The SENSEI Generic In Situ Interface

Utkarsh Ayachit[1], Brad Whitlock[3], Matthew Wolf[2], Burlen Loring[4], Berk Geveci[1], David Lonie[1], and E. Wes Bethel[4],
[1]Kitware Inc., USA
[2]Oak Ridge National Laboratory, USA
[3]Intelligent Light, USA
[4]Lawrence Berkeley National Laboratory, USA

November, 2016

Acknowledgment

This work was supported by the Director, Office of Science, Office of Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231, through the grant "Scalable Analysis Methods and *In Situ* Infrastructure for Extreme Scale Knowledge Discovery," program manager Dr. Lucy Nowell.

Legal Disclaimer

The SENSEI Generic In Situ Interface

Utkarsh Ayachit*, Brad Whitlock†, Matthew Wolf‡, Burlen Loring§, Berk Geveci*, David Lonie*, E. Wes Bethel§¶
*Kitware, Inc.
†Intelligent Light
‡Oak Ridge National Laboratory
§Lawrence Berkeley National Laboratory
¶Corresponding author: ewbethel@lbl.gov

***Abstract*—The SENSEI generic *in situ* interface is an API that promotes code portability and reusability. From the simulation view, a developer can instrument their code with the SENSEI API and then make make use of any number of *in situ* infrastructures. From the method view, a developer can write an *in situ* method using the SENSEI API, then expect it to run in any number of *in situ* infrastructures, or be invoked directly from a simulation code, with little or no modification. This paper presents the design principles underlying the SENSEI generic interface, along with some simplified coding examples.**

***Index Terms*—Application programming interfaces, Computer aided analysis, High performance computing, Scientific computing**

I. Introduction

For current extreme-scale computation environments, users already frequently have to make choices on how they export data from their simulation codes because of limitations of I/O bandwidth. This is generally done with simple sampling in time – rather than exporting data every 200 steps as they might prefer to do, they would export at every 1,000 or 2,000 steps so that the ratio of compute to I/O time stays at a reasonable level. The introduction of *in situ* processing frameworks as a way of reducing the quantity of data being output allows the user to have more control over the temporal resolution of the output data, for example by enabling automated feature detection so that only the key features of interest are output frequently, with the whole data volume only being output at large time intervals.

However, it is difficult enough as a data analyst to determine how to create new, scientifically relevant feature detection and/or feature-aware compression routines. Adding to this complexity when working *in situ*, though, is that each of the runtimes and hardware choices seem to have a different set of expectations and requirements, potentially causing one to write and rewrite the same algorithm with many different interfaces and choices. Addressing this problem, the multiplication of implementations of the same algorithm, is the primary goal for SENSEI.

A colloquial summary of the SENSEI infrastructure could be *write once, use everywhere*. It brings together three different *in situ* frameworks to explore the space of commonality between them and to regularize their interfaces in support of an ecosystem of end users. This introduces a number of fundamental goals for the project. First, if a simulation is instrumented with SENSEI, it should be able to *use any of the different runtimes seamlessly*. Second, if an analysis routine works with SENSEI, it should be portable, in the specific sense that it should be a straight forward process to *move that piece of analytics* to a different scientific simulation that uses SENSEI. The porting concerns should be at the level of data management (specifying the change in names of variable arrays), instead of wholesale rewriting of code.

ISAV16; Salt Lake City, Utah, USA; November 2016

An additional goal for SENSEI is *to simplify the creation of in situ* analysis for simulation scientists, data analysts, and visualization experts. This is relevant to the portability goal, but it is also worth pulling out separately. With multiple *in situ* frameworks, each with their own capabilities, advantages, and expected coding patterns, it is quite challenging for simulation scientists to instrument their code to each of the frameworks separately. The same can be true for analysis experts when deciding which framework to write their analysis routine under.

Our solution to achieve these goals is twofold. Firstly, we identify a data model as a shared intermediate form for all data that the infrastructure can process. Secondly, we define an API for instrumenting simulation and analysis codes to use within SENSEI. Instrumentation will typically involve providing implementations for the API which comprises the SENSEI interface (§II).

After describing these design aspects, we will discuss the implementation of the SENSEI model using the different infrastructures (§III). Following that, we will provide some performance discussion of different components (§IV), before concluding with an overview of related work and summary of future directions.

II. Interface Design

Data Model: A key part of the design of the common interface was a decision on a common data description model. Our choice was to extend a variant on the VTK data model. There were several reasons for this choice. The VTK data model is already widely used in applications like VisIt[1] and ParaView[2], which are important codes for the post-hoc development of the sorts of analysis and visualization that are required *in situ*. The VTK data model has native support for a plethora of common scientific data structures, including regular grids, curvilinear grids, unstructured grids, graphs, tables, and AMR. There is also already a dedicated community looking to carry forward VTK to exascale computing, so our efforts can cross-leverage those.

Despite its many strengths, there were some key additions we wanted for the SENSEI model. To minimize effort and memory overhead when mapping memory layouts for data arrays from applications to VTK, we extended the VTK data model to support arbitrary layouts for multicomponent arrays through a new API called *generic arrays*[3]. Through this work, this capability has been back-ported to the core VTK data model. VTK now natively supports the commonly encountered *structure-of-arrays* and *array-of-structures* layouts utilizing *zero-copy* memory techniques.

Interface: The SENSEI interface comprises of three components: *data adaptor* that helps map sim data to VTK data model, *analysis adaptor* that maps VTK data model for analysis methods, and *in situ bridge* that links together the data adaptor and the analysis adaptor, and provides the API that the simulation uses to trigger the *in situ* analysis.

The *data adaptor* defines an API to access the simulation data as VTK data objects. The analysis adaptor uses this API to access the data to pass to the analysis method. To instrument a simulation code for SENSEI, one has to provide a concrete implementation for this data adaptor API. The API treats connectivity and attribute array information separately, providing specific API calls for requesting each. This helps to avoid compute cycles needed to map the connectivity and/or data attributes to the VTK data model unless needed by active analysis methods. The main parts of the *sensei::DataAdaptor* API are as follows:

```
namespace sensei {
 class DataAdaptor : ... {
  /// provide the mesh. if structure_only is true,
  /// then only the container data object is
  /// returned without geometry or topology
  /// information.
  vtkDataObject* GetMesh(bool structure_only);
  /// add an attribute array to the mesh container,
  /// if not already added.
  bool AddArray(vtkDataObject* mesh,
       int association,
       const std::string& arrayname);
  /// enquire about available attribute arrays.
  unsigned int GetNumberOfArrays(int association);
  std::string GetArrayName(int association,
       unsigned int index);
  /// release data.
  void ReleaseData();
 }; }
```

The *analysis adaptor*'s role is to take the data adaptor and pass the data to the analysis method, doing any transformations as necessary. For a specific analysis method, the *analysis adaptor* is provided the data adaptor in its *Execute* method. Using the *sensei::DataAdaptor* API, the analysis adaptor can obtain the mesh (geometry, and connectivity) and attribute or field arrays necessary for the analysis method.

```
namespace sensei {
 class AnalysisAdaptor : ... {
  /// execute the analysis routine. this method is
  /// called to execute the analysis routine per
  /// simulation iteration.
  bool Execute(DataAdaptor* data);
 }; }
```

III. Coding and Usage Example

To better understand the steps involved in instrumenting a simulation and analysis code with SENSEI, we present coding examples showing how to use the SENSEI interface. We first present a view from the simulation code (§III-A), where we instrument that code to set up the "outbound" data bridge. The simulation code is the mini-application from Ayachit et al., 2016 [4], which is bulk-synchronous parallel computation of time-varying oscillators on a 3D structured mesh. At each timestep, this code invokes an *in situ* method to perform some computations. We present two different views of the *in situ* method. We present a view from the *in situ* method (§III-B), where we set up the "inbound" data bridge. This *in situ* method formulation can be thought of as the equivalent of a "subroutine" call since there is no *in situ* infrastructure in the picture. To round out the presentation, we discuss how the SENSEI interface would be used to connect to three specific *in situ* infrastructures (§III-C).

A. View from the Simulation Code

To instrument the miniapp, we started by defining the SENSEI bridge API. The bridge is custom to the code being instrumented. For the miniapp, the bridge uses four API calls to initialize and finalize once during the sim life cycle and pass data and request analysis per temporal iteration, as illustrated by the code below:

```
int main(...) {
  // ** initialize app, including domain decomposition
  // and global data structures. **
  bridge::initialize(...); //<- init SENSEI
  for (int timestep=first; timestep < last; ++timestep) {
    // ** advance simulation **
    bridge::set_data(...); //<- pass arrays
    bridge::analyze(...);  //<- request analysis
  }
  bridge::finalize(...);   //<- cleanup SENSEI
  // ** cleanup app **
}
```

Next, we implemented a custom *sensei::DataAdaptor* subclass called *oscillator::DataAdaptor*, that maps the data generated by the miniapp to a VTK data object. In our case, this is fairly straight forward since VTK already has a representation called *vtkImageData* that maps to a uniform rectilinear grid. We support arbitrary distribution across MPI ranks, we opted to build a composite dataset (*vtkMultiBlockDataSet*) comprising of block for each domain block. *oscillators::DataAdaptor* has custom API global extents as well as extents for a specific block along with data for the same which gets called in *bridge::initialize* and *bridge::set_data*.

B. View from the In Situ Method

When instrumenting for an *in situ* analysis method, one has to provide a *sensei::AnalysisAdaptor* subclass that simply implements the *Execute(sensei::DataAdaptor*)* method. In the simplest case, the analysis method is already based on the VTK data model. In which case, the AnalysisAdaptor subclass simply obtains the VTK data object using the *sensei::DataAdaptor* and does the necessary computation.

```
namespace sensei {
 bool Histogram::Execute(sensei::DataAdaptor* data) {
 ...
 // request light-weight mesh without connectivity info
 vtkDataObject* mesh = data->GetMesh(/*structure_only*/true);
 // request the array to histogram
 data->AddArray(mesh, this->Association, this->ArrayName);
 ...
 // * compute histogram using the array available on mesh
 // * cell or point data locally and then reduce local
 // * result across ranks using MPI Reduce.
 } }
```

If the analysis method assumes a different data model, for example, computes histogram for raw C/C++ arrays, then the *Histogram::Execute* method needs to obtain the raw array pointers from VTK data object and do any transformations needed.

The *sensei::AnalysisAdaptor*, using *sensei::DataAdaptor*, remains isolated from the data producer. Hence any simulation code, that is instrumented to provide a *sensei::DataAdaptor* implementation can easily invoke the analysis method.

C. Interfacing to Infrastructure: VisIt/Libsim, PV/Catalyst, ADIOS

The *sensei::AnalysisAdaptor* enables bringing in arbitrary analysis method within SENSEI. It also provides a convenient abstraction to interface with any other *in situ* infrastructure. Our implementation supports three of such infrastructures: VisIt/Libsim, ParaView/Catalyst and ADIOS. Libsim and Catalyst are both *in situ* visualization infrastructures while ADIOS is an I/O framework with *in situ* capabilities. Interfacing with the *in situ* visualization infrastructures entails providing access to the data made available via *sensei::DataAdaptor* to these for using analysis & visualization capabilities within these infrastructures. For ADIOS, the *AnalysisAdaptor* serializes the data provided by the *DataAdaptor* using ADIOS API. Additionally, we implemented a *analysis endpoint* which can read data using ADIOS API and provide a *DataAdaptor* interface. Now, any analysis method (including Libsim and Catalyst) can be added on to the analysis endpoint for posthoc or staged execution.

The *bridge* described in §III-A is responsible for setting up the *AnalysisAdaptor* for analysis to enable execution during a simulation. To make that easily configurable, we use a simple XML description:

```
<sensei>
 <analysis type="histogram"
  array="data" association="cell" bins="10" />
 <analysis type="catalyst" pipeline="slice" ... />
 <analysis type="libsim" plots="Pseudocolor" ... />
 <analysis type="adios"
  filename="oscillators.bp" method="FLEXPATH" />
</sensei>
```

In the following subsections, we discuss the details of the *AnalysisAdaptor* implementation for the three infrastructures.

1) VisIt/Libsim: SENSEI was used to build an analysis adaptor based on Libsim, which enables SENSEI to perform data analysis and rendering using VisIt. In this case, the SENSEI autocorrelation mini-app was integrated with a Libsim adaptor that could slice the incoming VTK data and render images of the sliced data. SENSEI was integrated with Libsim by subclassing *sensei::AnalysisAdaptor* to create a new class called *sensei::libsim::AnalysisAdaptor*. The new class implements a Libsim simulation adaptor, which couples SENSEI to Libsim so that VTK datasets passed to the class by SENSEI can be passed to Libsim via the typical Libsim callback function mechanism. The class' *Execute()* method sets up visualizations and renders the plots to an image which is then saved to disk. The analysis adaptor calls Libsim to set up VisIt plots directly or indirectly using a pre-created VisIt session file saved from the VisIt GUI.

The bulk of *sensei::libsim::AnalysisAdaptor* is devoted to passing VTK data from *sensei::DataAdaptor* to Libsim so VisIt can operate on the data. Both VisIt and SENSEI use VTK as their base data model. However, since Libsim was designed to interface with simulations using their own array-based data structures, its interface requires meshes and fields be specified using arrays instead of complete VTK datasets. This means that SENSEI's VTK datasets must be picked apart so that their array data may be passed to Libsim. The arrays are used inside VisIt to reconstitute VTK datasets, which are in this case zero-copy (for the bulk data) facsimiles of the SENSEI VTK datasets.

2) ParaView/Catalyst: When using Catalyst for *in situ* analysis, one has initialize and use a singleton called *vtkCPProcessor*, that sets up the ParaView/Catalyst engine, and add visualization pipelines, using subclasses of *vtkCPPipeline*, to it. The simulation is expected to pass a VTK data object to the coprocessor on each iteration. The coprocessor then executes relevant visualizations pipelines. Incorporating this workflow within the *sensei::AnalysisAdaptor* subclass was quite straight forward. The *sensei::catalyst::AnalysisAdaptor* in its *Execute(...)* method first ensures that the vtkCPProcessor is properly initialized and then passes it a *vtkDataObject* obtained from the *sensei::DataAdaptor*. Since Catalyst has mechanisms for a pipeline to only ask for specific arrays for specific timesteps, the *sensei::catalyst::AnalysisAdaptor* can leverage those to only ask the *sensei::DataAdaptor* for the fields of interest for timesteps where the pipelines are to be executed. Since Catalyst pipelines are already designed to work with the VTK data model, no other data transformations are needed.

The *sensei::catalyst::AnalysisAdaptor* also exposes an API to add any visualization pipelines, including a custom C++ pipeline or a Python script generated using the ParaView desktop application. To generate a slice and save a screenshot, for example, we created a custom vtkCPPipeline subclass that takes the input mesh and slices it using ParaView's API to create filters and then render it in a view.

3) ADIOS: As an *in situ* framework, ADIOS has a somewhat different approach from LibSim and Catalyst. Part of the goal of the interface is to make it so that invoking in situ processing and disk I/O look the same to the application; the actual specification of what happens for the *in situ* processing happens outside of the ADIOS layer itself. The processing is carried out by a separate executable that could be launched on the same core, the same node, or indeed a different node in the same machine. This flexibility adds a little to the complexity of the batch launch script that a user must use, but it fits very well with the Adaptor+Bridge+Execute structure of the SENSEI interface.

Within the ADIOS implementation of SENSEI, the key component is managing the subset of the VTK data model that we're using in a self-consistent manner. ADIOS doesn't natively understand meshes; instead, it supports arbitrarily named variable arrays, and the attachment of attributes (metadata) so that the reader/consumer of the data can reconstruct your intent. The ADIOS adaptor uses the introspection capabilities of VTK to be able to walk the arrays and meshes that are part of the passed-in object and labels all of the array, scalar variables, and attributes using a fixed namespace mapping. For example, the number of points in mesh #2 of a multi-patch mesh representation in VTK would become "block2/numpoints" in the ADIOS description.

The Adaptor thus mainly does clerical tasks or marshalling data once SENSEI is invoked. The Execute function allows ADIOS to then transmit data using one of several transports that support *in situ*; our current work has been focused on the FlexPath transport [5]. Because of the adaptor + endpoint strategy, the ADIOS method can be used to transport data to a different staging area (on node or off), where LibSim or Catalyst can then be invoked as an *in situ* function within the *endpoint* executable as a composite workflow.

IV. Results and Discussion

A. Performance Analysis

A natural question to ask is "how much overhead does use of a unified *in situ* interface add compared to direct invocation?" The answer to this question is the subject of a recent study by Ayachit et al., 2016 [4]. The following paragraphs contain a summary of the results and findings that are most relevant to the main theme of this paper.

One set of tests involve looking at this very question by running a bulk-synchronous parallel mini-application at varying levels of concurrency (812-, 6496-, 45440-way parallel). In one configuration, the mini-application did direct invocation of an autocorrelation computation, and in another configuration, invoked that method via the SENSEI interface. That study measured runtime performance and memory footprint. The results there show no discernable difference between these two configurations. In this case, the data models of data producer and consumer were identical (3D, structured mesh), so the SENSEI data bridge was able to broker *zero-copy* memory access.

Ayachit et al., 2016 [4] also presents studies where the SENSEI interface is used with three scientific applications, one of which demonstrates *in situ* at greater than 1M-way concurrency. Two of the examples are able to achieve zero-copy memory access due to similar data models between the simulation and the *in situ* methods and infrastructure. One of the applications, an unstructured memory code, ends up using zero-copy for mesh data but requires a deep copy for mesh connectivity data. The performance analysis for that application does a coarse-grained measure of the cost of *in situ*, which includes the cost for *in situ* processing as well as the cost of the SENSEI data bridge. While it is clear that a deep copy involves greater cost than a zero-copy configuration, future work would entail finer-grained instrumentation to shed more light on these costs.

B. Portability

A more difficult thing to measure is "the cost of portability." One of the goals for the SENSEI project is to make it possible write an *in situ* method once, then have it run in any of several different *in situ* infrastructures with little, if any, modification. The SENSEI project is still at an early stage in this regard, though studies like Ayachit et al., 2016 [4] show that it is possible to achieve, at least initially within a constrained set of circumstances.

V. Related Work

The subject of *in situ* methods and infrastructure is a vibrant research and development area, owing to the widening gap between our ability to compute data and our ability to save it to persistent storage. This topic was the subject of a 2016 EuroVis State-of-the-Art report [6], which presents an in-depth survey of previous work in this space, along with an assessment of current research challenges. One of those research challenges, namely the interface between simulation codes and *in situ* methods and infrastructure is the subject of this paper.

In the area of *in situ* interfaces, there are two broad approaches that one might consider. One approach is an explicit instrumentation, where the simulation code must be modified with API calls that connect the simulation to the *in situ* method or infrastructure. This is the approach is the one we are taking with this work, and is also the approach used by VisIt/Libsim [1], ParaView/Catalyst [7], as well as other projects like Damaris/Viz [8], *Free*processing [9] and Strawman [10]. Damaris/Viz's API has sharing semantics for arrays to be safely used by both the simulation and the *in situ* application. There, allocation is done through Damaris/Viz and works most efficiently when double-buffering is used when updating the simulation's data structures during time-stepping. Strawman supports Cartesian, rectilinear and unstructured grids and uses Conduit's [11] data model. It supports zero-copy arrays but requires a matching array layout.

The SENSEI interface model provides a data bridge, which serves to rectify differences between the data models of the data producer (e.g., the simulation) and consumer (e.g., the *in situ* method or infrastructure). Zero-copy operation is supported when the data models of each are closely aligned, with shallow or deep copies required depending upon the degree to which they are not aligned. In an application case study bridging between an unstructured mesh code and ParaView/Catalyst, use of the SENSEI interface resulted in zero-copy for mesh data but a deep copy of mesh connectivity [4].

Whereas many of these require that the simulation scientist specify the *in situ* processing choices at compile time, another approach is to utilize a more generic I/O interface that could support I/O or *in situ*, which allows the user to make the choice at run time. This is the approach taken by ADIOS [12], [5] and GLEAN [13]. This can be used to support relatively complex *in situ* deployments that can adapt their pipeline on the fly based on resource issues [14]. In this use case, the SENSEI interface has proven useful as the basis for having methods work together, or in connecting multiple *in situ* infrastructures in a chaining configuration [4].

VI. Conclusion and Future Work

The SENSEI generic *in situ* interface promotes code portability and reuse in *in situ* processing environments. It accomplishes "write once, use everywhere" through the use of a generic data model and user-definable mappings to/from that generic data model, which can result in highly efficient, zero-copy data access in many common use cases. The interface simplifies simulation codes' use of several different *in situ* infrastructures by encapsulating the complexity of using those infrastructures directly. From the simulation view, the result is a simplified, portable, flexible, and low-cost entry to use of *in situ* methods and infrastructure. This portability across different infrastructures offers some unique benefits: one such benefit we observed when doing scalability studies in Ayachit et al., 2016 [4] was that, from the simulation view, it was possible to switch from a strictly *in situ* approach to an in transit approach with no modification to the simulation code: we were able to swap out one *in situ* infrastructure for another with no modifications to the simulation code.

When developers of *in situ* methods use a generic interface that can be run in many *in situ* infrastructures, the result is an increased likelihood of collaboration and sharing of methods. Contrast this result with *in situ* approaches of the past, where a method developed for one code would require modification, potentially extensive, for use with a different code. Increased sharing and collaboration is one of the essential ingredients for fostering a community of developers and users within the *in situ* space.

There remains a great deal of work to be done in the future. One area that is highlighted by our performance discussion is the nature of costs associated with mapping to/from the generic data model and that used by simulation codes and *in situ* methods and infrastructures. Specifically, when data models are similar across all, then a zero-copy realization will result in the best performance as no data is moved. When data models diverge, then the cost will increase as a function of the degree to which data needs to be reorganized. We have not attemped to measure or quantify those costs, though we know those costs do exist.

The source code for the SENSEI generic interface will be released soon under an open source license, and will be accessible from the SENSEI project website [15]. We plan to also release code associated with the performance studies of Ayachit et al., 2016 [4], which includes codes for *in situ* methods (histogram, autocorrelation), as well as the code associated with creating the connections to the Libsim, Catalyst, and ADIOS *in situ* infrastructures.

Acknowledgment

This work was supported by the Director, Office of Science, Office of Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231, through the grant "Scalable Analysis Methods and *In Situ* Infrastructure for Extreme Scale Knowledge Discovery," program manager Dr. Lucy Nowell. This research used resources of the Argonne Leadership Computing Facility (ALCF), the Oak Ridge Leadership Computing Facility (OLCF), and the National Energy Research Scientific Computing Center (NERSC).

References

[1] "VisIt," June 2015, http://visit.llnl.gov, last accessed August, 2016. [Online]. Available: http://visit.llnl.gov

[2] Utkarsh Ayachit, *The ParaView Guide: A parallel visualization application*. Kitware, Inc., 2015.

[3] David Lonie, "vtkArrayDispatch and Related Tools," http://www.vtk.org/doc/nightly/html/VTK-7-1-ArrayDispatch.html, last accessed August, 2016. [Online]. Available: http://www.vtk.org/doc/nightly/html/VTK-7-1-ArrayDispatch.html

[4] U. Ayachit, A. Bauer, E. P. N. Duque, G. Eisenhauer, N. Ferrier, J. Gu, K. Jansen, B. Loring, Z. Lukić, S. Menon, D. Morozov, P. O'Leary, M. Rasquin, C. P. Stone, V. Vishwanath, G. H. Weber, B. Whitlock, M. Wolf, K. J. Wu, and E. W. Bethel, "Performance Analysis, Design Considerations, and Applications of Extreme-scale *In Situ* Infrastructures," in *Proceedings of SC16*, Salt Lake City, UT, USA, Nov. 2016, *To appear*.

[5] J. Dayal, D. Bratcher, G. Eisenhauer, K. Schwan, M. Wolf, X. Zhang, H. Abbasi, S. Klasky, and N. Podhorszki, "Flexpath: Type-based publish/subscribe system for large-scale science analytics," in *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*. IEEE, 2014, pp. 246–255.

[6] A. C. Bauer, H. Abbasi, J. Ahrens, H. Childs, B. Geveci, S. Klasky, K. Moreland, P. O'Leary, V. Vishwanath, B. Whitlock, and E. W. Bethel, "*In Situ* Methods, Infrastructures, and Applications on High Performance Computing Platforms, a State-of-the-art (STAR) Report," *Computer Graphics Forum, Proceedings of Eurovis 2016*, vol. 35, no. 3, Jun. 2016, lBNL-1005709.

[7] A. C. Bauer, B. Geveci, and W. Schroeder, *The ParaView Catalyst User's Guide v2.0*. Kitware, Inc., 2015.

[8] M. Dorier, R. Sisneros, T. Peterka, G. Antoniu, and D. Semeraro, "Damaris/viz: a nonintrusive, adaptable and user-friendly in situ visualization framework," in *Proceedings of the IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV '13)*, Oct. 2013, pp. 67–75.

[9] T. Fogal, F. Proch, A. Schiewe, O. Hasemann, A. Kempf, and J. Krüger, "Freeprocessing: Transparent in situ visualization via data interception," in *Eurographics Symposium on Parallel Graphics and Visualization: EG PGV:[proceedings] sponsored by Eurographics Association in co-operation with ACM SIGGRAPH. Eurographics Symposium on Parallel Graphics and Visualization*, vol. 2014. NIH Public Access, 2014, p. 49.

[10] M. Larsen, E. Brugger, H. Childs, J. Eliot, K. Griffin, and C. Harrison, "Strawman: A batch in situ visualization and analysis infrastructure for multi-physics simulation codes," in *Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, ser. ISAV2015. New York, NY, USA: ACM, 2015, pp. 30–35. [Online]. Available: http://doi.acm.org/10.1145/2828612.2828625

[11] "Conduit website," http://software.llnl.gov/conduit/, last accessed April, 2016. [Online]. Available: http://software.llnl.gov/conduit/

[12] Q. Liu, J. Logan, Y. Tian, H. Abbasi, N. Podhorszki, J. Y. Choi, S. Klasky, R. Tchoua, J. Lofstead, R. Oldfield *et al.*, "Hello adios: the challenges and lessons of developing leadership class i/o frameworks," *Concurrency and Computation: Practice and Experience*, vol. 26, no. 7, pp. 1453–1473, 2014.

[13] V. Vishwanath, M. Hereld, V. Morozov, and M. E. Papka, "Topology-aware data movement and staging for I/O acceleration on Blue Gene/P supercomputing systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: ACM, 2011, pp. 19:1–19:11. [Online]. Available: http://doi.acm.org/10.1145/2063384.2063409

[14] J. Dayal, J. Lofstead, G. Eisenhauer, K. Schwan, M. Wolf, H. Abbasi, and S. Klasky, "Soda: Science-driven orchestration of data analytics," in *e-Science (e-Science), 2015 IEEE 11th International Conference on*. IEEE, 2015, pp. 475–484.

[15] "SENSEI Project Website," http://www.sensei-insitu.org, last accessed August 2016.