Conduit Documentation

Release 0.5.0

LLNS

Nov 07, 2019

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*.

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 LLNL's Axom Toolkit (to be released).

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)
- Todd Gamblin (LLNL)
- Brad Whitlock (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 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

```

(continues on next page)

(continued from previous page)

```
If run in sub directory of a conduit install,
CONDUIT_DIR will default to ../../..

 mkdir build
 cd build
 cmake ..
 make
 ./conduit_example
```

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
```

8.1.3 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.

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
  },
  "list_example":
  [
    0,
    1,
    2,
    3,
```

(continues on next page)

(continued from previous page)

```
    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_json() << std::endl;
}

itr = n["list_example"].children();
while(itr.has_next())
{
    Node &cld = itr.next();
    std::cout << cld.to_json() << 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
0
1
2
```

(continues on next page)

(continued from previous page)

```
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":
  {
    "0x7fcc834044e0":
    {
      "path": "my",
      "type": "allocated",
      "bytes": 5
    },
    "0x7fcc83405f20":
    {
      "path": "a/b/c",
      "type": "allocated",
      "bytes": 2
    },
    "0x7fcc83405f10":
    {
      "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;
```

(continues on next page)

(continued from previous page)

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

```
{
  "test": 100
}

{
  "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>`.

Compatible Schemas

When a **set** method is called on a Node, if the data passed to the **set** is compatible with the Node's Schema the data is simply copied. No allocation or Schema changes occur. If the data is not compatible the Node will be reconfigured to store the passed data.

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.

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);
```

(continues on next page)

(continued from previous page)

```
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;
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]
```

Using Introspection and Conversion

In this example, we have an array in a node that we are interested in processing using an and 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)
{
    for(int i = 0; i < num_vals; i++)
    {
        std::cout << "vals[" << i << "] = " <<  vals[i] << std::endl;
    }
}

//-----------------------------------------------------------------------------
void process_doubles(Node & n)
{
    Node res;
```

(continues on next page)

(continued from previous page)

```
    // 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"]);
}

//-----------------------------------------------------------------------------
```

```
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
```

(continues on next page)

(continued from previous page)

```
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
```

Generators

Using *Generator* instances to parse JSON schemas

The *Generator* class is used to parse conduit JSON schemas into a *Node*.

```
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}
}
```

The *Generator* can also parse pure json. 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
}
```

Schemas can easily 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
}

{
  "mem_spaces":
  {
    "0x7fff5b4da040":
    {
      "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};
```

(continues on next page)

(continued from previous page)

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

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


```
{
  "mem_spaces":
  {
    "0x7fe5e2c05540":
    {
      "path": "",
      "type": "allocated",
      "bytes": 40
    }
  },
  "total_bytes_allocated": 40,
  "total_bytes_mmaped": 0,
```


(continues on next page)

(continued from previous page)

```
  "total_bytes_compact": 40,
  "total_strided_bytes": 40
}
[-100.0, -200.0, -300.0, -400.0, -500.0]

{
  "mem_spaces":
  {
    "0x7fe5e2c05d80":
    {
      "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);
```

(continues on next page)

(continued from previous page)

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

n.print();

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

```
{
  "mem_spaces":
  {
    "0x7fdebac044d0":
    {
      "path": "v_owned",
      "type": "allocated",
      "bytes": 40
    },
    "0x7fdebac045a0":
    {
      "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]
```

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.

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)
```

(continues on next page)

(continued from previous page)

```
{
    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;
    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 const fetch_child, 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);
```

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.

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 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.

```
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())))

for v in n["list_example"].children():
    print(v.node())
```

```
{
  "object_example":
  {
    "val1": "data",
    "val2": 10,
    "val3": 3.1415
  },
  "list_example":
  [
```

(continues on next page)

(continued from previous page)

```
    0,
    1,
    2,
    3,
    4
  ]
}
val1: "data"
val2: 10
val3: 3.1415
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":
  {
    "0x7fb538d51320":
    {
      "path": "my",
      "type": "allocated",
      "bytes": 5
    },
    "0x7fb538d31db0":
    {
      "path": "a/b/c",
      "type": "allocated",
      "bytes": 2
    },
    "0x7fb538daf890":
    {
      "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
n.print_detailed()
```

```
{
  "test": {"dtype":"int64", "number_of_elements": 1, "offset": 0, "stride": 8,
↪"element_bytes": 8, "endianness": "little", "value": 10}
}
```

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

Compatible Schemas

When a **set** method is called on a Node, if the data passed to the **set** is compatible with the Node's Schema the data is simply copied. No allocation or Schema changes occur. If the data is not compatible the Node will be reconfigured to store the passed data.

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.

Generators

Using *Generator* instances to parse JSON schemas

The *Generator* class is used to parse conduit JSON schemas into a *Node*.

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

```

{
  "test": 100.0
}
```

The *Generator* can also parse pure json. 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
}
```

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:**

```
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");

//load back from hdf5 using load
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:

{
  "a":
  {
    "my_data": 1.0,
    "b":
    {
      "my_string": "value"
    }
  }
}

Load result:

{
  "a":
  {
    "my_data": 1.0,
    "b":
    {
      "my_string": "value"
    }
  }
}
```

Save Merged

- **C++ Example:**

```
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");
```

(continues on next page)

(continued from previous page)

```
// 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,
    "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":
    {
```

(continues on next page)

(continued from previous page)

```
      "new_data": 42.0,
      "my_string": "value"
    },
    "my_data": 1.0
  }
}
```

Load from Subpath

- **C++ Example:**

```
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:**

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

(continues on next page)

(continued from previous page)

```
//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 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.

> **Danger:** Note: While you can read from and write to subpaths using a handle, IOHandle *does not* yet 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;
}
```

(continues on next page)

(continued from previous page)

```
// 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\""
          << 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"
```

(continues on next page)

(continued from previous page)

```
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
  }
}
```

- **Python Example:**

```python
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)
```

(continues on next page)

(continued from previous page)

```
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:

{
  "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
  }
}
```

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

// Read the entire tree:
```

(continues on next page)

(continued from previous page)

```
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 << "File \n'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

Data loaded:
1.0

Data loaded:

{
  "a":
  {
    "my_data": 3.1415,
    "b":
    {
      "my_string": "value",
      "c": 144
    }
  }
}
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();
```

(continues on next page)

(continued from previous page)

```
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_bin": "enabled",
    "hdf5": "enabled",
    "conduit_silo": "disabled",
    "conduit_silo_mesh": "disabled",
    "adios": "disabled"
  },
  "options":
  {
    "hdf5":
    {
      "compact_storage":
      {
        "enabled": "true",
        "threshold": 1024
      },
      "chunking":
      {
        "enabled": "true",
        "threshold": 2000000,
```

(continues on next page)

(continued from previous page)

```
        "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.

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:

- 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.

- 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.

Protocol Details

mcarray

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: struct of arrays 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}

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, separe 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`.

mesh

This section provides details about the Mesh Blueprint. Lots of them. We don't have a Mesh Blueprint tutorial yet, if you are looking to wrap your mind around the basic mechanics of describing a mesh, you may want to start by reviewing the *Detailed Uniform Example* and exploring the other *Examples* included in the blueprint library.

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 one or more single-domain mesh entries. 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}

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	an index count N, then indices to N coordinate tuples
polyhedral	polyhedron	a face count M, then M polygonal face definitions

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/{i0,j0,k0} (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/{i0,j0,k0} (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 structually identical to the other explicit topology shape types (see the *Single Shape Topologies* section above), but the contents of their `elements/connectivity` sections look slightly different. In particular, the shape index connectivity for each element in these topologies is **explicit**, which means that the index sequence for each element is prefixed by a count that specifies the total number of indices (polygonal) or faces (polyhedral) that comprise that element. This explicit shape index facilitates both the specification of non-standard shapes (e.g. octogons) and of highly mixed shape topologies (e.g. polygons/polyhedra of many different shapes in one topology).

In more explicit terms, the `elements/connectivity` lists for the `polygonal` and `polyhedral` topology shapes follow these rules:

- **polygonal** - The first element starts at the beginning of the `elements/connectivity` list. The first value `V` for each element `E` indicates the number of vertices that comprise polygon `E`. The next `V` values in the list are indices for the `V` coordinates that comprise `E`. The next element begins after this sequence of `V` values, and this specification continues until the connectivity list is exhausted of items.

```
// Example Diagram:
//
//       4-----5
//       |`\   |
// e1 -> |  \  | <- e0
//       |   \.|
//       7-----6
//

//    index count ---+     +--- coordinate index values
//                   |     |
//                   v  |-----|
int64 poly_data[] = {3, 4, 6, 5,   // element 0
                     3, 7, 6, 4};  // element 1

conduit::Node topology = mesh["topologies/poly_topo"];
topology["coordset"] = "coords";
topology["type"] = "unstructured";
topology["elements/shape"] = "polygonal";
topology["elements/connectivity"].set_int64_ptr(&poly_data[0], 8);
```

- **polyhedral** - The first element begins at the first index of the `elements/connectivity` list. The first value `F` for each element `E` specifies the number of faces that comprise polyhedron `E`. The next value `V` denotes the number of vertices that comprise the first polygonal face `F1` of polyhedron `E`. Exactly like the polygonal specification, the following sequence of `V` values contain the indices of the coordinates for face `F1`. The next face `F2` begins immediately after this sequence, and this process continues until `F` faces are enumerated. The next element then begins after this supersequence, and this specification continues until the connectivity list is exhausted of items.

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

//  face index count ---+
//                      |
//     face count ---+  |     +--- coordinate index values
//                   |  |     |
//                   v  v  |-----|
int64 poly_data[] = {5, 3, 0, 1, 2, 3, 0, 2, 4, 3, 0, 1, 3, 3, 0, 3, 4, 4, 1, 2,
↪4, 3,   // element 0
                     5, 3, 5, 1, 2, 3, 5, 2, 4, 3, 5, 1, 3, 3, 5, 3, 4, 4, 1, 2,
↪4, 3};  // element 1
```

(continues on next page)

(continued from previous page)

```
conduit::Node topology = mesh["topologies/poly_topo"];
topology["coordset"] = "coords";
topology["type"] = "unstructured";
topology["elements/shape"] = "polyhedral";
topology["elements/connectivity"].set_int64_ptr(&poly_data[0], 44);
```

(Optional) Element Offsets

Unstructured topologies can optionally include a child `elements/offsets` to indicate the starting position of each element defined in the `elements/connectivity` array. This list is most often specified for heterogeneous and polygonal/polyhedral topologies so that the elements don't need to be found by stepping through the input connectivity array.

- topologies/topo/elements/offsets: (index array)

To generate this array for a given unstructured topology `topo`, make the following call:

```
conduit::blueprint::mesh::topology::unstructured::generate_offsets(topo,
↪                    // input topology
                                                                   topo[
↪"elements/offsets"]);  // output node for offset array
```

Material Sets

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

A material set contains an **mcarray** that houses per-material, per-element volume fractions and a source topology over which these volume fractions are defined. To conform to protocol, each entry in the `matsets` section must be an *Object* that contains the following information:

- matsets/matset/topology: "topo"
- matsets/matset/volume_fractions: (mcarray)

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 in material-major and component-minor order. 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)

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)

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)

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":
```


(continues on next page)

(continued from previous page)


```
      {
        "dx": 10.0,
        "dy": 10.0
      }
    }
  },
  "topologies":
  {
    "mesh":
    {
      "type": "uniform",
      "coordset": "coords"
    }
  },
  "fields":
  {
    "field":
    {
      "association": "element",
      "topology": "mesh",
      "volume_dependent": "false",
      "values": [0.0, 1.0, 2.0, 3.0]
    }
  }
}
```


- **Visual**

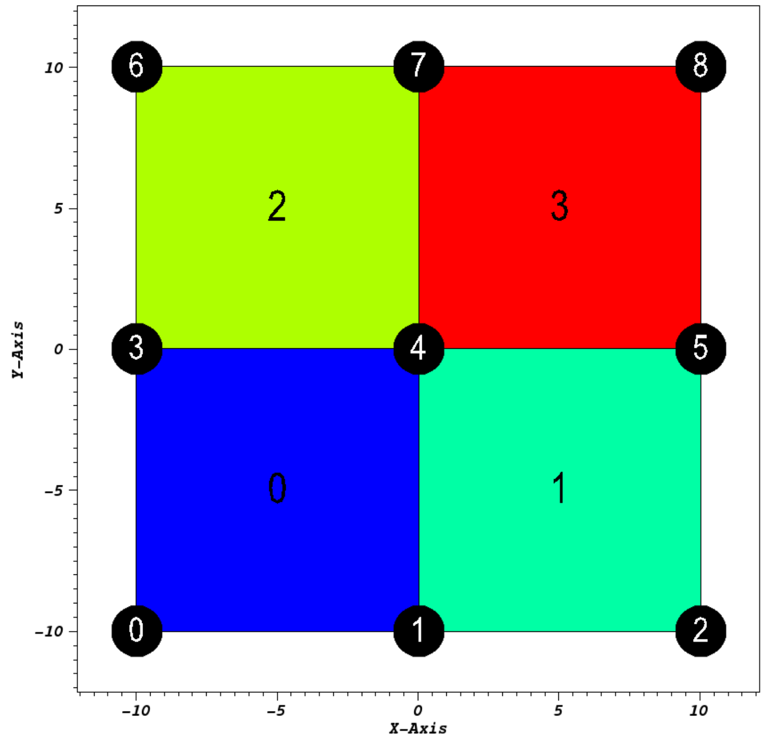

Fig. 1: Pseudocolor plot of `basic` (mesh type 'uniform')

Rectilinear

- **Usage Example**

```
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":
      {
        "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**

Structured

- **Usage Example**

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

(continues on next page)

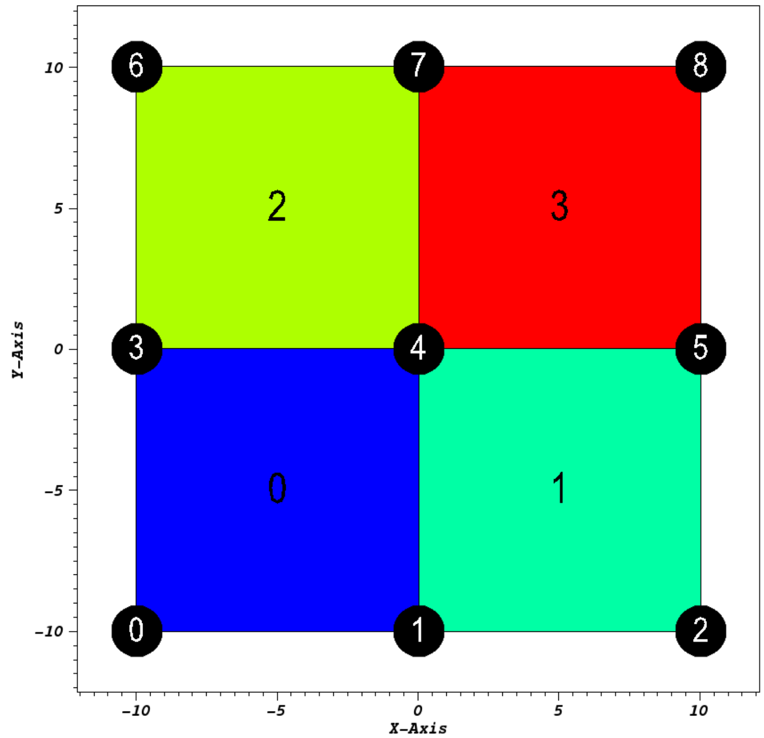

Fig. 2: Pseudocolor plot of `basic` (mesh type 'rectilinear')

(continued from previous page)

```
// 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":
      {
```

(continues on next page)

(continued from previous page)

```
        "dims":
        {
          "i": 2,
          "j": 2
        }
      }
    }
  },
  "fields":
  {
    "field":
    {
      "association": "element",
      "topology": "mesh",
      "volume_dependent": "false",
      "values": [0.0, 1.0, 2.0, 3.0]
    }
  }
}
```

- **Visual**

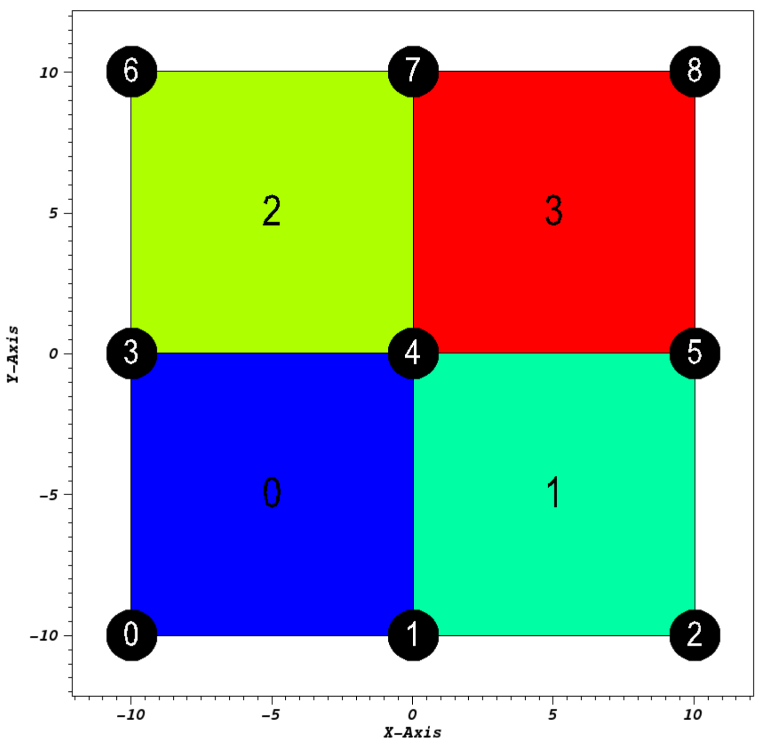

Fig. 3: Pseudocolor plot of `basic` (mesh type 'structured')

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;
```

(continues on next page)

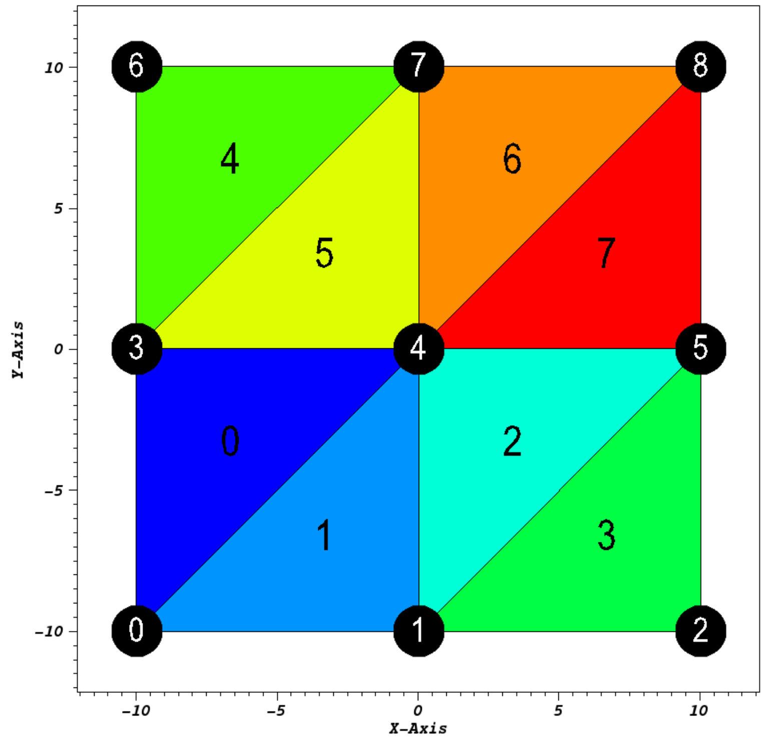

Fig. 4: Pseudocolor plot of `basic` (mesh type 'tris')

(continued from previous page)

```
// 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":
  {
    "mesh":
    {
      "type": "unstructured",
      "coordset": "coords",
```

(continues on next page)

(continued from previous page)

```
      "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**

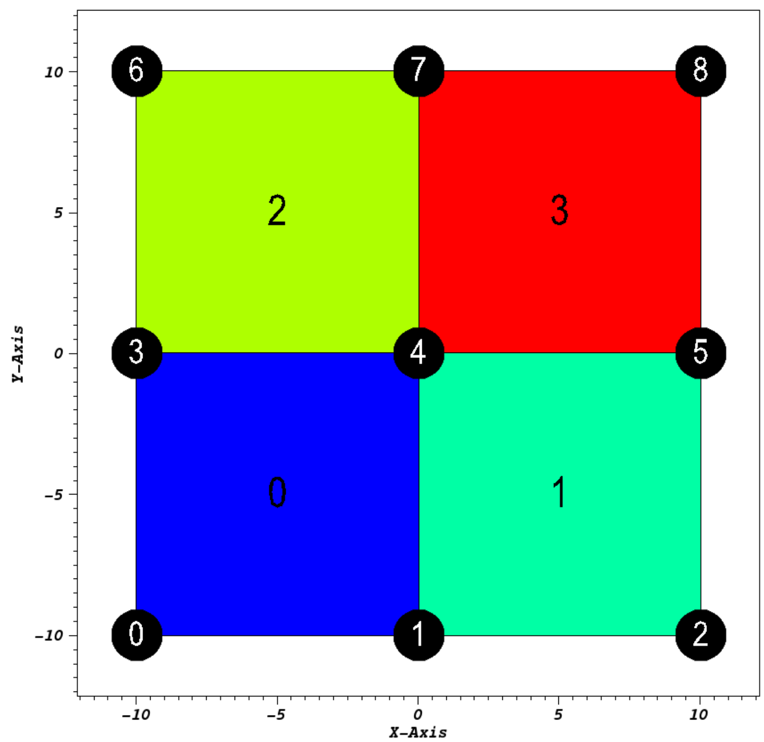

Fig. 5: Pseudocolor plot of `basic` (mesh type 'quads')

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();
```

- **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",
        "connectivity": [4, 0, 3, 4, 1, 4, 1, 4, 5, 2, 4, 3, 6, 7, 4, 4, 4, 7, 8, 5]
      }
    }
  },
  "fields": 
  {
    "field": 
    {
      "association": "element",
      "topology": "mesh",
      "volume_dependent": "false",
      "values": [0.0, 1.0, 2.0, 3.0]
    }
  }
}
```

- **Visual**

Tets

- **Usage Example**

```
// create container node
Node mesh;
// generate simple explicit tri-based 3d 'basic' mesh
```

(continues on next page)

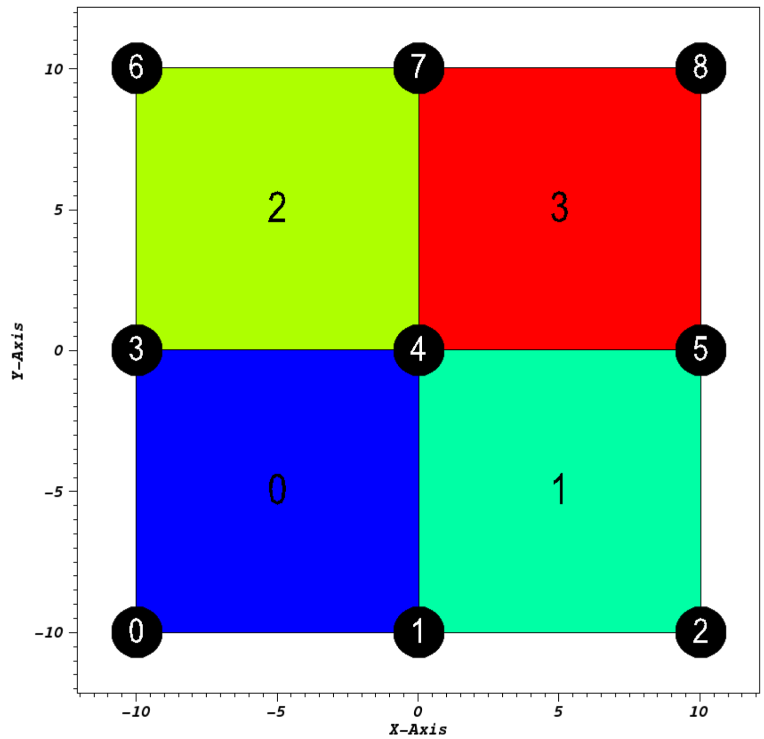

Fig. 6: Pseudocolor plot of `basic` (mesh type 'polygons')

(continued from previous page)

```
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, 10.0]
      }
    }
  },
```

(continues on next page)

(continued from previous page)


```
  "topologies":
  {
    "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**

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":
```


(continues on next page)

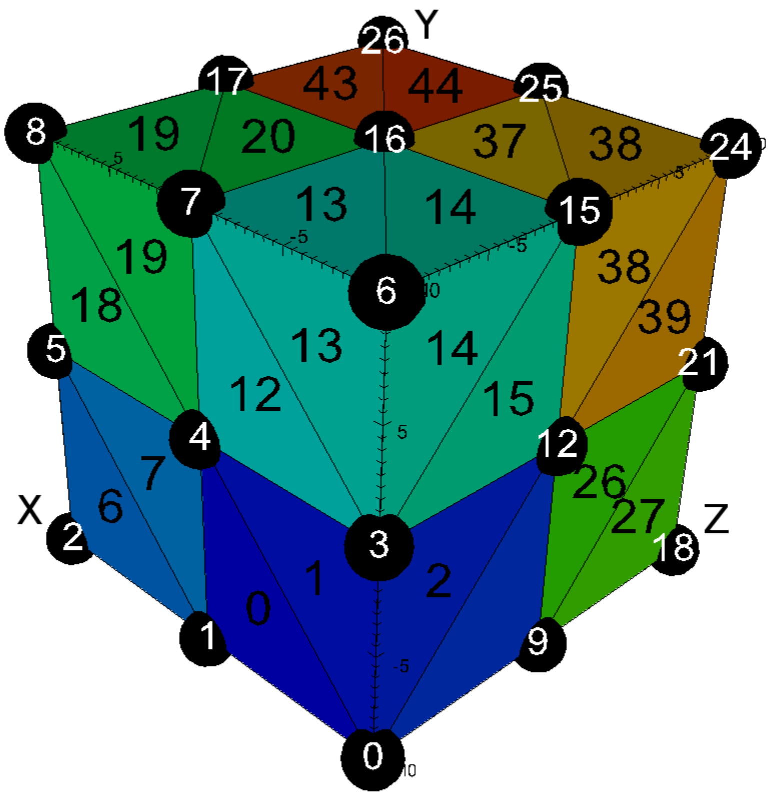

Fig. 7: Pseudocolor plot of `basic` (mesh type 'tets')

(continued from previous page)


```
    {
      "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();
```

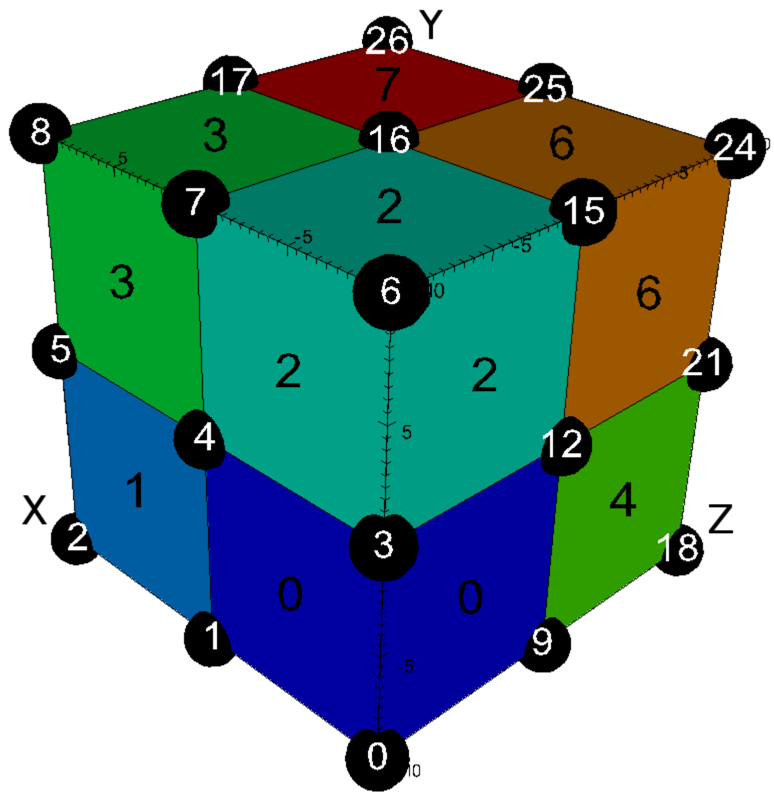

Fig. 8: Pseudocolor plot of `basic` (mesh type 'hexs')

- **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": "polyhedral",
        "connectivity": [6, 4, 0, 1, 4, 3, 4, 0, 1, 10, 9, 4, 1, 4, 13, 10, 4, 4, 3,␣
↪12, 13, 4, 3, 0, 9, 12, 4, 9, 10, 13, 12, 6, 4, 1, 2, 5, 4, 4, 1, 2, 11, 10, 4, 2,␣
↪5, 14, 11, 4, 5, 4, 13, 14, 4, 4, 1, 10, 13, 4, 10, 11, 14, 13, 6, 4, 3, 4, 7, 6, 4,
↪ 3, 4, 13, 12, 4, 4, 7, 16, 13, 4, 7, 6, 15, 16, 4, 6, 3, 12, 15, 4, 12, 13, 16, 15,
↪ 6, 4, 4, 5, 8, 7, 4, 4, 5, 14, 13, 4, 5, 8, 17, 14, 4, 8, 7, 16, 17, 4, 7, 4, 13,␣
↪16, 4, 13, 14, 17, 16, 6, 4, 9, 10, 13, 12, 4, 9, 10, 19, 18, 4, 10, 13, 22, 19, 4,␣
↪13, 12, 21, 22, 4, 12, 9, 18, 21, 4, 18, 19, 22, 21, 6, 4, 10, 11, 14, 13, 4, 10,␣
↪11, 20, 19, 4, 11, 14, 23, 20, 4, 14, 13, 22, 23, 4, 13, 10, 19, 22, 4, 19, 20, 23,␣
↪22, 6, 4, 12, 13, 16, 15, 4, 12, 13, 22, 21, 4, 13, 16, 25, 22, 4, 16, 15, 24, 25,␣
↪4, 15, 12, 21, 24, 4, 21, 22, 25, 24, 6, 4, 13, 14, 17, 16, 4, 13, 14, 23, 22, 4,␣
↪14, 17, 26, 23, 4, 17, 16, 25, 26, 4, 16, 13, 22, 25, 4, 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**

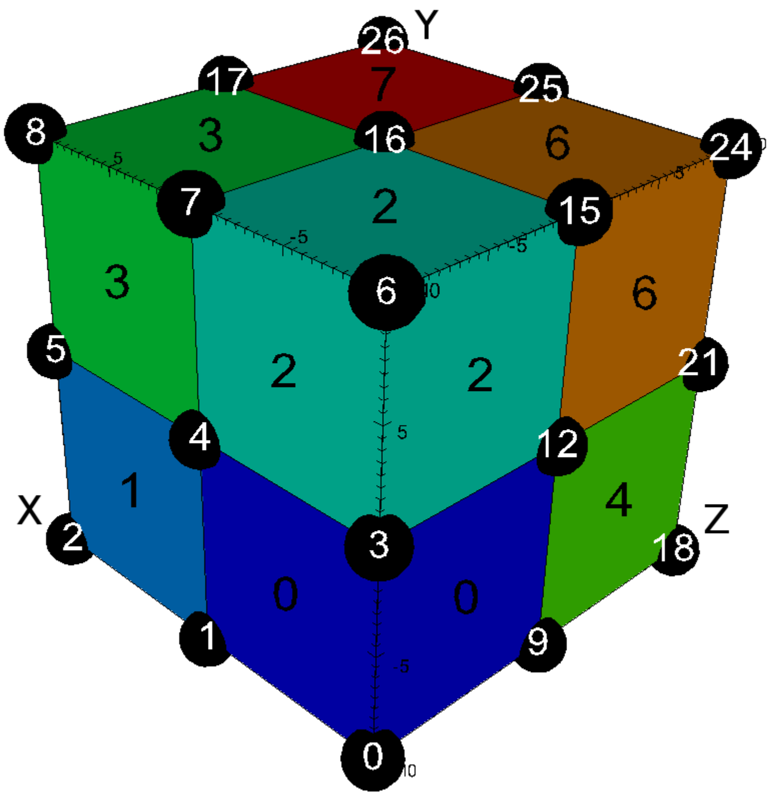

Fig. 9: Pseudocolor plot of `basic` (mesh type 'polyhedra')

braid

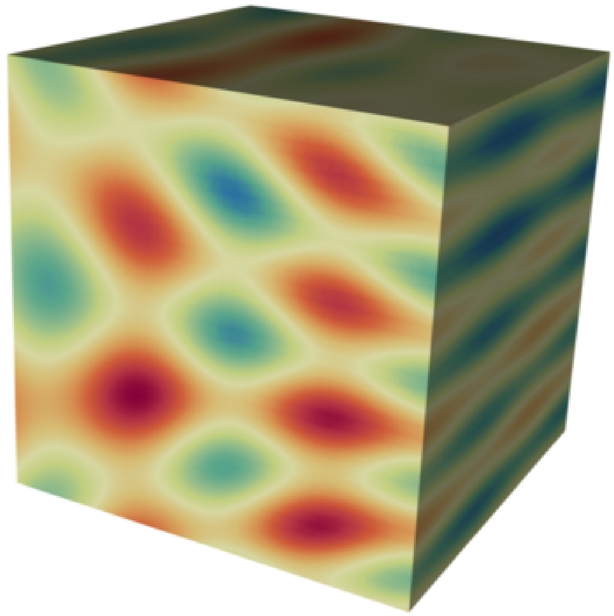

Fig. 10: Pseudocolor plot of a 3D braid example `braid` field

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,
                                          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

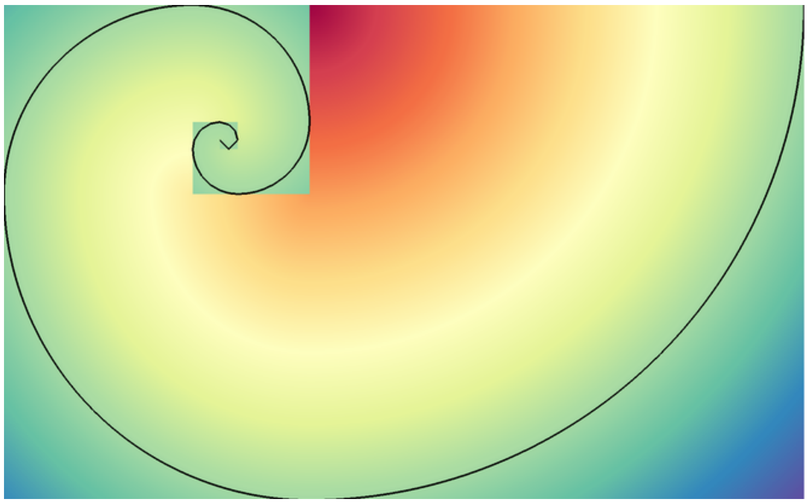

Fig. 11: Pseudocolor and Contour plots of the spiral example `dist` field.

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

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.

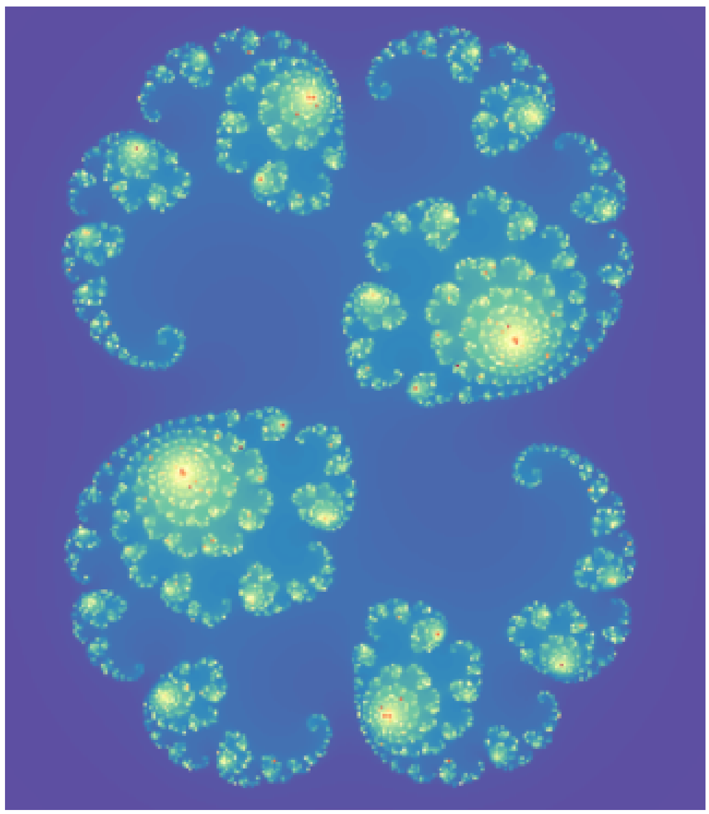

Fig. 12: Pseudocolor plot of the julia example `iter` field

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

polytess

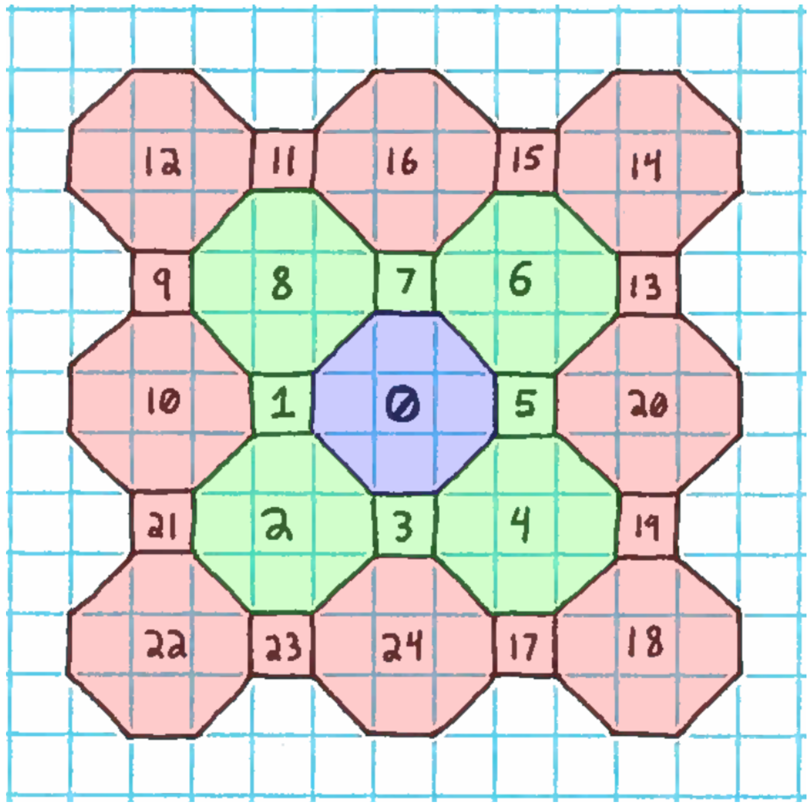

Fig. 13: Pseudocolor plot of the polytess example `level` field.

The `polytess()` function generates a polygonal tesselation in the 2D plane comprised of octogons and squares (known formally as a two-color truncated square tiling).

The scalar element-centered field `level` defined in the result mesh associates each element with its topological distance from the center of the tesselation.

```
conduit::blueprint::mesh::examples::polytess(index_t nlevels,
                                              Node &res);
```

`nlevels` specifies the number of tesselation levels/layers to generate. If this value is specified as 1 or less, only the central tesselation level (i.e. the octogon 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.

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). Provided that your mesh is sufficiently simple (see the note at the end of this section for details), you can output your mesh using one of the following `conduit::relay` library functions:

```
// saves the given mesh to disk at the given path (using the extension
// suffix in the path to inform the output data protocol)
conduit::relay::io_blueprint::save(const conduit::Node &mesh,
                                   const std::string &path);

// saves the given mesh to disk at the given path with the given explicit
// output data protocol (e.g. "json", "hdf5")
conduit::relay::io_blueprint::save(const conduit::Node &mesh,
                                   const std::string &path,
                                   const std::string &protocol);
```

It's important to note that both of these functions expect the given path to have a valid extension to properly output results. The valid extensions for these functions are as follows:

- `.blueprint_root` (JSON Extension)
- `.blueprint_root_hdf5` (HDF5 Extension)

Files output from these functions can be opened and subsequently visualized directly using VisIt.

Note: This automatic index generation and save functionality is under development. It handles most basic cases, but only supports `json` and `hdf5` output protocols and has limited multi-domain support. We are working on API changes and a more robust capability for future versions of Conduit.

Detailed Uniform Example

This snippet provides a complete C++ example that demonstrates:

- Describing a uniform mesh in a Conduit tree
- Verifying the tree conforms to the Mesh Blueprint
- Saving the result to a JSON file that VisIt 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";
```

(continues on next page)

(continued from previous page)

```
// 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));

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 json that can be read by VisIt
conduit::relay::io_blueprint::save(mesh, "basic_detailed_uniform.blueprint_root");
```

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();
```

```
{
  "coords":
  {
    "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":
  {
    "0x7fd6c0600100":
    {
      "path": "x",
      "type": "allocated",
      "bytes": 40
    },
    "0x7fd6c0600460":
    {
      "path": "y",
      "type": "allocated",
      "bytes": 40
    },
    "0x7fd6c0600130":
    {
      "path": "z",
      "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":
  {
    "0x7fd6c0602090":
    {
      "path": "",
      "type": "allocated",
      "bytes": 120
    }
  },
  "total_bytes_allocated": 120,
```

(continues on next page)

(continued from previous page)

```
  "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 `bootstrap-env.sh` and `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 `bootstrap-env.sh` (located at the root of the conduit repo) to help setup your development environment. This script uses `scripts/uberenv/uberenv.py`, which 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
source bootstrap-env.sh

#copy the generated host-config file into the standard location
cp uberenv_libs/`hostname`*.cmake to host-configs/

# run the configure helper script
./config-build.sh

# or you can run the configure helper script and give it the
# path to a host-config file
./config-build.sh uberenv_libs/`hostname`*.cmake
```

When `bootstrap-env.sh` runs `uberenv.py`, all command line arguments are forwarded:

```
python scripts/uberenv/uberenv.py $@
```

So any options to `bootstrap-env.sh` are effectively `uberenv.py` options.

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/spack_configs/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/spack_configs/
↪darwin/
```

The uberenv *–install* installs conduit@master (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 master branch run:

```
spack install conduit@master
```

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 3.3.1, 3.8.1 and 3.9.4. Other versions of CMake may work, however CMake 3.4.x to 3.7.x have specific issues with finding and using HDF5 and Python and C++11 support.

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`.

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.

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_child(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 master.
- Merge to master is only done via a pull request.
- The master should always compile and pass all tests.
- Releases are tagged off of master.

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 master. (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 v0.5.0

- 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

8.4.2 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`

8.4.3 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

8.4.4 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

8.4.5 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

8.4.6 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

8.5.1 Slides

- `SciPy 2016 talk on Conduit (July 2016)`
- `Conduit Introduction (February 2015)`

8.5.2 Talks

- SciPy 2016 talk on Conduit (July 2016)

8.5.3 Interviews

- RCE HPC Podcast on Conduit (October 2015)

8.5.4 Articles

- LLNL Article on the 2014-2015 Conduit Harvey Mudd CS Clinic Project (May 2015)

8.6 License Info

8.6.1 Conduit License

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)

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