SENSEI Documentation

Release 4.0.0-54-g4c28c54

Lawrence Berkeley Lab, Oak Ridge Lab, Argonne Lab, Intelligent

Dec 13, 2022

Contents

Fig. 1: SENSEI gives simulations access to a wide array of scalable data analytics and visualization solutions through a single API.

CHAPTER 1

Introduction

Write once run anywhere. SENSEI seamlessly & efficiently enables in situ data processing with a diverse set of languages, tools & libraries through a simple API, data model, and run-time configuration mechanism.

A SENSEI instrumented simulation can switch between different analysis back-ends such as ADIOS, Libsim, Ascent, Catalyst etc, at run time without modifying code. This flexibility provides options for users, who may have varying needs and preferences about which tools to use to accomplish a task.

Deploying a back end data consumer in SENSEI makes it usable by any SENSEI instrumented simulation. SENSEI is focused on being light weight, having low memory and execution overhead, having a simple API, and minimizing dependencies for each back-end that we support.

Scientists often want to add their own diagnostics in addition to or in place of other back-ends. We strive to make this easy to do in Python and C++. We don't aim to replace or compete against an individual vis/analysis tool, we aim to make the pie bigger by making their tool and capabilities available to a broader set of users.

With SENSEI the sum is greater than the parts. For instance for both simulations and back-end data consumers, which have not been designed for in transit use, can be run in transit with out modification. Configuring for in transit run makes use of the same simple configuration mechanism that is used to select back-end data consumer.

Write once run everywhere. SENSEI provides access to a diverse set of in situ analysis back-ends and transport layers through a simple API and data model. Simulations instrumented with the SENSEI API can process data using any of these back-ends interchangeably. The back-ends are selected and configured at run-time via an XML configuration file. This document is targeted at scientists and developers wishing to run simulations instrumented with SENSEI, instrument a new simulation, or develop new analysis back-ends.

CHAPTER 2

Source Code

SENSEI is open source and freely available on github at https://github.com/SENSEI-insitu/SENSEI.

CHAPTER 3

Online Documentation

SENSEI has autmated Doxygen documentation at https://sensei-insitu.readthedocs.io/en/latest/doxygen.

CHAPTER 4

Table of Contents

4.1 Installation

The base install of SENSEI depends on CMake, MPI, Python, SWIG, numpy, and mpi4py.

```
git clone https://github.com/SENSEI-insitu/SENSEI
mkdir sensei-build
cd sensei-build
cmake ../SENSEI
make -j
```

This base install enables one to perform in situ in Python using user provided Python scripts. For more information on Python based in situ see our ISAV 2018 paper.

Additional in situ and in transit processing capabilities are available by enabling various build options on the CMake command line.

Build Option	Default	Description
ENABLE_CUDA	OFF	Enables CUDA accelerated codes. Requires compute capability 7.5 and CUDA 11 or later.
ENABLE_PYTHON	ON	Enables Python bindings. Requires Python, Numpy, mpi4py, and SWIG.
ENABLE_CATALYST	OFF	Enables the Catalyst analysis adaptor. Depends on ParaView Catalyst. Set *ParaView_DIR*.
ENABLE_CATALYST_PYTHON	OFF	Enables Python features of the Catalyst analysis adaptor.
ParaView_DIR		Set to the directory containing ParaViewConfig.cmake.
ENABLE_ASCENT	OFF	Enables the Ascent analysis adaptor. Requires an Ascent install.
ASCENT_DIR		Set to the directory containing the Ascent CMake configuration.
ENABLE_ADIOS2	OFF	Enables ADIOS 2 in transit transport. Set *ADIOS2_DIR*.
ADIOS2_DIR		Set to the directory containing ADIOSConfig.cmake
ENABLE_HDF5	OFF	Enables HDF5 adaptors and endpoints. Set *HDF5_DIR*.
HDF5_DIR		Set to the directory containing HDF5Config.cmake
ENABLE_LIBSIM	OFF	Enables Libsim data and analysis adaptors. Requires Libsim. Set *VTK_DIR* and *LIBSIM_DIR*.
LIBSIM_DIR		Path to libsim install.
ENABLE_VTK_IO	OFF	Enables adaptors to write to VTK XML format.
ENABLE_VTK_MPI	OFF	Enables MPI parallel VTK filters, such as parallel I/O.
VTK_DIR		Set to the directory containing VTKConfig.cmake.
ENABLE_VTKM	OFF	Enables analyses that use VTK-m. Requires an install of VTK-m. Experimental, each implementation requires an exact version match

4.2 System Components

4.2.1 System Overview & Architecture

SENSEI is a light weight framework for in situ data analysis. SENSEI's data model and API provide uniform access to and run time selection of a diverse set of visualization and analysis back ends including VisIt Libsim, ParaView Catalyst, VTK-m, Ascent, ADIOS, Yt, and Python.

In situ architecture

Fig. 4.1: SENSEI's in situ architecture enables use of a diverse of back ends which can be selected at run time via an XML configuration file

The three major architectural components in SENSEI are *data adaptors* which present simulation data in SENSEI's data model, *analysis adaptors* which present the back end data consumers to the simulation, and *bridge code* from which the simulation manages adaptors and periodically pushes data through the system. SENSEI comes equipped with a number of analysis adaptors enabling use of popular analysis and visualization libraries such as VisIt Libsim, ParaView Catalyst, Python, and ADIOS to name a few. AMReX contains SENSEI data adaptors and bridge code making it easy to use in AMReX based simulation codes.

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, another Libsim, yet another Python with no changes to the code. This is depicted in figure Fig. 4.1. On the left side of the figure

AMReX produces data, the bridge code pushes the data through the configurable analysis adaptor to the back end that was selected at run time.

In transit architecture

Fig. 4.2: SENSEI's in transit architecture enables decoupling of analysis and simulation.

SENSEI's in transit architecture enables decoupling of analysis and simulation. In this configuration the simulation runs in one job and the analysis runs in a second job, optionally on a separate set of compute resources, optionally at a smaller or larger level of concurrency. The configuration is made possible by a variety of *transports* who's job is to move and repartitions data. This is depicted in figure Fig. 4.2.

In the in transit configuration, the simulation running in one job uses SENSEI's *configurable analysis adaptor* to select and configure the write side of the transport. When the simulation pushes data through the SENSEI API for analysis the transport deals with presenting and moving data needed for analysis across the network. In asynchronous mode the simulation proceeds while the data is processed.

A second job, running the SENSEI *in transit end-point*, uses the *configurable analysis adaptor* to select and configure one of the back-ends. A transport specific data adaptor presents the available data to the analysis. The analysis can select and request data to be moved across the network for processing.

SENSEI's design enables this configuration to occur with no changes to either the simulation or analysis back-end. The process is entirely seamless from the simulations point of view and can be so if desired on the analysis side as well. SENSEI supports in transit aware analyses, and provides API's for yielding control data repartitioning to the analysis.

4.2.2 Adaptor API's

SENSEI makes heavy use of the adaptor design pattern. This pattern is used to abstract away the details of complex and diverse systems exposing them through a single API. SENSEI has 2 types of adaptor. The DataAdaptor abstracts away the details of accessing simulation data. This let's analysis back-ends access any simulation's data through a single API. The AnalysisAdaptor abstarcts away the details of the analysis back-ends. This let's the simulation invoke all of the various analysis back-ends through a single API. When a simulation invokes an analysis back-end it passes it a DataAdaptor that can be used to access simulation data.

DataAdaptor API

SENSEI's data adaptor API abstracts away the differences between simulations allowing SENSEI's transports and analysis back ends to access data from any simulation in the same way. A simulation must implement the data adaptor API and pass an instance when it wishes to trigger in situ processing.

Through the data adaptor API the analysis back end can get metadata about what the simulation can provide. This metadata is examined and then the analysis can use the API to fetch only the data it needs to accomplish the tasks it has been configured to do.

Finally the data adaptor is a key piece of SENSEI's in transit system. The analysis back end can be run in a different parallel job and be given an in transit data adaptor in place of the simulation's data adaptor. In this scenario the in transit data adaptor helps move data needed by the analysis back end. The data adaptor API enables this scenario to appear the same to the simulation and the analysis back end. Neither simulation nor analysis need be modified for in transit processing.

Core API

Simulations need to implement the core API.

In transit API

In transit transports need to implement the in transit API.

AnalysisAdaptor API

Extending SENSEI for customized Analysis capabilities requires implementing a sensei::AnalysisAdaptor .

At a minimum one must implement the *sensei::AnalysisAdaptor::Execute* method. In your implementation you will make use of the passed sensei::DataAdaptor instance to fetch the necessary simulation data.

The following template can be used to add a new C++ based analysis capability.

```
class MyAnalysis : public sensei::AnalysisAdaptor
{
public:

    virtual bool Execute(DataAdaptor* dataIn, DataAdaptor** dataOut)
    {
        // YOUR ANALYSIS CODE HERE. USE THE PASSED DATA ADAPTOR TO ACCESS
        // SIMULATION DATA

        if (dataOut)
        {
            // IF YOUR CODE CAN RETURN DATA, CREATE AND RETURN A DATA
            // ADAPTOR HERE THAT CAN BE USED TO ACCESS IT
            *dataOut = nullptr;
        }

        return true;
    }

};
```

Python API

The sensei::PythonAnalysis adaptor enables the use of a Python scripts as an analysis back end in C,C++, and Fortran based simulation codes. It accomplishes this by embedding a Python interpreter and includes a minimal set of the sensei python bindings. To author a new python analysis one must provide a python script that implements three functions in a user provided Python script that is loaded at run time. The three functions are: *Inititalize*, *Execute* and *Finalize*. These functions implement the *sensei::AnalysisAdaptor* API.

The *Execute* function is required while *Initialize* and *Finalize* functions are optional. The *Execute* function is passed a *sensei::DataAdaptor* instance from which one has access to simulation data structures. If an error occurs during processing one should raise an exception. If the analysis required MPI communication, one must make use of the adaptor's MPI communicator which is stored in the global variable comm. Additionally one can provide a secondary script that is executed prior to the API functions. This script can set global variables that control runtime behavior.

End users will make use of the *sensei::ConfigurableAnalysis* and point to the python script containing the three functions described above. The script can be loaded in one of two ways: via python's import machinery or via a customized

mechanism that reads the file on MPI rank 0 and broadcasts it to the other ranks. The latter is the recommended approach.

```
def Initialize():
    """ Initialization code here """
    return

def Execute(dataAdaptor):
    """ Use sensei::DataAdaptor API to process data here """
    return

def Finalize():
    """ Finalization code here """
    return
```

4.2.3 Data model

The data model is a key piece of the system. It allows data to be packaged and shared between simulations and analysis back ends. SENSEI's data model relies on VTK's vtkDataObject class hierarchy to provide containers of array based data, VTK's conventions for mesh based data (i.e. ordering of FEM cells), and our own metadata object that is used to describe simulation data and it's mapping onto hardware resources.

Representing mesh based data

SENSEI makes use of VTK data object's to represent simulation data. VTK supports a diverse set of mesh and non-mesh based data. Figure *numref*:data_types shows a subset of the types of data supported in the VTK data model.

Fig. 4.3: A subset of the supported data types.

A key concept in understanding our use of VTK is that we view all data conceptually as multi-block. By multi-block we mean that each MPI rank has zero or more blocks of data. When we say blocks we really mean chunks or pieces, because the blocks can be anything ranging from point sets, to FEM cells, to hierarchical AMR data. to tables, to arrays. The blocks of a multi-block are distributed across the simulation's MPI ranks with each rank owning a subset of the blocks. An example is depicted in figure *numref*:multi_block where the 2 data blocks of a multi-block dataset are partitioned across 2 MPI ranks.

Fig. 4.4: Multi-block data. Each rank has zero or more data blocks. In VTK non-local blocks are nullptr's.

A strength of VTK is the diversity of data sets that can be represented. A challenge that comes with this lies in VTK's complexity. SENSEI's data model only relies on VTK's common, core and data libraries reducing surface area and complexity when dealing with VTK. While it is possible to use any class derived from vtkDataObject with SENSEI the following data sets are supported universally by all transports and analysis back-ends.

VTK Class	Description
vtkImageData	Blocks of uniform Cartesian geometry
vtkRectilinearGrid	Blocks of stretched Cartesian geometry
vtkUnstructuredGrid	Blocks of finite element method cell zoo and particle meshes
vtkPolyData	Blocks of particle meshes
vtkStructuredGrid	Blocks of logically Cartesian (aka Curvilinear) geometries
vtkOverlappingAMR	A collection of blocks in a block structured AMR hierarchy
vtkMultiBlockDataSet	A collection of data blocks distributed across MPI ranks

As mentioned VTK's data model is both rich and complex. VTK's capabilities go well beyond SENSEI's universal support. However, any dataset type derived from vtkDataObject can be used with SENSEI including those not listed in the table above. The successful use of classes not listed in the above table depends on support implemented by the back end or transport in question.

Representing array based data

Each block of a simulation mesh is expected to contain one or more data arrays that hold scalar, vector, and tensor fields generated by the simulation. VTK's data arrays are used to present array based data. VTK's data arrays are similar to the STL's std::vector, but optimized for high-performance computing. One such optimization is the support for zero-copy data transfer. With zero-copy data transfer it is possible to pass a pointer to simulation data directly to an analysis back-end without making a copy of the data.

All of the mesh based types in VTK are derived from vtkDataSet. vtkDataSet defines the common API's for accessing collections of VTK data arrays by geometric centering. SENSEI supports the following two containers in all back-ends and transports.

Class	Description
vtkPointData	Container of node centered arrays
vtkCellData	Container of cell centered arrays

VTK data arrays support use of any C++ POD type. The two main classes of VTK data arrays of interest here are:

Class	Description
vtkAOSDataArrayTemplate	Use with scalar, vector and tensor data in AOS layout
vtkSOADataArrayTemplate	Use with vector and tensor data in SOA layout

These classes define the API for array based data in VTK. Note the AOS layout is the default in VTK and that classes such as vtkFloatArray, vtkDoubleArray, vtkIntArray etc are aliases to vtkAOSDataArrayTemplate. For simplicity sake one can and should use these aliases anywhere an AOS layout is needed.

Zero-copy into VTK

The following snippet of code shows how to pass a 3 component vector field in the AOS layout from the simulation into VTK using the zero-copy mechanism:

```
// VTK's default is AOS, no need to use vtkAOSDataArrayTemplate
vtkDoubleArray *aos = vtkDoubleArray::New();
aos->SetNumberOfComponents(3);
aos->SetArray(v, 3*nxy, 0);
aos->SetName("velocity");

// add the array as usual
im->GetPointData()->AddArray(aos);

// give up our reference
aos->Delete();
```

The following snippet of code shows how to pass a 3 component vector field in the SOA layout from the simulation into VTK using the zero-copy mechanism:

```
// use the SOA class
vtkSOADataArrayTemplate<double> *soa = vtkSOADataArrayTemplate<double>::New();
soa->SetNumberOfComponents(3);

// pass a pointer for each array
soa->SetArray(0, vx, nxy, true);
soa->SetArray(1, vy, nxy);
soa->SetArray(2, vz, nxy);
soa->SetName("velocity");

// add to the image as usual
im->GetPointData()->AddArray(soa);

// git rid of our reference
soa->Delete();
```

In both these examples 'im' is a dataset for some block in a multiblock data set.

Accessing blocks of data

This section pertains to accessing data for analysis. During analysis one may obtain a mesh from the simulation. With the mesh in hand one can walk the blocks of data and access the array collections. Arrays in the array collection are accessed and a pointer to the data is obtained for processing. The collections of blocks in VTK are derived from vtkCompositeDataSet. vtkCompositeDataSet defines the API for generically access blocks via the vtkCompositeDataIterator class. The vtkCompositeDataIterator is used to visit all data blocks local to the MPI rank.

Getting help with VTK

For those new to VTK a good place to start is the VTK user guide which contains a chapter devoted to learning VTK data model as well as numerous examples. On the VTK community support forums volunteers, and often the VTK developers them selves, answer questions in an effort to help new users.

Metadata

SENSEI makes use of a custom metadata object to describe simulation data and its mapping onto hardware resources. This is in large part to support in transit operation where one must make decisions about how simulation data maps onto available analysis resources prior to accessing the data.

Applies to	Field name	Purpose
entire mesh	GlobalView	tells if the information describes data on this rank or all ranks
	MeshName	name of mesh
	MeshType	VTK type enum of the container mesh type
	BlockType	VTK type enum of block mesh type
	NumBlocks	global number of blocks
	NumBlocksLocal	number of blocks on each rank
	Extent	global index space extent †,§,*
	Bounds	global bounding box *
	CoordinateType	type enum of point data ‡
	NumPoints	total number of points in all blocks *
	NumCells	total number of cells in all blocks *
	CellArraySize	total cell array size in all blocks *

Continued on next page

Table 4.1 – continued from previous page

Applies to	Field name	Purpose
	NumArrays	number of arrays
	NumGhostCells	number of ghost cell layers
	NumGhostNodes	number of ghost node layers
	NumLevels	number of AMR levels (AMR)
	PeriodicBoundary	indicates presence of a periodic boundary
	StaticMesh	non zero if the mesh does not change in time
each array	ArrayName	name of each data array
	ArrayCentering	centering of each data array
	ArrayComponents	number of components of each array
	ArrayType	VTK type enum of each data array
	ArrayRange	global min,max of each array *
each block	BlockOwner	rank where each block resides *
	BlockIds	global id of each block *
	BlockNumPoints	number of points for each block *
	BlockNumCells	number of cells for each block *
	BlockCellArraySize	cell array size for each block ‡,*
	BlockExtents	index space extent of each block †,§,*
	BlockBounds	bounds of each block *
	BlockLevel	AMR level of each block §
	BlockArrayRange	min max of each array on each block *
each level	RefRatio	refinement ratio in i,j, and k direction §
	BlocksPerLevel	number of blocks in each level §

The metadata structure is intended to be descriptive and cover all of the supported scenarios. Some of the fields are potentially expensive to generate and not always needed. As a result not all fields are used in all scenarios. Flags are used by the analysis to specify which fields are required. The following table is used in conjunction with the above table to define under which circumstances the specific the fields are required.

symbol	required . . .
	always required
*	only if requested by the analysis
†	with Cartesian meshes
‡	with unstructured meshes
§	with AMR meshes

Simulations are expected to provide local views of metadata, and can optionally provide global views of metadata. The GlobalView field is used to indicate which is provided. SENSEI contains utilities to generate a global view form a local one.

Ghost zone and AMR mask array conventions

SENSEI uses the conventions defined by VisIt and recently adopted by VTK and ParaView for masking ghost zones and covered cells in overlapping AMR data. In accordance with VTK convention these arrays must by named svtkGhostType.

Mask values for cells and cell centered data:

Type	Bit
valid cell, not masked	0
Enhanced connectivity zone	1
Reduced connectivity zone	2
Refined zone in AMR grid	3
Zone exterior to the entire problem	4
Zone not applicable to problem	5

Mask values for points and point centered data:

Type	Bit
Valid node, not masked	0
Node not applicable to problem	1

For more information see the Kitware blog on ghost cells and the VisIt ghost data documentation.

Overhead due to the SENSEI data model

As in any HPC application we are concerned with the overhead associated with our design choices. To prove that we have minimal impact on a simulation we did a series of scaling and performance analyses up to 45k cores on a Cray supercomputer. We then ran a series of common visualization and analysis tasks up to 1M cores on second system. The results of our experiments that showed the SENSEI API and data model have negligible impact on both memory use and run-time of the simulation. A selection of the results are shown in figure Fig. 4.5.

Fig. 4.5: Run-time (left) and memory use (right) with (orange) and without (blue) SENSEI.

The full details of the performance and scaling studies can be found in our SC16 paper.

4.2.4 Analysis back-ends

Ascent back-end

Ascent is a many-core capable lightweight in-situ visualization and analysis infrastructure for multi-physics HPC simulations. The SENSEI AscentAnalysisAdaptor enables simulations instrumented with SENSEI to process data using Ascent.

SENSEI XML

The ascent back-end is activated using the `<analysis type="ascent">`. The supported attributes are:

attribute	description
actions	Path to ascent specific JSON file configuring ascent
options	Path to ascent specific JSON file configuring ascent

Back-end specific configurarion

SENSEI uses XML to select the specific back-end, in this case Ascent. The SENSEI XML will also contain references to Ascent specific configuration files that tell Ascent what to do. These files are native to Ascent. More information about configuring Ascent can be found in the Ascent documentation at https://ascent.readthedocs.io/en/latest/

Examples

Reaction rate in situ demo *Ascent in situ demo*.

Catalyst back-end

ParaView Catalyst (Catalyst) is an in situ use case library, with an adaptable application programming interface (API), that orchestrates the delicate alliance between simulation and analysis and/or visualization tasks. It brings the renown, scaling capabilities of VTK and ParaView to bear on the in situ use case. The analysis and visualization tasks can be implemented in C++ or Python, and Python scripts can be crafted from scratch or using the ParaView GUI to interactively setup Catalyst scripts (see Catalyst User Guide).

SENSEI XML Options

The Catalyst back-end is activated using the `<analysis type="catalyst">`.

Python Script

The supported attributes are:

attribute	description
pipeline	Use "pythonscript".
filename	pythonscript filename.
enabled	"1" enables this back-end.

Example XML

Catalyst Python script example. This XML configures a Catalyst with a Python script that creates a pipeline(s).

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

Back-end specific configuration

The easiest way to create a python script for Catalyst:

1. Load a sample of the data (possibly downsampled) into ParaView, including all the desired fields.
2. Create analysis and visualization pipeline(s) in ParaView by applying successive filters producing subsetted or alternative visual metaphors of data.

3. Define a Catalyst extracts with the menu choice *Catalyst→Define Exports*: this will pop up the *Catalyst Export Inspector* panel.
4. Export the Catalyst Python script using the menu *Catalyst→Export Catalyst Script*.

The Catalyst Export Inspector reference.

Slice Fixed Pipeline

For the Catalyst slice fixed pipeline the supported attributes are:

attribute	description
pipeline	Use "slice".
mesh	The name of the mesh to slice.
array	The data array name for coloring.
association	Either "cell" or "point" data.
image-filename	The filename template to write images.
image-width	The image width in pixels.
image-height	The image height in pixels.
slice-origin	The origin to use for slicing (optional).
slice-normal	The normal to use for slicing.
color-range	The color range of the array (optional).
color-log	Use logarithmic color scale (optional).
enabled	"1" enables this back-end

Example XML

This XML configures a C++-based fixed pipeline for a slice using Catalyst.

```
<sensei>
  <analysis type="catalyst"
            pipeline="slice" mesh="mesh" array="data" association="cell"
            image-filename="slice-%ts.png" image-width="1920" image-height="1080"
            slice-normal="0,0,1"
            color-range="0.0001,1.5" color-log="1"
            enabled="1" />
</sensei>
```

Particles Fixed Pipeline

For the Catalyst particle fixed pipeline the supported attributes are:

attribute	description
pipeline	Use “particle”.
mesh	The name of the mesh to slice.
array	The data array name for coloring.
association	Either “cell” or “point” data.
image-filename	The filename template to write images.
image-width	The image width in pixels.
image-height	The image height in pixels.
particle-style	The representation such as: “Gaussian Blur”, “Sphere”,
	“Black-edged circle”, “Plain circle”, “Triangle”, and
	“Square Outline”.
particle-radius	The normal to use for slicing.
color-range	The color range of the array (optional).
camera-position	The position of the camera (optional).
camera-focus	Where the camera points (optional).
enabled	“1” enables this back-end

Example XML

This XML configures a C++-based fixed pipeline for particles using Catalyst.

```
<sensei>
  <analysis type="catalyst"
     pipeline="particle" mesh="particles" array="data" association="point"
     image-filename="/tmp/particles-%ts.png" image-width="1920" image-height="1080"
     particle-style="Black-edged circle" particle-radius="0.5"
     color-range="0.0,1024.0" color-log="0"
     camera-position="150,150,100" camera-focus="0,0,0"
     enabled="1" />
</sensei>
```

Example

Reaction rate in situ demo *ParaView Catalyst in situ demo*.

Histogram back-end

As a simple analysis routine, the Histogram back-end computes the histogram of the data. At any given time step, the processes perform two reductions to determine the minimum and maximum values on the mesh. Each processor divides the range into the prescribed number of bins and fills the histogram of its local data. The histograms are reduced to the root process. The only extra storage required is proportional to the number of bins in the histogram.

SENSEI XML

The Histogram back-end is activated using the `<analysis type="histogram">`. The supported attributes are:

attribute	description
mesh	The name of the mesh for histogram.
array	The data array name for histogram.
association	Either "cell" or "point" data.
file	The filename template to write images.
bins	The number of histogram bins.

Example XML

Histogram example. This XML configures Histogram analysis.

```
<sensei>
  <analysis type="histogram"
    mesh="mesh" array="data" association="cell"
    file="hist.txt" bins="10"
    enabled="1" />
</sensei>
```

Back-end specific configurarion

No special back-end configuration is necessary.

Examples

VM Demo reference.

Autocorrelation back-end

As a prototypical time-dependent analysis routine, the Autocorrelation back-end computes the autocorrelation. Given a signal f(x) and a delay t, we find

$$\sum_x f(x)f(x+t).$$

Starting with an integer time delay t, we maintain in a circular buffer, for each grid cell, a window of values of the last t time steps. We also maintain a window of running correlations for each t t. When called, the analysis updates the autocorrelations and the circular buffer. When the execution completes, all processes perform a global reduction to determine the top k autocorrelations for each delay t t (k is specified by the user). For periodic oscillators, this reduction identifies the centers of the oscillators.

SENSEI XML

The Autocorrelation back-end is activated using the `<analysis type="autocorrelation">`. The supported attributes are:

attribute	description
mesh	The name of the mesh for autocorrelation.
array	The data array name for autocorrelation.
association	Either "cell" or "point" data.
window	The delay (t) for f(x).
k-max	The number of strongest autocorrelations to report.

Example XML

Autocorrelation example. This XML configures Autocorrelation analysis.

```
<sensei>
  <analysis type="autocorrelation"
    mesh="mesh" array="data" association="cell"
    window="10" k-max="3" enabled="1" />
</sensei>
```

Examples

VM Demo reference.

4.2.5 In transit transport layers and I/O

In transit data adaptor & control API

ADIOS-1

(Burlen)

ADIOS-2

(J.Logan)

Libis

(Silvio)

Data elevators

(Junmin)

4.2.6 Partitioners

4.2.7 The SENSEI end-point

4.3 Miniapps

4.3.1 oscillator

The oscillator mini-application computes a sum of damped, decaying, or periodic oscillators, convolved with (un-normalized) Gaussians, on a grid. It could be configured as a proxy for simulation of a chemical reaction on a two-dimensional substrate (see *Chemical reaction on a 2D substrate*).

option	description
-b, –blocks INT	Number of blocks to use [default: 1].
-s, –shape POINT	Number of cells in the domain [default: 64 64 64].
-e, –bounds FLOAT	Bounds of the Domain [default: {0,-1,0,-1,0,-1]}.
-t, –dt FLOAT	The time step [default: 0.01].
-f, –config STRING	SENSEI analysis configuration xml (required).
-g, –ghost-cells INT	Number of ghost cells [default: 1].
–t-end FLOAT	Request synchronize after each time step.
-j, –jobs INT	Number of threads [default: 1].
-o, –output STRING	Prefix for output [default: “”].
-p, –particles INT	Number of particles [default: 0].
-v, –v-scale FLOAT	Gradient to Velocity scale factor [default: 50].
-r, –seed INT	Random seed [default: 1].
–sync	The end time [default: 10].
-h, –help	Show help.

The oscillators’ locations and parameters are specified in an input file (see input folder for examples).

Note that the generate_input script can generate a set of randomly initialized oscillators.

The simulation code is in main.cpp while the computational kernel is in Oscillator.cpp.

To run:

There are a number of examples available in the SENSEI repositoory that leverage the oscillator mini-application.

4.3.2 newton

4.3.3 mandelbrot

4.4 Examples

4.4.1 Chemical reaction on a 2D substrate

This example illustrates how to select different back-ends at run time via XML, and how to switch in between in situ mode where the analysis runs in the same address space as the simulation and in transit mode where the analysis runs in a separate application called an end-point potentially on a different number of MPI ranks.

This example makes use of the oscillator mini-app configured as a proxy for simulation of a chemical reaction on a 2D substrate. The example uses different back-ends to make a pseudo coloring of the reaction rate with an iso-contour

of 1. The Python analysis computes the area of the substrate where the reaction rate is greater or equal to 1 and plots it over time.

In situ demos

In this part of the demo XML files are used to switch back-end data consumer. The back-end data consumers are running in the same process as the simulation. This enables the use of zero-copy data transfer between the simulation and data consumer.

Ascent in situ demo

Fig. 4.6: A pseudocolor plot rendered by Ascent of the rectaion rate field with an iso-contour plotted at a reaction rate of 1.0.

In the demo data from the reaction rate proxy simulation is processed using Ascent. Ascent is selected at run time via the following SENSEI XML:

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

XML to select the Ascent back-end and configure it using a Ascent JSON
configuration
```

The analysis element selects Ascent, the actions attribute points to the Ascent specific configuration. In this case a JSON configuration. The following shell script runs the demo on the VM.

```
#!/bin/bash

n=4
b=64
dt=0.25
bld=`echo -e '\e[1m'`
red=`echo -e '\e[31m'`
grn=`echo -e '\e[32m'`
blu=`echo -e '\e[36m'`
wht=`echo -e '\e[0m'`

echo "+ module load sensei/3.1.0-ascent-shared"
module load sensei/3.1.0-ascent-shared

set -x

export OMP_NUM_THREADS=1

cat ./configs/random_2d_${b}_ascent.xml | sed "s/.*/$blu&$wht/"

mpiexec -n ${n} \
    oscillator -b ${n} -t ${dt} -s ${b},${b},1 -g 1 -p 0 \
```

(continues on next page)

(continued from previous page)

```
    -f ./configs/random_2d_${b}_ascent.xml \
    ./configs/random_2d_${b}.osc 2>&1 | sed "s/.*/$red&$wht/"
```

During the run Ascent is configured to render a pseudocolor plot of the reaction rate field. The plot includes an iso-contour where the reaction rate is 1.

ParaView Catalyst in situ demo

Fig. 4.7: A pseudocolor plot rendered by ParaView Catalyst of the rectaion rate field with an iso-contour plotted at a reaction rate of 1.0.

In the demo data from the reaction rate proxy simulation is processed using ParaView Catalyst. Catalyst is selected at run time via the following SENSEI XML:

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

The analysis element selects ParaView Catalyst, the filename attribute points to the Catalyst specific configuration. In this case a Python script that was generated using the ParaView GUI. The following shell script runs the demo on the VM.

```
#!/bin/bash

n=4
b=64
dt=0.25
bld=`echo -e '\e[1m'`
red=`echo -e '\e[31m'`
grn=`echo -e '\e[32m'`
blu=`echo -e '\e[36m'`
wht=`echo -e '\e[0m'`

echo "+ module load sensei/3.0.0-catalyst-shared"
module load sensei/3.0.0-catalyst-shared

set -x

cat ./configs/random_2d_${b}_catalyst.xml | sed "s/.*/$blu&$wht/"

mpiexec -n ${n} \
    oscillator -b ${n} -t ${dt} -s ${b},${b},1 -g 1 -p 0 \
    -f ./configs/random_2d_${b}_catalyst.xml \
    ./configs/random_2d_${b}.osc 2>&1 | sed "s/.*/$red&$wht/"
```

During the run ParaView Catalyst is configured to render a pseudocolor plot of the reaction rate field. The plot includes an iso-contour where the reaction rate is 1.

VisIt Libsim in situ demo

In the demo data from the reaction rate proxy simulation is processed using VisIt Libsim. Libsim is selected at run time via the following SENSEI XML:

Fig. 4.8: A pseudocolor plot rendered by VisIt Libsim of the rectaion rate field with an iso-contour plotted at a reaction rate of 1.0.

```
<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>
```

The analysis element selects VisIt Libsim, the filename attribute points to the Libsim specific configuration. In this case a session file that was generated using the VisIt GUI. The following shell script runs the demo on the VM.

```
#!/bin/bash

n=4
b=64
dt=0.25
bld=`echo -e '\e[1m'`
red=`echo -e '\e[31m'`
grn=`echo -e '\e[32m'`
blu=`echo -e '\e[36m'`
wht=`echo -e '\e[0m'`

echo "+ module load sensei/3.0.0-libsim-shared"
module load sensei/3.0.0-libsim-shared

set -x

cat ./configs/random_2d_${b}_libsim.xml | sed "s/.*/$blu&$wht/"

mpiexec -n ${n} \
    oscillator -b ${n} -t ${dt} -s ${b},${b},1 -g 1 -p 0 \
    -f ./configs/random_2d_${b}_libsim.xml \
    ./configs/random_2d_${b}.osc 2>&1 | sed "s/.*/$red&$wht/"

Shell script that runs the Libsim in situ demo on the VM.
```

During the run VisIt Libsim is configured to render a pseudocolor plot of the reaction rate field. The plot includes an iso-contour where the reaction rate is 1.

Python in situ demo

Fig. 4.9: A plot of the time history of the area of the 2D substrate where the reaction rate is greater or equal to 1.0.

In the demo data from the reaction rate proxy simulation is processed using Python. Python is selected at run time via the following SENSEI XML:

```
<sensei>
  <analysis type="python" script_file="configs/volume_above_sm.py" enabled="1">
    <initialize_source>
threshold=1.0
```

(continues on next page)

(continued from previous page)

```
mesh='mesh'
array='data'
cen=1
out_file='random_2d_64_python.png'
     </initialize_source>
  </analysis>
</sensei>
```

The analysis element selects Python, the script_file attribute points to the user provided Python script and initialize_source contains run time configuration. The following shell script runs the demo on the VM.

```
#!/bin/bash

n=4
b=64
dt=0.01
bld=`echo -e '\e[1m'`
red=`echo -e '\e[31m'`
grn=`echo -e '\e[32m'`
blu=`echo -e '\e[36m'`
wht=`echo -e '\e[0m'`

export MPLBACKEND=Agg

echo "+ module load sensei/3.0.0-vtk-shared"
module load sensei/3.0.0-vtk-shared

set -x

cat ./configs/random_2d_${b}_python.xml | sed "s/.*/$blu&$wht/"

mpiexec -n ${n} oscillator -b ${n} -t ${dt} -s ${b},${b},1 -p 0 \
    -f ./configs/random_2d_${b}_python.xml \
    ./configs/random_2d_${b}.osc  2>&1 | sed "s/.*/$red&$wht/"
```

During the run this user provided Python script computes the area of the 2D substrate where the reaction rate is greater or equal to 1. The value is stored and at the end of the run a plot of the time history is made.

In transit demos

ParaView Catalyst

VisIt Libsim

Ascent

Python

4.4.2 Building Pipelines

Pipelines of in situ and in transit operations can be constructed by chaining *sensei::AnalysisAdaptors* using C++ or Python.

The following example shows a SENSEI PythonAnalysis script that configures and executes a simple pipeline that calculates a set of iso-surfaces using SENSEI's SliceExtract and then writes them to disk using SENSEI's VTKPosthocIO analysis adaptor.

```
from sensei import SliceExtract, VTKPosthocIO
from svtk import svtkDataObject
import sys

# control
meshName  = None
arrayName  = None
arrayCen  = None
isoVals  = None
outputDir  = None

# state
slicer = None
writer = None

def Initialize():
    global slicer,writer
    if comm.Get_rank() == 0:
        sys.stderr.write('PipelineExample::Initialize\n')
        sys.stderr.write('meshName=%s, arrayName=%s, arrayCen=%d, '
                         'isoVals=%s, outputDir=%s\n'%(meshName,
                         arrayName, arrayCen, str(isoVals), outputDir))

    slicer = SliceExtract.New()
    slicer.EnableWriter(0)
    slicer.SetOperation(SliceExtract.OP_ISO_SURFACE)
    slicer.SetIsoValues(meshName, arrayName, arrayCen, isoVals)
    slicer.AddDataRequirement(meshName, arrayCen, [arrayName])

    writer = VTKPosthocIO.New()
    writer.SetGhostArrayName('ghosts')
    writer.SetOutputDir(outputDir)
    writer.SetWriter(VTKPosthocIO.WRITER_VTK_XML)
    writer.SetMode(VTKPosthocIO.MODE_PARAVIEW)

def Execute(daIn):
    if comm.Get_rank() == 0:
        sys.stderr.write('PipelineExample::Execute %d\n'%(daIn.GetDataTimeStep()))

    # stage 1 - iso surface
    isoOk, iso = slicer.Execute(daIn)

    if not isoOk:
        return 0

    # stage 2 - write the result
    writer.Execute(iso)

    return 1

def Finalize():
    if comm.Get_rank() == 0:
        sys.stderr.write('PipelineExample::Finalize\n')
```

(continues on next page)

(continued from previous page)

```
    slicer.Finalize()
    writer.Finalize()
```

For the purposes of this demo, the above Python script is stored in a file named "iso_pipeline.py" and refered to in the XML used to configure the system. The XML used to configure the system to run the iso-surface and write pipeline is shown below:

```
<sensei>
  <analysis type="python" script_file="iso_pipeline.py" enabled="1">
    <initialize_source>
meshName = 'mesh'
arrayName = 'data'
arrayCen = svtkDataObject.CELL
isoVals = [-0.619028, -0.0739720000000001, 0.47108399999999995, 1.01614]
outputDir = 'oscillator_iso_pipeline_%s'%(meshName)
     </initialize_source>
  </analysis>
</sensei>
```

The variables *meshName*, *arrayName*, *arrayCen*, *isoVals*, and *outputDir* are used to configure for a specific simulation or run. For the purposes of this demo this XML is stored in a file name *iso_pipeline.xml* and passed to the SENSEI ConfigurableAnalaysis adaptor during SENSEI initialization.

One can run the above pipeline with the oscillator miniapp that ships with SENSEI using the following shell script.

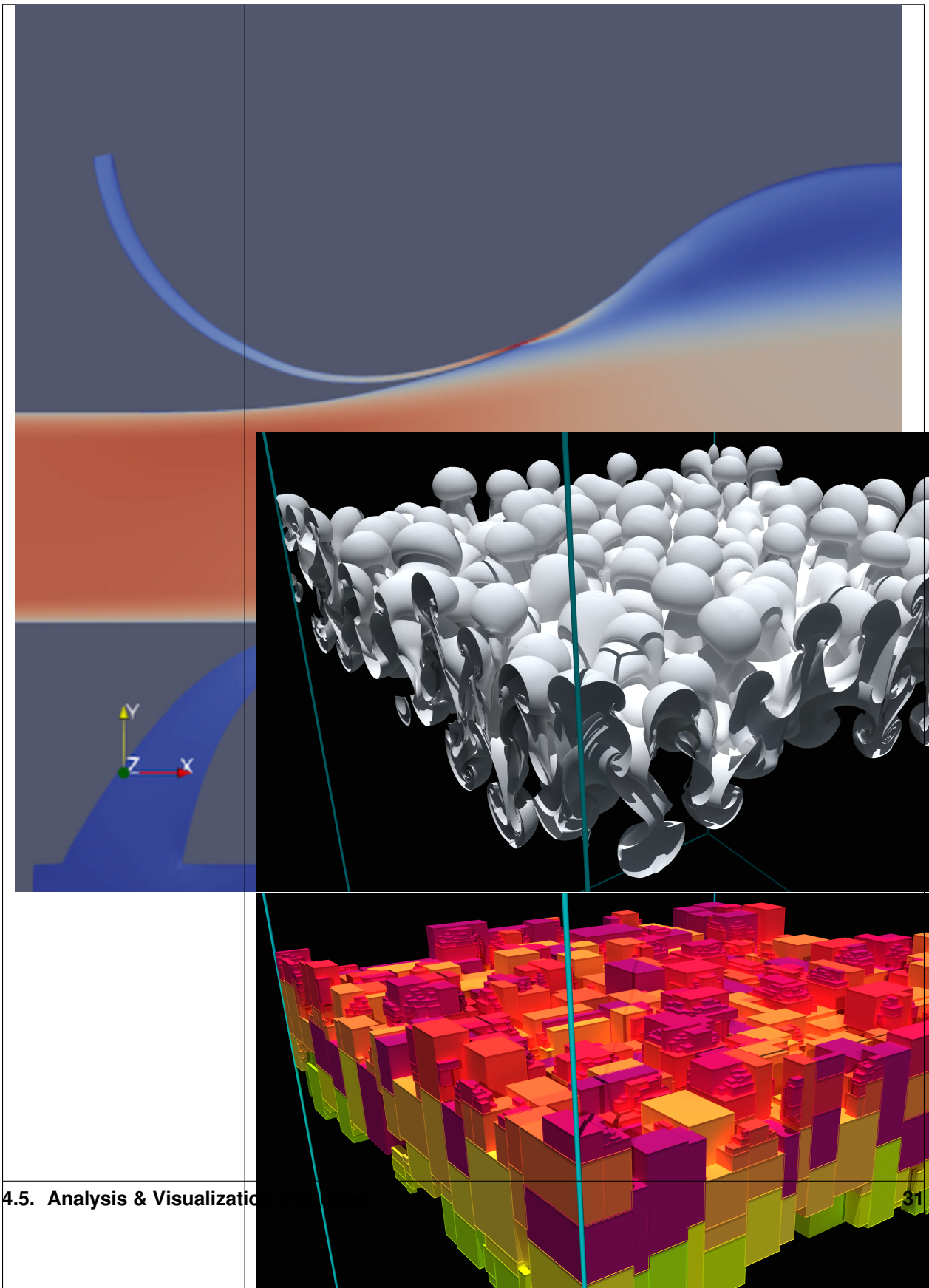
```
#!/bin/bash

 module load mpi/mpich-x86_64
 module use /work/SENSEI/modulefiles/
 module load sensei-vtk

 mpiexec -np 4 oscillator -t .5 -b 4 -g 1 -f iso_pipeline.xml simple.osc
```

4.5 Analysis & Visualization Vignettes

4.5.1 Papers



Software Tools to Enable Immersive Simulation

Felix Newberry, Corey Wetterer-Nelson, John A Evans, Alireza Doostan, Kenneth E Jansen

Full Text

Link to the full text PDF.

Abstract

There are two main avenues to design space exploration. In the first approach, a simulation is run, analyzed, the problem modified, and the simulation run again. In the second approach, an ensemble simulation is performed and the battery of results is leveraged to construct a surrogate model for a given quantity of interest (QoI). The first approach allows a practitioner to methodically move through the design space and analyze a solution field. A disadvantage of this technique is that each new simulation requires time consuming setup. The second approach provides the practitioner with a global view of the problem, but requires a priori design space limits and the QoI specification. In this work we introduce an immersive simulation software frame- work that enables practitioners to maintain the flexibility of the first approach, while eliminating the burden of setting up new simulations. Immersive simulation can also be used to inform the second approach, establishing limits and clarify- ing QoI selection prior to the launch of an ensemble simulation. We demonstrate live, reconfigurable visualization of on-going simulations coupled with live, reconfigurable problem definition that guides users in determining problem parameters. Ultimately, an immersive simulation framework enables more efficient design space exploration that reduces the gap between simulations, data analysis and insight ex- traction.

Improving Performance of M-to-N Processing and Data Redistribution in In Transit Analysis and Visualization

B. Loring 1, M. Wolf, J. Kress 2, S. Shudler, J. Gu , S. Rizzi, J. Logan, N. Ferrier, and E. W. Bethel

Full Text

Link to the full text PDF .

Abstract

In an in transit setting, a parallel data producer, such as a numerical simulation, runs on one set of ranks M, while a data consumer, such as a parallel visualization application, runs on a different set of ranks N. One of the central challenges in this in transit setting is to determine the mapping of data from the set of M producer ranks to the set of N consumer ranks. This is a challenging problem for several reasons, such as the producer and consumer codes potentially having different scaling characteristics and different data models. The resulting mapping from M to N ranks can have a significant impact on aggregate application performance. In this work, we present an approach for performing this M-to-N mapping in a way that has broad applicability across a diversity of data producer and consumer applications. We evaluate its design and performance with a study that runs at high concurrency on a modern HPC platform. By leveraging design characteristics, which facilitate an "intelligent" mapping from M-to-N, we observe significant performance gains are possible in terms of several different metrics, including time-to-solution and amount of data moved.

Instrumenting multiphysics blood flow simulation codes for in situ visualization and analysis

Anthony Bucaro, Connor Murphy, Nicola Ferrier, Joseph Insley, Victor Mateevitsi, Michael E Papka, Silvio Rizzi, Jifu Tan

Full Text

Link to the full text PDF.

Abstract

Blood flow simulations have important applications in engineering and medicine, requiring visualization and analysis for both fluid (blood plasma) and solid (cells). Recent advances in blood flow simulations highlight the need of a more efficient analysis of large data sets. Traditionally, analysis is performed after a simulation is completed, and any changes of simulation settings require running the simulation again. With bi-directional in situ analysis we aim to solve this problem by allowing manipulation of simulation parameters in run time. In this project, we describe our early steps toward this goal and present the in situ instrumentation of two coupled codes for blood flow simulation using the SENSEI in situ framework.

Spack meets singularity: creating movable in-situ analysis stacks with ease

Sergei Shudler, Nicola Ferrier, Joseph Insley, Michael E Papka, Silvio Rizzi

Full Text

Link to the full text PDF.

Abstract

In situ data analysis and visualization is a promising technique to handle the enormous amount of data an extreme-scale application produces. One challenge users often face in adopting in situ techniques is setting the right environment on a target machine. Platforms such as SENSEI require complex software stacks that consist of various analysis packages and visualization applications. The user has to make sure all these prerequisites exist on the target machine, which often involves compiling and setting them up from scratch. In this paper, we leverage the containers technology (eg, light-weight virtualization images) and provide users with Singularity containers that encapsulate ready-to-use, movable in situ software stacks. Moreover, we make use of Spack to ease the process of creating these containers. Finally, we evaluate this solution by running in situ analysis from within a container on an HPC system.

Fast Mesh Validation in Combustion Simulations through In-Situ Visualization.

Sergei Shudler, Nicola J Ferrier, Joseph A Insley, Michael E Papka, Saumil Patel, Silvio Rizzi

Full Text

Link to the full text PDF.

Abstract

In situ visualization and analysis is a powerful concept that aims to give users the ability to process data while it is still resident in memory, thereby vastly reducing the amount of data left for posthoc analysis. The problem of having too much data for posthoc analysis is exacerbated in large-scale high-performance computing applications such as Nek5000, a massively-parallel CFD (Computational Fluid Dynamics) code used primarily for thermal hydraulics problems. Specifically, one problem users of Nek5000 often face is validating the mesh, that is identifying the exact location of problematic mesh elements within the whole mesh. Employing the standard posthoc approach to address this problem is both time consuming and requires vast storage space. In this paper, we demonstrate how in situ visualization, produced with SENSEI, a generic in situ platform, helps users quickly validate the mesh. We also provide a bridge between Nek5000 and SENSEI that enables users to use any existing and future analysis routines in SENSEI. The approach is evaluated on a number of realistic datasets

Low-overhead in situ visualization using halo replay

Jeff Ames, Silvio Rizzi, Joseph Insley, Saumil Patel, Benjamín Hernández, Erik W Draeger, Amanda Randles

Full Text

Link to the full text PDF.

Abstract

In situ visualization and analysis is of increasing importance as the compute and I/O gap further widens with the advance to exascale capable computing. Yet, currently in situ methods impose resource constraints leading to the difficult task of balancing simulation code performance and the quality of analysis. Applications with tightly-coupled in situ visualization often achieve performance through spatial and temporal downsampling, a tradeoff which risks not capturing transient phenomena at sufficient fidelity. Determining a priori visualization parameters such as sampling rate is difficult without time and resource intensive experimentation. We present a method for reducing resource contention between in situ visualization and stencil codes on heterogeneous systems. This method permits full-resolution replay through recording halos and the communication-free reconstruction of interior values uncoupled from the main simulation. We apply this method in the computational fluid dynamics (CFD) code HARVEY on the Summit supercomputer. We demonstrate minimal overhead, in situ visualization relative to simulation alone, and compare the Halo Replay performance to tightly-coupled in situ approaches.

Python-based In Situ Analysis and Visualization

Burlen Loring, Andrew Myers, David Camp, E. Wes Bethel

Full Text

Link to the full text PDF.

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 userwritten 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.

In situ visualization and analysis to design large scale experiments in computational fluid dynamics

Bennett Bernardoni, Nicola Ferrier, Joseph Insley, Michael E Papka, Saumil Patel, Silvio Rizzi

Full Text

Link to the full text PDF.

Abstract

Nek5000 is a massively-parallel computational fluid dynamics code, which is widely used and researched, including as part of a co-design center of the Exascale Computing Project (ECP). As computation capacity reaches exascale, storage bandwidth remains stable leading to a larger percentage of time spent performing I/O. In situ analysis overcomes this issue by processing the data before it is written to disk. One method for accomplishing in situ analysis is through SENSEI, a generic in situ interface that enables the use of many existing in situ infrastructures with little modification to the simulation. In this work, we present the instrumentation of Nek5000 with SENSEI and evaluate its ability to accelerate the development of large scale simulation campaigns.

libIS: a lightweight library for flexible in transit visualization

Will Usher, Silvio Rizzi, Ingo Wald, Jefferson Amstutz, Joseph Insley, Venkatram Vishwanath, Nicola Ferrier, Michael E Papka, Valerio Pascucci

Full Text

Link to the full text PDF.

Abstract

As simulations grow in scale, the need for in situ analysis methods to handle the large data produced grows correspondingly. One desirable approach to in situ visualization is in transit visualization. By decoupling the simulation and visualization code, in transit approaches alleviate common difficulties with regard to the scalability of the analysis, ease of integration, usability, and impact on the simulation. We present libIS, a lightweight, flexible library which lowers the bar for using in transit visualization. Our library works on the concept of abstract regions of space containing data, which are transferred from the simulation to the visualization clients upon request, using a client-server model. We also provide a SENSEI analysis adaptor, which allows for transparent deployment of in transit visualization. We demonstrate the flexibility of our approach on batch analysis and interactive visualization use cases on different HPC resources.

In Situ Summarization with VTK-m

David Thompson, Sebastien Jourdain, Andrew Bauer, Berk Geveci, Robert Maynard, Ranga Raju Vatsavai, and Patrick O'Leary

Full Text

Link to the full text PDF.

Abstract

Summarization and compression at current and future scales requires a framework for developing and benchmarking algorithms. We present a framework created by integrating existing, production- ready projects and provide timings of two particular algorithms that serve as exemplars for summarization: a wavelet-based data reduction filter and a generator for creating image-like databases of extracted features (isocontours in this case). Both support browser-based, post-hoc, interactive visualization of the summary for decision- making. A study of their weak-scaling on a distributed multi-GPU system is included.

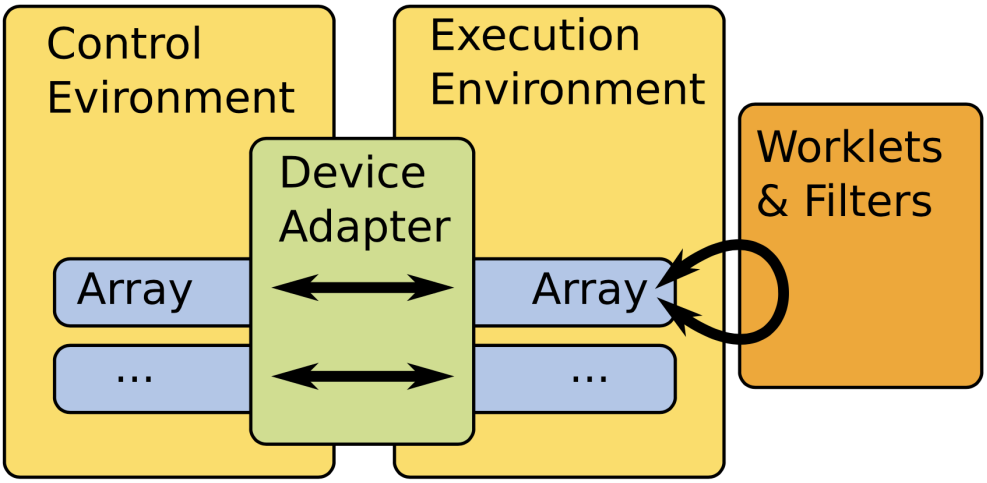


Fig. 4.10: VTKm's architecture lazily transfers arrays to streaming processors for analysis.

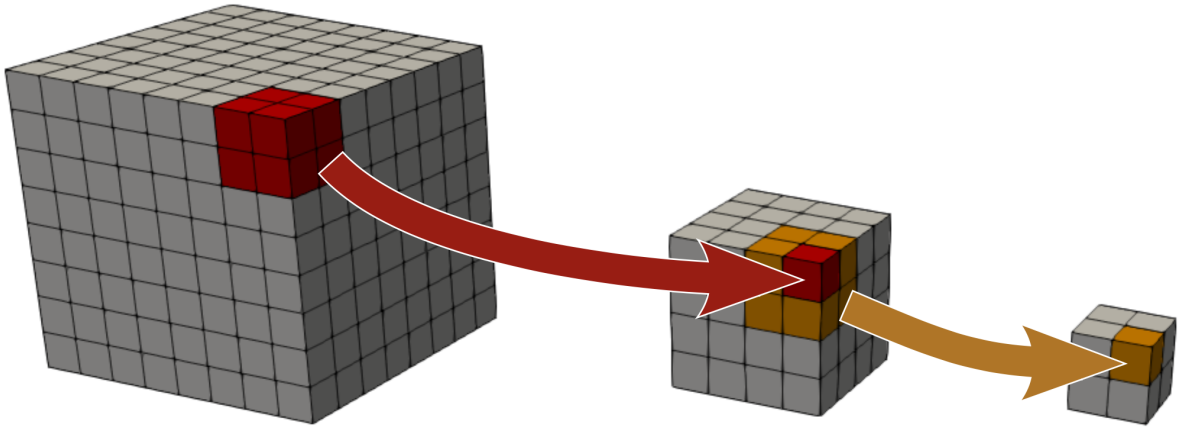
Summarization Examples

Performance Analysis, Design Considerations, and Applications of Extreme-scale In Situ Infrastructures

Utkarsh Ayachit, Andrew Bauer, Earl P. N. Duque, Greg Eisenhauer, Nicola Ferrier, Junmin Gu, Kenneth E. Jansen, Burlen Loring, Zarija Lukic , Suresh Menon, Dmitriy Morozov, Patrick O'Leary, Reetesh Ranjan, Michel Rasquin, Christopher P. Stone, Venkat Vishwanath, Gunther H. Weber, Brad Whitlock, Matthew Wolf, K. John Wu, and E. Wes Bethel

Full Text

Link to the full text PDF.



(a)

(b)

Fig. 4.11: Data Reduction - (a) By running a Haar wavelet filter multiple times, we can reduce the volume to a fixed size that is appropri- ate for rendering in a browser via ArcticViewer (depicted in Figure 5b). (b) Our wavelet-based reduction pipeline uses VTKm to compute multiple passes of the DHW transform. Circular dots indicate a zero-copy handoff of data. The tri- angle and diamond indicate where VTKm and VTK transfer data to/from the GPU, respectively.

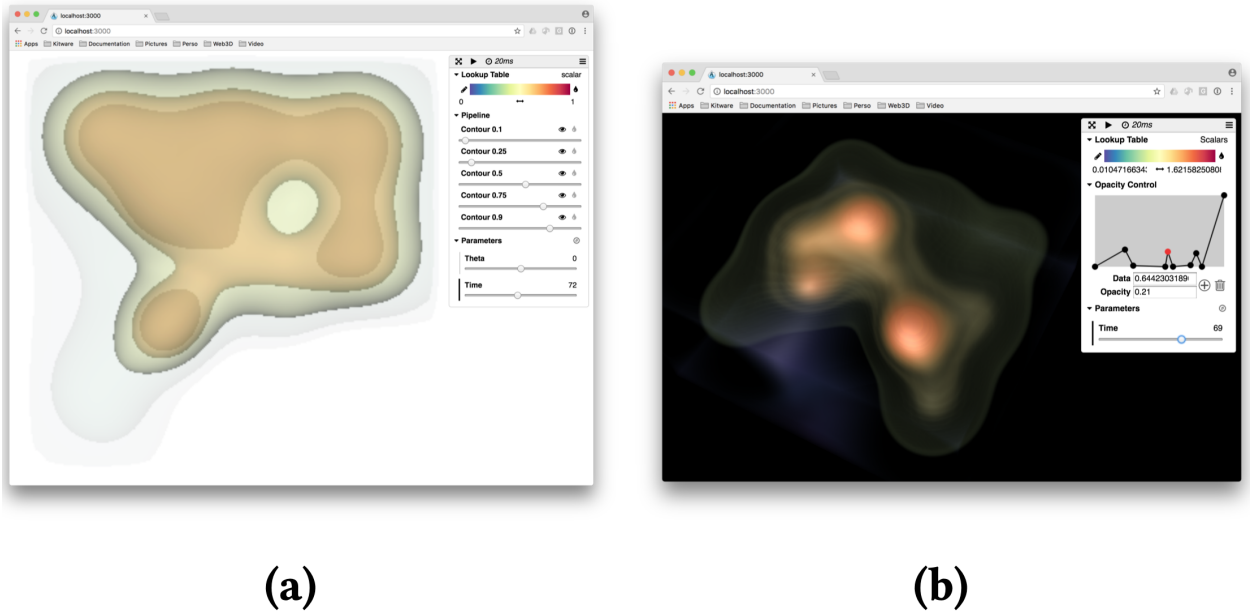


Fig. 4.12: Isocontours - Complex periodic system composed of a grid of nine big oscillators and four interleaved smaller ones with various frequencies. Visualizations via (a) translucent contours and (b) in-browser volume rendering.

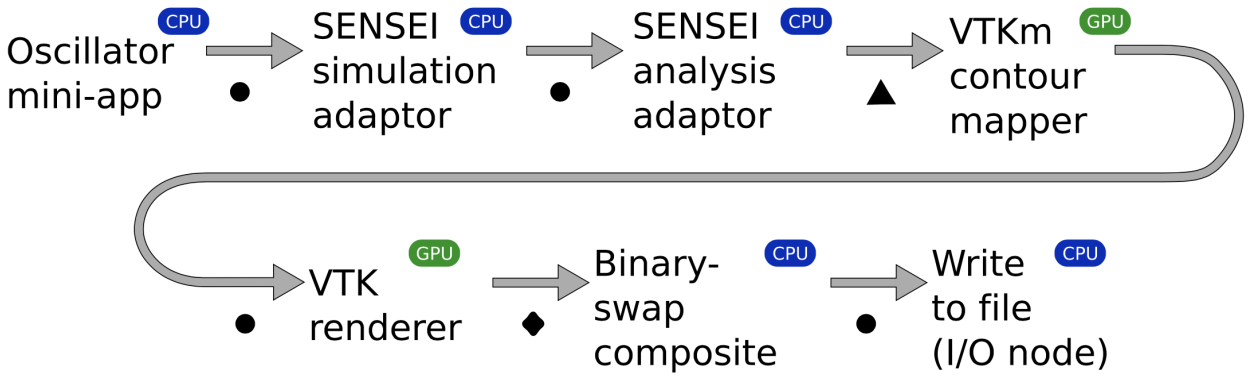


Fig. 4.13: Isocontours - Our visual summary rendering pipeline uses VTKm to compute renderings of isocontours. Circular dots indicate a zero-copy handoff of data. The triangle and dia- mond indicate where VTKm and VTK transfer data to/from the GPU, respectively.

Abstract

A key trend facing extreme-scale computational science is the widening gap between computational and I/O rates, and the challenge that follows is how to best gain insight from simulation data when it is increasingly impractical to save it to persistent storage for subsequent visual exploration and analysis. One approach to this challenge is centered around the idea of in situ processing, where visualization and analysis processing is performed while data is still resident in memory. This paper examines several key design and performance issues related to the idea of in situ processing at extreme scale on modern platforms: scalability, overhead, performance measurement and analysis, comparison and contrast with a traditional post hoc approach, and interfacing with simulation codes. We illustrate these principles in practice with studies, conducted on large-scale HPC platforms, that include a miniapplication and multiple science application codes, one of which demonstrates in situ methods in use at greater than 1M-way concurrency.

Mini-Application

As a prototypical data source, we implemented a miniapplication, an MPI code in C++, that simulates a collection of periodic, damped, or decaying oscillators. Placed on a grid, each oscillator is convolved with a Gaussian of a prescribed width. The oscillator parameters are specified as the input, which is read and broadcast from the root process. The user also specifies the time resolution, duration of the simulation, and the dimensions of the grid, partitioned between the processes using regular decomposition. The code iteratively fills the grid cells with the sum of the convolved oscillator values; the computation on each rank takes O(mN3) per time step, where m is the number of oscillators and N3 is the size of the subgrid on the rank. The computation is embarrassingly parallel; optionally, the ranks may synchronize after every time step, but this synchronization is off in the experiments below.

Pipelines

The miniapplication test configurations, listed below, show the various combinations of the oscillator miniapplication, in situ the ParaView/Catalyst, VisIt/Libsim, and ADIOS infrastructures, different in situ analysis methods, and with/without use of the SENSEI data interface.

Pipeline	Description
Original	miniapplication with no SENSEI interface and no I/O. In some test configurations, we do perform in situ analysis, but that coupling is done directly via subroutine call and does not use any in situ interface. The distinction of with vs. without analysis will be called out when needed in the subsections that follow.
Baseline	miniapplication with SENSEI interface enabled, but no in situ analysis or I/O. This configuration is useful in measuring the overhead of the SENSEI data interface in isolation from other processing.
Histogram	miniapplication with the SENSEI interface enabled, and connected directly to an in situ histogram calculation, but without any of the in situ infrastructures.
Autocorrelation	miniapplication with the SENSEI interface enabled, and connected directly to an in situ autocorrelation calculation, but without any of the in situ infrastructures.
Catalyst-slice	miniapplication with SENSEI interface enabled, and connected to Catalyst, which performs in situ rendering of a 2D slice from a 3D volume, then writes the image to disk.
Libsim-slice	miniapplication with SENSEI interface enabled, and connected to Libsim, which performs in situ rendering of a 2D slice from a 3D volume, then writes the image to disk.
ADIOS-FlexPath	miniapplication with SENSEI interface en- abled, and connected to the ADIOS FlexPath in situ infrastructure. Within this miniapplication/in situ infrastructure combination, we further refine the configuration in §§4.1.4 to include in situ workloads for histogram, autocorrelation, and Catalyst-slice.

Science Application Examples

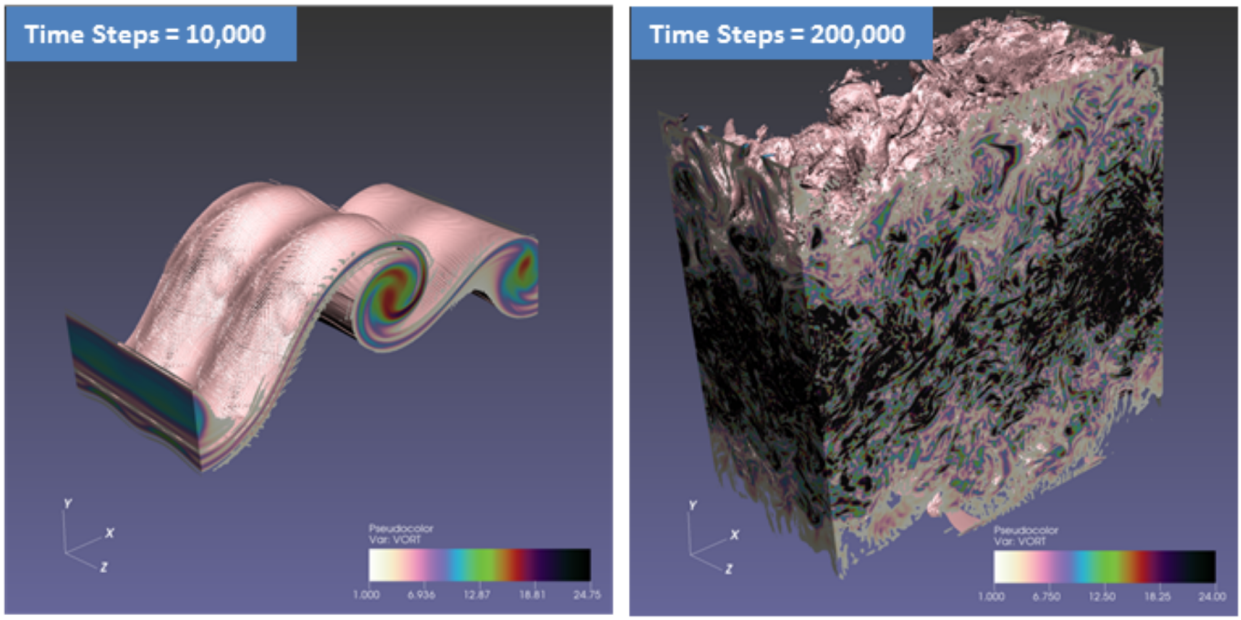


Fig. 4.14: Computational Fluid Dynamics - The Evolution of Temporal Mixing Layer from Initial to Vortex Breakdown using AVF-LESLIE.

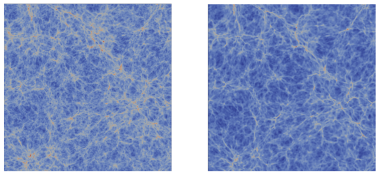


Fig. 4.15: Computational Cosmology - Time steps 200 and 300 of the 10243 Nyx Lyman α forest simulation. Simulations often only save every 100th time step. The difference between these time steps is considerable, hampering feature tracking.

Fig. 4.16: Computational Fluid Dynamics - 6.33 Billion element grid with 1,048,576 MPI ranks on 32,768 nodes (32 MPI ranks per node) with output size of 2900x725 and 30 time steps.

The SENSEI Generic In Situ Interface

Utkarsh Ayachit, Brad Whitlock, Matthew Wolf, Burlen Loring, Berk Geveci, David Lonie, and E. Wes Bethel

Full Text

Link to the full text PDF.

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.

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 and ParaView, 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.

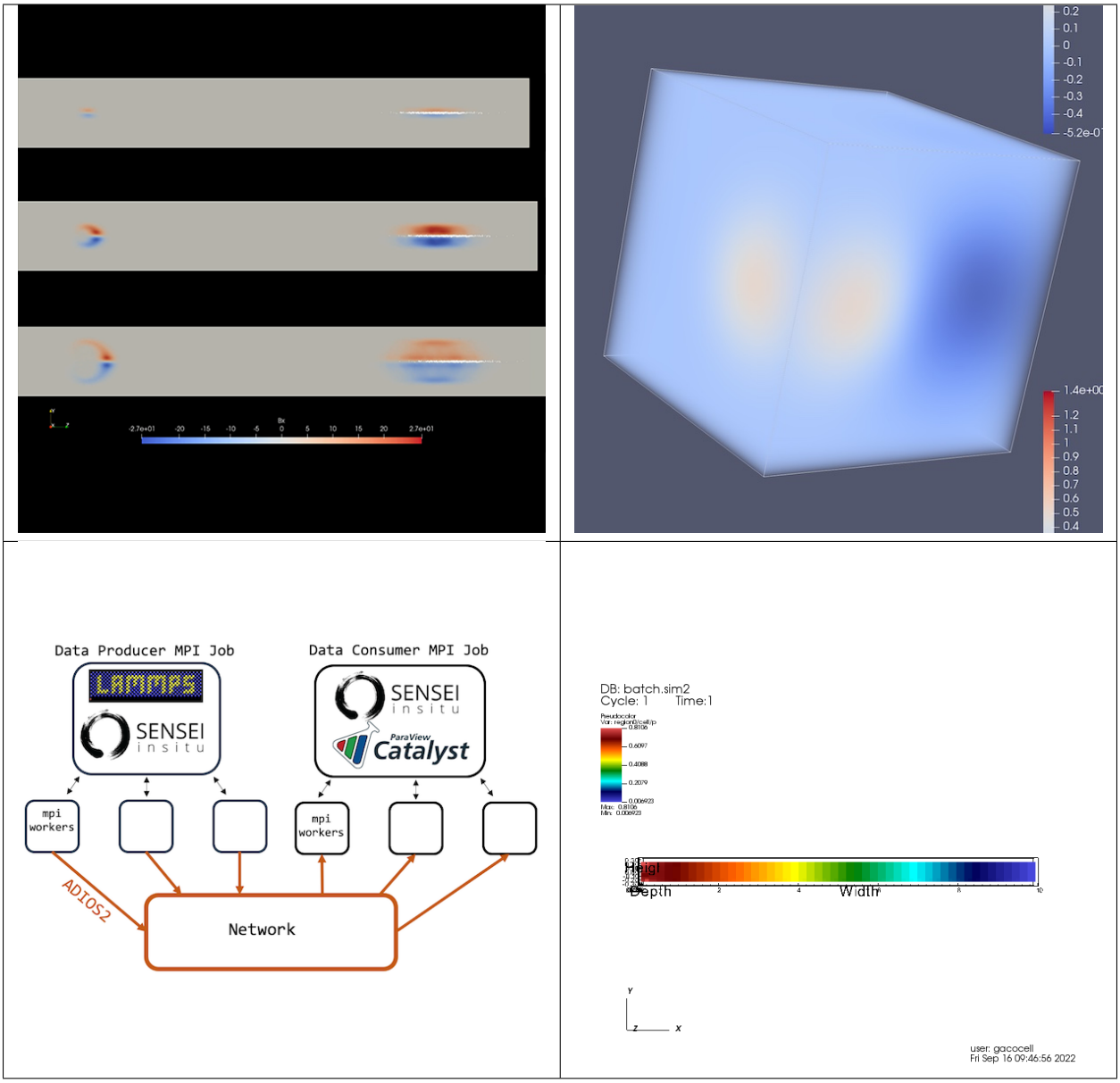
Despite its many strengths, there were some key additions we wanted for the SENSEI model. To minimize effort and memory overhead when mapping memory layouts for data arrays from applications to VTK, we extended the VTK data model to support arbitrary layouts for multicomponent arrays through a new API called generic arrays. 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 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.

4.5.2 Tutorials and Examples



SENSEI + AMReX

Synopsis

AMReX is a freely available simulation software framework designed to enable massively parallel block-structured adaptive mesh refinement (AMR) applications.

The AMReX Tutorials repository houses several tutorial examples demonstrating the use of SENSEI in AMReX based simulation codes. The full list of tutorials are documented here.

Overview

Two distinct tutorial examples are available with multiple configurations:

Name	Description
Advection_AmrCore	This tutorial illustrates an explicit SENSEI instrumentation of a code that makes use of *amrex::AmrMesh*.
Advection_AmrLevel	This tutorial illustrates 3 scenarios with a code that makes use of *amrex::Amr*. The first, *ImplcitAmr*, illustrates using SENSEI with the built-in instrumentation in *amrex::Amr*. The second, *ExplcitAmr*, illustrates using SENSEI with an explicit instrumentation. The third, *ExplicitParticlesAndAmr*, illustrates using SENSEI from a simulation that generates both particle and meshed based data.

Note that the *Advection_AmrLevel* contains code for 3 different scenarios. Which of these is active/available depends on how AMReX is compiled. See below for the details on configuring the build.

Setting Up

Compiling the AMReX SENSEI tutorials requires that SENSEI is previously installed. The options that SENSEI was built with determine the specific in situ capabilities available. Additional CMake options must also be passed when compiling AMReX to activate the SENSEI bridge and adaptors bundled with AMReX. The AMReX library is available on GitHub.

Build options

The options that AMReX is compiled with determine which SENSEI tutorials are available. The following table summarizes the various combinations and results.

CMake Options	What gets built
-DAMReX_SENSEI=ON -DAMReX_FORTRAN=ON -DSENSEI_DIR=<path to install>	Enables SENSEI features in AMReX. Required to compile SENSEI tutorials. Enables the AmrCore tutorial and AmrLevel implicit tutorial.
-DAMReX_SENSEI=ON -DAMReX_PARTICLES=ON -DAMReX_NO_SENSEI_AMR_INST=TRUE -DAMReX_FORTRAN=ON -DSENSEI_DIR=<path to install>	Enables the AmrCore tutorial, AmrLevel explicit tutorial, and particle based tutorials to be compiled.
-DAMReX_SENSEI=ON -DAMReX_NO_SENSEI_AMR_INST=TRUE -DAMReX_FORTRAN=ON -DSENSEI_DIR=<path to install>	Enables the AmrCore tutorial, AmrLevel explicit tutorial.

Running the Example

Once the tutorials are compiled they can be run from their corresponding directory. The executable is passed an AMReX parm-parse *inputs* file configuring the run. Options inside the *inputs* file configure the SENSEI instrumentation inside AMReX. Additionally SENSEI needs to configure the back-end that will process the data generated. This is done with a SENSEI XML file. Within each tutorial the *sensei* directory contains a number of SENSEI XML configuration files. The *inputs* file must be modified to point to one of these. Which one depends on how SENSEI was compiled. For instance the following snippet from an *inputs* file would configure SENSEI to send data to ParaView Catalyst,

```
sensei.enabled = 1                                  # turn SENSEI in situ on/off
sensei.config = sensei/render_iso_catalyst_3d.xml   # render simulation data with␣
↪ParaView Catalyst
sensei.frequency = 1                                # number of level 0 steps between␣
↪in situ processing
```

while the following snippet would configure SENSEI to send data to VisIt Libsim,

```
sensei.enabled = 1                                  # turn SENSEI in situ on/off
sensei.config = sensei/render_iso_libsim_3d.xml     # render simulation data with␣
↪ParaView Catalyst
sensei.frequency = 1                                # number of level 0 steps between␣
↪in situ processing
```

There are a number of XML files providing the configuration for a number of the available back-ends. A given SENSEI XML configuration is only valid when the SENSEI install has been compiled with the requisite back-end enabled.

Note that the *Advection_AmrLevel_ExplicitParticlesAndAmr* uses the file *inputs.tracers* while the others use the file *inputs*.

The tutorials are run by switching into the tutorial's build directory and issuing the launching command. For instance the *Advection_AmrLevel_ImplicitAmr* tutorial is launched by a command similar to:

```
mpiexec -np 4 ./Advection_AmrLevel_ImplicitAmr inputs
```

Results

Computational Steering with Catalyst and SENSEI

Synopsis

In this example, we demonstrate using SENSEI with a bidirectional Catalyst 1.0 Analysis Adaptor to perform computational steering. Here, we employ the oscillator miniapp. The list of oscillator objects are exposed to Catalyst where each oscillator can be manipulated. Position of oscillators as well as oscillator properties can be modified on the fly, changing the behavior of the simulated domain.

Setting Up

You will need ParaView 5.10 installed, and SENSEI compiled with the options *ENABLE_CATALYST* and *ENABLE_CATALYST_PYTHON* turned on.

Note, only Catalyst 1.0 scripts are compatible with computational steering in SENSEI. Unfortunately, those scripts cannot be generated automatically in the most recent editions of ParaView. The provided example script can be modified manually for your own purposes.

Running the Example

Open the ParaView application and navigate to *Tools*->'Manage Plugins'. Press the *Load New...* button, and navigate the file browser to the location of the *oscillator_catalyst_steering_proxies.xml* file, and select that file.

Next, navigate to *Catalyst*->'Connect' and accept incoming connections on port 22222. This will tell ParaView to begin listening for Catalyst connections on that port. If a different port number is desired, you will need to edit the

port number in the Catalyst python script *oscillator_catalyst_steering.py* to the desired port, and then start Catalyst in ParaView with the same desired port number.

In a terminal, navigate to your desired run directory (this testing directory is fine to run from), and start the oscillator miniapp with the SENSEI config xml *oscillator_catalyst_steering.xml*. Oscillator miniapp options can be set as desired, but a good starting point is:

```
$ mpirun -np 1 /path/to/oscillator -g 1 -t 0.01 -f oscillator_catalyst_steering.xml␣
↪simple.osc
```

With the Oscillator miniapp running, ParaView should automatically detect a new Catalyst connection and add several items to the catalsyt server list in the *Pipeline Browser*. Clicking the icon next to *mesh* and *oscillator* will display the data to the 3D viewport, updating as the miniapp progresses.

Click on the *Steering Parameters* item in the *Pipeline Browser*. The *Properties* panel will display several controls over each oscillator which can be modified, manipulating the oscillator parameters as the miniapp executes.

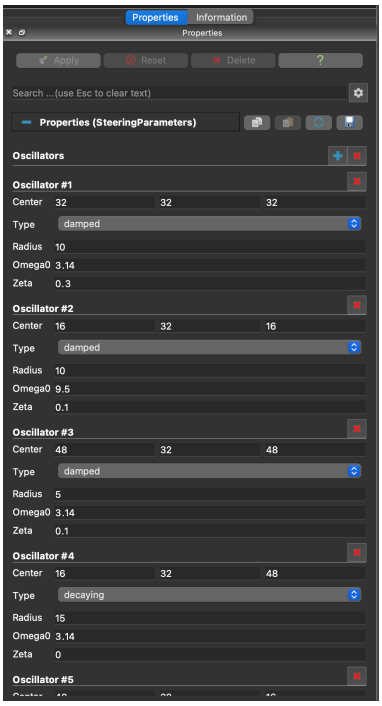


Fig. 4.17: The Properties panel provides parameters to add or delete oscillators in the domain, and change the parameters of the oscillators independently.

Results

The key takeaway from this example is that Catalyst and SENSEI can be used to perform computational steering tasks with in situ visualization. The oscillators, whose properties and locations can be modified in situ, respond to the user's modifications. Setting up such a computational steering workflow in your own simulation code requires exposing desired parameters to SENSEI, and writing XML instructions for ParaView to generate the GUI for modifying the parameters.

In Transit MxN communication with LAMMPS, SENSEI, and Paraview/Catalyst

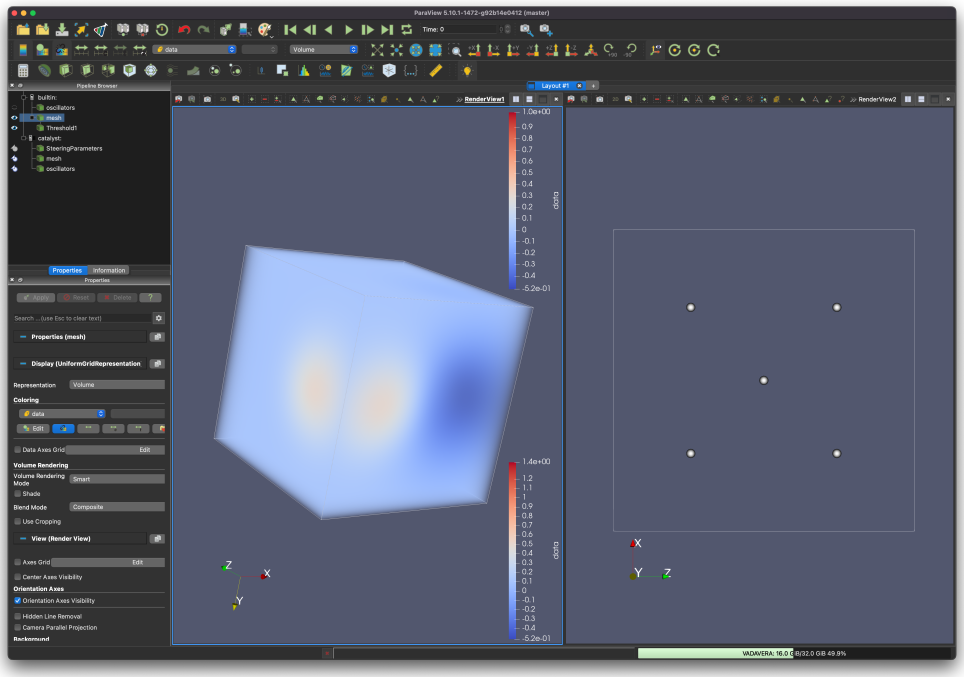


Fig. 4.18: ParaView's GUI contains the Properties panel, where oscillator parameters can be edited, a center 3D Viewport where the oscillators are rendered using ray-traced volume rendering, and a second 3D Viewport where the 5 existing oscillators' locations are visualized with respect to one another.

Synopsis

In this example we instrument the molecular dynamics simulation code LAMMPS with SENSEI and demonstrate in transit capabilities. Our example showcases M to N ranks redistribution and the Catalyst analysis adaptor to generate a Cinema database.

Setting Up

This example uses Docker containers. There are two containers: (1) Producer, which uses the LAMMPS molecular dynamics simulator instrumented with SENSEI; and (2) Consumer, which uses the SENSEI endpoint and the Paraview/Catalyst analysis adaptor.

A Zenodo artifact is available at https://zenodo.org/record/6336286 , containing Dockerfile recipes to build the Producer and Consumer. Please refer to file "containers.zip"

You should be able to build your own images with the Dockerfiles provided. If you would like to use prebuilt docker containers you can get them from the Docker hub:

```
docker pull srizzi/woiv22producer_runtime
docker pull srizzi/woiv22consumer_runtime
```

If your site uses Singularity you can build Singularity images pulling from the Docker hub

Running the Example

From the same Zenodo artifact, you can download "in_transit_demo_files.zip" , containing SENSEI xml configuration files and scripts to run the demo.

The LAMMPS producer is configured to run a simple simulation configured in file in_lj

With the parameters in in_lj the simulation evolves about 16 million hydrogen atoms. If you would like to change the size of the simulation, edit the multipliers for xx, yy, and zz in file in_lj. The multipliers are 16 in the file provided, which results in 67108864 atoms simulated.

The producer is launched with producer.sh . Notice that settings in this script are specific for ThetaGPU, but you should not find any major difficulties if you wish to adapt it for your system. The script also relies on a local build of mpich.

Notice that SENSEI uses the file adios_write_sst.xml to configure its backend. You will likely need to change the NetworkInterface in this xml file with an appropiate value for your own system.

The consumer side contains SENSEI with ParaView/Catalyst. For simplicity, in this demo we use the PosthocIO backend, which saves the received data in VTK format.

The consumer is launched with script consumer.sh and there are two xml configuration files required for SENSEI. The first defines the network transport and its called adios_transport_sst.xml . Once again, you may need to change the NetworkInterface parameter in this file according to your system. The second xml file, vtk_io.xml in this case, activates the PosthocIO analysis adaptor in SENSEI and specifies a directory to save the data.

These are intended to run on different machines and different ranks on producer (M) and consumer (N). The scripts provided will launch 16 MPI ranks on the producer and 4 MPI ranks on the consumer.

Results

The simulation is configure to run five timesteps. The SENSEI endpoint should receive data for each timestep and save it as VTK files.

SENSEI OpenFOAM Interface

Synopsis

The SENSEI OpenFOAM Interface integrates SENSEI into OpenFOAM. The interface is implemented as an OpenFOAM function object which subclasses the SENSEI VTKDataAdaptor. The interface was developed using Version 7 of OpenFOAM at openfoam.org and SENSEI 3.2.1. This interface may work with other OpenFOAM and SENSEI versions, but this has not been tested.

Setting Up

These instructions assume that SENSEI and OpenFOAM have already been built and installed.

To build the OpenFOAM function object that implements the SENSEI interface, edit the file functionObjects/sensei/Allwmake and specify paths to SENSEI and to VTK by setting the following variables:

```
SENSEI_DIR="SENSEI installation path"
VTK_INC="path to VTK include files"
VTK_LIB="path to VTK libraries"
```

Note that you must point the the same VTK version used by SENSEI.

These paths will be used to set the compile and link flags used by the build as follows. These typically will not need to be modified:

```
export SENSEI_CXXFLAGS="-O2 -I$SENSEI_DIR/include -I$VTK_INC"
export SENSEI_CXXLIBS="-L$SENSEI_DIR/lib -lsensei -lsenseiCore -lpugixml"
export SENSEI_CXXLIBS="$SENSEI_CXXLIBS -Wl,-rpath=$VTK_LIB"
```

To build the interface, you set up your OpenFOAM environment and execute the Allwmake script:

```
cd functionObjects/sensei
source "OpenFOAM installation path"/etc/bashrc
./Allwmake
```

Running Allwmake builds the interface and installs it into the users OpenFOAM library directory.

Running the Example

Copy the pipeCyclic tutorial from the OpenFOAM distribution to use as an example. You can find it here:

```
"OpenFOAM installation path"/tutorials/incompressible/simpleFoam/pipeCyclic
```

Edit the system/controlDict file in the tutorial and add a senseiFunctionObject block into the functions section. This will load the interface during the solver run and export the requested fields. For example:

```
functions
{
    senseiFunctionObject
    {
        type        senseiFunctionObject;
        libs        ("libSenseiAdaptor.so");

        enabled     true;

        name        simpleFoam;
        desc        "simpleFoam pipeCyclic tutorial SENSEI export";
        fields          ( U p k epsilon );
    }
}
```

Next, create a file called constants/sensei_config.xml to control what SENSEI will export. Here is an example sensei_config.xml file that outputs PNG images of a slice using Libsim"

```
<sensei>
    <analysis type="libsim"
        plots="Pseudocolor"
        plotvars="region0/cell/p"
        slice-origin="0,0,0.25" slice-normal="1,1,1"
        image-filename="openfoam_slice_%ts"
        image-width="800" image-height="600"
        image-format="png"
        frequency="1"
        enabled="1" />
</sensei>
```

Then run the tutorial using the Allrun script:

```
./Allrun
```

Results

Here are two of the images produced running the tutorial with the above sensei_config.xml file:

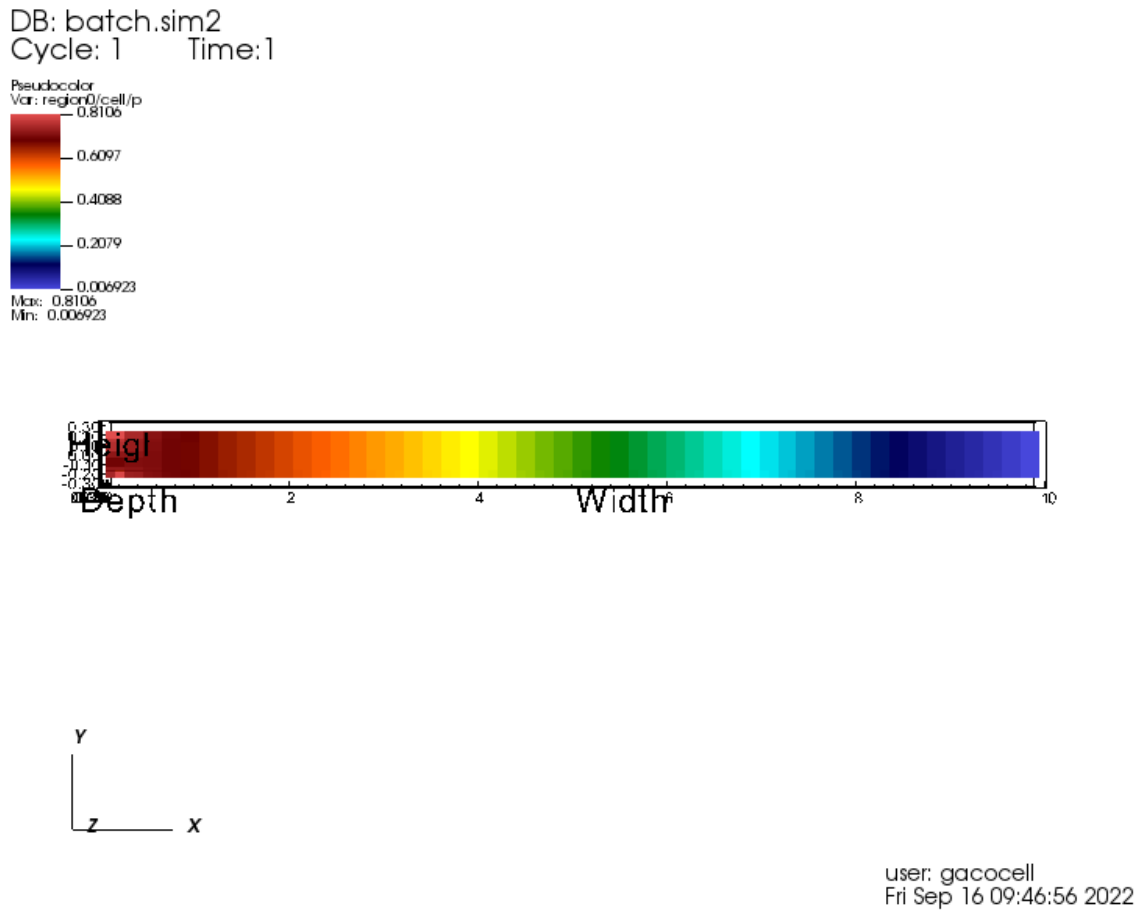


Fig. 4.19: Z = 0.25 slice at time step 1

4.6 Developer Guidelines

4.6.1 Git workflow

When working on a feature for a future release make a pull request targeting the develop branch. Only features to be included in the next release should be merged. A code review should be made prior to merge. New features should always be accompanied with read the docs documentation and at least one regression test.

We use the branching model described here: https://nvie.com/posts/a-successful-git-branching-model/ and detailed below in case the above link goes away in the future.

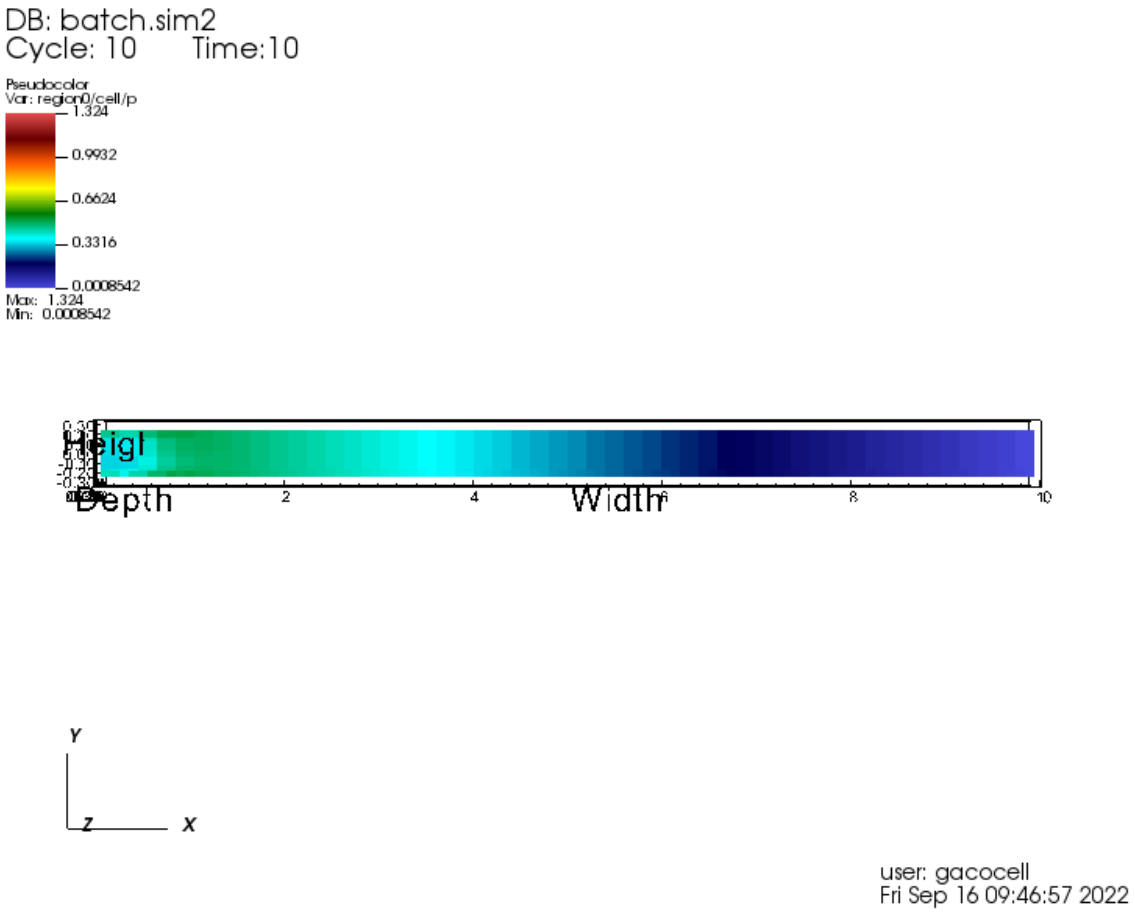


Fig. 4.20: Z = 0.25 slice at time step 10

The main branches

The central repo holds two main branches with an infinite lifetime:

- master
- develop

The master branch at origin should be familiar to every Git user. Parallel to the master branch, another branch exists called develop.

We consider origin/master to be the main branch where the source code of HEAD always reflects a production-ready state.

We consider origin/develop to be the main branch where the source code of HEAD always reflects a state with the latest delivered development changes for the next release. Some would call this the "integration branch".

When the source code in the develop branch reaches a stable point and is ready to be released, all of the changes should be merged back into master somehow and then tagged with a release number. How this is done in detail will be discussed further on.

Therefore, each time when changes are merged back into master, this is a new production release by definition. We tend to be very strict at this, so that theoretically, we could use a Git hook script to automatically build and roll-out our software to our production servers every time there was a commit on master.

Feature branches

May branch off from: develop Must merge back into: develop Branch naming convention: anything except master, develop, or release-*

Feature branches (or sometimes called topic branches) are used to develop new features for the upcoming or a distant future release. When starting development of a feature, the target release in which this feature will be incorporated may well be unknown at that point. The essence of a feature branch is that it exists as long as the feature is in development, but will eventually be merged back into develop (to definitely add the new feature to the upcoming release) or discarded (in case of a disappointing experiment).

Creating a feature branch

When starting work on a new feature, branch off from the develop branch.

```
$ git checkout -b myfeature develop
```

Incorporating a finished feature on develop

Finished features may be merged into the develop branch to definitely add them to the upcoming release:

```
$ git checkout develop
$ git merge --no-ff myfeature
$ git branch -d myfeature
$ git push origin develop
```

The –no-ff flag causes the merge to always create a new commit object, even if the merge could be performed with a fast-forward. This avoids losing information about the historical existence of a feature branch and groups together all commits that together added the feature.

Release branches

May branch off from: develop Must merge back into: develop and master Branch naming convention: release-*

Release branches support preparation of a new production release. They allow for last-minute dotting of i's and crossing t's. Furthermore, they allow for minor bug fixes and preparing meta-data for a release (version number, build dates, etc.). By doing all of this work on a release branch, the develop branch is cleared to receive features for the next big release.

The key moment to branch off a new release branch from develop is when develop (almost) reflects the desired state of the new release. At least all features that are targeted for the release-to-be-built must be merged in to develop at this point in time. All features targeted at future releases may not—they must wait until after the release branch is branched off.

It is exactly at the start of a release branch that the upcoming release gets assigned a version number—not any earlier. Up until that moment, the develop branch reflected changes for the "next release", but it is unclear whether that "next release" will eventually become 0.3 or 1.0, until the release branch is started. That decision is made on the start of the release branch and is carried out by the project's rules on version number bumping.

Creating a release branch

Release branches are created from the develop branch. For example, say version 1.1.5 is the current production release and we have a big release coming up. The state of develop is ready for the "next release" and we have decided that this will become version 1.2 (rather than 1.1.6 or 2.0). So we branch off and give the release branch a name reflecting the new version number:

```
$ git checkout -b release-1.2 develop
$ ./bump-version.sh 1.2
$ git commit -a -m "Bumped version number to 1.2"
```

After creating a new branch and switching to it, we bump the version number. Here, bump-version.sh is a fictional shell script that changes some files in the working copy to reflect the new version. (This can of course be a manual change—the point being that some files change.) Then, the bumped version number is committed.

This new branch may exist there for a while, until the release may be rolled out definitely. During that time, bug fixes may be applied in this branch (rather than on the develop branch). Adding large new features here is strictly prohibited. They must be merged into develop, and therefore, wait for the next big release.

Finishing a release branch

When the state of the release branch is ready to become a real release, some actions need to be carried out. First, the release branch is merged into master (since every commit on master is a new release by definition, remember). Next, that commit on master must be tagged for easy future reference to this historical version. Finally, the changes made on the release branch need to be merged back into develop, so that future releases also contain these bug fixes.

```
$ git checkout master
$ git merge --no-ff release-1.2
$ git tag -a 1.2
```

The release is now done, and tagged for future reference.

4.6.2 Code style

Here are some of the guidelines

- use 2 spaces for indentation, no tabs or trailing white space
- use *CamelCase* for names
- variable names should be descriptive, with in reason
- class member variables, methods, and namespace functions start with an upper case character, free functions start with a lower case
- use the this pointer to access class member variables and methods
- generally operators should be separated from operands by a single white space
- for loop and conditional braces are indented 2 spaces, and the contained code is written at the same indentation level as the braces
- functions and class braces are not indented, but contained code is indented 2 spaces.
- a comment containing one space and 77 - precede class method definitions
- pointers and reference markers should be preceded by a space.
- generally wrap code at 80 chars
- treat warnings as errors, compile with -Wall and clean all warnings
- avoid 1 line conditionals

there are surely other details to this style, but I think that is the crux of it.

and a snippet to illustrate:

```
// a free function
void fooBar()
{
  printf("blah");
}

// a namespace
namespace myNs
{
// with a function
void Foo()
{
  pritf("Foo");
}

}

// a class
class Foo
{
public:
  // CamelCase methods and members
  void SetBar(int bar);
  int GetBar();

  // pointer and reference arguments
  void GetBar(int &bar);
  void GetBar(int *bar);

private:
  int Bar;
```

(continues on next page)

(continued from previous page)

```
};

// ---------------------------------------------------------------------------
void Foo::SetBar(int bar)
{
  // a member function
  this->Bar = bar;
}

// ---------------------------------------------------------------------------
int Foo::GetBar()
{
  return this->Bar;
}

int main(int argc, char **argv)
{
  // a conditional
  if (strcmp("foo", argv[1]) == 0)
    {
    foo();
    }
  else
    {
    bar();
    }
  return 0;
}
```

4.6.3 Regressions tests

New classes should be submitted with a regression test.

4.6.4 User guide code style

Please use this style https://documentation-style-guide-sphinx.readthedocs.io/en/latest/style-guide.html

4.7 Glossary

List of all terms we defined