The SENSEI Generic *In Situ* Interface: Tool and Processing Portability at Scale

E. Wes Bethel[1], Burlen Loring[1], Utkarsh Ayatchit[4], David Camp[1], Earl P. N. Duque[5], Nicola Ferrier[2], Joseph Insley[2], Junmin Gu[1], James Kress[3], Patrick O'Leary[4], David Pugmire[3], Silvio Rizzi[2], David Thompson[4], Gunther H. Weber[1], Brad Whitlock[5], Matthew Wolf[3], Kesheng Wu[1]

Abstract One key challenge when doing *in situ* processing is the investment required to add code to numerical simulations needed to take advantage of *in situ* processing. Such instrumentation code is often specialized, and tailored to a specific *in situ* method or infrastructure. Then, if a simulation wants to use other *in situ* tools, each of which has its own *bespoke* API [4], then the simulation code team will quickly become overwhelmed with having a different set of instrumentation APIs, one per *in situ* tool or method. In an ideal situation, such instrumentation need happen only once, and then the instrumentation API provides access to a large diversity of tools. In this way, a data producer's instrumentation need not be modified if the user desires to take advantage of a different set of *in situ* tools. The SENSEI generic *in situ* interface addresses this challenge, which means that SENSEI-instrumented codes enjoy the benefit of being able to use a diversity of tools at scale, tools that include Libsim, Catalyst, Ascent, as well as user-defined methods written in C++ or Python. SENSEI has been shown to scale to greater than 1M-way concurrency on HPC platforms, and provides support for a rich and diverse collection of common scientific data models. This chapter presents the key design challenges that enable tool and processing portability at scale, some performance analysis, and example science applications of the methods.

1 Introduction and Overview

A fact of life in *in situ* processing is the need to add instrumentation code to data producers, such as numerical simulations, in order to invoke tools for *in situ* processing. Over the years, tools like Libsim and Catalyst have evolved to include *in situ* APIs, but these are incompatible with one another. As a result, an application that wants to use both Libsim and Catalyst would need to have tool-specific instrumentation.

[1]Lawrence Berkeley National Laboratory · [2]Argonne National Laboratory · [3]Oak Ridge National Laboratory · [4]Kitware, Inc. · [5]Intelligent Light

This problem compounds as we consider the use of more and more tools, tools that include not only visualization, but also those for statistical analysis, machine learning, preparation of derived data products, among others.

The SENSEI generic *in situ* interface project has focused on solving this problem of *tool portability* in a direct way. Its key objective is to make it possible for a SENSEI-instrumented data producer, such as a numerical simulation, to make use of any of a number of different external tools and applications for *in situ* processing, and to do so without requiring any instrumentation code changes when going from one tool to another. This concept may be thought of as “tool portability”, or more colloquially as “write once, run everywhere”. A related concept, *proximity portability*, refers to the notion of being able to run either *in situ* on the same set of nodes, or *in transit* on different set of nodes, and also without any instrumentation changes.

In this chapter, we focus on the design and implementation issues for the purposes of achieving tool portability. One central idea is the design of the SENSEI *in situ* interface itself (Sect. 2), which includes a solution to a challenging data modeling problem. We present several examples that illustrate tool portability (Sect. 3), explore the costs of *in situ* processing using this generic interface at scale (Sect. 4), and illustrate its application to specific science applications (Sect. 5).

The SENSEI project website[1] provides direct access to the SENSEI interface source code, documentation, code examples, and other project-related information.

2 The SENSEI Generic *In Situ* Interface Design

Given the high level objective of tool portability — being able to have a SENSEI-instrumented code connect *in situ* with various tools like Libsim, Catalyst, or custom analysis codes — we identify three design considerations.

First, if a simulation is instrumented with SENSEI, it should be able to use any of the different runtimes transparently, without any coding changes needed on the simulation side to use a different tool. In other words, in an ideal world, once a simulation has been instrumented with SENSEI, then any effort needed to leverage any other *in situ* tool should occur outside the simulation instrumentation code.

Second, if an analysis routine works with SENSEI, it should be portable, in the specific sense that it should be a straightforward 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.

Third is the desire to simplify the creation of *in situ* methods and tools for simulation scientists, data analysts, and visualization experts. This concept is related to the tool portability objective, but it is also worth mentioning separately. Given that there exist multiple *in situ* frameworks, each with its own capabilities, advantages, and expected coding patterns, it is quite challenging for simulation scientists to

[1] `http://www.sensei-insitu.org/`

instrument their code to each of the frameworks separately. The same idea applies for *in situ* tool/method developers as they consider which of the *in situ* frameworks for implementing and deploying their method.

As presented in earlier work [2], the approach used to meet these design considerations focuses on two separate, but related, issues. First, we need to solve a data model problem so that producers and consumers are able to exchange data. Second, we need to define an API that is suitable for use in instrumenting both data producers and consumers in a way that is representative of common design patterns and use scenarios.

2.1 SENSEI 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 [5] and ParaView [16], 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 [12].

Despite its many strengths, there were some key additions we added 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* [6]. 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.

2.2 SENSEI Interface

The SENSEI interface is comprised of three components, which are shown in Fig. 1. The *data adaptor* performs a mapping from the simulation data model to the VTK data model. The *analysis adaptor* performs a mapping from the VTK data model to that used by the *in situ* analysis methods. The bridge links together the data adaptor and the analysis adaptor, and provides the API that a simulation uses to trigger the invocation of the *in situ* methods. In this design and implementation, the VTK data model is the bridge data model between producer and consumer.

The data adaptor defines an API to fetch the simulation data packaged 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

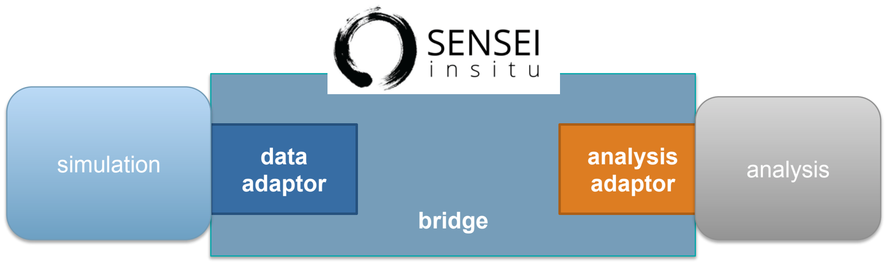


Fig. 1: The SENSEI bridge includes a `DataAdaptor`, seen by the simulation code or data producer, an `AnalysisAdaptor`, seen by the analysis code or data consumer, a bridge data model, and machinery to link the two adaptors.

implementation for this data adaptor API. The API treats connectivity and attribute array information separately, providing specific API calls for requesting each. As a result, we can avoid using compute cycles needed to map the connectivity and/or data attributes to the VTK data model unless actually needed by active analysis methods. The main parts of the originally released `sensei::DataAdaptor` API are shown in Listing 1. Subsequent releases of the `sensei::DataAdaptor` API have added methods: for exposing multiple named datasets; for fetching ghost zone and adaptive mesh refinement (AMR) covered cell masks information; and, for fetching light weight metadata useful for load balancing and planning data movement in *in transit* configurations[10].

```
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();
 }; }
```

Listing 1: SENSEI Data Adaptor API.

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. The main elements of the analysis adaptor API is shown in Listing 2.

```
namespace sensei {
class AnalysisAdaptor : ... {
   public:
   /// Execute the analysis routine.
   virtual int Execute(DataAdaptor* data) = 0;

   /// Finalize the analyis routine
   virtual int Finalize() = 0;
}; }
```

Listing 2: SENSEI Analysis Adaptor API.

2.3 Data Types Supported in the SENSEI Interface

Conceptually SENSEI expects a simulation to expose a number of "meshes" for *in situ* processing. Here a "mesh" can represent a spatially geometric data set. However, collections of non-spatially oriented data such as arrays, tables, and graphs are also supported. Irrespective of the type of data, each mesh that a simulation exposes represents a logical grouping of array based data partitioned and distributed for parallel execution. Meshes are therefor comprised of collections of distributed "blocks" such that each MPI rank has zero or more "blocks" of data. We liberally use the term "block" when referring to the subsets of a mesh which are distributed among a simulation's MPI ranks for parallel execution. The so called "blocks" can, but do not need to, have a Cartesian structure. For instance the subsets of points allocated to each MPI rank in particle in cell (PIC) simulation are referred to as blocks of data.

The SENSEI data model makes use of VTK to internally represent simulation data. VTK is widely used in the analysis of HPC simulation data already and supports a diverse array of dataset types ranging for finite element method (FEM) datasets to graphs. VTK supports zero copy transfer of array based data and is extensible at both compile and run time. The SENSEI APIs make use of the base class in the VTK data model, `vtkDataObject`, so that any VTK dataset type including user defined types may be passed through the API without modification. A high level depiction of some of the types of data supported in the SENSEI data model are enumerated in Table 1.

Type	**Description**
AMR	Adaptive Mesh Refinement (AMR) is a specialization of a multi-block dataset where blocks have differing resolutions. Different organization schemes exist such as block structured overlapping and oct-tree.
Multi-“block”	The general mesh type used in the SENSEI data model. All array based data passed through the simulation interface is multi-block. Blocks are used to partition subsets of the data to MPI ranks for parallel execution.
Uniform Cartesian	A block type where data exists on regular Cartesian mesh. Geometry is fully implicit.
Stretched Cartesian	A block type where data exists on a stretched Cartesian mesh. Geometry is defined by 3 coordinate axes.
Curvilinear	A block type with hexahedral elements in a regular ordering such that element indexing is logically Cartesian. Geometry is fully explicit.
Unstructured/FEM	A block type with collections of potentially mixed types of finite element method (FEM) cells with an arbitrary ordering. Geometry is full explicit.
PIC/Point cloud	A block type where data exists at points in space. Implemented as unstructured mesh.
Molecular	A block type specifically designed for molecular dynamics with representations of atoms and bonds between them.
Tabular	A block type where data is organized as a collection of rows and columns.
Graph	A block type where data is organized on nodes and edges without spatial information.
Array collection	A block type consisting of an arbitrary set of arrays without any spatial information.

Table 1: SENSEI supports a rich collection of common scientific data models, ranging from simple, like the uniform Cartesian mesh, to more complex, like AMR.

2.4 SENSEI Data Producer Coding 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. Here, we focus on the steps needed to instrument a data producer (e.g., a simulation) for use with SENSEI. We call this instrumentation the “bridge” code. The bridge code is not a part of SENSEI. Rather it is a concept that enables us to discuss simulation instrumentation in a general way.

The bridge code does three things: initializes SENSEI, including passing user provided XML that selects the data consumer; periodically invokes *in situ* processing through the SENSEI APIs as the simulation state evolves; and finalizes SENSEI. Listing 3 shows an example in C++. Conditionals protect each of the three bridge code blocks, as the code is only executed if and when the simulation determines it would like to do *in situ* processing.

```
int main(...) {
  // initialize the simulation
  ...

  // initialize SENSEI
  if (doInSitu) {
    ca = sensei::ConfigurableAnalysisAdaptor::New();
    ca->Initialize(userXMLFile);
  }

  // simulation main loop
  for (int timestep=first; timestep < last; ++timestep) {

    //  advance simulation
    ...

    if (doInSitu) {
      // create and initialize the data adaptor
      DataAdaptor *da = DataAdaptor::New();
      da->Initialize(...);
      // invoke in situ processing
      ca->Execute(da);
      // clean up
      da->Delete();
    }
  }

  // SENSEI shutdown and cleanup
  if (doInSitu) {
    ca->Finalize();
    ca->Delete();
  }

  // simulation specific cleanup
  ...
}
```

Listing 3: Data producer, data bridge setup and use.

2.5 SENSEI Data Consumer Coding Example

Next, in Listing 4 we present the view from an *in situ* method where we set up the analysis adaptor. This particular example is from the SENSEI `histogram` endpoint, which is also part of the SENSEI code distribution. When instrumenting for an *in situ* analysis method, one has to provide a `sensei::AnalysisAdaptor` subclass that implements the `Execute(sensei::DataAdaptor*)` method. In the simplest case, the analysis method's data model is based on the VTK data model, in which case the `AnalysisAdaptor` subclass obtains the VTK data object using the `sensei::DataAdaptor` and then does the necessary computation. This listing does not show computation of the histogram, only setting up the "inbound" bridge.

```
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.
 } }
```

Listing 4: *In Situ* data consumer, data bridge setup and use.

If the analysis method uses a different data model other than the VTK data model, then the `Histogram::Execute` method needs to obtain the raw array pointers from the VTK data object and then do any needed transformations.

3 SENSEI Tool Portability

One of the main strengths of the SENSEI design and implementation is the idea of *tool portability*. The design objective is to be able to instrument a data producer code once with SENSEI, and then use any number of different *in situ* or *in transit* methods without any coding changes. This section explores the different ways that SENSEI achieves tool portability by presenting examples of use with a diverse set of *in situ* tools, ranging from user-written Python, to *in situ* endpoints from both the SENSEI and Ascent projects.

3.1 Configurable Analysis Adaptor

Reviewing briefly, in SENSEI, there are data producers that are instrumented with SENSEI to invoke a `DataAdaptor` that is specific to the data producer code. There are also data consumers, or endpoints, which have associated `AnalysisAdaptors`. The purpose of the `DataAdaptor` and `AnalysisAdaptor` is to perform any necessary transformations between a native data model and the bridge data model.

In addition, SENSEI provides a *configurable analysis adaptor*, which uses an XML file to select and configure one or more back ends at run time. Run time selection of the back end via XML means one user can access Catalyst, Libsim, or a Python-based method with no changes to the code. In Fig. 2, we see the data producer, the AMReX code invokes an AMReX specific `DataAdaptor`. Its bridge code pushes the data through the configurable analysis adaptor to the back end that was selected at run time via the SENSEI configuration file.

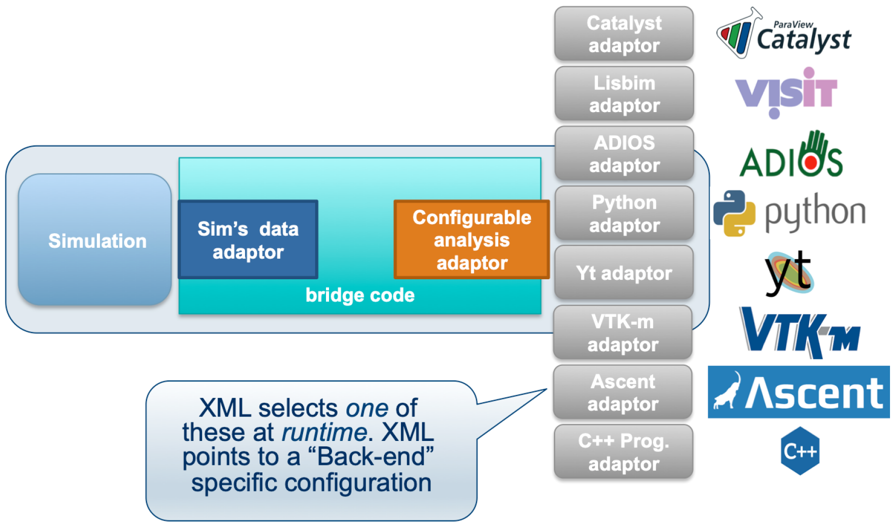
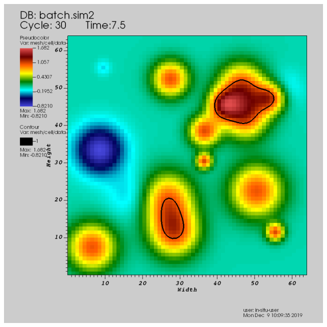


Fig. 2: Leveraging SENSEI's configurable analysis adaptor, a single data producer has access to any number of potential *in situ* or *in transit* methods. The runtime choice of which *in situ* or *in transit* method or endpoint, along with its associated parameters, is specified in a human-readable XML configuration file. Image courtesy B. Loring.

3.2 Connecting SENSEI to Libsim, Catalyst, Ascent, or ADIOS

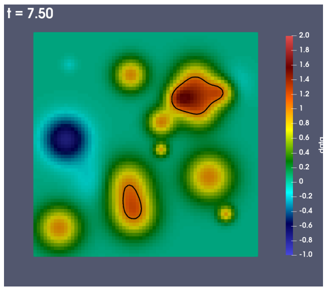
To illustrate tool portability, Fig. 3 presents an example of a single data producer, the `oscillators` miniapplication, which is part of the SENSEI software distribution, coupled to three different *in situ* endpoints that perform visualization. Figs. 3a, 3b, and 3c show sample images and the associated XML configuration file used to produce the image with Libsim, Catalyst, and Ascent, respectively. The runtime selection of endpoint is performed by SENSEI's configurable analysis adaptor. This example reinforces the idea of tool portability, one of SENSEI's design objectives, whereby no instrumentation code changes are needed on the data producer side when changing between *in situ* endpoints.

For *in transit* configurations, where data must be explicitly moved from producer ranks to consumer ranks, SENSEI can take advantage of several different potential



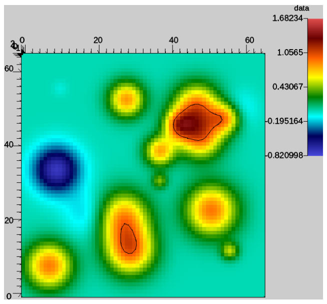
```
<sensei>
  <analysis type="libsim" mode="batch" frequency="1"
            session="configs/random_2d_64_libsim.session"
            image-filename="random_2d_64_libsim_%ts"
            image-width="800" image-height="800"
            image-format="png"
            options="-debug 0" enabled="1" />
</sensei>
```

(a) Sample image and XML that was used to configure the run with Libsim.



```
<sensei>
  <analysis type="catalyst" pipeline="pythonscript"
    filename="configs/random_2d_64_catalyst.py"
    enabled="1" />
</sensei>
```

(b) Sample image and XML that was used to configure the run with Catalyst.



```
<sensei>
  <analysis type="ascent"
    actions="configs/random_2d_64_ascent.json"
    enabled="1" />
</sensei>
```

(c) Sample image and XML that was used to configure the run with Ascent.

Fig. 3: Example of SENSEI's tool portability, where the `oscillators` miniapplication is used with three different *in situ* infrastructures: Libsim, Catalyst, and Ascent. In all cases, there are no coding changes needed to the `oscillators` miniapplication. Instead, the only difference is in the configuration file, which specifies the specific *in situ* method to be run and using what parameters. Note that these three backends all make use of their own separate configuration files as well. Images courtesy B. Loring.

```xml
<sensei>
  <transport type="adios1" filename="random_2d_64.bp"
    method="FLEXPATH" enabled="1" />
</sensei>
```

Listing 5: This XML configures runs to use ADIOS as a data transport. The filename in this example is used to establish a connection between sender and receiver. SENSEI has the ability to switch between several different potential data transport tools at runtime via XML-based configuration files.

data transport mechanisms. Continuing the example of connecting the `oscillators` miniapplication to one of several potential endpoints, Listing 5 shows the SENSEI configuration file that connects the data producer to the ADIOS data transport. That data transport could then be connected to any of the endpoints shown earlier in Fig. 3 to produce exactly the same visual results as when those endpoints are invoked *in situ*.

Using this idea to implement *in transit* processing, and building on the adaptor and endpoint strategy, one can construct *in situ* and *in transit* workflows by "daisy chaining" together transport layers and endpoints that perform specific types of processing. This has been demonstrated at scale on HPC systems with several different scientific simulations and different types of endpoints [1, 2, 10].

3.3 Coupling with User-written Python Tools

Due to the growing collection of Python-based tools and methods for diverse activities ranging from visualization to analysis and machine learning, the SENSEI project includes the ability to invoke Python-based methods for use *in situ*, including parallel Python-based methods and use of such methods at scale on HPC platforms. This section provides a high-level overview of how to invoke user-supplied Python code *in situ*, and in parallel, from a SENSEI-instrumented application, and is a consolidation of a more detailed discussion of design principles and implementation details discussed elsewhere [11].

Focusing on the user-supplied Python analysis code only, the basic idea is that the user-supplied Python code needs to contain three functions: `Initialize`, `Execute`, and `Finalize`. The `PythonAnalysis` forwards calls from SENSEI's C++ `AnalysisAdaptor` API to those three user-provided Python functions. Those three functions are contained in the user Python 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 6.

The runtime selection of the user-supplied Python method is accomplished via SENSEI's `ConfigurableAnalysis`. The `ConfigurableAnalysis` allows users

```python
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 6: The user-supplied Python file must contain three functions: `Initialize`, `Execute`, and `Finalize`, which are invoked by the SENSEI AnalysisAdaptor at runtime.

to select one of the analysis back ends at run time via an XML configuration file. Example XML is shown in Listing 7, which contains the name of the user-supplied Python script, along with some initialization values that are specific to the variables in that script.

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

Listing 7: The XML initilization file used by SENSEI's `ConfigurableAnalaysis` to invoke the user-supplied Python code, as well as to provide some initialization values specific to the user script. Having initialization values in the configuration file helps to avoid having hard-coded parameters inside the Python code itself.

The user-supplied XML file shown in Listing 7 includes Python code that can be thought of as initialization steps. The idea is that this *initialization source* is injected by SENSEI and executed as source code in 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 *initialization source* has been run, the user provided `Initialize` function, if present, is invoked.

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 the case of a user-supplied Python code running *in situ*, when the C++ implementation's `Execute` override is called (by the simulation), 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. A complete example is show in Listing 8.

The parallel user Python code may need to make use of MPI for tasks like 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 [9] and VisIt [17]. SENSEI's `DataAdaptor` provides methods for querying the presence of ghost zones and accessing mask arrays identifying them.

We leverage this capability to perform a conditional *in situ* computation of a time-varying data producer running in parallel on an HPC system. We configured the `oscillator` miniapplication 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 miniapplication at simulation time 1 is shown on the left of Fig. 4, including an isoline at 1.0. In the original study [11], we ran this miniapplication at four concurrency levels (512, 1024, 2048 and 4096 cores) for 100 timesteps and invoked the *in situ* method, in this case our custom Python code shown in Listing 8, at each timestep. Note that this code uses SENSEI's ghost zone mask array to support selective computations. In each invocation, we calculate the area of the domain where the reaction rate exceeds a given threshold, here set to 1.0, 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 Fig. 4, which shows the time-evolving area computation.

```python
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 8: A complete, working example of user-written Python code run *in situ* by SENSEI to produce the results shown in Fig. 4. This code uses SENSEI's ghost zone mask array to perform a conditional computation.

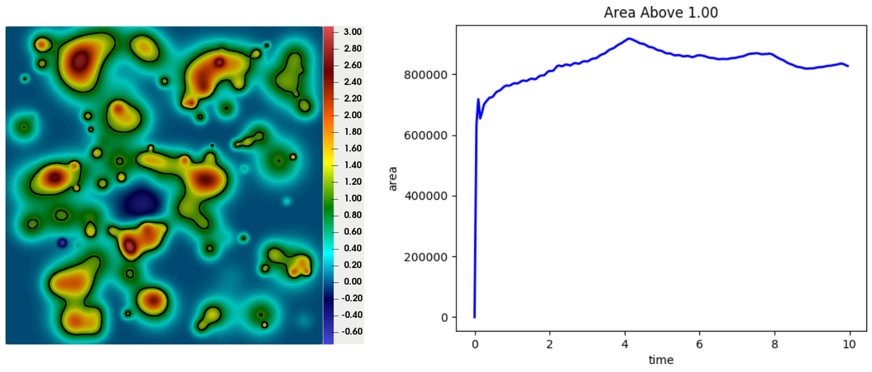


Fig. 4: The reaction rate on a planar substrate as computed in the miniapp is shown here at simulation time 1, including an iso-line of 1 in black (left). At each time step the *in situ* analysis (Listing 8) computes the area of the substrate where the reaction rate is greater or equal to 1. The area is accumulated and plotted at the end of the run (right). Images courtesy B. Loring, et al. [11]

3.4 *In Situ* analysis of AMR data

Adaptive mesh refinement (AMR) is a computational technique introduced by Berger and Colella [3]. It is used for solving systems of partial differential equations whereby computational resources are targeted to areas of the simulated domain where numerical errors are unacceptably large. Periodically during the simulation, estimates of the numerical error are obtained, cells with unacceptably large errors are flagged for refinement, and a new mesh discretization is generated where the flagged cells have increased spatial resolution. The increased spatial resolution reduces discretization errors. Cells with acceptable errors remain at their existing resolution. In overlapping block structured AMR, the domain is decomposed in to groups of blocks organized by refinement level. The blocks in higher levels fully overlap those in lower levels. Metadata accompanying the set of blocks is used to describe the hierarchical nesting of the blocks.

The SENSEI data and analysis adaptor API's and data model includes support for AMR-capable data producers. Among these capabilities are APIs for fetching mask arrays, which indicate where mesh cells are covered by more refined cells, and for fetching metadata, which describes the hierarchical structure of the AMR data [10]. These extensions enable seamless movement and processing of AMR data by the supported *in situ* and in transit methods and endpoints.

Fig. 5 shows two examples of *in situ* processed AMR data. In the left panel a pseudocolored image produced using SENSEI's Python back-end in conjunction with the Yt visualization module shows the hierarchical structure of an AMR advection miniapp that ships with the AMReX framework [11]. In the right panel, blue lines on an extracted isosurface show the refined nature of the mesh. In both cases the mask arrays obtained through SENSEI's data adaptor API are critical for correct

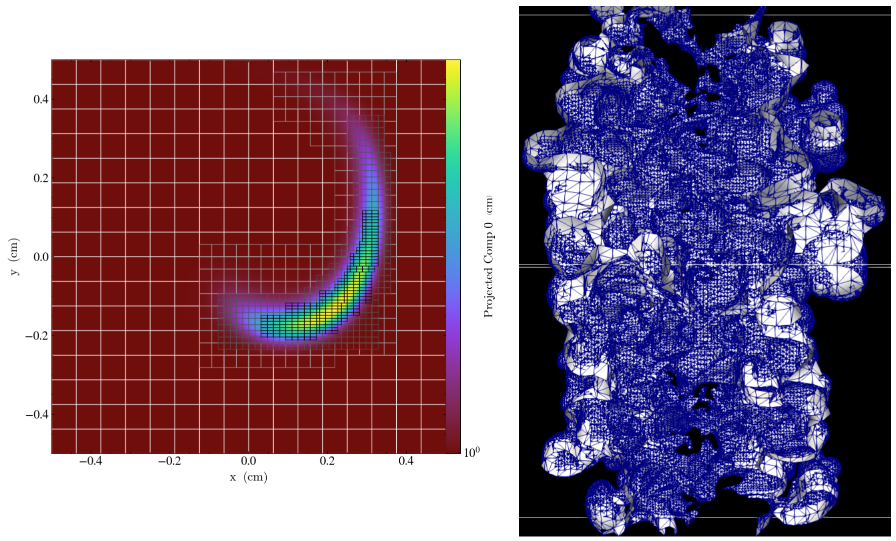
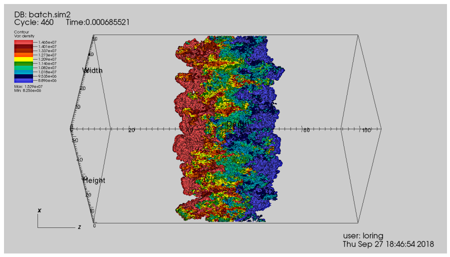


Fig. 5: Adaptive mesh refined (AMR) data has a hierarchical structure where mesh cells in regions of the simulation where numerical error is unacceptably large are refined, increasing the spatial resolution, to reduce the error. SENSEI's data model and adaptor API's are designed for use with AMR data producers. Left: In this *in situ* pseudocolored rendering, using SENSEI's Yt back-end, over set lines show the hierarchical mesh structure. Data produced by an AMReX advection mini-application. Right: In this image an isosurface extracted with SENSEI's Catalyst back-end is rendered. The blue lines show the refined mesh structure. Data produced with the AMReX IAMR compressible Navier-Stokes simulation. Images courtesy B. Loring.

processing of the refined data. Cells that are covered by refined cells in higher levels as indicated by the mask need to be discarded during *in situ* processing so that only the highest resolution data is used.

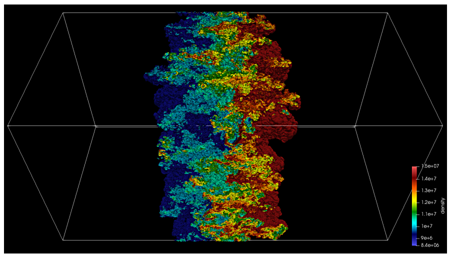
The SENSEI analysis adaptors for *in situ* and *in transit* methods and endpoints with AMR processing capabilities handle translation of AMR data from SENSEI's data model.As in the case of other SENSEI instrumented data producers, AMR simulations make use of run time provided XML to select one of the *in situ* or *in transit* data processing, analysis, or movement methods. No changes to the simulation are required to switch between the various *in situ* and *in transit* methods and endpoints.

Fig. 6 shows renderings of results produced by IAMR, a compressible Navier-Stokes solver implemented on top of AMReX. These IAMR runs were made on NERSC's Cray XC40 Cori with 2048 MPI ranks. The simulation was configured using a Rayleigh-Taylor instability initial condition and set to use 3 levels with a base mesh of 256^2 x 512 giving an effective resolution of 1024^2 x 2048. The top most panel shows 10 isosurfaces rendered with Libsim, the middle panel shows the same isosurfaces rendered with Catalyst, and the bottom most panel shows them again rendered with Ascent. The SENSEI XML used to switch between the different renderers is shown next to the corresponding image. In each case the XML file points to a library specific configuration file which contains the configuration information



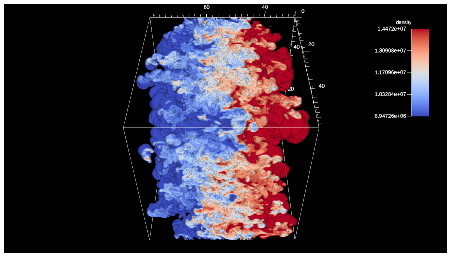
```
<sensei>
  <analysis type="libsim"
    session_file="rt_contour.session"
    enabled="1" />
</sensei>
```

(a) Rendering the IAMR Rayleigh-Talyor simulation with Libsim



```
<sensei>
  <analysis type="catalyst"
    script_file="rt_contour.py"
    enabled="1" />
</sensei>
```

(b) Rendering the IAMR Rayleigh-Talyor simulation with Catalyst

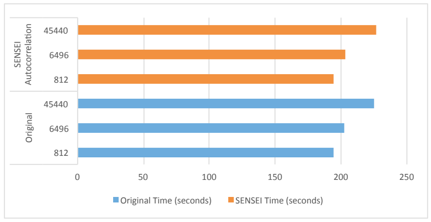


```
<sensei>
  <analysis type="ascent"
    actions="rt_contour.json"
    enabled="1" />
</sensei>
```

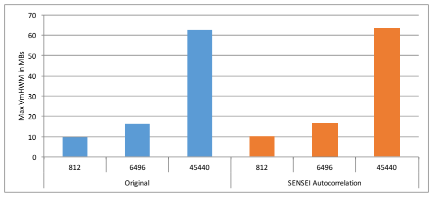
(c) Rendering the IAMR Rayleigh-Talyor simulation with Ascent

Fig. 6: This SENSEI data producer, the AMReX IAMR Rayleigh-Taylor simulation instrumented with the SENSEI interface, is shown here running in parallel on a large HPC platform with 3 levels of refinement. Three runs were made, the first using the SENSEI Analysis Endpoint that invokes a Libsim-based renderer (top), the second one that invokes a Catalyst-based renderer (middle), and the third using one that invokes an Ascent-based renderer (bottom). Switching between the different rendering back-ends required no changes to the simulation code. Instead, runtime provided XML, shown to the right of each figure, was used to effect the change. Images courtesy B. Loring.

in the rendering library's native format. These library specific configurations, in the case of Libsim and Catalyst, can be generated using the respective GUIs.



(a) Time to solution for the 1K, 6K and 45K configurations.



(b) Memory footprint for the 1K, 6K, and 45K configurations.

Fig. 7: Performance analysis to measure the potential impact to a code of invoking a method directly as a subroutine call compared to invoking the method through the SENSEI interface. In this particular test configuration, which entails using *in situ* processing of a structured mesh using an `autocorrelation` operation, there are no significant differences between the two when measuring and comparing time to solution (Fig. 7a) and memory footprint (Fig. 7b). Image courtesy Ayachit et al., 2016 [1].

4 SENSEI *In Situ* Performance Analysis at Scale

There are several potential different dimensions of performance analysis of *in situ* systems and methods. Of these, we focus on two specific questions in this section. The first is to examine the degree to which use of *in situ* methods impacts the performance of a simulation running at scale on an HPC system. The second is to compare the performance of a typical scientific visualization task when run in *post hoc* and *in situ* configurations so as to gain an understanding of potential gains that might result from using an *in situ* approach.

4.1 Performance Impact of *In Situ* Processing

One key question is "how much does the use of *in situ* methods impact simulation code" in terms of memory footprint and runtime. Ayachit, et al., 2016 [1] approach this problem by setting up a test matrix that consists of the several configurations aimed to revealing performance differences between invoking methods directly vs. invoking them using *in situ* infrastructure. The study involves the use of several different *in situ* analysis methods that have embarrassingly parallel scaling characteristics. The test battery includes runs at varying concurrencies (1K-, 6K-, and 45K-way parallel) on a large HPC platform, and include measurements of runtime and memory use.

The results of those tests, shown in Fig. 7, reveal there was no appreciable difference in either runtime or memory footprint when invoking the method directly as a subroutine compared to invoking the method through the SENSEI infrastructure. In other words, the use of *in situ* methods at scale had no appreciable difference when compared to the application just making a subroutine call. This result means the *in situ* interface did not “get in the way“ of the application, or otherwise incur any significant cost in terms of runtime or memory footprint.

Some of the characteristics of the SENSEI interface and the study that enabled this type of result are as follows. First, this particular study focused on processing of structured, 3D meshes. Such data types require relatively little metadata, and are amenable to *zero copy* sharing between the simulation and *in situ* method. The results would have likely been somewhat different with a different type of data model that carries a heavier cost in terms of metadata, such as AMR meshes. At the time of that study, the SENSEI interface provided support for *zero copy* data sharing between the simulation and *in situ* method for structure mesh data models. More recent work enables shallow- and zero-copy data sharing for other types of data whenever possible and feasible. Some of the science examples studied include unstructured meshes, where portions of the data required shallow- or deep-copies, including a large-scale CFD run at greater than 1M-way parallel, shown in Fig. 8.

This particular study from Ayachit, et al., 2016 [1] focuses on a relatively narrow set of configurations to explore performance impact at scale for *in situ* processing. Other studies have examined different dimensions of performance in *in transit* processing. For example, Morozov and Lukić, 2016 [13] compare time to solution *in situ* and *in transit* work processes. Kress, et al., 2019 [8] examine cost estimation for *in transit* configurations of some common visualization rendering scenarios. This particular area, performance analysis of *in situ* and *in transit* processing, is a vibrant area of research with a long history in distributed computing.

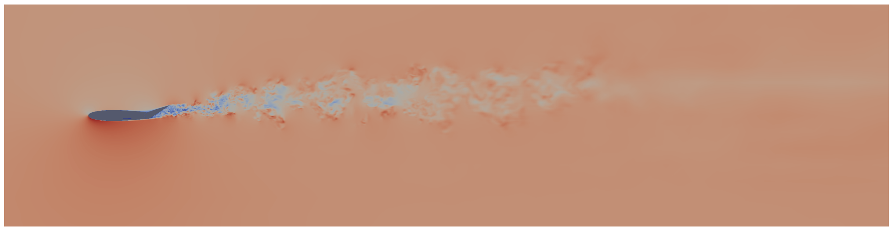


Fig. 8: From a 1M-way parallel PHASTA run on `mira` at Argonne National Laboratory, this image is a zoomed-in view of a 2D slice from 3D volume. This numerical simulation models the flow over a vertical tail-rudder assembly for a geometry that exactly matches the configuration of an ongoing wind tunnel experiment. Image courtesy K. Jensen, A. Bauer, A. Ayachit, and P. O’Leary [1].

4.2 Cost Savings of *In Situ* over *Post Hoc*

Another dimension of performance analysis for *in situ* processing is to examine time-to-solution for a workflow consisting of a simulation producing numerical results that are then processed either *in situ* or in a *post hoc* configuration. Ayachit, et al., 2016 [1] include such a study, which shows significant differences in time-to-solution for *in situ* and *post hoc* configurations. In their study, they run a data producer over 100 timesteps at a range of concurrencies: 1K-, 6K-, and 45K-way parallel. The computation results are then analyzed using a set of simple, embarrassingly parallel methods (histogram, point-wise autocorrelation) and visualized with ParaView/Catalyst.

In the *post hoc* configuration, data from each time step is written to disk using a file-per-processor configuration. Then, these datafiles are read back in from disk and processed by the analysis and visualization methods. The *post hoc* configuration uses $1/10$ the number of nodes used to produce the data, since this ratio is typical of many *post hoc* workflows where there are typically far fewer ranks used for analysis than were used to compute the data.

In the *in situ* configuration, the analysis and visualization are performed at the same concurrency as the simulation. This configuration is also representative of many common *in situ* configurations: using a differing number of ranks for producer and consumer would likely entail data movement or redistribution, which is essentially an *in transit* use scenario.

The findings from this study, presented in detail in Ayachit, et al., 2016 [1], show that the expensive cost of disk I/O, both for writing and reading, is a significant impediment. It is no surprise that avoiding disk I/O altogether results in a huge performance gain. In addition, being able to run the analysis and visualization at the same scale as the simulation results in significantly lower runtimes for that part of the processing. The disparity between *in situ* and *post hoc* configurations grows with scale: as concurrency increases, the difference in performance between these two modalities also increases. This study makes a strong case for performance gains that result when using *in situ* processing.

5 SENSEI *In Situ* Applications to Science Problems

The next two sections present results where the SENSEI interface facilities *in situ* processing for specific science applications. The first, in §5.1, shows how *in situ* processing helps to validate mesh configurations for the Nek5000 code. The second, in §5.2, shows use of *in situ* processing to produce data extracts for post-hoc analysis, where the data extract is significantly smaller in size than the underlying computational mesh.

5.1 In Situ Mesh Validation in Combustion Simulations

Nek5000 is a massively-parallel high-order spectral element computational fluid dynamics (CFD) solver written in Fortran and used for more than 30 years [14]. Domain scientists use it for large-scale simulations of fluid flow, thermal convection, combustion, magnetohydrodynamics, and electromagnetics. During a typical simulation, spectral (mesh) elements may deform, sometimes causing the Jacobian of some of the elements to become zero or negative. In those cases the simulation can not proceed.

Users of Nek5000 frequently encounter the problem of identifying the exact location of problematic mesh elements within the whole mesh. Employing the standard post-hoc approach to address this problem is both time consuming and requires vast storage space. To alleviate this problem, Shudler et al. demonstrated Nek5000 instrumented with SENSEI as a tool to enable users to find problematic regions of the simulation mesh *in situ* [15].

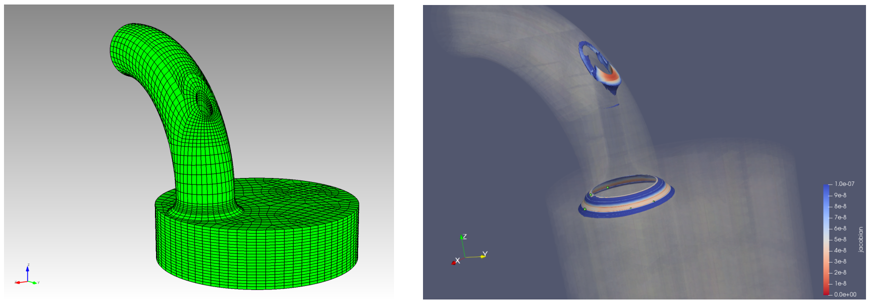


Fig. 9: (a) Simulation mesh. Image courtesy Saumil Patel, Argonne National Laboratory; (b) In situ visualization highlighting very small and vanishing Jacobians. Image courtesy Sergei Shudler, Argonne National Laboratory [15].

The left side of Fig. 9 shows the input mesh for a Nek5000 simulation of an experimental combustion engine. The hexahedral elements of the mesh may be valid (i.e., non-vanishing Jacobians) at the start of a production run, but may become highly skewed or distorted (vanishing Jacobians) during the mesh motion. The goal of this *in situ* workflow with SENSEI is to determine when and where vanishing Jacobians, if any, occur. The desired outcome is to provide a capability to users where they can validate the mesh before running any production-level simulations at scale. The right side of Fig. 9 shows an in situ rendering with SENSEI and Catalyst where the problematic areas of the mesh are highlighted in red (low values of the Jacobian). *In Situ* visualization of vanishing Jacobians allows users to locate the problematic regions of the mesh quicker, providing a list of the problematic elements that can be fixed in preparation for large-scale production runs.

5.2 *In Situ* Processing and Analysis in Wind Energy Applications

One area of study in wind energy applications is the numerical modeling of wind turbines as individuals and as groups in wind farms. As individuals, some studies focus on stresses and other factors of the wind turbine elements, including the blades, the nacelle, and the tower. As groups, in farms, some studies seek to analyze the interplay of turbulence and its impact when the downstream wake from one wind turbine intersects other wind turbines.

A recent study from Kirby, et al., 2018 [7] explores the use of *in situ* methods in a numerical modeling and analysis workflow. Here, the W^2A^2KE3D code is used to model the NREL WindPACT-1.5MW wind turbine (as well as other models, which are presented in cited work). Fig. 10 demonstrates *in situ* extracts visualized in VisIt showing the wake evolution through seven cut-planes across the wake as a function of rotor diameter at fixed locations downstream of the wind turbine. Here, the *in situ* operation is to extract and produce 7 cut-planes at 400×400 resolution, each containing five flow variables and three coordinates. Additionally, a center plane of resolution 2000×400 with the same variables is extracted and is shown in the top of Fig. 11. An isocontour of velocity magnitude is shown the bottom of Fig. 11, which reveals the complex tip vortex phenomenon evolving downstream of the wind turbine.

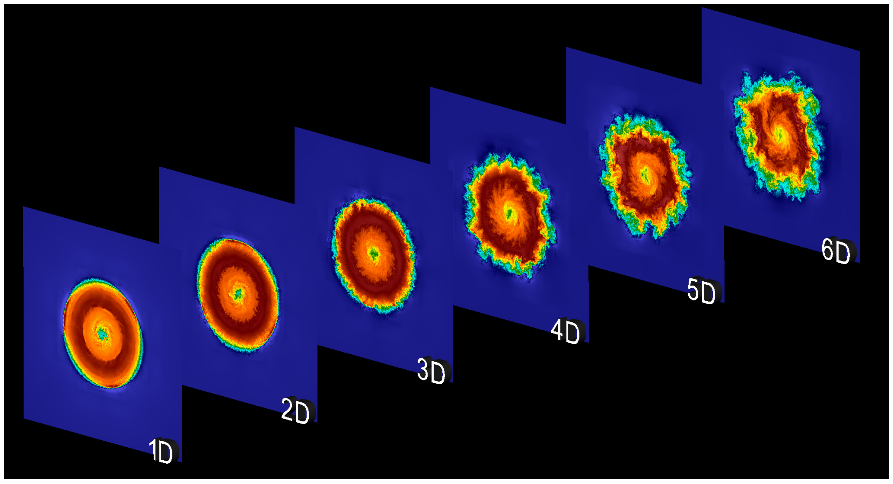


Fig. 10: Downstream cross flow cut-planes at various positions of the NREL WindPACT-1.5MW wind turbine. Image courtesy Kirby et al., 2018 [7].

For this study, one of the benefits reported by the authors is a significant cost savings. For their particular workflow, which entails performing analysis of wake turbulence, they realize a data reduction factor of approximately 246.2 that results

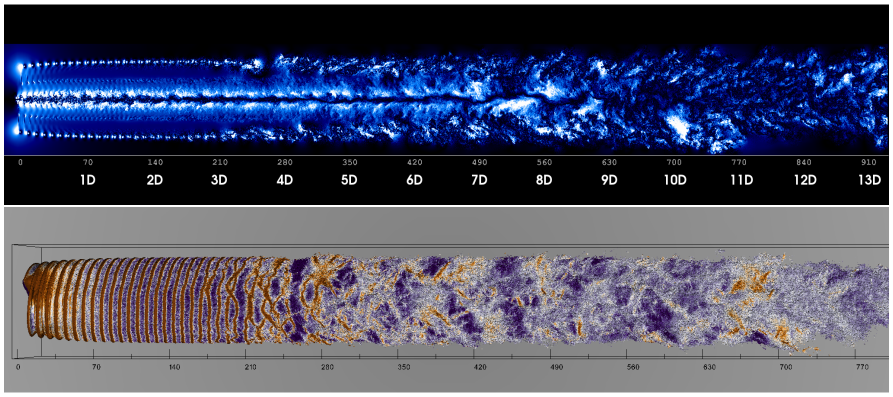


Fig. 11: (Top) Instantaneous flow visualization of the the NREL WindPACT-1.5MW wind turbine. The absolute tangential ow velocity is visualized demonstrating the wake propagation downstream annotated by rotor diameter lengths. (Bottom) An isocontour of the velocity magnitude demonstrating the vortex structure evolution of the NREL WindPACT-1.5WM wind turbine. The data extracts were created *in situ* using Libsim, and were rendered in a post-processing step using VisIt. Image courtesy Kirby et al., 2018 [7].

when using *in situ* methods to produce data extracts at each timestep as opposed to doing I/O at full spatiotemporal resolution.

For problems such wind turbine and wind farm modeling, the authors suggest that *in situ* workflows "provide opportunities for increased productivity as the data extraction and visualization is co-produced with simulation results at run-time." The productivity increase results from a decrease in data sizes, and provides the ability to do analysis at high spatiotemporal resolution with only a fraction of the I/O and computational cost required by a *post hoc* workflow.

6 Conclusion

The SENSEI generic *in situ* interface has made it possible for parallel simulation code to be instrumented once but then take advantage of any number of different potential *in situ* or *in transit* methods or tools for analysis and visualization. The primary design feature that makes this code coupling possible is the adaptor model, whereby implementation-specific code that, say, maps from native to bridge data models, provides a degree of portability between data producers and data consumers. The examples in this chapter show SENSEI being used with a diverse set of visualization and analysis endpoints, including Libsim, Catalyst, Ascent, and user-written Python code. This latter capability, being able to invoke user-written Python code both *in* *situ* and in parallel, opens the doors to a vast world of tools for analysis, computer vision, and machine learning that exist in the Python world.

This system has been studied extensively at scale, including at over 1M-way concurrency. The findings show that when there is a high degree of alignment between data models, as would be the case with something like a 3D structure mesh, that the *in situ* operations are highly efficient due to SENSEI's ability to use *zero copy* wherever possible. SENSEI's rich support for a wide diversity of different scientific data models means that it is readily usable by nearly all scientific simulation codes in existence today.

References

1. Ayachit, U., Bauer, A., Duque, E.P.N., Eisenhauer, G., Ferrier, N., Gu, J., Jansen, K.E., Loring, B., Lukić, Z., Menon, S., Morozov, D., O'Leary, P., Ranjan, R., Rasquin, M., Stone, C.P., Vishwanath, V., Weber, G.H., Whitlock, B., Wolf, M., Wu, K.J., Bethel, E.W.: Performance analysis, design considerations, and applications of extreme-scale in situ infrastructures. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '16, pp. 79:1–79:12. IEEE Press, Piscataway, NJ, USA (2016). URL `http://dl.acm.org/citation.cfm?id=3014904.3015010`
2. Ayachit, U., Whitlock, B., Wolf, M., Loring, B., Geveci, B., Lonie, D., Bethel, E.W.: The SENSEI Generic in Situ Interface. In: Proceedings of the 2nd Workshop on In Situ Infrastructures for Enabling Extreme-scale Analysis and Visualization, ISAV '16, pp. 40–44. IEEE Press, Piscataway, NJ, USA (2016). DOI 10.1109/ISAV.2016.13. URL `https://doi.org/10.1109/ISAV.2016.13`
3. Berger, M., Colella, P.: Local adaptive mesh refinement for shock hydrodynamics. Journal of Computational Physics **82**(1), 64 – 84 (1989). DOI https://doi.org/10.1016/0021-9991(89)90035-1. URL `http://www.sciencedirect.com/science/article/pii/0021999189900351`
4. Childs, H., Ahern, S., Ahrens, J., Bauer, A.C., Bennett, J., Bethel, E.W., Bremer, P.T., Brugger, E., Cottam, J., Dorier, M., Dutta, S., Favre, J., Fogal, T., Frey, S., Garth, C., Geveci, B., Godoy, W.F., Hansen, C.D., Harrison, C., Hentschel, B., Insley, J., Johnson, C., Klasky, S., Knoll, A., Kress, J., Larsen, M., Lofstead, J., Ma, K.L., Malakar, P., Meredith, J., Moreland, K., Navratil, P., O'Leary, P., Parashar, M., Pascucci, V., Patchett, J., Peterka, T., Petruzza, S., Podhorszki, N., Pugmire, D., Rasquin, M., Rizzi, S., Rogers, D.H., Sane, S., Sauer, F., Sisneros, R., Shen, H.W., Usher, W., Vickery, R., Vishwanath, V., Wald, I., Wang, R., Weberr, G.H., Whitlock, B., Wolf, M., Yu, H., Ziegler, S.B.: A Terminology for In Situ Visualization and Analysis Systems. International Journal of High Performance Computing Applications **0**(0) (2020). DOI 10.1177/1094342020935991. URL `https://doi.org/10.1177/1094342020935991`
5. Childs, H., Brugger, E., Whitlock, B., Meredith, J., Ahern, S., Pugmire, D., Biagas, K., Miller, M., Weber, G.H., Krishnan, H., Fogal, T., Sanderson, A., Garth, C., Bethel, E.W., Camp, D., Rübel, O., Durant, M., Favre, J., Navratil, P.: VisIt: An End-User Tool for Visualizing and Analyzing Very Large Data. In: E.W. Bethel, H. Childs, C. Hansen (eds.) High Performance Visualization—Enabling Extreme-Scale Scientific Insight, Chapman & Hall, CRC Computational Science, pp. 357–372. CRC Press/Francis–Taylor Group, Boca Raton, FL, USA (2012). URL `http://www.crcpress.com/product/isbn/9781439875728`. LBNL-6320E
6. David Lonie: vtkArrayDispatch and Related Tools. URL `http://www.vtk.org/doc/nightly/html/VTK-7-1-ArrayDispatch.html`. `http://www.vtk.org/doc/nightly/html/VTK-7-1-ArrayDispatch.html`, last accessed August, 2016
7. Kirby, A.C., Yang, Z., Mavriplis, D.J., Duque, E.P., Whitlock, B.J.: Visualization and data analytics challenges of large-scale high-fidelity numerical simulations of wind energy applica-

tions. In: 2018 AIAA Aerospace Sciences Meeting (2018). DOI 10.2514/6.2018-1171. URL `https://arc.aiaa.org/doi/abs/10.2514/6.2018-1171`

8. Kress, J., Larsen, M., Choi, J., Kim, M., Wolf, M., Podhorszki, N., Klasky, S., Childs, H., Pugmire, D.: Comparing the efficiency of in situ visualization paradigms at scale. In: International Conference on High Performance Computing, pp. 99–117. Springer (2019)
9. Lipsa, D., Geveci, B.: Ghost and blanking (visibility) changes. `https://blog.kitware.com/ghost-and-blanking-visibility-changes/` (2015). Last accessed: Aug 2018
10. Loring, B., Gu, J., Ferrier, N., Rizzi, S., Shudler, S., Kress, J., Logan, J., Wolf, M., Bethel, E.W.: Improving performance of m-to-n processing and data redistribution in in transit analysis and visualization. In: EuroGraphics Symposium on Parallel Graphics and Visualization (EGPGV). Norrköping, Sweden (2020)
11. Loring, B., Myers, A., Camp, D., Bethel, E.: Python-based in situ analysis and visualization. In: Proceedings of the Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization - ISAV '18. ACM Press (2018). DOI 10.1145/3281464.3281465
12. Moreland, K., Sewell, C., Usher, W., Lo, L., Meredith, J., Pugmire, D., Kress, J., Schroots, H., Ma, K.L., Childs, H., Larsen, M., Chen, C.M., Maynard, R., Geveci, B.: VTK-m: Accelerating the Visualization Toolkit for Massively Threaded Architectures. IEEE Computer Graphics and Applications (CG&A) **36**(3), 48–58 (2016)
13. Morozov, D., Lukić, Z.: Master of puppets: Cooperative multitasking for in situ processing. In: Proceedings of High-Performance Parallel and Distributed Computing (HPDC) (2016)
14. Offermans, N., Marin, O., Schanen, M., Gong, J., Fischer, P., Schlatter, P., Obabko, A., Peplinski, A., Hutchinson, M., Merzari, E.: On the Strong Scaling of the Spectral Element Solver Nek5000 on Petascale Systems. In: Proceedings of the Exascale Applications and Software Conference 2016, EASC '16, pp. 5:1–5:10. ACM, New York, NY, USA (2016). DOI 10.1145/2938615.2938617
15. Shudler, S., Ferrier, N., Insley, J., Papka, M.E., Patel, S., Rizzi, S.: Fast Mesh Validation in Combustion Simulations through In-Situ Visualization. In: H. Childs, S. Frey (eds.) Eurographics Symposium on Parallel Graphics and Visualization. The Eurographics Association (2019). DOI 10.2312/pgv.20191105
16. Utkarsh Ayachit: The ParaView Guide: A parallel visualization application. Kitware, Inc. (2015)
17. Whitlock, B.: Representing ghost data. `http://www.visitusers.org/index.php?title=Representing_ghost_data` (2012). Last accessed: June 2020