Conduit

A Scientific Data Exchange Library for HPC Simulations

http://software.llnl.gov/conduit

SciPy 2016 – The 2016 Scientific Computing with Python Conference

Wednesday July 13, 2016
Austin, Texas

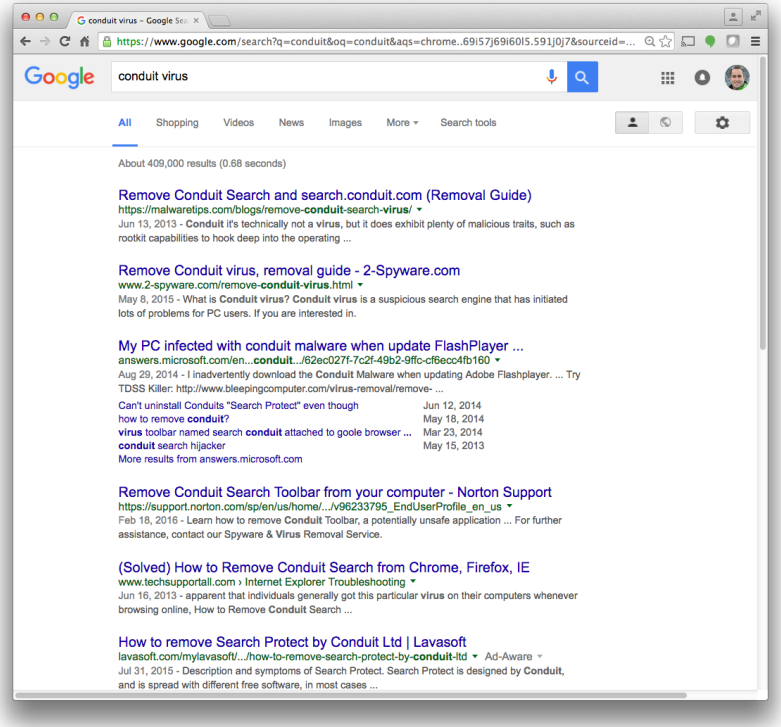
Cyrus Harrison, Brian Ryujin, Adam Kunen

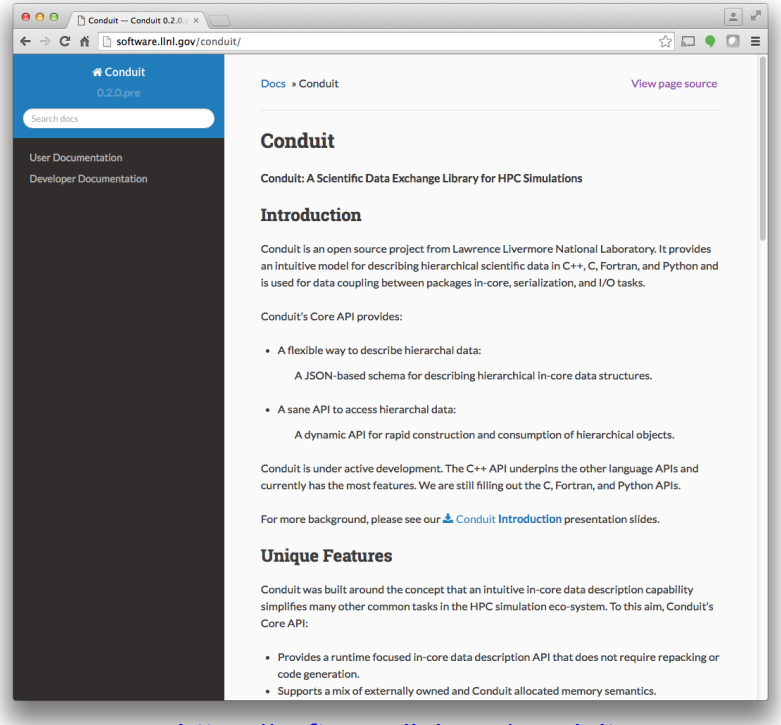
LLNL-PRES-697177

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC

Lawrence Livermore National Laboratory

Our Conduit project is not focused on developing a virus (or malware, adware, etc).





https://software.llnl.gov/conduit

Conduit is an open source project focused on simplifying in-core data exchange in the HPC simulation ecosystem.

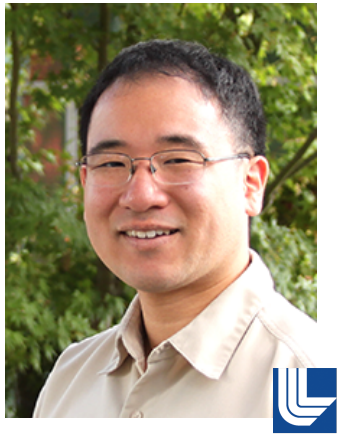
Project Info:

- Languages: C++, Python, C, Fortran
- Docs: https://software.llnl.gov/conduit
- Github Repo: https://github.com/llnl/conduit
- License: BSD Style
- Timeline:
  - 2011 – 2013: Neurons start firing and developing concepts
  - November 2013: Code development starts at a LLNL Hackathon
  - Fall 2014 – Spring 2015: Harvey Mudd Clinic Project
  - January 2015: Released Open Source
  - 2015 – Present: Early use in applications at LLNL

Conduit is small collaborative development effort with in LLNL's Weapons Simulation and Computing (WSC) program.

Core Team:

Cyrus Harrison
cyrush@llnl.gov

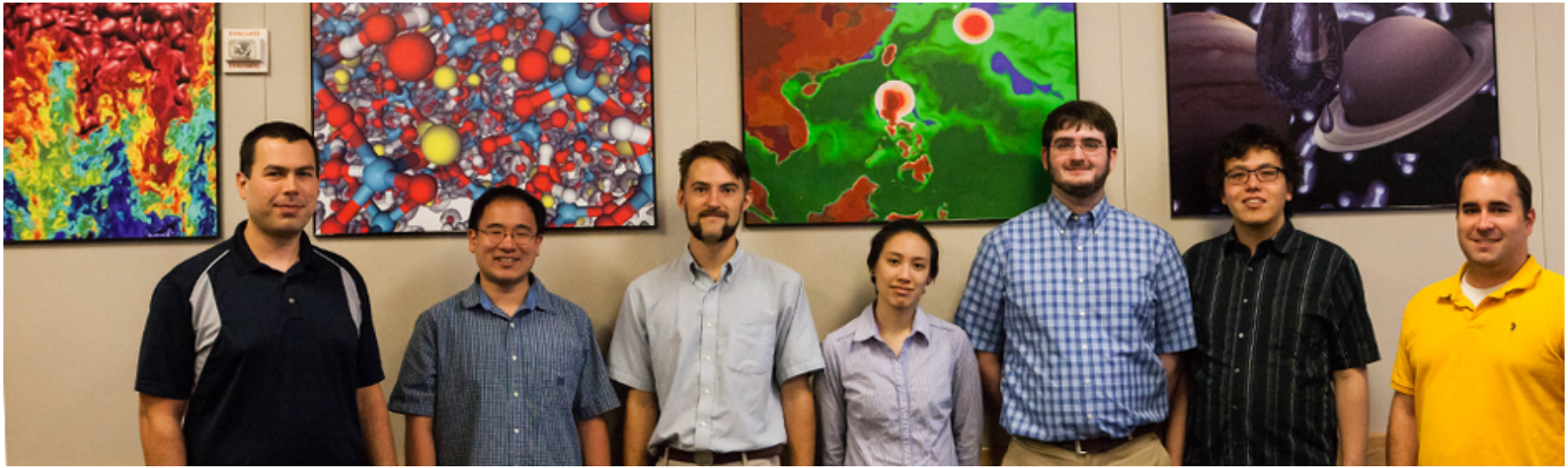


Brian Ryujin
ryujin1@llnl.gov

Adam Kunen
kunen1@llnl.gov

Many other people have contributed ideas and code!

Conduit development was boosted by a 2014-2015 Harvey Mudd Clinic Project exploring its use in HPC proxy applications.



The Harvey Mudd Clinic program helped us refine Conduit for important use cases

What motivated Conduit?

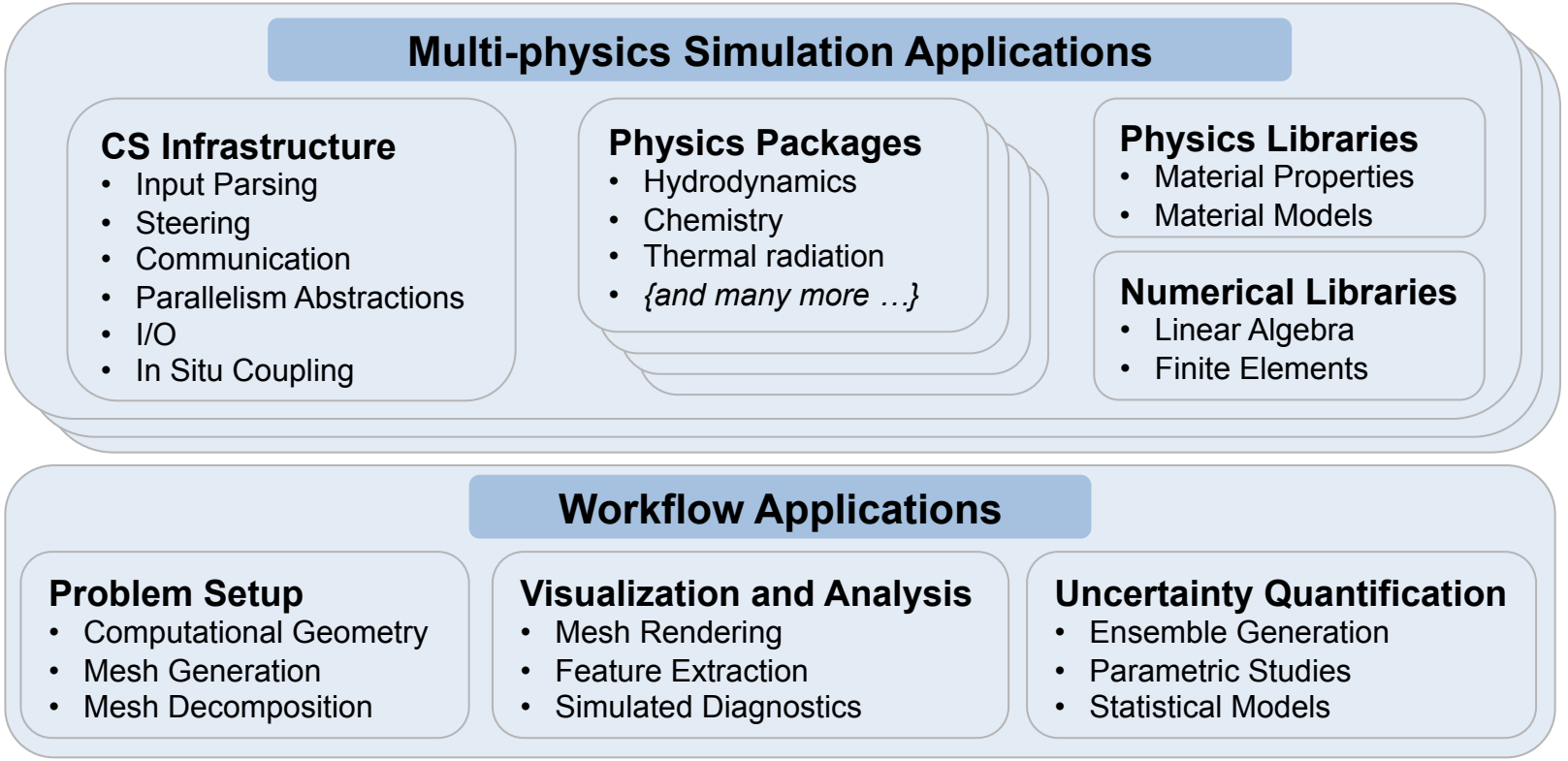
LLNL's WSC Computational Physics (CP) program develops production HPC multi-physics simulation applications.

Snapshot of Scale and Complexity of LLNL's WSC CP Program:

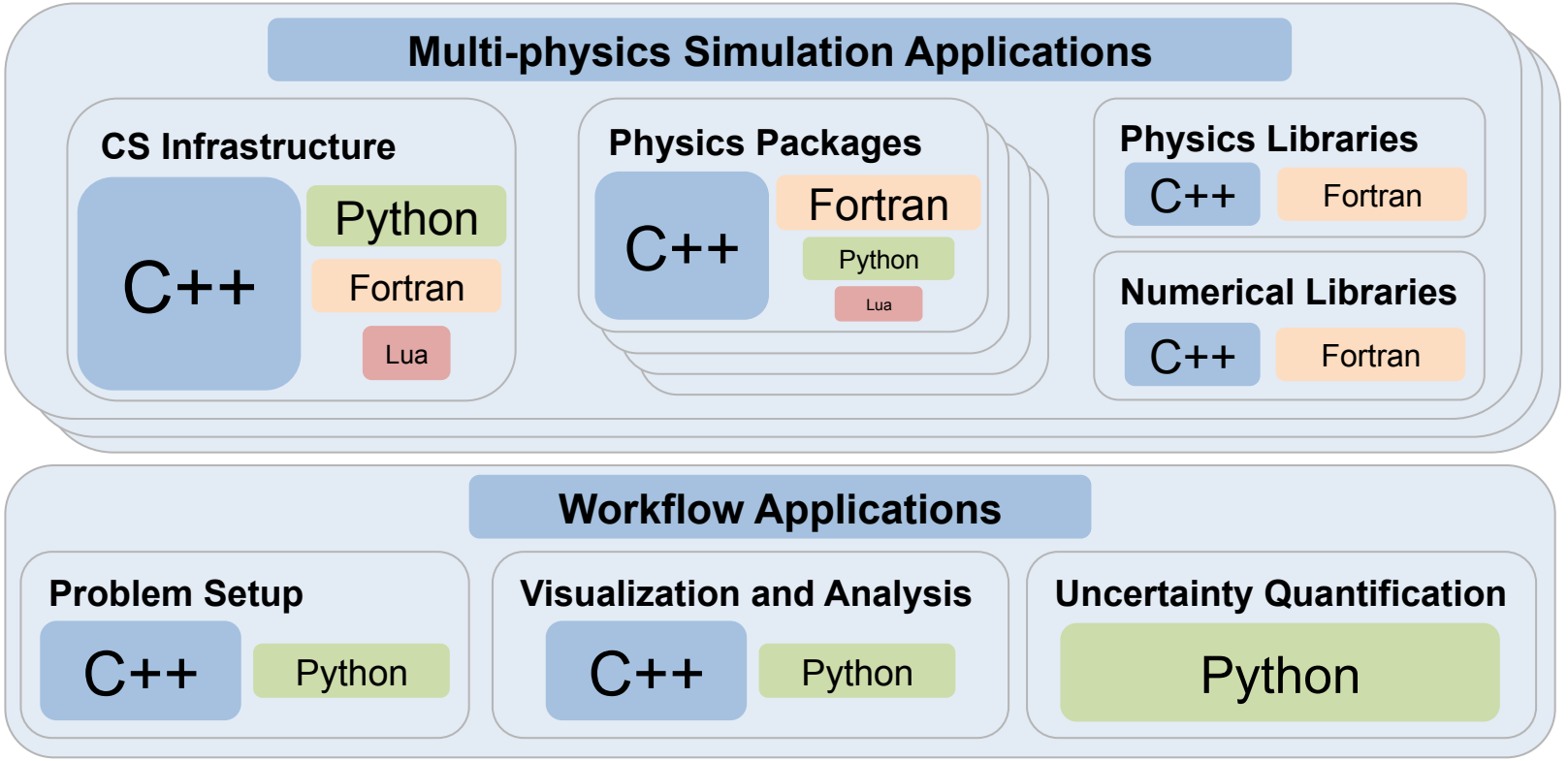
- Development Efforts
  - —~120 Developers (Physicists, Engineers, Computer Scientists, Software Quality, etc …)
  - —~15 project teams
  - —~15 – 30+ year application lifetimes
  - —~12 million lines of production code across projects (w/ more than 100 third-party dependencies)
- Diversity of Programming Languages
  - —C++/C, Fortran, Python, Lua
- Diversity of Data
  - —Scalars, Arrays, Strings
  - —Tables, Contours
  - —CAD Geometry, Computational Meshes

CP's efforts are a microcosm of the broader HPC simulation community.

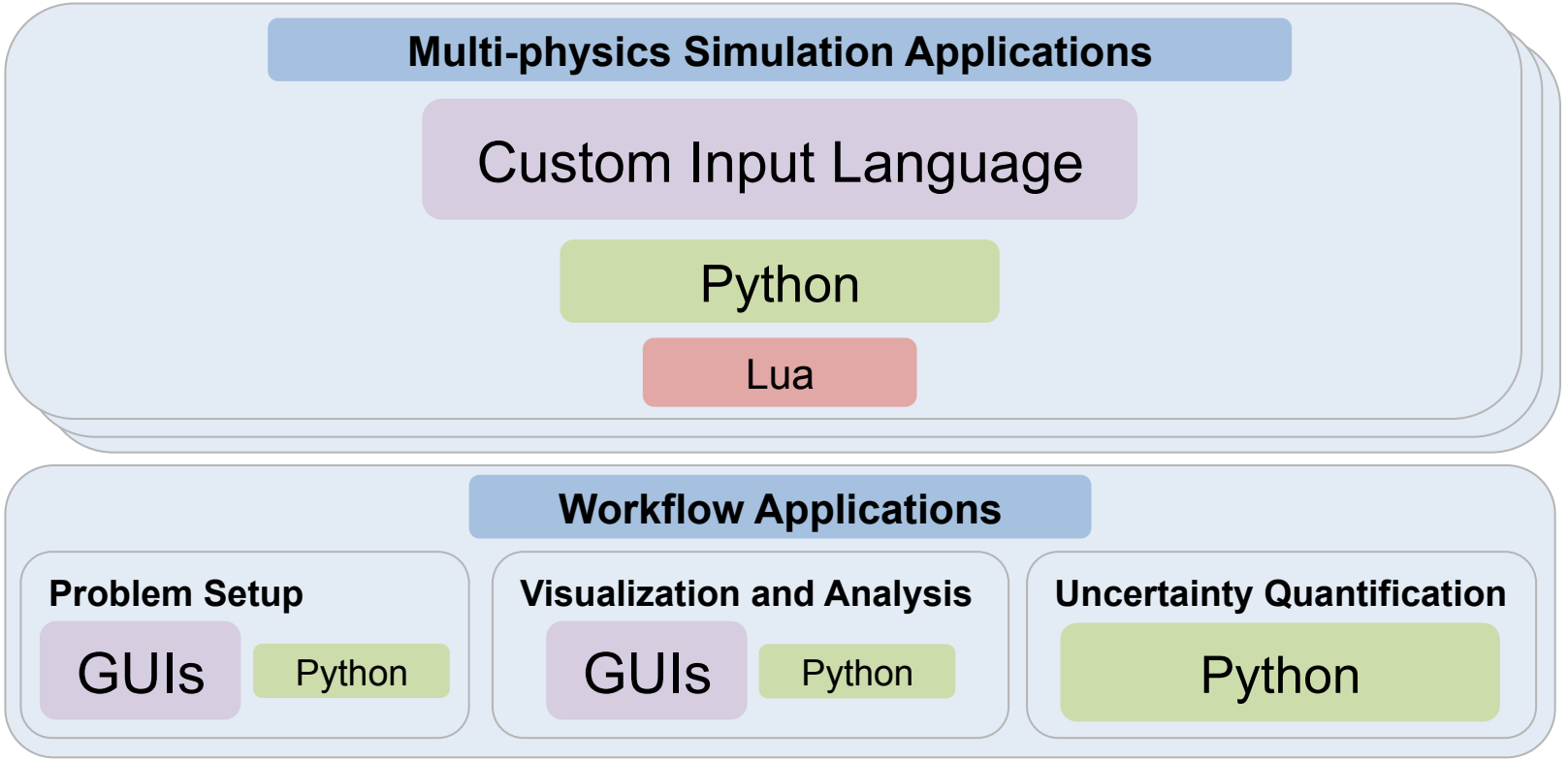
The software ecosystem supporting HPC multi-physics simulations is very complex.



A mix of C++, Python, Fortran and Lua are used to develop applications.



A mix of custom input languages, GUIs, Python and Lua are used to run and script applications.



Data coupling is a key aspect in simulation software design and user workflows.

In-core data exchange has received less focus than file-based data exchange:

- File-based I/O libraries have evolved into defacto interfaces between components.
- File-based I/O is acceptable as a coarse-grain and low-frequency data coupling solution.

To achieve fine-grained data coupling we need tools to help with in-core data exchange.

Supporting in-core data exchange throughout the simulation eco-system is quite different from trivial single component use cases.

Key Requirements:

- Provide a description mechanism for numeric primitives
  - Scalars, strided arrays, etc with explicit precision
- Support mixed memory ownership semantics
  - Enable zero-copy where feasible
  - Play friendly with existing data structures in multiple languages
- Enable higher level conventions
  - Hierarchical context (“paths” are great interface…)
  - Human readable descriptions

What is Conduit?

Conduit provides a set of tools focused on in-core data description to aid with data coupling across the HPC ecosystem.

Conduit's multi-language data model provides:

- A flexible way to ***describe*** complex data
  - A JSON-based schema for describing the layout of hierarchical in-core data.
- A sane API to ***access*** complex data
  - Dynamic APIs for rapid construction and consumption of hierarchical objects.

Conduit provides a multi-language data model,
not a multi-language wrapping solution.

The heart of Conduit is a hierarchical variant type named *Node*.

A *Node* acts as one of following basic roles:

- ***Object***: An ordered associative array mapping names to children
- ***List***: An ordered list of unnamed children
- ***Leaf***: Scalar or 1D Array of a bitwidth-specified primitive:
  - *Signed Integers*: int8, int16, int32, int64
  - *Unsigned Integers*: uint8, uint16, uint32, uint64
  - *Floating Point Numbers*: float32, float64
  - *Strings*: char8_str
- ***Empty***: No data

Experience with NumPy and JSON motivated Conduit's data model.

Conduit Python API Example: Create a tree of NumPy data arrays

```
import numpy as np
from conduit import Node

# create a "Node", the primary actor in conduit
n = Node()

# create example numpy data
x_coords = np.array(range(3),dtype='float64')
y_coords = np.array(range(3),dtype='float64')
den      = np.ones(4,dtype='float64')
```

Conduit Python API Example: Create a tree of NumPy data arrays

```python
# create dynamic a tree hierarchy with a Node by
# copying our numpy data into named paths
n["coords/x"] = x_coords
n["coords/y"] = y_coords
n["fields/density/values"] = den
# show the tree
print(n)
```

```
{
  "coords":
  {
    "x": [0.0, 1.0, 2.0],
    "y": [0.0, 1.0, 2.0]
  },
  "fields":
  {
    "density":
    {
      "values": [1.0, 1.0, 1.0, 1.0]
    }
  }
}
```

Conduit Python API Example:
Mixing externally allocated and Conduit owned data

```
# you can mix external and conduit owned data within a
# Node hierarchy
vel_u = np.ones(9,dtype='float64')
vel_v = np.ones(9,dtype='float64')
# use set_external() to inits the "u" and "v" nodes of
# the tree to point to the same memory location as their
# source numpy arrays
n["fields/velocity/values/u"].set_external(vel_u)
n["fields/velocity/values/v"].set_external(vel_v)
```

Conduit Python API Example:
Mixing externally allocated and Conduit owned data

```
# show the elements of the "u" array
print(n["fields/velocity/values"])
```

```
{
  "u": [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0],
  "v": [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
}
```

```
# change the first element of the u array (via src)
vel_u[0] = 3.14159;
# change the first element of the v array (via conduit)
n["fields/velocity/values/v"][0] = 2.71828

# show the elements of the "u" array again
print(n["fields/velocity/values"])
```

```
{
  "u": [3.14159, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0],
  "v": [2.71828, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
}
```

Conduit Python API Example:
Mixing externally allocated and Conduit owned data

```
# mixed ownership semantics allow you to organize,
# extend, and annotate existing data
n["coords/type"] = "uniform"
n["fields/density/units"] = "g/cc"
n["fields/velocity/units"] = "m/s"
# show the entire tree
print(n)
```

Conduit Python API Example:
Mixing externally allocated and Conduit owned data

```
{
  "coords":
  {
    "x": [0.0, 1.0, 2.0],
    "y": [0.0, 1.0, 2.0],
    "type": "uniform"
  },
  "fields":
  {
    "density":
    {
      "values": [1.0, 1.0, 1.0, 1.0],
      "units": "g/cc"
    },
    "velocity":
    {
      "values":
      {
        "u": [3.14159, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0],
        "v": [2.71828, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
      },
      "units": "m/s"
    }
  }
}
```

We are aiming to provide intuitive C++, Python, and Fortran APIs.

Python

```
den = np.ones(4,dtype='float64')

n = Node();

n["fields/density/values"] = den;
n["fields/density/units"]  = "g/cc";

n_density = n["fields/density"]

den_vals  = n_density["values"];
den_units = n_density["units"];

print(n_density)

print('\nDensity ({}):\n{}'.format(den_units,
                                   den_vals))
```

C++

```
std::vector<float64> den(4,1.0);

Node n;
n["fields/density/values"] = den;
n["fields/density/units"]  = "g/cc";

Node &n_density = n["fields/density"];

float64 *den_ptr = n_density["values"].value();
std::string den_units = n_density["units"].as_string();

n_density.print();

std::cout <<"\nDensity (" << den_units << "):\n";

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

Fortran

```
do i = 1,4
    den(i) = 1.0
enddo

n = conduit_node_obj_create()
call n%set_path_ptr("fields/density/values",den,4_8)
call n%set_path("fields/density/units","g/cc")

n_den = n%fetch("fields/density")

call n_den%fetch_path_as_float64_ptr("values",d_arr)

n_den_units = n%fetch("fields/density/units")
call n_den_units%as_char8_str(units)
units_len = n_den_units%number_of_elements()

call n_den%print()
print *,
print *,"Density (",(units(i),i=1,units_len),"): "

do i = 1,4
    write(*,"(f5.2,1x)",advance="no"), d_arr(i)
enddo
print *

call conduit_node_obj_destroy(n)
```

We are aiming to provide intuitive C++, Python, and Fortran APIs.

Python

```
{
  "values": [1.0, 1.0, 1.0, 1.0],
  "units": "g/cc"
}

Density (g/cc):
[ 1.  1.  1.  1.]
```

C++

```
{
  "values": [1.0, 1.0, 1.0, 1.0],
  "units": "g/cc"
}

Density (g/cc):
1 1 1 1
```

Fortran

```
{
  "values": [1.0, 1.0, 1.0, 1.0],
  "units": "g/cc"
}

 Density (g/cc):
 1.00  1.00  1.00  1.00
```

Conduit was designed with software engineering ecosystem logistics in mind.

- Completely runtime focused
  - Avoids incompatible (or unsharable) code-generation solutions
- Emphasizes data description as a core capability
  - Does not require repacking
  - Provides a foundation on which to build serialization, I/O, and messaging features
- Provides a multi-language data model
  - Currently includes APIs for C++, Python, C, and Fortran
  - JSON friendly

Philosophy: Share data without massive code infrastructure.

Conduit's *Relay* and *Blueprint* libraries provide features built on top of Conduit's core data model.

Conduit

Implements interfaces to Conduit's in-core data model.

- Core Objects
- JSON parsing
- Basic I/O
- Basic transforms

Relay

Provides advanced I/O features built on top of Conduit's data model.

- HDF5
- MPI
- WebSockets

Blueprint

Supports shared higher level conventions for using Conduit to represent data.

- Computational Meshes
- Multi-Component Arrays

Where are we using Conduit?

Our current production simulation code projects develop and maintain their own CS infrastructure.

Multi-physics Simulation Applications

CS Infrastructure
- Input Parsing
- Steering
- Communication
- Parallelism Abstractions
- I/O
- In Situ Coupling

Physics Packages
- Hydrodynamics
- Chemistry
- Thermal radiation
- *{and many more …}*

Physics Libraries
- Material Properties
- Material Models

Numerical Libraries
- Linear Algebra
- Finite Elements

Workflow Applications

Problem Setup
- Computational Geometry
- Mesh Generation
- Mesh Decomposition

Visualization and Analysis
- Mesh Rendering
- Feature Extraction
- Simulated Diagnostics

Uncertainty Quantification
- Ensemble Generation
- Parametric Studies
- Statistical Models

LLNL's ASC CS Toolkit is a new project working to refine and share CS infrastructure across simulation applications.

ASC CS Toolkit

CS Infrastructure
- Data Management
- Communication
- Parallelism Abstractions
- I/O
- In Situ Coupling

Multi-physics Simulation Applications

CS Infra.

Physics Packages
- Hydrodynamics
- Chemistry
- Thermal radiation
- *{and many more …}*

Physics Libraries
- Material Properties
- Material Models

Numerical Libraries
- Linear Algebra
- Finite Elements

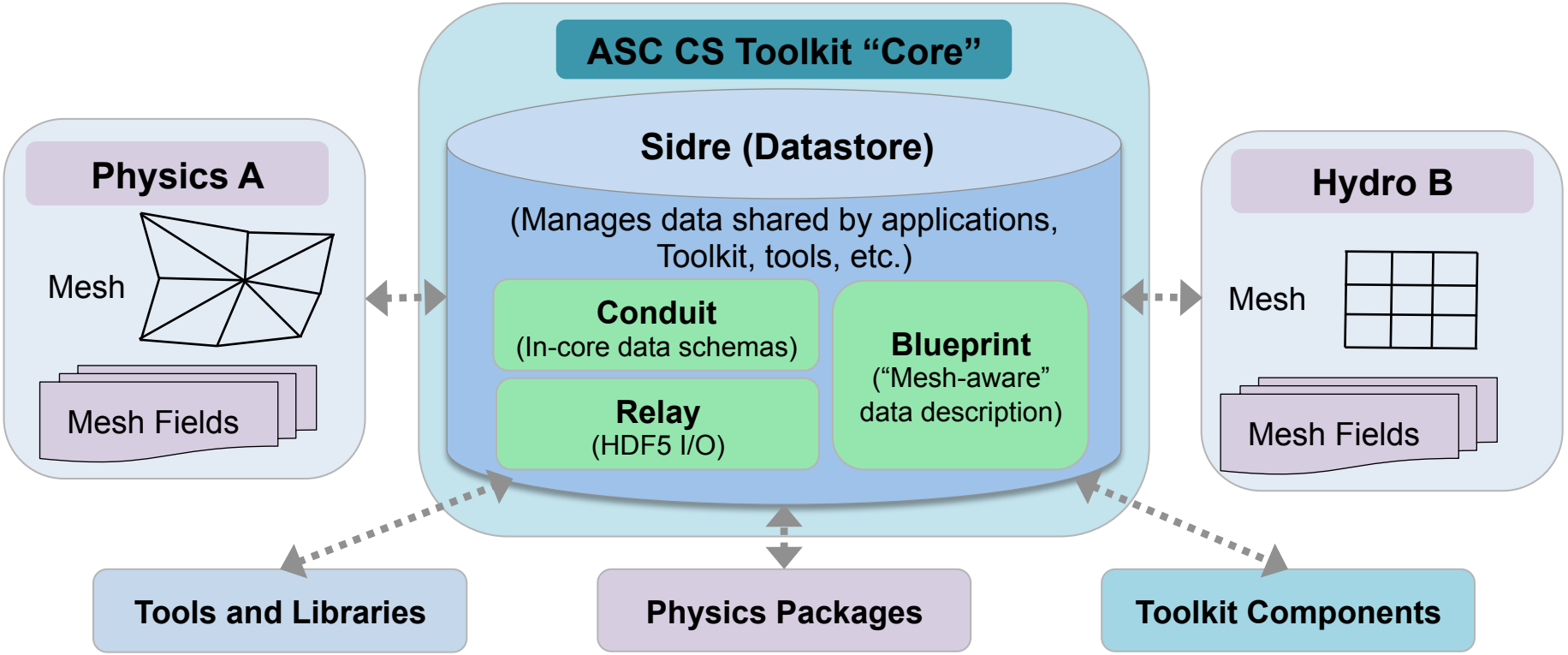
Workflow Applications

Problem Setup
- Computational Geometry
- Mesh Generation
- Mesh Decomposition

Visualization and Analysis
- Mesh Rendering
- Feature Extraction
- Simulated Diagnostics

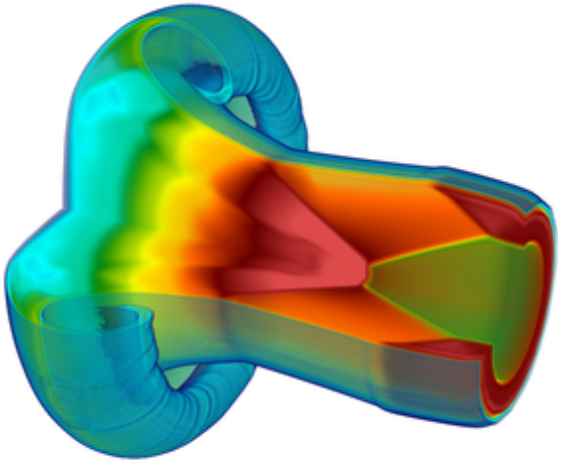
Uncertainty Quantification
- Ensemble Generation
- Parametric Studies
- Statistical Models

Simulation components will use the ASC CS Toolkit and common conventions to share data.

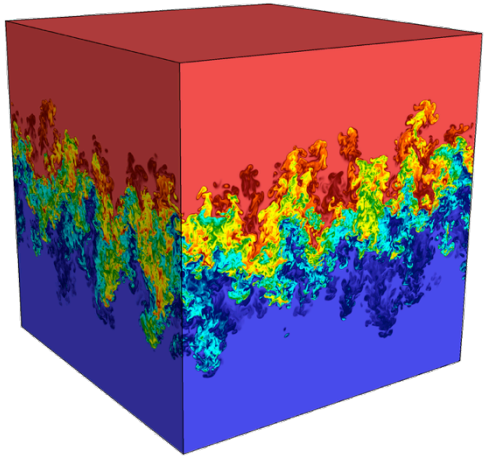


LLNL is developing a new code, MARBL, which employs two modular hydrodynamics algorithms.

BLAST is an unstructured mesh finite element ALE code that uses high-order elements and geometry



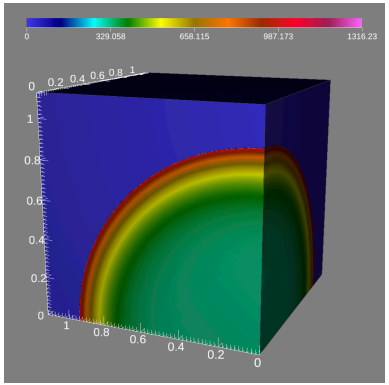
MIRANDA is a structured mesh Eulerian code that uses high-order finite difference methods

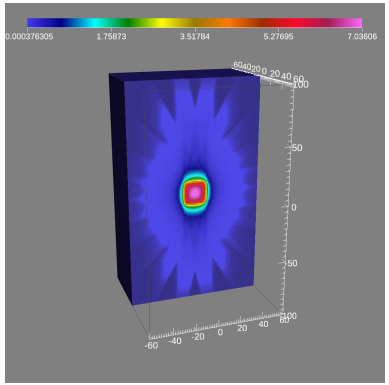


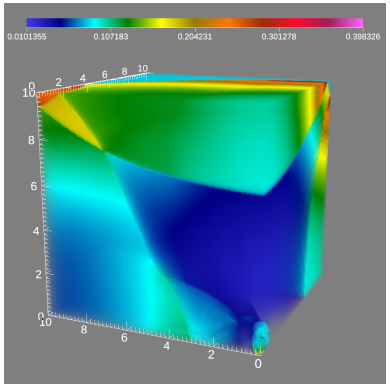
MARBL will be the first application to incorporate the data management and I/O features of the ASC CS Toolkit.

Strawman is an proxy application for in situ rendering in mesh-based HPC simulations.

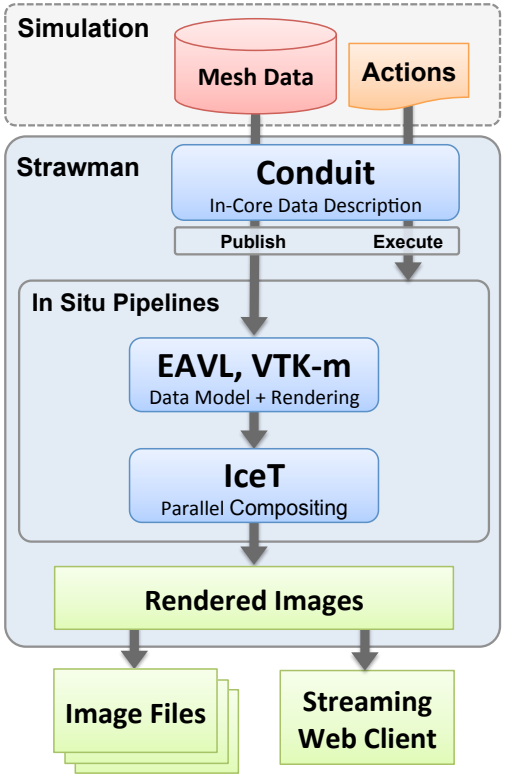
Example renders from built-in sample simulations:







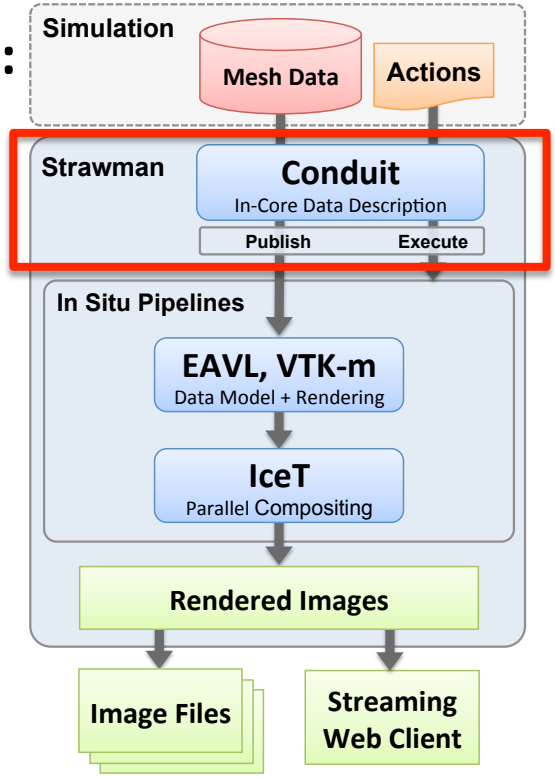
LULESH	Kripke	CloverLeaf3D
Hydrodynamics	Neutron Transport	Hydrodynamics
Unstructured	Uniform	Rectilinear
C++	C++	Fortran



Strawman uses Conduit to describe and pass simulation mesh data structures to MPI+X rendering pipelines.

Example: Describing mesh data using Conduit (C++):

```
conduit::Node data;
data["state/time"].set_external(&time);
data["state/cycle"].set_external(&cycle);
data["state/domain"] = my_mpi_rank;
data["coords/type"] = "explicit";
data["coords/x"].set_external(x);
data["coords/y"].set_external(y);
data["coords/z"].set_external(z);
data["topology/type"] = "unstructured";
data["topology/coordset"] = "coords";
data["topology/elements/shape"] = "hexs";
data["topology/elements/connectivity"].set_external(nodelist);
data["fields/e/association"] = "element";
data["fields/e/type"]        = "scalar";
data["fields/e/values"].set_external(e);
```



Conduit is an open source project focused on simplifying in-core data exchange in the HPC simulation ecosystem.

Project Info:

- Docs: https://software.llnl.gov/conduit
- Repo: https://github.com/llnl/conduit
- License: BSD Style

Contact Info:

- Cyrus Harrison (cyrush@llnl.gov)
- Brian Ryujin (ryujin1@llnl.gov)
- Adam Kunen (kunen1@llnl.gov)

Conduit is a work in progress!

- Future Work:
  - Using Conduit in HPC applications
  - Expanding API Support
    - Current State: C++ > Python > C/Fortran
    - Lua (?)
  - Refining *Relay* and *Blueprint*
  - Windows support
  - Binary distributions (?)


Lawrence Livermore
National Laboratory