ABSTRACT
We present Strawman, a system designed to explore the in situ visualization and analysis needs of simulation code teams planning for multi-physics calculations on exascale architectures. Strawman’s design derives from key requirements from a diverse set of simulation code teams, including lightweight usage of shared resources, batch processing, ability to leverage modern architectures, and ease-of-use both for software integration and for usage during simulation runs. We describe the Strawman system, the key technologies it depends on, and our experiences integrating Strawman into three proxy simulations. Our findings show that Strawman’s design meets our target requirements, and that some of its concepts may be worthy of integration into our community in situ implementations.

1. INTRODUCTION
Like many other institutions, Lawrence Livermore National Laboratory (LLNL) and the Atomic Weapons Establishment (AWE) are moving towards in situ processing to handle the growing divide between computational power and I/O capabilities on modern supercomputers. At these laboratories, the simulation codes are often very complex — meaning multi-physics codes incorporating many libraries — and diverse — e.g., multiple programming languages, data models, etc. With this short paper, we describe our experiences designing, implementing, and evaluating a system to respond to these simulation code teams’ requirements. We call this system Strawman. We consider Strawman to be a mini-app, i.e., it is designed to be lightweight and allow us to perform research.

LLNL and AWE are home to more than fifty different simulation code groups. While our requirements are not distilled from conversations with all of these groups, they do represent commonalities from conversations across many of them. Further, in this paper, we present results from integrating Strawman with three prominent proxy simulations: LLNL’s Lulesh [6, 16] and Kripke [17, 8] and AWE’s CloverLeaf3D [23, 4]. Our experiences have led us to believe the system has promising elements for meeting their needs.

The remainder of the paper describes Strawman and our experiences with it. Section 2 describes technologies that we used within Strawman, and Section 3 describes the requirements and design of Strawman. Then, in Section 4, we describe the results of integrating Strawman into the three proxy simulations.

2. BACKGROUND
Both Paraview Catalyst [15, 22] and VisIt’s LibSim [31] provide full-featured tightly-coupled in situ visualization and analysis solutions for simulations running on current HPC platforms. As they are both fully featured and have complex designs, they are not well suited for quick evaluation of new technologies. With Strawman, we have developed a small system that enabled us to explore solutions that may help us evolve the state of the art in tightly-coupled in situ processing for batch use cases on modern and future HPC architectures. We hope to test if solutions are viable enough to be considered for use in larger production systems. To build Strawman, we heavily leveraged three libraries: Conduit, EAVL, and IceT. This section provides background information on these libraries and why they were selected.

2.1 Conduit
Conduit [5] is an open source development effort from LLNL that provides an interface for in-core description of scientific data that can be used across C++, C, Python, and Fortran. It is used for data coupling between packages in-core, serialization, and I/O tasks.

Conduit provides a hierarchical object model similar to JSON [7], but differs from JSON and other message exchange formats such as BSON [2], Protocol Buffers [10], and Apache
Thrift [1] in a few fundamental ways:

- **Bit-width Style Leaf Types** To describe scientific data in-core, you must capture the specific details about your data in memory, such as the bit-width used for numeric primitives, offsets, endianness, etc. To address this, Conduit provides bit-width specified scalar and array leaf types. Arrays can be contiguous or strided. The bit-width descriptions follow the style of NumPy [30]. Leaf types are always concretely specified in Conduit, however the API also allows you to use native types. For example in C++, when using native double data, Conduit will map this to the concrete bit-width type for the current platform, usually Conduit’s float64 type.

- **Separation of Description from Data** To describe existing in-core data, you cannot use a solution that embeds a description mechanism into the data. Because of this, Conduit separates description metadata from data and does not impose a packed data representation. This design supports zero-copy and allows a Conduit to bring context to chunks of data allocated in different regions of memory.

- **Runtime Focus** Many existing message exchange formats utilize code generation to implement an object model. Conduit provides a runtime API for describing and accessing data and introspection features that allow you to query and discover the structure of described data. Conduit does not use compile time code generation. This avoids complex build system requirements and makes it easier for components across the simulation ecosystem to share data. This design does have runtime costs and shifts some errors to runtime. These tradeoffs are similar to properties of weakly typed languages like Python. The runtime costs are acceptable for simulation applications where descriptions of the data will be small compared to the large arrays that hold the simulation state.

These features make Conduit a suitable multi-language solution for passing mesh data and commands from a simulation to a visualization pipeline.

### 2.2 EAVL

Scientific visualization libraries often contain hundreds of algorithms. As a result, achieving portable performance across varied supercomputing architectures is difficult. To address this problem, a number of visualization infrastructures (e.g., DAX [26], EAVL [24], and PISTON [20]), were created that leverage data parallel primitives [13]. By using an abstraction that encapsulates data-parallel concepts, these visualization infrastructures provide node level parallelism for CPU, GPU, and MIC architectures. Further, the libraries have shown that the architecture agnostic approach can achieve performance of within roughly a factor of two when compared to highly optimized, architecture specific implementations [19]. These three projects have merged their efforts into a single library called VTK-m. For Strawman, we chose to use EAVL, since VTK-m is in the initial stages of development and lacks a rendering infrastructure.

For building an in situ visualization application, EAVL is an ideal building block for hybrid parallelism, since it decouples architecturally specific node level parallelism and distributed-memory parallelism. EAVL’s data model, as described in [24], is expected to consume 70 to 75 percent less memory because of its use of an underlying array class that can do in-place memory writes. The flexibility of the data model also opens up more memory layouts for zero-copy. Further, EAVL has a rendering infrastructure that includes OpenGL, ray tracing, and volume rendering options [18]. These features make EAVL a natural choice for rendering the subset of a distributed mesh on each MPI task in a distributed-memory simulation.

### 2.3 IceT

Producing a coherent image from a distributed render requires parallel compositing. IceT [25, 27] provides a scalable solution for compositing using MPI on distributed-memory parallel systems. It implements a sort-last compositing with a suite of efficient parallel partitioning and reduction strategies [28]. We selected IceT as our compositing solution for this effort since it has been used at large scales for years in production visualization tools including Paraview, VisIt, and their in situ offerings.

### 3. SYSTEM OVERVIEW

This section provides details on the requirements and design of Strawman.

#### 3.1 System Requirements

To guide the development of our mini-app, we selected a set of important batch in situ visualization and analysis requirements extracted from our interactions and experiences with simulation code teams. Here are our 11 requirements broken out into three broader categories:

- **Support a diverse set of simulations on many-core architectures.**
  - R1: Support execution on many-core architectures.
  - R2: Support usage within a batch environment (i.e., no simulation user involvement once the simulation has begun running).
  - R3: Support the four most common languages used by simulation code teams: C, C++, Python, and Fortran.
  - R4: Support for multiple data models, including uniform, rectilinear, and unstructured grids.

- **Provide a streamlined interface to improve usability.**
  - R5: Provide straight forward data ownership semantics between simulation routines and visualization and analysis routines.
  - R6: Provide a low-barrier to entry with respect to developer time for integration.
  - R7: Ease-of-use in terms of directing visualization and analysis actions to occur during runtime.
  - R8: Ease-of-use in terms of consuming visualization results, including delivery mechanisms both for images on a file system and for streaming to a web browser.

- **Minimize the resource impacts on host simulations.**
  - R9: Synchronous in situ processing, meaning that visualization and analysis routines can directly access the memory of a simulation code.
The Strawman API consists of only a few function calls. The simplicity of the actions interface supports \textbf{R7}. These actions are then translated into a simple pipeline and executed as described in 3.2.2.

\subsection*{3.2.2 In Situ Pipeline}

The in situ pipeline converts the mesh data described using Conduit and passed via \textit{Publish} into concrete EAVL data sets and then uses the action descriptions passed via \textit{Execute} to construct a simple rendering pipeline. The pipeline uses EAVL’s filters and rendering infrastructure, including surface, volume, and ray-casting rendering techniques, to render the subsets of the data owned by each MPI task. As discussed in 2.2, algorithms in EAVL are designed to run on many-core architectures, supporting \textbf{R1} and EAVL’s data model has zero-copy features supporting \textbf{R9} and \textbf{R11}.

We made enhancements to EAVL to support rendering in a distributed-memory parallel setting. For example, we added visibility ordering calculations to support volume rendering image compositing and data extent reductions in order to apply consistent color tables across MPI tasks. IceT is then used to composite these renders using MPI to create a final image.

\subsection*{3.2.3 Presentation of Results}

To satisfy \textbf{R8}, Strawman saves rendered images to PNG files, and includes a lightweight embedded web server [3] that can be used to optionally stream results over a WebSocket. To allow a simulation user to view the rendered images via a web browser, it also provides a WebSocket client that displays images as they are streamed from the WebSocket connection.

\section{4. RESULTS}

To test in situ integration with a range of simulations and mesh types, we selected three proxies for larger production multi-physics simulations: \textit{LULESH}, \textit{Kripke}, and \textit{CloverLeaf3D}. \textit{LULESH} is written in C++ and uses a Lagrangian approach on a 3D unstructured hex mesh to simulate shock hydrodynamics. \textit{Kripke} is written in C++ and uses a deterministic discrete ordinates solver on a 3D uniform mesh to simulate particle transport. \textit{CloverLeaf3D} is written in Fortran and solves the compressible Euler equations on a 3D rectilinear mesh to simulate hydrodynamics. Figure 2 shows images produced from the integrations with each of the simulations.
4.1 Integration Experiences

To evaluate the barrier of entry R6 and ease-of-use R7 requirements, we present the lines of code required to integrate Strawman into each of these proxy simulations to produce the simplest and most common visualization result, a Pseudocolor rendering.

As outlined in 3.2.1, integrating Strawman into a simulation can be broken down into three steps: describing simulation mesh data, describing the set of in situ actions for Strawman to perform, and calling the Strawman API to Publish the mesh data and Execute the specified actions. To demonstrate these three steps, we show code fragments for each step from our LULESH Strawman integration.

**Listing 1: Describing the simulation data.**

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

**Listing 2: Describing the actions to perform.**

```cpp
conduit::Node actions;
conduit::Node &add = actions.append();
add["action"] = "AddPlot";
add["var"] = "p";
conduit::Node &draw = actions.append();
draw["action"] = "DrawPlots";
conduit::Node &save = actions.append();
char output_filename[30];
sprintf(output_filename,"image%04d",cycle);
save["action"] = "SaveImage";
save["fileName"]=output_filename;
save["format"] = "png";
save["width"] = 1024;
save["height"] = 1024;
```

**Listing 3: Calling the Strawman API to publish data and execute the specified actions.**

```cpp
Strawman strawman;
Node options;
options["mpi_comm"] = mpi_comm_handle(MPI_COMM_WORLD);
strawman.Open(options);
strawman.Publish(data);
strawman.Execute(actions);
strawman.Close();
```

In the case of LULESH, the simulation data model exactly matches EAVL’s data model and all the data can be passed zero-copy to Strawman. Because of this, LULESH requires the least amount of Strawman integration code. In the case of Kripke, it was necessary to copy the field data since the array ordering did not match EAVL’s data model. In the case of CloverLeaf3D, it was necessary to copy the coordinate and field data to remove the embedded ghost zones, which Strawman currently does not support. The difference in lines of code required between the proxy simulations is due entirely to code related to modifying the simulation data to match EAVL’s data model.

Looking at the complexity of listings 2 and 3 we assert it is straightforward to instruct Strawman on the actions to perform. Listing 1 demonstrates the most complex step, describing simulation mesh data. This resembles conventional transformations simulation developers are already familiar with from supporting post processed visualization and analysis tools.

Based on the lines of code to instrument the three simulation codes and the complexity of the code fragments we believe we have satisfied R6 and R7.

4.2 Performance

To evaluate the performance of our infrastructure, we ran Strawman with batch jobs using all three proxy simulations on 4096 cores with mesh sizes ranging between 4 billion and 8 billion cells, satisfying R2. Table 2 shows the average time per cycle of both visualization and simulation. We will explore the performance characteristics in more depth in future work to understand how Strawman addresses R10.

5. CONCLUSION

We described our mini-app for in situ, Strawman, which enabled us to explore a system meeting our stakeholders requirements: modern architectures, streamlined user interface, and minimal burden on the simulation. The results were promising, especially in terms of streamlined user interface, although we believe more effort is needed to reduce the time to carry out visualization operations.

<table>
<thead>
<tr>
<th>Data Description</th>
<th>LULESH</th>
<th>Kripke</th>
<th>CloverLeaf3D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vis</td>
<td>15 loc</td>
<td>21 loc</td>
<td>39 loc</td>
</tr>
<tr>
<td>Sim</td>
<td>14 loc</td>
<td>14 loc</td>
<td>14 loc</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Strawsman API Calls</th>
<th>LULESH</th>
<th>Kripke</th>
<th>CloverLeaf3D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vis</td>
<td>7 loc</td>
<td>7 loc</td>
<td>9 loc</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Simulation</th>
<th>Vis</th>
<th>Sim</th>
</tr>
</thead>
<tbody>
<tr>
<td>CloverLeaf3D 4B Cells (Ray Tracing)</td>
<td>1.68</td>
<td>5.77</td>
</tr>
<tr>
<td>Kripke 4B Cells (OSMesa)</td>
<td>3.80</td>
<td>16.55</td>
</tr>
<tr>
<td>LULESH 8B Cells (Vol. Ren.)</td>
<td>30.85</td>
<td>12.62</td>
</tr>
</tbody>
</table>

Table 1: Lines of code needed to instrument the three selected proxy apps to produce a PNG file with a Pseudocolor rendering of a scalar variable.

Table 2: Simulation burden from visualization for large runs using 4096 cores on 256 nodes of LLNL’s Cab machine using various renderers. Times are listed in average seconds per cycle for both the visualization and simulation. All images were generated at a resolution of $1024^2$. 
Figure 2: Images produced using Strawman from the three integrations. From left to right, a volume rendered image from CloverLeaf3D, a ray traced image of Kripke, and rasterized image (OSMesa) from LULESH.

6. ACKNOWLEDGMENTS
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. LLNL-CONF-676409. This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) program and the U.S. Department of Energy, National Nuclear Security Agency. Advanced Simulation & Computing (ASC) program. Hank Childs is grateful for support from the DOE Early Career Award, Contract No. DEFG02-13ER26150, Program Manager Lucy Nowell.

7. REFERENCES
[22] B. Lorendeau, Y. Fournier, and A. Ribes. In-situ visualization in fluid mechanics using catalyst: A case study for code samuraine. In Large-Scale Data Analysis


