_mixed	**One allocation, using interleaved memory layout with:** • float32 x components • float64 y components • uint8 z components

The number of components per tuple is always three (x,y,z).

npts specifies the number tuples created.

The resulting data is placed the Node *out*, which is passed in via a reference.

For more details, see the unit tests that exercise these examples in `src/tests/blueprint/t_blueprint_mcarray_examples.cpp`.

Table Blueprint

The *table* blueprint protocol provides a convention for expressing tabular data in Conduit. Each data entry in a *table* represents a column and each column contains the same number of rows. Nodes that conform to the *table* blueprint protocol are easily translated to and from CSV files.

Protocol

To conform to the *table* blueprint protocol, a Node must have a "values" child which is a *list* OR an *object* and:

- All of "values" children are data arrays OR *mcarrays*
- All of "values" children must have the same number of elements

A node will also conform to the *table* blueprint protocol if it is a collection of tables. A valid collection of *tables* must be a *list* OR an *object* and:

- All of its children are valid *tables* as defined above.

Table Examples

An example of a *table* blueprint in yaml format:

```
values:
    scalar_column: [0, 1, 2, 3]
    vector_column:
        x: [0, 1, 2, 3]
        y: [0, 1, 2, 3]
        z: [0, 1, 2, 3]
```

An example of a collection of *tables* in yaml format:

```
point_data:
    values:
        points:
            x: [0, 1, 2, 3]
            y: [0, 1, 2, 3]
            z: [0, 1, 2, 3]
        scalar_data: [0, 1, 2, 3]
element_data:
    values:
        scalar_data: [0, 1]
        vector_data:
            a: [0, 1]
            b: [0, 1]
            c: [0, 1]
```

The table blueprint namespace includes a function *basic()*, that generates a simple example of tabular data.

```
conduit::blueprint::table::examples::basic(conduit::index_t nx,
                                           conduit::index_t ny,
                                           conduit::index_t nz,
                                           Node &res);
```

This function will generate points (points/x, points/y, points/z) in a uniform manner based off the arguments *nx*, *ny*, and *nz*. Also included in the output table is a point_data column that starts at 0 and increases by 1 for each point.

The resulting data is placed the Node *res*, which is passed in via a reference.

For more details, see the unit tests that exercise these examples in `src/tests/blueprint/t_blueprint_table_verify.cpp` and `src/tests/blueprint/t_blueprint_table_examples.cpp`.

Partitioning

Partitioning meshes is commonly needed in order to evenly distribute work among many simulation ranks. Blueprint provides two `partition()` functions that can be used to split or recombine Blueprint meshes in serial or parallel. Full M:N repartioning is supported. The `partition()` functions are in the serial and parallel Blueprint libraries, respectively.

```
// Serial
void conduit::blueprint::mesh::partition(const Node &mesh,
                                         const Node &options,
                                         Node &output);

// Parallel
void conduit::blueprint::mpi::mesh::partition(const Node &mesh,
                                              const Node &options,
                                              Node &output,
                                              MPI_Comm comm);
```

Partitioning meshes using Blueprint will use any options present to determine how the partitioning process will behave. Typically, a caller would pass options containing selections if pieces of domains are desired. The partitioner processes any selections and then examines the desired target number of domains and will then decide whether domains must be moved among ranks (only in parallel version) and then locally combined to achieve the target number of domains. The combining process will attempt to preserve the input topology type for the output topology. However, in cases where lower topologies cannot be used, the algorithm will promote the extracted domain parts towards more general topologies and use the one most appropriate to contain the inputs.

In parallel, the `partition()` function will make an effort to redistribute data across MPI ranks to attempt to balance how data are assigned. Domains produced from selections are assigned round-robin across ranks from rank 0 through rank N-1 until all domains have been assigned. This assignment is carried out after extracting selections locally so they can be restributed among ranks before being combined into the target number of domains.

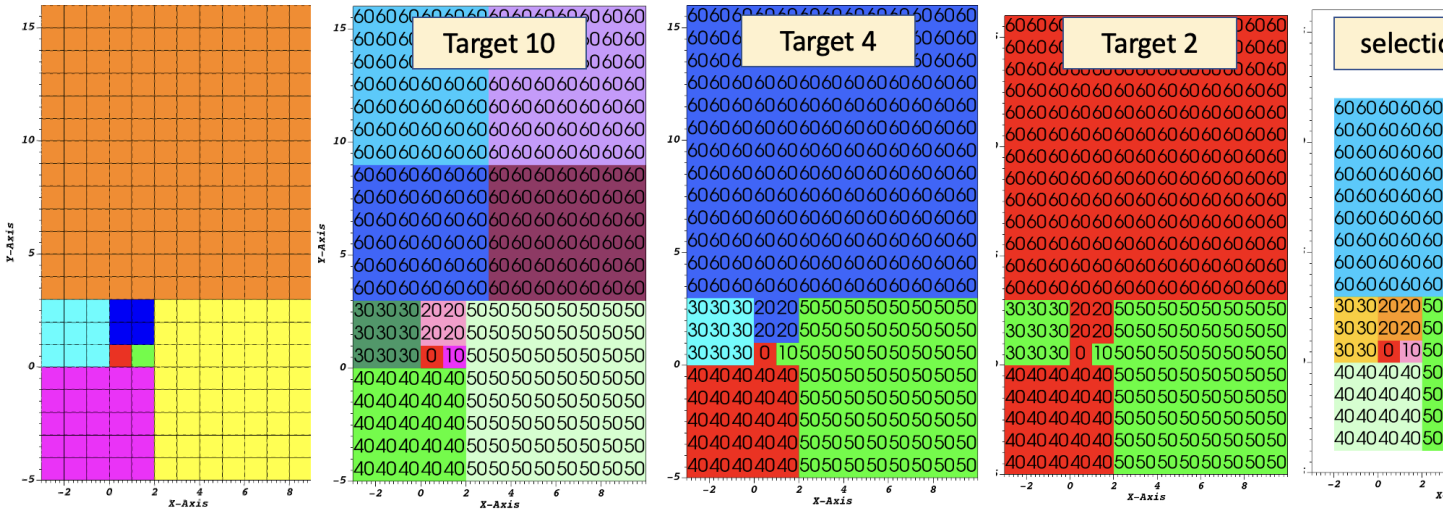

Fig. 22: Partition used to re-partition a 7 domain mesh (left) to different target numbers of domains and to isolate logical subsets.

Options

The `partition()` functions accept a node containing options. The options node can be empty and all options are optional. If no options are given, each input mesh domain will be fully selected. It is more useful to pass selections as part of the option node with additional options that tell the algorithm how to split or combine the inputs. If no selections are present in the options node then the partitioner will create selections of an appropriate type that selects all elements in each input domain.

The `target` option is useful for setting the target number of domains in the final output mesh. If the target value is larger than the number of input domains or selections then the mesh will be split to achieve that target number of domains. This may require further subdividing selections. Alternatively, if the target is smaller than the number of selections then the selections will be combined to yield the target number of domains. The combining is done such that smaller element count domains are combined first.

Option	Description		Example
selections	A list of selection objects that identify regions of interest from the input domains. Selections can be different on each MPI rank.		

```
selections:
  -
    type: logical
    start: [0,0,0]
    end: [9,9,9]
    domain_id: 10
```

If not given,the output will contain the number of selections. In parallel,the largest target value from the ranks will be

used for all ranks.

```
target: 4
```

---

fields An list of strings that indicate the names of the fields to extract in the output. If this option is not provided,all fields will be extracted.

```
fields: ["dist", "pressure"]
```

---

mapping An integer that determines whether fields containing original domains and ids will be added in the output. These fields enable one to know where each vertex and element came from originally. Mapping is on by default. A non-zero value turns it on and a zero value turns it off.

```
mapping: 0
```

---

merge_tolerance A double value that indicates the max allowable distance between 2 points before they are considered to be separate. 2 points spaced smaller than this distance will be merged when explicit coordsets are combined.

```
merge_tolerance: 0.000001
```

Selections

Selections can be specified in the options for the `partition()` function to select regions of interest that will participate in mesh partitioning. If selections are not used then all elements from the input meshes will be selected to partitipate in the partitioning process. Selections can be further subdivided if needed to arrive at the target number of domains. Selections can target specific domains and topologies as well. If a selection does not apply to the input mesh domains then no geometry is produced in the output for that selection.

The `partition()` function's options support 4 types of selections:

Selection Type	Topologies	Description
logical	uniform,rectilinear,structured	Identifies start and end logical IJK ranges to select sub-bricks of uniform, rectilinear, or structured topologies. This selection is not compatible with other topologies.
explicit	all	Identifies an explicit list of element ids and it works with all topologies.
range	all	Identifies ranges of element ids, provided as pairs so the user can select multiple contiguous blocks of elements. This selection works with all topologies
field	all	Uses a specified field to indicate destination domain for each element.

By default, a selection does not apply to any specific domain_id. A list of selections applied to a single input mesh will extract multiple new domains from that original input mesh. Since meshes are composed of many domains in practice, selections can also be associated with certain domain_id values. Selections that provide a domain_id value will only match domains that either have a matching state/domain_id value or match its index in the input node's list of children (if state/domain_id is not present).

Selections can apply to certain topology names as well. By default, the first topology is used but if the `topology` name is provided then the selection will operate on the specified topology only.

Option	Description	Example

type	The selection type	`selections:`
		`  -`
		`  type: logical`

domain_id	The domain_id to which the selection will apply. This is almost always an unsigned integer value.
	For field selections, domain_id is allowed to be a string "any" so a single selection can apply to many domains.
		`selections:`
		`  -`
		`  type: logical`
		`  domain_id: 10`
		
		`selections:`
		`  -`
		`  type: logical`
		`  domain_id: any`

topology	The topology to which the selection will apply.	`selections:`
		`  -`
		`  type: logical`
		`  domain_id: 10`
		`  topology: mesh`

Logical Selection

The logical selection allows the partitioner to extract a logical IJK subset from uniform, rectilinear, or structured topologies. The selection is given as IJK start and end values. If the end values extend beyond the actual mesh's logical extents, they will be clipped. The partitioner may automatically subdivide logical selections into smaller logical selections, if needed, preserving the logical structure of the input topology into the output.

```
selections:
  -
   type: logical
   start: [0,0,0]
   end: [9,9,9]
```

Explicit Selection

The explicit selection allows the partitioner to extract a list of elements. This is used when the user wants to target a specific set of elements. The output will result in an explicit topology.

```
selections:
  -
   type: explicit
   elements: [0,1,2,3,100,101,102]
```

Range Selection

The range selection is similar to the explicit selection except that it identifies ranges of elements using pairs of numbers. The list of ranges must be a multiple of 2 in length. The output will result in an explicit topology.

```
selections:
  -
   type: range
   range: [0,3,100,102]
```

Field Selection

The field selection enables the partitioner to use partitions done by other tools using a field on the mesh as the source of the final domain number for each element. The field must be associated with the mesh elements. When using a field selection, the partitioner will make a best attempt to use the domain numbers to extract mesh pieces and reassemble them into domains with those numberings. If a larger target value is specified, then field selections can sometimes be partitioned further as explicit partitions. The field selection is unique in that its `domain_id` value can be set to "any" if it is desired that the field selection will be applied to all domains in the input mesh. The domain_id value can still be set to specific integer values to limit the set of domains over which the selection will be applied.

```
selections:
  -
   type: field
   domain_id: any
   field: fieldname
```

Top Level Blueprint Interface

Blueprint provides a generic top level `verify()` method, which exposes the verify checks for all supported protocols.

```
bool conduit::blueprint::verify(const std::string &protocol,
                                const Node &node,
                                Node &info);
```

`verify()` returns true if the passed Node *node* conforms to the named protocol. It also provides details about the verification, including specific errors in the passed *info* Node.

```
// setup our candidate and info nodes
Node n, info;

//create an example mesh
conduit::blueprint::mesh::examples::braid("tets",
                                          5,5,5,
                                          n);
// check if n conforms
if(conduit::blueprint::verify("mesh",n,info))
    std::cout << "mesh verify succeeded." << std::endl;
else
    std::cout << "mesh verify failed!" << std::endl;

// show some of the verify details
info["coordsets"].print();
```

```
mesh verify succeeded.

valid: "true"
coords:
  type:
```

(continues on next page)

(continued from previous page)

```
    valid: "true"
  info:
    - "mesh::coordset::explicit: 'type' has valid value 'explicit'"
    - "mesh::coordset::explicit: 'values' is an mcarray"
  values:
    valid: "true"
  valid: "true"
```

Methods for specific protocols are grouped in namespaces:

```
// setup our candidate and info nodes
Node n, verify_info, mem_info;

// create an example mcarray
conduit::blueprint::mcarray::examples::xyz("separate",5,n);

std::cout << "example 'separate' mcarray " << std::endl;
n.print();
n.info(mem_info);
mem_info.print();

// check if n conforms
if(conduit::blueprint::verify("mcarray",n,verify_info))
{
    // check if our mcarray has a specific memory layout
    if(!conduit::blueprint::mcarray::is_interleaved(n))
    {
        // copy data from n into the desired memory layout
        Node xform;
        conduit::blueprint::mcarray::to_interleaved(n,xform);
        std::cout << "transformed to 'interleaved' mcarray " << std::endl;
        xform.print_detailed();
        xform.info(mem_info);
        mem_info.print();
    }
}
```

```
example 'separate' mcarray

x: [1.0, 1.0, 1.0, 1.0, 1.0]
y: [2.0, 2.0, 2.0, 2.0, 2.0]
z: [3.0, 3.0, 3.0, 3.0, 3.0]


mem_spaces:
  0x7f83a7404e10:
    path: "x"
    type: "allocated"
    bytes: 40
  0x7f83a7404f10:
    path: "y"
    type: "allocated"
    bytes: 40
  0x7f83a7405070:
    path: "z"
```

(continues on next page)

(continued from previous page)

```
    type: "allocated"
    bytes: 40
total_bytes_allocated: 120
total_bytes_mmaped: 0
total_bytes_compact: 120
total_strided_bytes: 120

transformed to 'interleaved' mcarray

{
  "x":
{
"dtype":"float64",
"number_of_elements": 5,
"offset": 0,
"stride": 24,
"element_bytes": 8,
"endianness": "little"
, "value": [1.0, 1.0, 1.0, 1.0, 1.0]},
  "y":
{
"dtype":"float64",
"number_of_elements": 5,
"offset": 8,
"stride": 24,
"element_bytes": 8,
"endianness": "little"
, "value": [2.0, 2.0, 2.0, 2.0, 2.0]},
  "z":
{
"dtype":"float64",
"number_of_elements": 5,
"offset": 16,
"stride": 24,
"element_bytes": 8,
"endianness": "little"
, "value": [3.0, 3.0, 3.0, 3.0, 3.0]}
}

mem_spaces:
  0x7f83a7406b60:
    path: ""
    type: "allocated"
    bytes: 120
total_bytes_allocated: 120
total_bytes_mmaped: 0
total_bytes_compact: 120
total_strided_bytes: 312
```

8.2.4 Building

This page provides details on several ways to build Conduit from source.

For the shortest path from zero to Conduit, see *Quick Start*.

If you are building features that depend on third party libraries we recommend using *uberenv* which leverages Spack or *Spack directly*. We also provide info about *building for known HPC clusters using uberenv*. and a *Docker example* that leverages Spack.

Obtain the Conduit source

Clone the Conduit repo from Github:

```
git clone --recursive https://github.com/llnl/conduit.git
```

`--recursive` is necessary because we are using a git submodule to pull in BLT (https://github.com/llnl/blt). If you cloned without `--recursive`, you can checkout this submodule using:

```
cd conduit
git submodule init
git submodule update
```

Configure a build

Conduit uses CMake for its build system. These instructions assume `cmake` is in your path. We recommend CMake 3.9 or newer, for more details see *Supported CMake Versions*.

`config-build.sh` is a simple wrapper for the cmake call to configure conduit. This creates a new out-of-source build directory `build-debug` and a directory for the install `install-debug`. It optionally includes a `host-config.cmake` file with detailed configuration options.

```
cd conduit
./config-build.sh
```

Build, test, and install Conduit:

```
cd build-debug
make -j 8
make test
make install
```

Build Options

The core Conduit library has no dependencies outside of the repo, however Conduit provides optional support for I/O and Communication (MPI) features that require externally built third party libraries.

Conduit's build system supports the following CMake options:

- **BUILD_SHARED_LIBS** - Controls if shared (ON) or static (OFF) libraries are built. *(default = ON)*
- **ENABLE_TESTS** - Controls if unit tests are built. *(default = ON)*
- **ENABLE_EXAMPLES** - Controls if examples are built. *(default = ON)*
- **ENABLE_UTILS** - Controls if utilities are built. *(default = ON)*
- **ENABLE_TESTS** - Controls if unit tests are built. *(default = ON)*
- **ENABLE_DOCS** - Controls if the Conduit documentation is built (when sphinx and doxygen are found). *(default = ON)*
- **ENABLE_COVERAGE** - Controls if code coverage compiler flags are used to build Conduit. *(default = OFF)*

- **ENABLE_PYTHON** - Controls if the Conduit Python module is built. *(default = OFF)*
- **CONDUIT_ENABLE_TESTS** - Extra control for if Conduit unit tests are built. Useful for in cases where Conduit is pulled into a larger CMake project *(default = ON)*

The Conduit Python module can be built for Python 2 or Python 3. To select a specific Python, set the CMake variable **PYTHON_EXECUTABLE** to path of the desired python binary. The Conduit Python module requires Numpy. The selected Python instance must provide Numpy, or PYTHONPATH must be set to include a Numpy install compatible with the selected Python install. Note: You can not use compiled Python modules built with Python 2 in Python 3 and vice versa. You need to compile against the version you expect to use.

- **ENABLE_MPI** - Controls if the conduit_relay_mpi library is built. *(default = OFF)*

  We are using CMake's standard FindMPI logic. To select a specific MPI set the CMake variables **MPI_C_COMPILER** and **MPI_CXX_COMPILER**, or the other FindMPI options for MPI include paths and MPI libraries.

  To run the mpi unit tests on LLNL's LC platforms, you may also need change the CMake variables **MPIEXEC** and **MPIEXEC_NUMPROC_FLAG**, so you can use srun and select a partition. (for an example see: src/host-configs/chaos_5_x86_64.cmake)

Warning: Starting in CMake 3.10, the FindMPI **MPIEXEC** variable was changed to **MPIEXEC_EXECUTABLE**. FindMPI will still set **MPIEXEC**, but any attempt to change it before calling FindMPI with your own cached value of **MPIEXEC** will not survive, so you need to set **MPIEXEC_EXECUTABLE** [reference].

- **HDF5_DIR** - Path to a HDF5 install *(optional)*.

  Controls if HDF5 I/O support is built into *conduit_relay*.

- **SILO_DIR** - Path to a Silo install *(optional)*.

  Controls if Silo I/O support is built into *conduit_relay*. When used, the following CMake variables must also be set:

  - **HDF5_DIR** - Path to a HDF5 install. (Silo support depends on HDF5)

- **ADIOS_DIR** - Path to an ADIOS install *(optional)*.

  Controls if ADIOS I/O support is built into *conduit_relay*. When used, the following CMake variables must also be set:

  - **HDF5_DIR** - Path to a HDF5 install. (ADIOS support depends on HDF5)

- **BLT_SOURCE_DIR** - Path to BLT. *(default = "blt")*

  Defaults to "blt", where we expect the blt submodule. The most compelling reason to override is to share a single instance of BLT across multiple projects.

Installation Path Options

Conduit's build system provides an **install** target that installs the Conduit libraires, headers, python modules, and documentation. These CMake options allow you to control install destination paths:

- **CMAKE_INSTALL_PREFIX** - Standard CMake install path option *(optional)*.
- **PYTHON_MODULE_INSTALL_PREFIX** - Path to install Python modules into *(optional)*.

  When present and **ENABLE_PYTHON** is ON, Conduit's Python modules will be installed to `${PYTHON_MODULE_INSTALL_PREFIX}` directory instead of `${CMAKE_INSTALL_PREFIX}/python-modules`.

Host Config Files

To handle build options, third party library paths, etc we rely on CMake's initial-cache file mechanism.

```
cmake -C config_file.cmake
```

We call these initial-cache files *host-config* files, since we typically create a file for each platform or specific hosts if necessary.

The `config-build.sh` script uses your machine's hostname, the SYS_TYPE environment variable, and your platform name (via *uname*) to look for an existing host config file in the `host-configs` directory at the root of the conduit repo. If found, it passes the host config file to CMake via the *-C* command line option.

```
cmake {other options} -C host-configs/{config_file}.cmake ../
```

You can find example files in the `host-configs` directory.

These files use standard CMake commands. To properly seed the cache, CMake *set* commands need to specify `CACHE` as follows:

```
set(CMAKE_VARIABLE_NAME {VALUE} CACHE PATH "")
```

Building Conduit and Third Party Dependencies

We use **Spack** (http://software.llnl.gov/spack) to help build Conduit's third party dependencies on OSX and Linux. Conduit builds on Windows as well, but there is no automated process to build dependencies necessary to support Conduit's optional features.

Uberenv (`scripts/uberenv/uberenv.py`) automates fetching spack, building and installing third party dependencies, and can optionally install Conduit as well. To automate the full install process, Uberenv uses the Conduit Spack package along with extra settings such as Spack compiler and external third party package details for common HPC platforms.

Building Third Party Dependencies for Development

Note: Conduit developers use `scripts/uberenv/uberenv.py` to setup third party libraries for Conduit development. For info on how to use the Conduit Spack package see *Building Conduit and its Dependencies with Spack*.

On OSX and Linux, you can use `scripts/uberenv/uberenv.py` to help setup your development environment. This script leverages **Spack** to build all of the external third party libraries and tools used by Conduit. Fortran support is optional and all dependencies should build without a fortran compiler. After building these libraries and tools, it writes an initial *host-config* file and adds the Spack built CMake binary to your PATH so can immediately call the `config-build.sh` helper script to configure a conduit build.

```
#build third party libs using spack
python scripts/uberenv/uberenv.py

# run the configure helper script and give it the
# path to a host-config file
./config-build.sh uberenv_libs/`hostname`*.cmake
```

Uberenv Options for Building Third Party Dependencies

`uberenv.py` has a few options that allow you to control how dependencies are built:

Option	Description	Default
–prefix	Destination directory	`uberenv_libs`
–spec	Spack spec	linux: **%gcc** osx: **%clang**
–spack-config-dir	Folder with Spack settings files	linux: (empty) osx: `scripts/uberenv_configs/spack_configs/config/darwin/`
-k	Ignore SSL Errors	**False**
–install	Fully install conduit, not just dependencies	**False**
–run_tests	Invoke tests during build and against install	**False**

The `-k` option exists for sites where SSL certificate interception undermines fetching from github and https hosted source tarballs. When enabled, `uberenv.py` clones spack using:

```
git -c http.sslVerify=false clone https://github.com/llnl/spack.git
```

And passes `-k` to any spack commands that may fetch via https.

Default invocation on Linux:

```
python scripts/uberenv/uberenv.py --prefix uberenv_libs \
                                  --spec %gcc
```

Default invocation on OSX:

```
python scripts/uberenv/uberenv.py --prefix uberenv_libs \
                                  --spec %clang \
                                  --spack-config-dir scripts/uberenv_configs/spack_
↪configs/configs/darwin/
```

The uberenv *–install* installs conduit@develop (not just the development dependencies):

```
python scripts/uberenv/uberenv.py --install
```

To run tests during the build process to validate the build and install, you can use the `--run_tests` option:

```
python scripts/uberenv/uberenv.py --install \
                                  --run_tests
```

For details on Spack's spec syntax, see the Spack Specs & dependencies documentation.

You can edit yaml files under `scripts/uberenv/spack_config/{platform}` or use the **–spack-config-dir** option to specify a directory with compiler and packages yaml files to use with Spack. See the Spack Compiler Configuration and Spack System Packages documentation for details.

For OSX, the defaults in `spack_configs/darwin/compilers.yaml` are X-Code's clang and gfortran from https://gcc.gnu.org/wiki/GFortranBinaries#MacOS.

Note: The bootstrapping process ignores `~/.spack/compilers.yaml` to avoid conflicts and surprises from a user's specific Spack settings on HPC platforms.

When run, `uberenv.py` checkouts a specific version of Spack from github as `spack` in the destination directory. It then uses Spack to build and install Conduit's dependencies into `spack/opt/spack/`. Finally, it generates a host-config file `{hostname}.cmake` in the destination directory that specifies the compiler settings and paths to all of the dependencies.

Building with Uberenv on Known HPC Platforms

To support testing and installing on common platforms, we maintain sets of Spack compiler and package settings for a few known HPC platforms. Here are the commonly tested configurations:

System	OS	Tested Configurations (Spack Specs)
pascal.llnl.gov	Linux: TOSS3	%gcc %gcc~shared
lassen.llnl.gov	Linux: BlueOS	%clang@coral~python~fortran
cori.nersc.gov	Linux: SUSE / CNL	%gcc

See `scripts/spack_build_tests/` for the exact invocations used to test on these platforms.

Building Conduit and its Dependencies with Spack

As of 1/4/2017, Spack's develop branch includes a recipe to build and install Conduit.

To install the latest released version of Conduit with all options (and also build all of its dependencies as necessary) run:

```
spack install conduit
```

To build and install Conduit's github develop branch run:

```
spack install conduit@develop
```

The Conduit Spack package provides several variants that customize the options and dependencies used to build Conduit:

Variant	Description	Default
shared	Build Conduit as shared libraries	ON (+shared)
cmake	Build CMake with Spack	ON (+cmake)
python	Enable Conduit Python support	ON (+python)
mpi	Enable Conduit MPI support	ON (+mpi)
hdf5	Enable Conduit HDF5 support	ON (+hdf5)
silo	Enable Conduit Silo support	ON (+silo)
adios	Enable Conduit ADIOS support	OFF (+adios)
doc	Build Conduit's Documentation	OFF (+docs)

Variants are enabled using + and disabled using ~. For example, to build Conduit with the minimum set of options (and dependencies) run:

```
spack install conduit~python~mpi~hdf5~silo~docs
```

You can specify specific versions of a dependency using ^. For Example, to build Conduit with Python 3:

```
spack install conduit+python ^python@3
```

Supported CMake Versions

We recommend CMake 3.9 or newer. We test building Conduit with CMake 3.9 and 3.14. Other versions of CMake may work, however CMake 3.18.0 and 3.18.1 have known issues that impact HDF5 support. CMake 3.18.2 resolved the HDF5 issues.

Using Conduit in Another Project

Under `src/examples` there are examples demonstrating how to use Conduit in a CMake-based build system (`using-with-cmake`) and via a Makefile (`using-with-make`).

Building Conduit in a Docker Container

Under `src/examples/docker/ubuntu` there is an example `Dockerfile` which can be used to create an ubuntu-based docker image with a build of the Conduit. There is also a script that demonstrates how to build a Docker image from the Dockerfile (`example_build.sh`) and a script that runs this image in a Docker container (`example_run.sh`). The Conduit repo is cloned into the image's file system at `/conduit`, the build directory is `/conduit/build-debug`, and the install directory is `/conduit/install-debug`.

Building Conduit with pip

Conduit provides a setup.py that allows pip to use CMake to build and install Conduit and the Conduit Python module. This script assumes that CMake is in your path.

Example Basic Build:

```
pip install . --user
```

Or for those with certificate woes:

```
pip install  --trusted-host pypi.org --trusted-host files.pythonhosted.org . --user
```

You can enable Conduit features using the following environment variables:

Option	Description	Default
HDF5_DIR	Path to HDF5 install for HDF5 Support	IGNORE
ENABLE_MPI	Build Conduit with MPI Support	OFF

Example Build with MPI and HDF5 Support:

```
env ENABLE_MPI=ON HDF5_DIR={path/to/hdf5/install} pip install . --user
```

Notes for Cray systems

HDF5 and gtest use runtime features such as `dlopen`. Because of this, building static on Cray systems commonly yields the following flavor of compiler warning:

```
Using 'zzz' in statically linked applications requires at runtime the shared␣
↪libraries from the glibc version used for linking
```

You can avoid related linking warnings by adding the `-dynamic` compiler flag, or by setting the CRAYPE_LINK_TYPE environment variable:

```
export CRAYPE_LINK_TYPE=dynamic
```

Shared Memory Maps are read only on Cray systems, so updates to data using `Node::mmap` will not be seen between processes.

Notes for using OpenMPI in a container as root

By default OpenMPI prevents the root user from launching MPI jobs. If you are running as root in a container you can use the following env vars to turn off this restriction:

```
OMPI_ALLOW_RUN_AS_ROOT=1
OMPI_ALLOW_RUN_AS_ROOT_CONFIRM=1
```

8.2.5 Glossary

This page aims to provide succinct descriptions of important concepts in Conduit.

children

Used for Node instances in the *Object* and *List* role interfaces. A Node may hold a set of indexed children (List role), or indexed and named children (Object role). In both of these cases the children of the Node can be accessed, or removed via their index. Methods related to this concept include:

- Node::number_of_children()
- Node::child(index_t)
- Node::child_ptr(index_t)
- Node::operator=(index_t)
- Node::remove(index_t)
- Schema::number_of_children()
- Schema::child(index_t)
- Schema::child_ptr(index_t)
- Schema::operator=(index_t)
- Schema::remove(index_t)

paths

Used for Node instances in *Object* role interface. In the Object role, a Node has a collection of indexed and named children. Access by name is done via a *path*. The path is a forward-slash separated URI, where each segment maps to Node in a hierarchal tree. Methods related to this concept include:

- Node::fetch(string)
- Node::fetch_ptr(string)
- Node::operator=(string)
- Node::has_path(string)
- Node::remove(string)

- Schema::fetch(string)
- Schema::fetch_existing(string)
- Schema::fetch_ptr(string)
- Schema::operator=(string)
- Schema::has_path(string)
- Schema::remove(string)

external

Concept used throughout the Conduit API to specify ownership for passed data. When using Node constructors, Generators, or Node::set calls, you have the option of using an external variant. When external is specified, a Node does not own (allocate or deallocate) the memory for the data it holds.

8.3 Developer Documentation

8.3.1 Source Code Repo Layout

- **src/libs/**
- **conduit/** - Main Conduit library source
- **relay/** - Relay libraries source
- **blueprint/** - Blueprint library source
- **src/tests/**
- **conduit/** - Unit tests for the main Conduit library
- **relay/** - Unit tests for Conduit Relay libraries
- **blueprint/** - Unit tests for Blueprint library
- **thirdparty/** - Unit tests for third party libraries
- **src/examples/** - Basic examples related to building and using Conduit
- **src/docs/** - Documentation
- **src/thirdparty_builtin/** - Third party libraries we build and manage directly

8.3.2 Build System Info

Configuring with CMake

See *Building* in the User Documentation.

Important CMake Targets

- **make**: Builds Conduit.
- **make test**: Runs unit tests.
- **make docs**: Builds sphinx and doxygen documentation.

- **make install**: Installs conduit libraries, headers, and documentation to `CMAKE_INSTALL_PREFIX`

Adding a Unit Test

- Create a test source file in `src/tests/{lib_name}/`
- All test source files should have a `t_` prefix on their file name to make them easy to identify.
- Add the test to build system by editing `src/tests/{lib_name}/CMakeLists.txt`

Running Unit Tests via Valgrind

We can use ctest's built-in valgrind support to check for memory leaks in unit tests. Assuming valgrind is automatically detected when you run CMake to configure conduit, you can check for leaks by running:

```
ctest -D ExperimentalBuild
ctest -D ExperimentalMemCheck
```

The build system is setup to use **src/cmake/valgrind.supp** to filter memcheck results. We don't yet have all spurious issues suppressed, expect to see leaks reported for python and mpi tests.

BLT

Conduit's CMake-based build system uses BLT (https://github.com/llnl/blt).

8.3.3 Git Development Workflow

Conduit's primary source repository and issue tracker are hosted on github:

https://github.com/llnl/conduit

We are using a **Github Flow** model, which is a simpler variant of the confusingly similar sounding **Git Flow** model.

Here are the basics:

- Development is done on topic branches off the develop.
- Merge to develop is only done via a pull request.
- The develop should always compile and pass all tests.
- Releases are tagged off of develop.

More details on GitHub Flow:

https://guides.github.com/introduction/flow/index.html

Here are some other rules to abide by:

- If you have write permissions for the Conduit repo, you *can* merge your own pull requests.
- After completing all intended work on branch, please delete the remote branch after merging to develop. (Github has an option to do this after you merge a pull request.)

8.4 Releases

Source distributions for Conduit releases are hosted on github:

https://github.com/LLNL/conduit/releases

Note: Conduit uses BLT as its core CMake build system. We leverage BLT as a git submodule, however github does not include submodule contents in its automatically created source tarballs. To avoid confusion, starting with v0.3.0 we provide our own source tarballs that include BLT.

8.4.1 v0.8.1

- Source Tarball

Highlights

(Extracted from Conduit's `Changelog`)

Added

- **General**
- Added `CONDUIT_DLL_DIR` env var support on windows, for cases where Conduit DLLs are not installed directly inside the Python Module.
- **Blueprint**
- Allow adjsets to be used in `blueprint::mesh::partition` to determine global vertex ids.
- Added partial matset support to `blueprint::mesh::partition` and `blueprint::mesh::combine`.

Fixed

- **General**
- Fixed CMake bug with `ENABLE_RELAY_WEBSERVER` option.
- Fixed build and test issues with Python >= 3.8 on Windows.
- **Blueprint**
- Fixed a bug in `blueprint::mesh::partition` where adjsets could be missing in new domains.
- Fixed a bug with `blueprint::mesh::matset::to_silo` and uni-buffer matsets.

8.4.2 v0.8.0

- Source Tarball

Highlights

(Extracted from Conduit's `Changelog`)

Added

- **General**
- Added `setup.py` for building and installing Conduit and its Python module via pip
- Added DataAccessor class that helps write generic algorithms that consume data arrays using expected types.
- Added support to register custom memory allocators and a custom data movement handler. This allows conduit to move trees of data between heterogenous memory spaces (e.g. CPU and GPU memory). See conduit_utils.hpp for API details.
- **Blueprint**
- Added `conduit::blueprint::{mpi}::partition` function that provides a general N-to-M partition capability for Blueprint Meshes. This helps with load balancing and other use cases, including fusing multi-domain data to simplifying post processing. This capability supports several options, see (https://llnl-conduit.readthedocs.io/en/latest/blueprint_mesh_partition.html) for more details.
- Added a `Table` blueprint used to represent tables of numeric data. See (https://llnl-conduit.readthedocs.io/en/latest/blueprint_table.html) more details.
- Added `conduit::blueprint::{mpi}::flatten` which transforms Blueprint Meshes into Blueprint Tables. This transforms Mesh Blueprint data into a form that is more easily digestible in machine learning applications.
- Added `conduit::blueprint::mpi::generate_partition_field`, which uses Parmetis to create a field that identifies how to load balance an input mesh elements. This field can be used as a Field selection input to `conduit::blueprint::mpi::partition` function.
- Added the‘‘blueprint::mesh::examples::polychain‘‘ example. It is an example of a polyhedral mesh. See Mesh Blueprint Examples docs (https://llnl-conduit.readthedocs.io/en/latest/blueprint_mesh.html#polychain) for more details.
- Added a new function signature for `blueprint::mesh::topology::unstructured::generate_sides`, which performs the same task as the original and also takes fields from the original topology and maps them onto the new topology.
- Added `blueprint::mpi::mesh::to_polygonal`, which provides a MPI aware conversion Blueprint Structured AMR meshes to a Blueprint Polyhedral meshes.
- Added a host of `conduit::blueprint::mpi::mesh::generate_*` methods, which are the MPI parallel equivalents of the `conduit::blueprint::mesh::topology::unstructured::generate_*` functions.
- Added the `conduit::blueprint::mpi::mesh::find_delegate_domain` function, which returns a single delegate domain for the given mesh across MPI ranks (useful when all ranks need mesh information and some ranks can have empty meshes).
- Added check and transform functions for the newly-designated `pairwise` and `maxshare` variants of `adjsets`. For more information, see the `conduit::blueprint::mesh::adjset` namespace.
- Added `mesh::topology::unstructured::to_polytopal` as an alias to `mesh::topology::unstructured::to_polygonal`, to reflect that both polygonal and polyhedral are supported.

- Added `conduit::blueprint::mpi::mesh::to_polytopal` as an alias to `conduit::blueprint::mpi::mesh::to_polygonal` and `conduit::blueprint::mpi::mesh::to_polyhedral`.
- **Relay**
- Added `conduit::relay::io::hdf5_identifier_report` methods, which create conduit nodes that describes active hdf5 resource handles.

Changed

- **General**
- Updated CMake logic to provide more robust Python detection and better support for HDF5 installs that were built with CMake.
- Improved Node::diff and Node::diff_compatible to show string values when strings differ.
- `conduit::Node::print()` and in Python Node `repr` and `str` now use `to_summary_string()`. This reduces the output for large Nodes. Full output is still supported via `to_string()`, `to_yaml()`, etc methods.
- **Blueprint**
- The `blueprint::mesh::examples::polytess` function now takes a new argument, called `nz`, which allows it to be extended into 3 dimensions. See Mesh Blueprint Examples docs (https://llnl-conduit.readthedocs.io/en/latest/blueprint_mesh.html#polytess) for more details.
- Added support for both `const` and non-`const` inputs to the `conduit::blueprint::mesh::domains` function.
- Improved mesh blueprint index generation logic (local and MPI) to support domains with different topos, fields, etc.
- Deprecated accepting `npts_z !=0` for 2D shape types in `conduit::blueprint::mesh::examples::{braid, basic,grid}`. They issue a `CONDUIT_INFO` message when this detected and future versions will issue a `CONDUIT_ERROR`.
- An empty Conduit Node is now considered a valid multi-domain mesh. This change was made to make serial uses cases better match sparse MPI multi-domain use cases. Existing code that relied `mesh::verify` to exclude empty Nodes will now need an extra check to see if an input mesh has data.
- Added MPI communicator argument to `conduit::blueprint::mpi::mesh::to_polygonal` and `conduit::blueprint::mpi::mesh::to_polyhedral`.
- **Relay**
- Added CMake option (`ENABLE_RELAY_WEBSERVER`, default = `ON`) to control if Conduit's Relay Web Server support is built. Down stream codes can check for support via header ifdef `CONDUIT_RELAY_WEBSERVER_ENABLED` or at runtime in `conduit::relay::about`.
- Added support to compile against HDF5 1.12.

Fixed

- **General**
- Avoid compile issue with using `_Pragma()` with Python 3.8 on Windows

- `conduit_node` and `conduit_datatype` in the C API are no longer aliases to `void` so that callers cannot pass just any pointer to the APIs.
- Fixed memory over read issue with Fortran API due to int vs bool binding error. Fortran API still provides logical returns for methods like conduit_node_has_path() however the binding implementation now properly translates C_INT return codes into logical values.
- Fixed a subtle bug with Node fetch and Object role initialization.
- **Blueprint**
- Fixed a bug that was causing the `conduit::blueprint::mesh::topology::unstructured::generate_*` functions to produce bad results for polyhedral input topologies with heterogeneous elements (e.g. tets and hexs).
- Fixed a bug with `conduit::relay::io::blueprint::write_mesh` that undermined `truncate=true` option for root-only style output.
- Fixed options parsing bugs and improved error messages for the `conduit_blueprint_verify` exe.
- **Relay**
- Changed HDF5 offset support to use 64-bit unsigned integers for offsets, strides, and sizes.
- Fixed a bug with `conduit::relay::mpi::io::blueprint::save_mesh` where `file_style=root_only` could crash or truncate output files.
- Fixed a bug with inconsistent HDF5 handles being used in some cases when converting existing HDF5 Datasets from fixed to extendable.

8.4.3 v0.7.2

- Source Tarball

Highlights

(Extracted from Conduit's `Changelog`)

Added

- **General**
- Added the `cpp_fort_and_py` standalone example. It demos passing Conduit Nodes between C++, Fortran, and Python. See the related tutorial docs (https://llnl-conduit.readthedocs.io/en/latest/tutorial_cpp_fort_and_py.html) for more details.
- Added `conduit::utils::info_handler()`, `conduit::utils::warning_handler()`, and `conduit::utils::error_handler()` methods, which provide access to the currently registered info, warning, and error handlers.
- Added DataType::index_t method. Creates a DataType instance that describes an `index_t`, which is an alias to either `int32`, or `int 64` controlled by the `CONDUIT_INDEX_32` compile time option.
- Added several more methods to Python DataType interface
- Removed duplicate install of CMake exported target files that served as a bridge for clients using old style paths.

Changed

- **General**
- Updated to newer version of uberenv and changed to track spack fork https://github.com/alpine-dav/spack (branch: conduit/develop).
- Updated to newer version of BLT to leverage CMake's FindMPI defined targets when using CMake 3.15 or newer.
- Changed `rapidjson` namespace to `conduit_rapidjson` to avoid symbol collisions with other libraries using RapidJSON.
- **Blueprint**
- The semantics of `conduit::blueprint::mesh::verify` changed. An empty conduit Node is now considered a valid multi-domain mesh with zero domains. If you always expect mesh data, you can add an additional check for empty to craft code that works for both the old and new verify semantics.
- **Relay**
- Added Relay HDF5 support for reading and writing to an HDF5 dataset with offset.
- Added `conduit::relay::io::hdf5::read_info` which allows you to obtain metadata from an HDF5 file.
- Added configure error when conduit lacks MPI support and HDF5 has MPI support

Fixed

- **General**
- Fixed missing implementation of DataType::is_index_t
- Fixed issue with compiling t_h5z_zfp_smoke.cpp against an MPI-enabled HDF5.
- **Blueprint**
- Fixed a bug that caused HDF5 reference paths to appear twice in Relay HDF5 Error messages.
- **Blueprint**
- `conduit::relay::io::blueprint.read_mesh` now uses read only I/O handles.

8.4.4 v0.7.1

- Source Tarball

Highlights

(Extracted from Conduit's `Changelog`)

Fixed

- **General**
- Fixed a bug with Conduit's C interface including C++ headers.
- **Blueprint**

- Fixed a bug with `blueprint::mesh::matset::to_silo` and `blueprint::mesh::field::to_silo` that could modify input values.

8.4.5 v0.7.0

- Source Tarball

Highlights

(Extracted from Conduit's `Changelog`)

Changed

- **General**
- Conduit now requires C++11 support.
- Python Node repr string construction now uses `Node.to_summary_string()`

Added

- CMake: Added extra check for include dir vs fully resolved hdf5 path.
- **General**
- Added a builtin sandboxed header-only version of fmt. The namespace and directory paths were changed to `conduit_fmt` to avoid potential symbol collisions with other codes using fmt. Downstream software can use by including `conduit_fmt/conduit_fmt.h`.
- Added support for using C++11 initializer lists to set Node and DataArray values from numeric arrays. See C++ tutorial docs (https://llnl-conduit.readthedocs.io/en/latest/tutorial_cpp_numeric.html#c-11-initializer-lists) for more details.
- Added a Node::describe() method. This method creates a new node that mirrors the current Node, however each leaf is replaced by summary stats and a truncated display of the values. For use cases with large leaves, printing the describe() output Node is much more helpful for debugging and understanding vs wall of text from other to_string() methods.
- Added conduit::utils::format methods. These methods use fmt to format strings that include fmt style patterns. The formatting arguments are passed as a conduit::Node tree. The `args` case allows named arguments (args passed as object) or ordered args (args passed as list). The `maps` case also supports named or ordered args and works in conjunction with a `map_index`. The `map_index` is used to fetch a value from an array, or list of strings, which is then passed to fmt. The `maps` style of indexed indirection supports generating path strings for non-trivial domain partition mappings in Blueprint. This functionality is also available in Python, via the `conduit.utils.format` method.
- Added `DataArray::fill` method, which set all elements of a DataArray to a given value.
- Added `Node::to_summary_string` methods, which allow you to create truncated strings that describe a node tree, control the max number of children and max number of elements shown.
- Added python support for `Node.to_summary_string`
- **Relay**

- Added Relay IO Handle mode support for `a` (append) and `t` (truncate). Truncate allows you to overwrite files when the handle is opened. The default is append, which preserves prior IO Handle behavior.
- Added `conduit::relay::io::blueprint::save_mesh` variants, these overwrite existing files (providing relay save semantics) instead of adding mesh data to existing files. We recommend using `save_mesh` for most uses cases, b/c in many cases `write_mesh` to an existing HDF5 file set can fail due to conflicts with the current HDF5 tree.
- Added `conduit::relay::io::blueprint::load_mesh` variants, these reset the passed node before reading mesh data (providing relay load semantics). We recommend using `load_mesh` for most uses cases.
- Added `truncate` option to `conduit::relay::io::blueprint::write_mesh`, this is used by `save_mesh`.
- Improve capture and reporting of I/O errors in `conduit::relay::[mpi::]io::blueprint::{save_mesh|write_m` Now in the MPI case, If any rank fails to open or write to a file all ranks will throw an exception.
- Added yaml detection support to `conduit::relay::io:identify_file_type`.
- **Blueprint**
- Added `conduit::blueprint::mesh::matset::to_silo()` which converts a valid blueprint matset to a node that contains arrays that follow Silo's sparse mix slot volume fraction representation.
- Added `conduit::blueprint::mesh::field::to_silo()` which converts a valid blueprint field and matset to a node that contains arrays that follow Silo's sparse mix slot volume fraction representation.
- Added `material_map` to `conduit::blueprint::mesh:matset::index`, to provide an explicit material name to id mapping.
- Added `mat_check` field to `blueprint::mesh::examples::venn`. This field encodes the material info in a scalar field and in the `matset_values` in a way that can be used to easily compare and verify proper construction in other tools.

Fixed

- **Relay**
- Fixed bug in the Relay IOHandle Basic that would create unnecessary "_json" schema files to be written to disk upon open().

Removed

- **General**
- Removed `Node::fetch_child` and `Schema::fetch_child` methods for v0.7.0. (Deprecated in v0.6.0 – prefer `fetch_existing`)
- Removed `Schema::to_json` method variants with `detailed` for v0.7.0. (Deprecated in v0.6.0 – prefer standard `to_json`)
- Removed `Schema::save` method variant with `detailed` for v0.7.0. (Deprecated in v0.6.0 – prefer standard `save`)
- The `master` branch was removed from GitHub (Deprecated in v0.6.0 – replaced by the `develop` branch)
- **Relay**
- Removed `conduit::relay::io_blueprint::save` methods for v0.7.0. (Deprecated in v0.6.0 – prefer `conduit::relay::io::blueprint::save_mesh`)

8.4.6 v0.6.0

- Source Tarball

Highlights

(Extracted from Conduit's `Changelog`)

Added

- **General**
- Added support for children with names that include `/`. Since slashes are part of Conduit's hierarchical path mechanism, you must use explicit methods (add_child(), child(), etc) to create and access children with these types of names. These names are also supported in all basic i/o cases (JSON, YAML, Conduit Binary).
- Added Node::child and Schema::child methods, which provide access to existing children by name.
- Added Node::fetch_existing and Schema::fetch_existing methods, which provide access to existing paths or error when given a bad path.
- Added Node::add_child() and Node::remove_child() to support direct operations and cases where names have `/` s.
- Added a set of conduit::utils::log::remove_* filtering functions, which process conduit log/info nodes and strip out the requested information (useful for focusing the often verbose output in log/info nodes).
- Added to_string() and to_string_default() methods to Node, Schema, DataType, and DataArray. These methods alias either to_yaml() or to_json(). Long term yaml will be preferred over json.
- Added helper script (scripts/regen_docs_outputs.py) that regenerates all example outputs used Conduit's Sphinx docs.
- Added to_yaml() and to_yaml_stream methods() to Schema, DataType, and DataArray.
- Added support for C++-style iterators on node children. You can now do `for (Node &node : node.children()) {}`. You can also do `node.children.begin()` and `node.children.end()` to work with the iterators directly.
- **Relay**
- Added an open mode option to Relay IOHandle. See Relay IOHandle docs (https://llnl-conduit.readthedocs.io/en/latest/relay_io.html#relay-i-o-handle-interface) for more details.
- Added the conduit.relay.mpi Python module to support Relay MPI in Python.
- Added support to write and read Conduit lists to HDF5 files. Since HDF5 Groups do not support unnamed indexed children, each list child is written using a string name that represents its index and a special attribute is written to the HDF5 group to mark the list case. On read, the special attribute is used to detect and read this style of group back into a Conduit list.
- Added preliminary support to read Sidre Datastore-style HDF5 using Relay IOHandle, those grouped with a root file.
- Added `conduit::relay::io::blueprint::read_mesh` functions, were pulled in from Ascent's Blueprint import logic.
- Added `conduit::relay::mpi::wait` and `conduit::relay::mpi::wait_all` functions. These functions consolidate the logic supporting both `isend` and `irecv` requests. `wait_all` supports cases where both sends and receives were posted, which is a common for non-trivial point-to-point communication use cases.

- **Blueprint**
- Added support for sparse one-to-many relationships with the new `blueprint::o2mrelation` protocol. See the `blueprint::o2mrelation::examples::uniform` example for details.
- Added sparse one-to-many, uni-buffer, and material-dominant specification support to Material sets. See the Material sets documentation
- Added support for Adjacency sets for Structured Mesh Topologies. See the `blueprint::mesh::examples::adjset_uniform` example.
- Added `blueprint::mesh::examples::julia_nestsets_simple` and `blueprint::mesh::examples::julia_nestsets_complex` examples represent Julia set fractals using patch-based AMR meshes and the Mesh Blueprint Nesting Set protocol. See the Julia AMR Blueprint docs
- Added `blueprint::mesh::examples::venn` example that demonstrates different ways to encode volume fraction based multi-material fields. See the Venn Blueprint docs
- Added `blueprint::mesh::number_of_domains` property method for trees that conform to the mesh blueprint.
- Added MPI mesh blueprint methods, `blueprint::mpi::mesh::verify` and `blueprint::mpi::mesh::number_of_domains` (available in the `conduit_blueprint_mpi` library)
- Added `blueprint::mpi::mesh::examples::braid_uniform_multi_domain` and `blueprint::mpi::mesh::examples::spiral_round_robin` distributed-memory mesh examples to the `conduit_blueprint_mpi` library.
- Added `state/path` to the Mesh Blueprint index, needed for consumers to know the proper path to read extended state info (such as `domain_id`)

Fixed

- **General**
- Updated to newer BLT to resolve BLT/FindMPI issues with rpath linking commands when using OpenMPI.
- Fixed internal object name string for the Python Iterator object. It used to report `Schema`, which triggered both puzzling and concerned emotions.
- Fixed a bug with `Node.set` in the Python API that undermined setting NumPy arrays with sliced views and complex striding. General slices should now work with `set`. No changes to the `set_external` case, which requires 1-D effective striding and throws an exception when more complex strides are presented.
- Fixed a bug with auto detect of protocol for Node.load
- Fixed bugs with auto detect of protocol for Node.load and Node.save in the Python interface
- **Relay**
- Use H5F_ACC_RDONLY in relay::io::is_hdf5_file to avoid errors when checking files that already have open HDF5 handles.
- Fixed compatibility check for empty Nodes against HDF5 files with existing paths

Changed

- **General**

- Conduit's main git branch was renamed from `master` to `develop`. To allow time for folks to migrate, the `master` branch is active but frozen and will be removed during the `0.7.0` release.
- We recommend a C++11 (or newer) compiler, support for older C++ standards is deprecated and will be removed in a future release.
- Node::fetch_child and Schema::fetch_child are deprecated in favor of the more clearly named Node::fetch_existing and Schema::fetch_existing. fetch_child variants still exist, but will be removed in a future release.
- Python str() methods for Node, Schema, and DataType now use their new to_string() methods.
- DataArray<T>::to_json(std::ostring &) is deprecated in favor DataArray<T>::to_json_stream. to_json(std::ostring &) will be removed in a future release.
- Schema::to_json and Schema::save variants with detailed (bool) arg are deprecated. The detailed arg was never used. These methods will be removed in a future release.
- Node::print() now prints yaml instead of json.
- The string return variants of `about` methods now return yaml strings instead of json strings.
- Sphinx Docs code examples and outputs are now included using start-after and end-before style includes.
- Schema to_json() and to_json_stream() methods were expanded to support indent, depth, pad and end-of-element args.
- In Python, conduit.Node() repr now returns the YAML string representation of the Node. Perviously verbose `conduit_json` was used, which was overwhelming.
- conduit.about() now reports the git tag if found, and `version` was changed to add git sha and status (dirty) info to avoid confusion between release and development installs.
- **Relay**
- Provide more context when a Conduit Node cannot be written to a HDF5 file because it is incompatible with the existing HDF5 tree. Error messages now provide the full path and details about the incompatibility.
- `conduit::relay::io_blueprint::save` functions are deprecated in favor of `conduit::relay::io::blueprint::write_mesh`
- `conduit::relay::io::blueprint::write_mesh` functions were pulled in from Ascent's Blueprint export logic.
- `conduit_relay_io_mpi` lib now depends on `conduit_relay_io`. Due to this change, a single build supports either ADIOS serial (no-mpi) or ADIOS with MPI support, but not both. If conduit is configured with MPI support, ADIOS MPI is used.
- The functions `conduit::relay::mpi::wait_send` and `conduit::relay::mpi::wait_recv` now use `conduit::relay::mpi::wait`. The functions `wait_send` and `wait_recv` exist to preserve the old API, there is no benefit to use them over `wait`.
- The functions `conduit::relay::mpi::wait_all_send` and `conduit::relay::mpi::wait_all_recv` now use `conduit::relay::mpi::wait_all`. The functions `wait_all_send` and `wait_all_recv` exist to preserve the old API, there is no benefit to use them over `wait_all`.
- **Blueprint**
- Refactored the Polygonal and Polyhedral mesh blueprint specification to leverage one-to-many concepts and to allow more zero-copy use cases.
- The `conduit_blueprint_mpi` library now depends on `conduit_relay_mpi`.
- The optional Mesh Blueprint structured topology logical element origin is now specified using `{i,j,k}` instead of `{i0,j0,k0}`.

8.4.7 v0.5.1

- Source Tarball

Highlights

(Extracted from Conduit's `Changelog`)

Added

- **General**
- Added Node::parse() method, (C++, Python and Fortran) which supports common json and yaml parsing use cases without creating a generator instance.
- Use FOLDER target property to group targets for Visual Studio
- Added Node load(), and save() support to the C and Fortran APIs

Changed

- **General**
- Node::load() and Node::save() now auto detect which protocol to use when protocol argument is an empty string
- Changed Node::load() and Node::save() default protocol value to empty (default now is to auto detect)
- Changed Python linking strategy to defer linking for our compiler modules
- Conduit Error Exception message strings now print cleaner (avoiding nesting doll string escaping headaches)
- Build system improvements to support conda-forge builds for Linux, macOS, and Windows

Fixed

- **General**
- Fixed install paths for CMake exported target files to follow standard CMake find_package() search conventions. Also perserved duplicate files to support old import path structure for this release.
- python: Fixed Node.set_external() to accept conduit nodes as well as numpy arrays
- Fixed dll install locations for Windows

8.4.8 v0.5.0

- Source Tarball

Highlights

(Extracted from Conduit's `Changelog`)

Added

- **General**
- Added support to parse YAML into Conduit Nodes and to create YAML from Conduit Nodes. Support closely follows the "json" protocol, making similar choices related to promoting YAML string leaves to concrete data types.
- Added several more Conduit Node methods to the C and Fortran APIs. Additions are enumerated here: https://github.com/LLNL/conduit/pull/426
- Added Node set support for Python Tuples and Lists with numeric and string entires
- Added Node set support for Numpy String Arrays. String Arrays become Conduit lists with child char8_str arrays
- **Blueprint**
- Added support for a "zfparray" blueprint that holds ZFP compressed array data.
- Added the the "specsets" top-level section to the Blueprint schema, which can be used to represent multi-dimensional per-material quantities (most commonly per-material atomic composition fractions).
- Added explicit topological data generation functions for points, lines, and faces
- Added derived topology generation functions for element centroids, sides, and corners
- Added the basic example function to the conduit.mesh.blueprint.examples module
- **Relay**
- Added optional ZFP support to relay, that enables wrapping and unwraping zfp arrays into conduit Nodes.
- Extended relay HDF5 I/O support to read a wider range of HDF5 string representations including H5T_VARIABLE strings.

Changed

- **General**
- Conduit's automatic build process (uberenv + spack) now defaults to using Python 3
- Improved CMake export logic to make it easier to find and use Conduit install in a CMake-based build system. (See using-with-cmake example for new recipe)
- **Relay**
- Added is_open() method to IOHandle in the C++ and Python interfaces
- Added file name information to Relay HDF5 error messages

Fixed

- **General**
- Fixed bug that caused memory access after free during Node destruction
- **Relay**
- Fixed crash with mpi broadcast_using_schema() when receiving tasks pass a non empty Node.
- Fixed a few Windows API export issues for relay io

8.4.9 v0.4.0

- Source Tarball

Highlights

(Extracted from Conduit's `Changelog`)

Added

- **General**
- Added Generic IO Handle class (relay::io::IOHandle) with C++ and Python APIs, tests, and docs.
- Added `rename_child` method to Schema and Node
- Added generation and install of conduit_config.mk for using-with-make example
- Added datatype helpers for long long and long double
- Added error for empty path fetch
- Added C functions for setting error, warning, info handlers.
- Added limited set of C bindings for DataType
- Added C bindings for relay IO
- Added several more functions to conduit node python interfaces
- **Blueprint**
- Added implicit point topology docs and example
- Added julia and spiral mesh bp examples
- Added mesh topology transformations to blueprint
- Added polygonal mesh support to mesh blueprint
- Added verify method for mesh blueprint nestset
- **Relay**
- Added ADIOS Support, enabling ADIOS read and write of Node objects.
- Added a relay::mpi::io library that mirrors the API of relay::io, except that all functions take an MPI communicator. The functions are implemented in parallel for the ADIOS protocol. For other protocols, they will behave the same as the serial functions in relay::io. For the ADIOS protocol, the save() and save_merged() functions operate collectively within a communicator to enable multiple MPI ranks to save data to a single file as separate "domains".
- Added an add_time_step() function to that lets the caller append data collectively to an existing ADIOS file
- Added a function to query the number of time steps and the number of domains in a ADIOS file.
- Added versions of save and save_merged that take an options node.
- Added C API for new save, save_merged functions.
- Added method to list an HDF5 group's child names
- Added save and append methods to the HDF5 I/O interface
- Added docs and examples for relay io

Changed

- **General**
- Changed mapping of c types to bit-width style to be compatible with C++11 std bit-width types when C++11 is enabled
- Several improvements to uberenv, our automated build process, and building directions
- Upgraded the type system with more explicit signed support
- **Relay**
- Improvements to the Silo mesh writer
- Refactor to support both relay::io and relay::mpi::io namespaces.
- Refactor to add support for steps and domains to I/O interfaces
- Changed to only use `libver latest` setting for for hdf5 1.8 to minimize compatibility issues

Fixed

- **General**
- Fixed bugs with std::vector gap methods
- Fixed A few C function names in conduit_node.h
- Fixed bug in python that was requesting unsigned array for signed cases
- Fixed issue with Node::diff failing for string data with offsets
- Fixes for building on BlueOS with the xl compiler
- **Blueprint**
- Fixed validity status for blueprint functions
- Fixed improper error reporting for Blueprint references
- **Relay**
- Relay I/O exceptions are now forwarded to python
- Fixed MPI send_with_schema bug when data was compact but not contiguous
- Switched to use MPI bit-width style data type enums in `relay::mpi`

8.4.10 v0.3.1

- Source Tarball

Highlights

- **General**
- Added new `Node::diff` and `Node::diff_compatible` methods
- Updated uberenv to use a newer spack and removed several custom packages
- C++ `Node::set` methods now take const pointers for data

- Added Python version of basic tutorial
- Expanded the Node Python Capsule API
- Added Python API bug fixes
- Fixed API exports for static libs on Windows
- **Blueprint**
- Mesh Protocol
  - Removed unnecessary state member in the braid example
- Added Multi-level Array Protocol (conduit::blueprint::mlarray)
- **Relay**
- Added bug fixes for Relay HDF5 support on Windows

8.4.11 v0.3.0

- Source Tarball

Highlights

- **General**
- Moved to use BLT (https://github.com/llnl/blt) as our core CMake-based build system
- Bug fixes to support building on Visual Studio 2013
- Bug fixes for `conduit::Node` in the List Role
- Expose more of the Conduit API in Python
- Use ints instead of bools in the Conduit C-APIs for wider compiler compatibility
- Fixed memory leaks in *conduit* and *conduit_relay*
- **Blueprint**
- Mesh Protocol
  - Added support for multi-material fields via *matsets* (volume fractions and per-material values)
  - Added initial support for domain boundary info via *adjsets* for distributed-memory unstructured meshes
- **Relay**
- Major improvements *conduit_relay* I/O HDF5 support
  - Add heuristics with knobs for controlling use of HDF5 compact datasets and compression support
  - Improved error checking and error messages
- Major improvements to *conduit_relay_mpi* support
  - Add support for reductions and broadcast
  - Add support zero-copy pass to MPI for a wide set of calls
  - Harden notion of *known schema* vs *generic* MPI support

8.4.12 v0.2.1

- Source Tarball

Highlights

- **General**
- Added fixes to support static builds on BGQ using xlc and gcc
- Fixed missing install of fortran module files
- Eliminated separate fortran libs by moving fortran symbols into their associated main libs
- Changed `Node::set_external` to support const Node references
- Refactored path and file systems utils functions for clarity.
- **Blueprint**
- Fixed bug with verify of mesh/coords for rectilinear case
- Added support to the blueprint python module for the mesh and mcarray protocol methods
- Added stand alone blueprint verify executable
- **Relay**
- Updated the version of civetweb used to avoid dlopen issues with SSL for static builds

8.4.13 v0.2.0

- Source Tarball

Highlights

- **General**
- Changes to clarify concepts in the `conduit::Node` API
- Added const access to `conduit::Node` children and a new `NodeConstIterator`
- Added support for building on Windows
- Added more Python, C, and Fortran API support
- Resolved several bugs across libraries
- Resolved compiler warnings and memory leaks
- Improved unit test coverage
- Renamed source and header files for clarity and to avoid potential conflicts with other projects
- **Blueprint**
- Added verify support for the mcarray and mesh protocols
- Added functions that create examples instances of mcarrays and meshes
- Added memory layout transform helpers for mcarrays
- Added a helper that creates a mesh blueprint index from a valid mesh

- **Relay**
- Added extensive HDF5 I/O support for reading and writing between HDF5 files and conduit Node trees
- Changed I/O protocol string names for clarity
- Refactored the `relay::WebServer` and the Conduit Node Viewer application
- Added entangle, a python script ssh tunneling solution

8.5 Presentations and Publications

8.5.1 Related Publications

- The ALPINE In Situ Infrastructure: Ascending from the Ashes of Strawman Presented at the ISAV 2017 Workshop, held in conjunction with SC 17, on November 12th 2017 in Denver, CO.
- Strawman: A Batch In Situ Visualization and Analysis Infrastructure for Multi-Physics Simulation Codes Presented at the ISAV 2015 Workshop, held in conjunction with SC 15, on November 16th 2015 in Austin, TX.

8.5.2 Presentation Slides

- Conduit and Mesh Blueprint Intro (July 2021) Presented at LANL 2021 Data Science at Scale Summer School.
- The Conduit Mesh Blueprint: Drafting a New Way to Share Simulation Meshes Presented at DOE Computer Graphics Forum April 2019.
- `SciPy 2016 talk on Conduit (July 2016)`
- `Conduit Introduction (February 2015)`

8.5.3 Recorded Talks

- SciPy 2016 talk on Conduit (July 2016)

8.5.4 Interviews

- RCE HPC Podcast on Conduit (October 2015)

8.6 License Info

8.6.1 Conduit License

Please also read conduit/LICENSE

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the disclaimer below.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the disclaimer (as noted below) in the documentation and/or other materials provided with the distribution.
- Neither the name of the LLNS/LLNL nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL LAWRENCE LIVERMORE NATIONAL SECURITY, LLC, THE U.S. DEPARTMENT OF ENERGY OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Additional BSD Notice

1. This notice is required to be provided under our contract with the U.S. Department of Energy (DOE). This work was produced at Lawrence Livermore National Laboratory under Contract No. DE-AC52-07NA27344 with the DOE.
2. Neither the United States Government nor Lawrence Livermore National Security, LLC nor any of their employees, makes any warranty, express or implied, or assumes any liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately-owned rights.
3. Also, reference herein to any specific commercial products, process, or services by trade name, trademark, manufacturer or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

Third Party Builtin Libraries

Here is a list of the software components used by conduit in source form and the location of their respective license files in our source repo.

C and C++ Libraries

- *gtest*: From BLT - (BSD Style License)
- *libb64*: src/thirdparty_builtin/libb64/LICENSE (Public Domain)
- *rapidjson*: src/thirdparty_builtin/rapidjson/license.txt (MIT License)
- *civetweb*: src/thirdparty_builtin/civetweb-0a95342/LICENSE.md (MIT License)
- *libyaml*: src/thirdparty_builtin/libyaml-690a781/LICENSE (MIT License)

- *fmt*: src/thirdparty_builtin/fmt-7.1.0/LICENSE.rst (MIT License)

JavaScript Libraries

- *fattable*: src/libs/relay/web_clients/rest_client/resources/fattable/LICENSE (MIT License)
- *pure*: src/libs/relay/web_clients/rest_client/resources/pure/LICENSE.md (BSD Style License)
- *d3*: src/libs/relay/web_clients/rest_client/resources/d3/LICENSE (BSD Style License)
- *jquery*: src/libs/relay/web_clients/wsock_test/resources/jquerty-license.txt (MIT License)

Fortran Libraries

- *fruit*: From BLT - (BSD Style License)

Build System

- *CMake*: http://www.cmake.org/licensing/ (BSD Style License)
- *BLT*: https://github.com/llnl/blt (BSD Style License)
- *Spack*: http://software.llnl.gov/spack (LGPL License)

Documentation

- *doxygen*: http://www.stack.nl/~dimitri/doxygen/index.html (GPL License)
- *sphinx*: http://sphinx-doc.org/ (BSD Style License)
- *breathe*: https://github.com/michaeljones/breathe (BSD Style License)
- *rtd sphinx theme*: https://github.com/snide/sphinx_rtd_theme/blob/master/LICENSE (MIT License)

CHAPTER 9

Indices and tables

- genindex
- search