Title

Python-based in situ analysis and visualization

Authors

Loring, Burlen
Myers, Andrew
Camp, David
et al.

Publication Date

2018-11-11

DOI

10.1145/3281464.3281465

Peer reviewed

Python-based *In Situ* Analysis and Visualization

Burlen Loring Andrew Myers David Camp E. Wes Bethel
Lawrence Berkeley National Laboratory
Berkeley, CA
{BLoring,ATMyers,DCamp,EWBethel}@lbl.gov

ABSTRACT

This work focuses on enabling the use of Python-based methods for the purpose of performing *in situ* analysis and visualization. This approach facilitates access to and use of a rapidly growing collection of Python-based, third-party libraries for analysis and visualization, as well as lowering the barrier to entry for user-written Python analysis codes. Beginning with a simulation code that is instrumented to use the SENSEI *in situ* interface, we present how to couple it with a Python-based data consumer, which may be run *in situ*, and in parallel at the same concurrency as the simulation. We present two examples that demonstrate the new capability. One is an analysis of the reaction rate in a proxy simulation of a chemical reaction on a 2D substrate, while the other is a coupling of an AMR simulation to Yt, a parallel visualization and analysis library written in Python. In the examples, both the simulation and Python *in situ* method run in parallel on a large-scale HPC platform.

CCS CONCEPTS

• **Software and its engineering** → **Massively parallel systems**; • **Theory of computation** → *Parallel computing models*; • **Computing methodologies** → *Massively parallel algorithms*; *Massively parallel and high-performance simulations*;

KEYWORDS

Python, *in situ* analysis, *in situ* visualization

ACM Reference Format:
Burlen Loring Andrew Myers David Camp E. Wes Bethel. 2018. Python-based *In Situ* Analysis and Visualization. In *ISAV: In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization (ISAV '18), November 12, 2018, Dallas, TX, USA.* ACM, New York, NY, USA, Article 4, 6 pages. https://doi.org/10.1145/3281464.3281465

1 INTRODUCTION

There is an increasing interest in being able to leverage a large and growing body of useful Python-based analysis and visualization tools, and to take advantage of these capabilities in an *in situ* context. While there is some limited progress in this area, our approach is infrastructure for a Python-based analysis endpoint in the SENSEI *in situ* infrastructure. The idea is that a simulation code is instrumented once with the SENSEI interface, and can then at runtime switch between any number of different *in situ* "backends", and that the work we present here offers the possibility of new *in situ* backends that are based on Python.

The contribution of this work is the design of a new Python-based *in situ* back end for the SENSEI infrastructure, along with two examples that show this methodology in use on a large-scale HPC platform. These examples show diversity in terms of simulation type, data model being generated, and the use of a combination of user-provided Python analysis code as well as two different popular third-party Python libraries for analysis and visualization. In addition, we present results of a preliminary effort that aims to shed light on a performance analysis of this new Python-based *in situ* approach.

2 PREVIOUS WORK

Virtually all major *in situ* infrastructures, e.g., ADIOS [8], ParaView Catalyst [3, 15], VisIt Libsim [5, 17], SENSEI [2] through Libsim and Catalyst, and Ascent [6], provide some form of support for Python use by employing one of two approaches. In one, a framework exposes capabilities through a set of wrapped objects or functions that can be used in Python programs. Frameworks that embed an interpreter can then use Python code to configure *in situ* analysis and processing. In the other, the framework provides a channel to inject user supplied Python code directly into an analysis pipeline written in C++. Here, user supplied Python codes have direct access to simulation data and can apply the full gamut of Python analysis capabilities to do the heavy lifting in the analysis.

Both Libsim and Catalyst provide Python wrapped proxy objects for control and configuration, but these do not have access directly to data. In both, the control and configuration Python scripts can be automatically generated by recording a series of actions in the GUI. Libsim's Python features have been successfully used to control and configure large scale *in situ* analysis [12]. With its server side embedded interpreter and Python programmable filter, Catalyst has the capability to run user provided Python code that is written to map to VTK's pipeline abstraction so as to have direct access to the simulation data [11]. Ascent [6] is a many-core capable lightweight *in situ* visualization and analysis infrastructure being developed in conjunction with VTK-m [10]. In addition to providing access to VTK-m's many core accelerated algorithms, Ascent embeds a Python interpreter and can run user provided code that has direct access to the simulation's data.

SENSEI [2] is a lightweight, generic *in situ* interface providing normalized access to a diverse set of analysis back ends via a simple API and rich data model, and that has been shown to scale to over 1M cores [1]. With no coding changes, a simulation instrumented with the SENSEI interface has runtime-switchable access to multiple *in situ* backends, such as Libsim, Catalyst, ADIOS, VTK-m, and a host of others. The work in this paper details the design of a new

ISAV '18, November 12, 2018, Dallas, TX, USA

ACM ISBN 978-1-4503-6579-6/18/11…$15.00
https://doi.org/10.1145/3281464.3281465

Python based analysis backend for SENSEI that gives user-provided Python code direct access to simulation data for *in situ* analysis.

3 DESIGN AND IMPLEMENTATION

After a review of the SENSEI interface design (§3.1), we present the design pattern (§3.2) for our Python *in situ* infrastructure, which is part of the SENSEI infrastructure. This design pattern accommodates embedding the Python interpreter (§3.2.1), accommodating user-supplied Python code (§3.2.2), provides for runtime initialization of variables in the Python code (§3.2.3), accessing simulation data from within the Python code (§3.2.4), parallel execution of Python code (§3.2.5), and error handling (§3.2.6).

```
namespace sensei {
class PythonAnalysis : public AnalysisAdaptor
{
public:
  void SetScriptFile(const std::string &file);
  void SetInitializeSource(const std::string &code);

  int Initialize();

  int Execute(sensei::DataAdaptor *da) override;
  int Finalize() override;
};
}
```

Listing 1: The `PythonAnalysis` class.

3.1 The SENSEI *In Situ* Interface

SENSEI generically presents *in situ* methods to simulations through its `AnalysisAdaptor` API. Analysis codes then access simulation data structures through SENSEI's `DataAdaptor` API. A "bridge" code integrates SENSEI into the simulation, manages the adaptors and periodically invokes the analysis. Via an XML configuration file, SENSEI enables the simulation to select between different analysis back ends at run time.

SENSEI's data model is based on the VTK data model. The VTK data model supports a wide variety of mesh based data ranging from unstructured finite element meshes to AMR hierarchies and includes support for non-geometric data, such as tables, graphs, and array collections. Critically, the VTK data model also supports zero copy data transfer including zero copy mapping to and from structure-of-arrays (SOA) and array-of-structures (AOS) layouts, which enables simulations using one layout to work efficiently with an analysis requiring the other.

3.2 Core Design Pattern

For our Python based *in situ* solution, called the `PythonAanalysis`, we are primarily interested in SENSEI's `AnalysisAdaptor` API, which consists of two virtual methods, `Execute` and `Finalize`, for which we must provide implementations. Integration into SENSEI's `ConfigurableAnalysis`, which selects the analysis back end at runtime via a user provided XML configuration file, requires implementation of a non-virtual `Initialize` method and a snippet of code to parse user provided XML tags from which run time configuration is received. The class for these is shown in Listing 1.

3.2.1 Embedding Python. The `PythonAnalysis` embeds a Python interpreter, which is used to run user provided analysis code. We forward calls made to SENSEI's C++ `AnalysisAdaptor` API to user provided functions written in Python. This strategy allows us to manage the compile and run time complexity associated with the embedded interpreter in a way that is invisible to the simulation. Our solution can be used from C, C++, and Fortran simulations without those codes having any knowledge of Python internals.

3.2.2 User supplied analysis code. The `PythonAnalysis` forwards calls from SENSEI's C++ `AnalysisAdaptor` API to three user provided Python functions: `Initialize`, `Execute`, and `Finalize`. The three functions are contained in a script file that is passed as an argument to the `PythonAnalysis` class. During initialization, the `PythonAnalysis` class reads the user Python script file on rank 0, and then broadcasts that script to all other ranks. No specific action is required on the part of the user Python code for this to happen. The function signatures are shown in Listing 2.

```
def Initialize():
  # your initialization code here
  return

def Execute(dataAdaptor):
  # your in situ analysis code here
  return

def Finalize():
  # your tear down code here
  return
```

Listing 2: Python user interface, were user supplied Python code.

3.2.3 Runtime configuration. We have exposed the `PythonAnalysis` through SENSEI's `ConfigurableAnalysis`. The `ConfigurableAnalysis` allows users to select one of the analysis back ends at run time via an XML configuration file. Example XML is shown in listing 3.

```
<sensei>
  <analysis type="python" script_file="area_above.py"
    enabled="1">
    <initialize_source>
threshold=1.
mesh='mesh'
array='data'
cen=1
    </initialize_source>
  </analysis>
</sensei>
```

Listing 3: `ConfigurableAnalaysis` XML for the `PythonAnalysis`.

In order to avoid hard-coding runtime parameters that might be vastly different between different simulation runs, we provide a second channel called *initialization source* to inject and execute source code into the interpreter. This channel is only used for initialization, and as a result, is only run once at start up. This channel can be used to set global variables that control execution of the user defined analysis script. Once the initialize source has been run, the user provided `Initialize` function is invoked.

3.2.4 Accessing data during analysis. During a simulation run, the simulation periodically invokes the analysis back end passing a data adaptor instance, which enables the analysis to access the data it needs from the simulation. In our case, when the C++ implementation's `Execute` override is called, it creates a SWIG wrapped instance of the data adaptor passed to it, builds an argument list containing the wrapped adaptor instance, and invokes the user supplied Python `Execute` function. The Python analysis code uses the wrapped data adaptor to query metadata and then selectively access data objects containing the desired set of arrays. The data adaptor returns a VTK-wrapped `vtkDataObject` instance. The user Python code makes use of VTK's `numpy_support` module to access simulation data. This is shown in Listing 4.

3.2.5 Parallel analysis. In a parallel analysis, the user-provided Python code may need to use MPI for interprocess communication. SENSEI uses an isolated MPI communication space, which can be overridden by the simulation if desired. The communicator is accessible in the Python script via a global variable named `comm`.

Since many parallel simulations make use of ghost zones, the corresponding analysis methods will require access to them for their computations. SENSEI has adopted the ghost zone convention now used by both ParaView and VisIt[7, 16]. SENSEI's `DataAdaptor` provides methods for querying the presence of ghost zones and accessing mask arrays identifying them. Listing 4 shows an example of using the ghost zone mask array to do selective analysis computations.

3.2.6 Error handling. Error reporting and handling in Python is typically accomplished via exceptions. In our C++ code, after every invocation of user provided sources we check for a Python exception using the Python C-API. When one is detected we obtain the Python stack, construct an informative error report, and return an error code giving the simulation a chance to shutdown or disable the failing analysis.

```
import numpy as np, matplotlib.pyplot as plt
from vtk.util.numpy_support import *
from vtk import vtkDataObject, vtkCompositeDataSet

# default values of control parameters
threshold = 0.5
mesh = ''
array = ''
cen = vtkDataObject.POINT
out_file = 'area_above.png'
times = []
area_above = []

def pt_centered(c):
    return c == vtkDataObject.POINT

def Execute(adaptor):
    # get the mesh and arrays we need
    dobj = adaptor.GetMesh(mesh, False)
    adaptor.AddArray(dobj, mesh, cen, array)
    adaptor.AddGhostCellsArray(dobj, mesh)
    time = adaptor.GetDataTime()

    # compute area above over local blocks
    vol = 0.
    it = dobj.NewIterator()
    while not it.IsDoneWithTraversal():
        # get the local data block and its props
        blk = it.GetCurrentDataObject()

        # get the array container
        atts = blk.GetPointData() if pt_centered(cen) \
           else blk.GetCellData()

        # get the data and ghost arrays
        data = vtk_to_numpy(atts.GetArray(array))
        ghost = vtk_to_numpy(atts.GetArray('vtkGhostType'))

        # compute the area above
        ii = np.where((data > threshold) & (ghost == 0))
        vol += len(ii[0])*np.prod(blk.GetSpacing())

        it.GoToNextItem()

    # compute global area
    vol = comm.reduce(vol, root=0, op=MPI.SUM)

    # rank zero writes the result
    if comm.Get_rank() == 0:
        times.append(time)
        area_above.append(vol)

def Finalize():
    if comm.Get_rank() == 0:
        plt.plot(times, area_above, 'b-', linewidth=2)
        plt.xlabel('time')
        plt.ylabel('area')
        plt.title('area Above %0.2f'%(threshold))
        plt.savefig(out_file)
    return 0
```

Listing 4: Example of accessing simulation data and ghost zone data. Analysis that computes the area where a field exceeds a threshold.

4 RESULTS

We focus our evaluation of the new `PythonAnalysis` capability in the following ways. First, we show that the new method is usable by parallel simulations, and the examples in the following subsections show operation, *in situ* and in parallel, at up to 4096-way concurrency on two large HPC platforms, Cori and Edison, at NERSC, by two different codes and with two different Python-based *in situ* backends. Second, each of these case studies show use of both a custom user-analysis method written in Python as well as use of popular third-party Python libraries. Third, we present results of a preliminary effort that aims to shed light on a performance analysis of this new Python-based *in situ* approach. Finally, we show the ability to accommodate a diversity in data models, where one producer generates a uniform, structured mesh, while the other generates an Adaptive Mesh Refinement (AMR) mesh.

4.1 Case Study 1: Parallel Rendering with Yt

In this first case study, we use the `PythonAnalysis` adaptor to enable the *in situ* visualization of an AMR-based simulation using `yt` [14], a Python-based visualization and analysis toolkit for volumetric data that has been used in domains such as astrophysics, seismology, nuclear engineering, molecular dynamics, and oceanography. `Yt` uses Numpy arrays to represent the data on a given patch and heavily leverages Cython [4] for expensive tasks such as pixelization, box intersections, and volume rendering. To enable *in situ* visualization using SENSEI, we created a new code frontend for Yt that maps between the VTK-based data model returned by SENSEI's

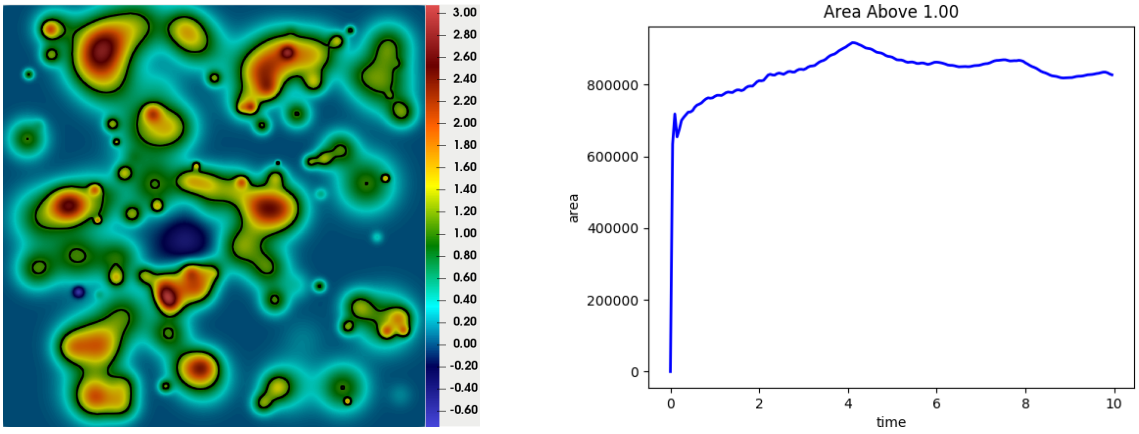

Figure 1: The reaction rate on a planar substrate as computed in the miniapp is shown here at simulation time 1, including an iso-line of 1 in black (left). At each time step the *in situ* analysis computes the area of the substrate where the reaction rate is greater or equal to 1. The area is accumulated and plotted at the end of the run (right).

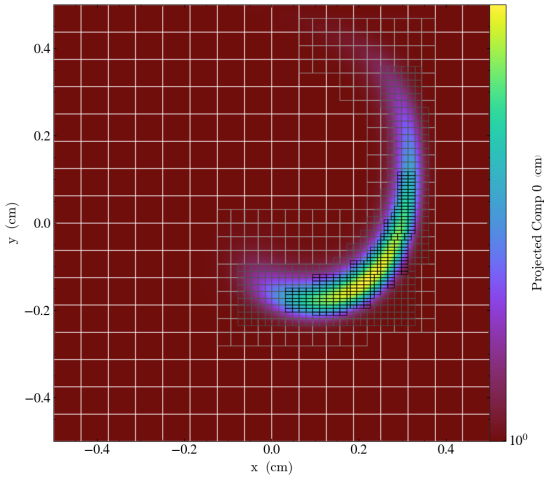

Figure 2: A slice through the simulation domain made with `Yt` showing the advection of scalar field through a vortex-like velocity field. The over-plotted boxes represent the AMR patches used for parallel domain decomposition.

data adaptor and the internal classes used by Yt to represent AMR hierarchies.

For most operations on patch-based AMR data, `Yt` uses demand-driven data access along with a grid-based approach to domain decomposition, in which each MPI process is assigned a set of AMR patches to work on. To enable *in situ* visualization with SENSEI, we simply instructed `Yt` to use the same communicator and box-process assignment mapping as the running simulation. This ensured that whenever `Yt`'s plotting routines tried to access the Numpy arrays corresponding to a given patch, that data would already exist in memory without the need for additional communication.

For the simulation code, we used AMReX, a software framework for parallel block-structured AMR calculations. We added SENSEI instrumentation to AMReX's AMR library, exposing information about the state variables and AMR Hierarchy. To test both the analysis and data adaptors, we ran the simulation for 236 time steps on 256 cores on Cori, using a base grid of 128^2 and 3 levels of refinement with a refinement ratio of 2. Each time step, we used Yt to generate a slice through the simulation *in situ* and saved the resulting image to disk. An example slice is shown in figure 2.

4.2 Case Study 2: Proxy Reaction Rate Analysis

In this case study, the simulation, or data source, is the oscillator miniapp that is part of the SENSEI software distribution, available at the SENSEI project website [13]. We configured this miniapp with 256 randomly positioned and initialized harmonic oscillators on a 16384^2 plane. This configuration serves as a proxy for a simulation of a chemical reaction on a 2D substrate where the output represents the reaction rate. Data generated by the miniapp at simulation time 1 is shown on the left of Figure 1 including an iso-line of 1. We ran this miniapp at four concurrency levels (512, 1024, 2048 and 4096 cores) on Edison for 100 timesteps, and invoked the *in situ* analysis method at each timestep. For each simulated time step we calculate the area of the domain where the reaction rate exceeds a given threshold, here set to 1, and accumulate the value over all timesteps. At the end of the run we use `matplotlib` to generate the x-y plot shown in the right of Figure 1, which shows the time-evolving area computation. The complete analysis code for this example is shown in Listing 4, and it uses the XML configuration shown in Listing 3.

4.3 Preliminary Performance Analysis

One well justified concern with using interpreted-language tools such as Python on HPC systems is the potentially high cost of startup, which is due to the need to load many different shared libraries and packages in an on-demand fashion. A common solution to this problem is to stage the needed files on a locally mounted filesystem, such as the approach described by MacLean et al., 2017 [9]. While solving this problem is well beyond the scope of this paper, we have collected some preliminary performance numbers for our second case study in which we vary the number of

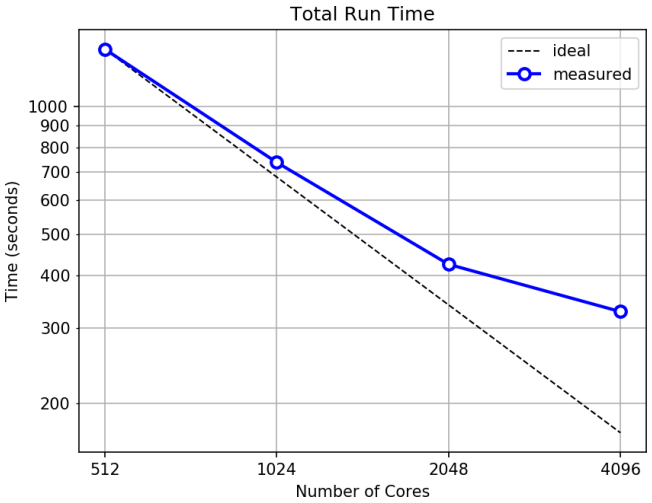

Figure 3: Total run time as a function of compute cores.

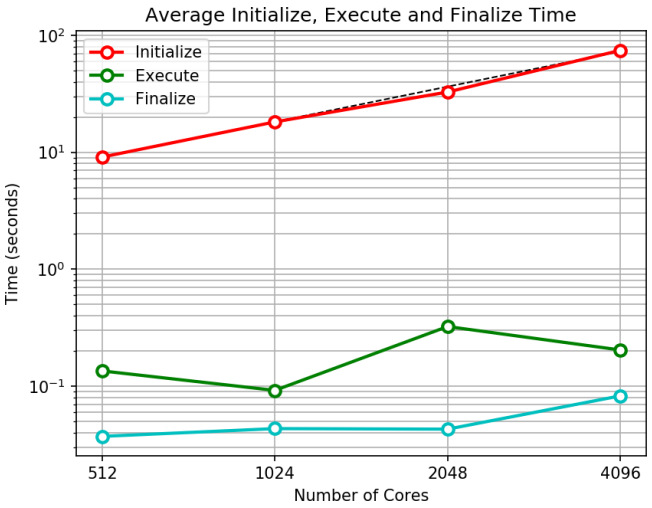

Figure 4: Average initialization time as a function of compute cores.

compute cores from 512 up to 4096 in 4 runs. In each run, in addition to the total run time, we measure the time spent in the Initialize, Execute, and Finalize methods. Measurements are made from the C++ layer in the `ConfigurableAnalysis` class so that all of the relevant overhead, such as Python interpreter initialization, script loading, and conversions of data structures from C++ to Python, is captured.

Figure 3 shows total run time as a function of compute cores (solid blue line). We observe that scaling is less than ideal as indicated by deviation from the perfect scaling line (dashed black line). Figure 4, which shows the average initialize (red line), execute (green line), and finalize (cyan line) time as a function of compute cores, sheds some light on the situation. With increasing compute cores the average finalize time is approximately flat, the average execute time increases slightly, however, the average initialization time increases linearly. Doubling the number of compute cores nearly perfectly doubles initialization times across the range of cores used here.

While this performance data is helpful in understanding the one-time and recurring runtime costs of a modest-sized case study, there are a number of interesting questions that remain. These include the relative performance difference of Python-based and compiled-language *in situ* methods in terms of both runtime and memory footprint. Such a study would be useful, and is beyond the scope of this paper.

5 CONCLUSION

Our work enables *in situ* use of a rapidly growing body of Python-based analysis and visualization methods. SENSEI's new `PythonAnalysis` can accommodate user-supplied Python code as well as use of third-party Python libraries for analysis and visualization. We have demonstrated this capability with two different example simulation codes and two different sets of Python-based analysis, running at modest concurrency on a modern HPC platform. This work serves as a starting point for others who desire to create, use, and deploy Python based *in situ* methods, and part of the open source SENSEI software distribution [13].

Future work will focus on studying and improving performance at scale, as well as development and deployment of new Python-based *in situ* methods for use by computational science projects.

ACKNOWLEDGEMENT

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. Laura Biven. This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

REFERENCES

[1] Utkarsh Ayachit, Andrew Bauer, Earl P. N. Duque, Greg Eisenhauer, Nicola Ferrier, Junmin Gu, Kenneth Jansen, Burlen Loring, Zarija Lukić, Suresh Menon, Dmitriy Morozov, Patrick O'Leary, Michel Rasquin, Christopher P. Stone, Venkat Vishwanath, Gunther H. Weber, Brad Whitlock, Matthew Wolf, K. John Wu, and E. Wes Bethel. 2016. Performance Analysis, Design Considerations, and Applications of Extreme-scale *In Situ* Infrastructures. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC16)*. Salt Lake City, UT, USA. https://doi.org/10.1109/SC.2016.78 LBNL-1007264.

[2] Utkarsh Ayachit, Brad Whitlock, Matthew Wolf, Burlen Loring, Berk Geveci, David Lonie, and E. Wes Bethel. 2016. The SENSEI Generic In Situ Interface. In *Proceedings of In Situ Infrastructures for Enabling Extreme-scale Analysis and Visualization (ISAV 2016)*. Salt Lake City, UT, USA. https://doi.org/10.1109/ISAV.2016.13 LBNL-1007263.

[3] Andrew C. Bauer, Berk Geveci, and Will Schroeder. 2015. *The ParaView Catalyst User's Guide v2.0.* Kitware, Inc.

[4] Stefan Behnel, Robert Bradshaw, Lisandro Dalcin, Mark Florisson, Vitja Makarov, and Dag Sverre Seljebotn. [n. d.]. Cython: C-extensions for Python. http://www.cython.org. Last accessed: Aug 2018.

[5] Hank Childs, Eric Brugger, Brad Whitlock, Jeremy Meredith, Sean Ahern, David Pugmire, Kathleen Biagas, Mark Miller, Gunther H. Weber, Hari Krishnan, Thomas Fogal, Allen Sanderson, Christoph Garth, E. Wes Bethel, David Camp, Oliver Rübel, Marc Durant, Jean Favre, and Paul Navratil. 2012. VisIt: An End-User Tool for Visualizing and Analyzing Very Large Data. In *High Performance Visualization—Enabling Extreme-Scale Scientific Insight*, E. Wes Bethel, Hank Childs, and Charles Hansen (Eds.). CRC Press/Francis–Taylor Group, Boca Raton, FL, USA, 357–372. http://www.crcpress.com/product/isbn/9781439875728 LBNL-6320E.

[6] Matthew Larsen, James Ahrens, Utkarsh Ayachit, Eric Brugger, Hank Childs, Berk Geveci, and Cyrus Harrison. 2017. The ALPINE In Situ Infrastructure: Ascending from the Ashes of Strawman. In *Proceedings of the In Situ Infrastructures on Enabling Extreme-Scale Analysis and Visualization (ISAV'17)*. ACM, New York, NY, USA, 42–46. https://doi.org/10.1145/3144769.3144778

[7] Dan Lipsa and Berk Geveci. 2015. Ghost and Blanking (Visibility) Changes. https://blog.kitware.com/ghost-and-blanking-visibility-changes/. Last accessed: Aug 2018.

[8] Qing Liu, Jeremy Logan, Yuan Tian, Hasan Abbasi, Norbert Podhorszki, Jong Youl Choi, Scott Klasky, Roselyne Tchoua, Jay Lofstead, Ron Oldfield, et al. 2014. Hello ADIOS: the challenges and lessons of developing leadership class I/O frameworks. *Concurrency and Computation: Practice and Experience* 26, 7 (2014), 1453–1473.

[9] Colin A. MacLean, HonWai Leong, and Jeremy Enos. 2017. Improving the Start-Up Time of Python Applications on Large Scale HPC Systems. In *Proceedings of the HPC Systems Professionals Workshop (HPCSYSPROS'17)*. ACM, New York, NY, USA, Article 2, 8 pages. https://doi.org/10.1145/3155105.3155107

[10] K. Moreland, C. Sewell, W. Usher, L. Lo, J. Meredith, D. Pugmire, J. Kress, H. Schroots, K. Ma, H. Childs, M. Larsen, C. Chen, R. Maynard, and B. Geveci. 2016. VTK-m: Accelerating the Visualization Toolkit for Massively Threaded Architectures. *IEEE Computer Graphics and Applications* 36, 3 (May 2016), 48–58. https://doi.org/10.1109/MCG.2016.48

[11] Cory Quammen. 2015. Scientific Data Analysis and Visualization with Python, VTK, and ParaView. In *Proceedings of the 14th Python in Science Conference (SCIPY 2015)*.

[12] Oliver Rübel, Burlen Loring, Jean-Luc Vay, David P. Grote, Remi Lehe, Stepan Bulanov, Henri Vincenti, and E. Wes Bethel. 2016. WarpIV: In Situ Visualization and Analysis of Ion Accelerator Simulations. *IEEE Computer Graphics and Applications* 36, 3 (May 2016), 22–35. https://doi.org/10.1109/MCG.2016.62 Selected for presentation in IEEE VIS session of selected Computer Graphica and Applications Papers. LBNL-1005718.

[13] SENSEI Project Team. 2018. SENSEI: Scalable in situ analysis and visualization, Project website. http://www.sensei-insitu.org. Last accessed August 2018.

[14] M. J. Turk, B. D. Smith, J. S. Oishi, S. Skory, S. W. Skillman, T. Abel, and M. L. Norman. 2011. yt: A Multi-code Analysis Toolkit for Astrophysical Simulation Data. *The Astrophysical Journal Supplement Series* 192, Article 9 (Jan. 2011), 9 pages. https://doi.org/10.1088/0067-0049/192/1/9 arXiv:astro-ph.IM/1011.3514

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

[16] Visit Users Website. 2012. Representing ghost data. http://www.visitusers.org/index.php?title=Representing_ghost_data. Last accessed: August 2018.

[17] Brad Whitlock, Jean M. Favre, and Jeremy S. Meredith. 2011. Parallel in Situ Coupling of Simulation with a Fully Featured Visualization System. In *Proceedings of the 11th Eurographics Conference on Parallel Graphics and Visualization (EGPGV '11)*. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 101–109. https://doi.org/10.2312/EGPGV/EGPGV11/101-109