SAND2020-8456B

Leveraging Production Visualization Tools In Situ

Kenneth Moreland, Andrew C. Bauer, Berk Geveci, Patrick O'Leary, and Brad Whitlock

Abstract The visualization community has invested decades of research and development into producing large-scale production visualization tools. Although in situ is a paradigm shift for large-scale visualization, much of the same algorithms and operations apply regardless of whether the visualization is run post hoc or in situ. Thus, there is a great benefit to taking the large-scale code originally designed for post hoc use and leveraging it for use in situ.

This chapter describes two in situ libraries, Libsim and Catalyst, that are based on mature visualization tools, VisIt and ParaView, respectively. Because they are based on fully-featured visualization packages, they each provide a wealth of features. For each of these systems we outline how the simulation and visualization software are coupled, what the runtime behavior and communication between these components are, and how the underlying implementation works. We also provide use cases demonstrating the systems in action. Both of these in situ libraries, as well as the underlying products they are based on, are made freely available as open-source products. The overviews in this chapter provide a toehold to the practical application of in situ visualization.

Kenneth Moreland
Sandia National Laboratories, Albuquerque, NM, USA, e-mail: kmorel@sandia.gov

Andrew C. Bauer
United States Army Corps of Engineers, e-mail: andrew.c.bauer9.civ@mail.mil

Berk Geveci
Kitware, Inc., Clifton Park, NY, USA, e-mail: berk.geveci@kitware.com

Patrick O'Leary
Kitware, Inc., Clifton Park, NY, USA, e-mail: patrick.oleary@kitware.com

Brad Whitlock
Intelligent Light, Rutherford, NJ, USA, e-mail: bjw@ilight.com

1 Introduction

Although in situ is a paradigm shift for large-scale visualization, much of the same algorithms and operations apply regardless of whether the visualization is run post hoc or in situ. Thus, there is a great benefit to taking the large-scale code originally designed for post hoc use and leveraging it for use in situ. Two of the most popular post hoc visualization tools are VisIt [9] and ParaView [2]. Contributing to the success of these tools is that they each are feature rich, have proven parallel scalability, have automated scripting capabilities, are free, and have a large development community. To leverage these capabilities for an in situ environment, each tool now provides a library that allows data and control to pass from another software tool. VisIt provides a library named Libsim [32], and ParaView provides a library named Catalyst [4]. In this chapter we review these libraries and demonstrate how they are used to implement in situ visualization.

The introduction of this book lists many important features of in situ visualization that motivate the implementation and use of Libsim and Catalyst. However, the introduction also lists several limitations of in situ visualization that do not apply to post hoc visualization. The upshot is that for the foreseeable future both in situ and post hoc visualization will be important for discovery at large computing scales, and so providing both types of visualization are important. Because Libsim and Catalyst each derive functionality from their respective classic tools, they immediately make available both in situ and post hoc visualization. Furthermore, visualizations made post hoc are easily made in situ and vice versa.

We present Libsim and Catalyst together in this chapter because there are many common features the two libraries share. The two systems share the same in situ taxonomy described in the introduction.

Integration Type Both tools are general purpose and designed to work well with a variety of simulation codes. However, their primary function is specific to visualization and the simulation must be modified to use the library.

Proximity Libsim and Catalyst assume they are running in close proximity using the same resources as the simulation.

Access Because Libsim and Catalyst are libraries that share the same memory space as the simulation, it is possible for these codes to directly access the simulation's memory. However, they only access memory specifically given to them, and the data must be in a specified format.

Division of Execution Libsim and Catalyst use time division to alternate use of the simulation's resources.

Operation Controls The main mode of operation is to perform visualizations according to a predefined batch script. However, both Libsim and Catalyst are capable of performing human-in-the-loop visualization by attaching a remote GUI to a running simulation.

Output Type Libsim and Catalyst are each capable of producing a wide variety of outputs. Images and image databases [3] are common outputs, but derived geometric structures and statistics are also possible data products.

In addition to having similar properties, Libsim and Catalyst share similar methods to interface with simulations, to specify what visualization operations to perform, and to instantiate the visualization operation. Both Libsim and Catalyst are interfaced to a simulation by writing an "adapter." The adapter is primarily responsible for converting the data representation used by the simulation to the data representation used by Libsim and Catalyst. Both Libsim and Catalyst use VTK [25] as their underlying implementation, and thus the adapter for either must convert the simulation's data format to VTK's data format. VTK can reference data in external arrays, so often the adaption of a simulation's data structures to VTK's data structures can be done without copying the data.

Also similar among the two libraries is their runtime behavior. Each allows the simulation to operate in its own execution loop. At the simulation's discretion, it periodically invokes Libsim or Catalyst with an updated collection of data. Under typical batch operation, the library processes the data, saves whatever visualization product is generated, and returns control back to the simulation. Both libraries also support a mode in which a live, remote GUI is updated. In this mode control can either be immediately returned to the simulation, or the simulation may be blocked while a remote user interactively explores the data, which is particularly useful for debugging the simulation.

The following two sections provide details for Libsim and Catalyst. Each section describes how the respective library is integrated with a simulation, how the library behaves at runtime, and the underlying implementation of the library. Because of their similarity there is redundancy in these descriptions. For clarity, we have repeated descriptions in each section to provide a thorough explanation of each.

2 Libsim

Libsim [33] is a library that enables in situ visualization using VisIt [9], a massively parallel visualization and data analysis tool built on VTK. VisIt contains a rich set of data readers, operators, and plots. These features read, filter or transform data, and ultimately provide a visual representation of the data to allow for exploration and analysis. Many of these features can be chained together to build pipelines that create sophisticated visualizations. Libsim satisfies multiple use cases, shown in Figure 1. Libsim was conceived originally as an online visualization mechanism for debugging simulations with the aid of the VisIt GUI. Over time, Libsim evolved to allow both interactive and batch uses cases that allows it to generate a host of data products without a user in the loop. Today, virtually anything that is possible in the VisIt GUI is also possible from Libsim. This flexibility has enabled Libsim to be integrated into diverse simulations related to fields of study such as Computational Fluid Dynamics (CFD) or Cosmology. Libsim is highly scalable and has been run at levels of concurrency surpassing 130K cores.

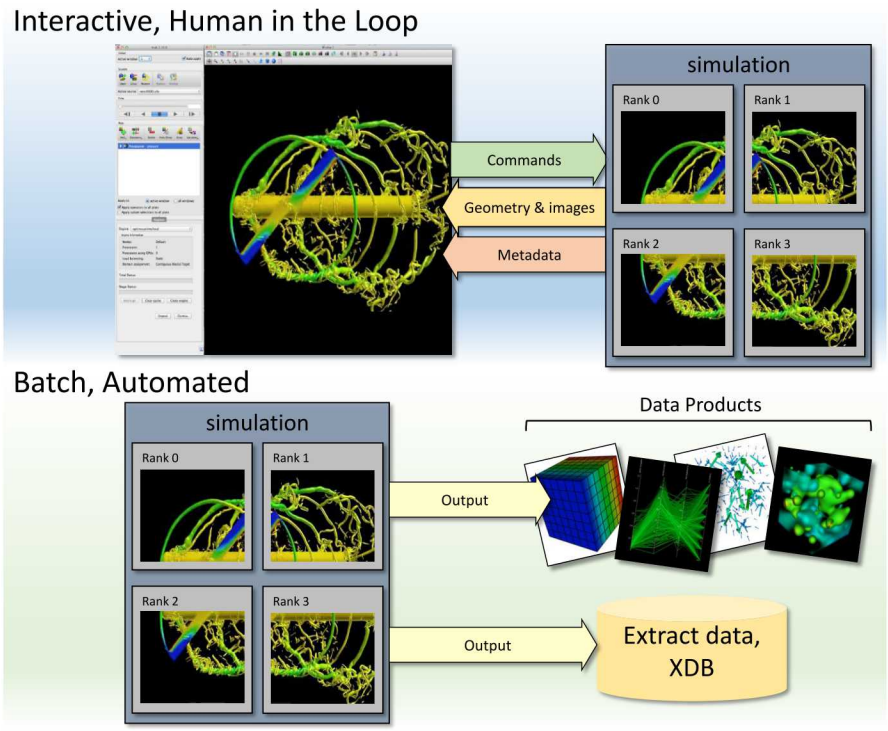

Fig. 1 Libsim supports interactive and batch use cases.

2.1 Integration with Simulation

Libsim integrates with applications as a set of library calls that are usually encapsulated into a module called a data adaptor (depicted in Figure 2). Libsim provides C, FORTRAN, and Python bindings to minimize the amount of cross-language programming that is asked of application scientists. Libsim provides a relatively low-level application programming interface (API) so it can be integrated flexibly into host simulations. Libsim can be used directly, or it can be used within other infrastructures that integrate into the simulation, such SENSEI [5] or Damaris [10]. The typical procedure for instrumenting a simulation with Libsim involves writing a data adaptor and proceeding through four stages: initialization, exposing data, iteration, and adding user interface. During initialization, the simulation sets up the relevant environment and calls functions to either prepare for interactive connections or for batch operations. Writing the data adaptor involves exposing simulation data to Libsim. The next stages are optional. Iteration involves adding code that will produce any plots or data extracts. The final stage adds a user interface and registers simulation functions to respond to user-interaction via the VisIt GUI.

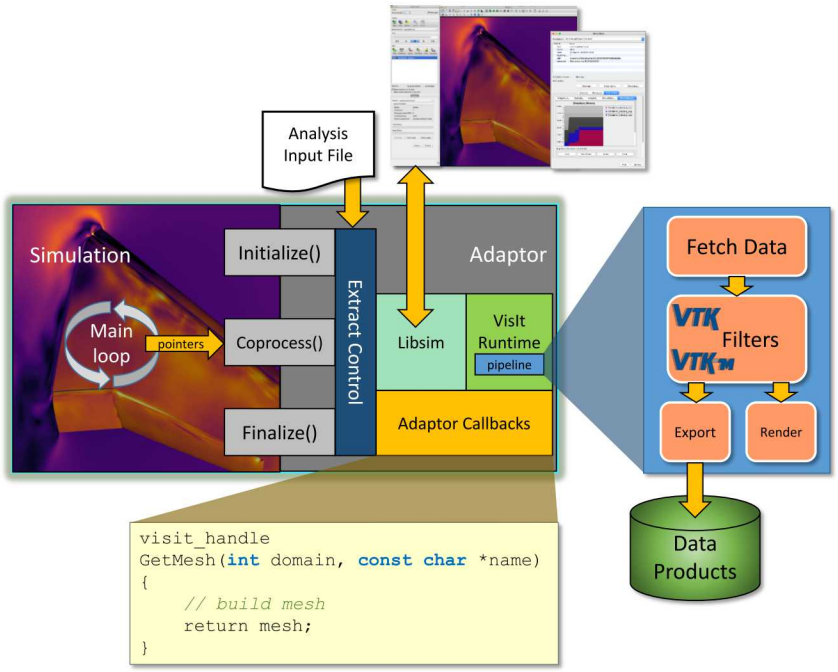

Fig. 2 Simulations instrumented with Libsim link to the Libsim library. The simulation supplies data adaptor functions that expose data to VisIt pipelines. The VisIt pipelines can supply a running VisIt instance with data or produce in situ data products.

The Libsim API can be thought of as having 2 components: a control interface and a data interface. The control interface is responsible for setting up environment, event handling, and registering data callback functions. The data interface is responsible for annotating simulation memory and packaging related data arrays into mesh data structures that can be used as inputs to VisIt. Data arrays are passed by pointer, allowing both zero-copy access to simulation data and transfer of array ownership to VisIt so data can be freed when no longer needed. Arrays can be contiguous in memory as in structure of array (SOA) data layouts or they can use combinations of strides and offsets to access simulation data as in array of structures (AOS) data structures. Libsim supports commonly used mesh types including rectilinear, curvilinear, Adaptive Mesh Refinement (AMR), and unstructured grids consisting of finite element cell types. Libsim also can also support computational domains that are not actually meshes such as Constructive Solid Geometry (CSG). However data are represented, Libsim usually relies on the simulation's data decomposition when exposing data to VisIt, and the simulation can expose multiple meshes with their own domain decompositions. Libsim permits simulations to add field data on the mesh centered on the cells or on the points. Field data consists of scalars, vectors, tensors, labels, and arrays with an arbitrary number of tuples per element. Libsim includes

additional data model concepts, allowing simulations to specify domain adjacency, ghost data, mixed material cells, and material species.

During the instrumentation process, a decision must be made whether to support interactive connections or batch operations via Libsim, or both. The paths differ somewhat, though in both cases there are some upfront calls that can be made to set up the environment for Libsim. This consists of VisIt's environment and the parallel environment. When interactive connections are expected, Libsim will write a small *.sim2* file containing network connection information that VisIt can use to initiate a connection to the simulation. This file is not needed for batch-only operation. The following code example includes the Libsim header files, sets up Libsim for parallel operation, discovers environment settings needed to load VisIt runtime libraries, and finally creates the .sim2 file needed for interactive connections.

```
#include <VisItControlInterface_V2.h>
#include <VisItDataInterface_V2.h>

/* Broadcast callbacks */
static int
bcast_int(int *value, int sender, void *cbdata)
{
    return MPI_Bcast(value, 1, MPI_INT, sender, MPI_COMM_WORLD);
}
static int
bcast_string(char *str, int len, int sender, void *cbdata)
{
    return MPI_Bcast(str, len, MPI_CHAR, sender, MPI_COMM_WORLD);
}
void libsim_initialize(int interactive)
{
    /* Parallel setup */
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    VisItSetBroadcastIntFunction2(bcast_int, NULL);
    VisItSetBroadcastStringFunction2(bcast_string, NULL);
    VisItSetParallel(size > 1);
    VisItSetParallelRank(rank);
    /* Get VisIt environment */
    char *env = NULL;
    if(rank == 0)
        env = VisItGetEnvironment();
    VisItSetupEnvironment2(env);
    if(env != NULL)
        free(env);
    if(rank == 0 && interactive)
    {
        /* Write out .sim2 file that VisIt uses to connect. */
        VisItInitializeSocketAndDumpSimFile(
            "simulation_name"
            "Comment about the simulation",
            "/path/to/where/sim/was/started",
            NULL, NULL, "simulation_name.sim2");
```

```
    }
}
```

When integrating Libsim for interactive operation, calls to the control interface to handle events must be inserted into the simulation. Libsim provides the `VisItDetectInput()` function for this purpose. It listens for connections from a VisIt client. Simulations can build event loops using the VisItDetectInput function or call it in a polling manner from their own event loops. When a connection request is detected, the function will return a value indicating that other Libsim functions must be called to complete the connection request and load the runtime library. Once the runtime library is loaded, the developer may register data callback functions that expose simulation data as Libsim objects. Data callback functions are called by VisIt's runtime library to inquire about simulation metadata and when specific meshes and fields are needed in order to create a specific data product. Data callbacks must be installed once the VisIt runtime library has been loaded. In a batch-style integration, this can be immediately after the call to *VisItInitializeRuntime()* whereas for interactive, the data callbacks must be installed after a successful call to *VisItAttemptToCompleteConnection()*, which signifies a successful connection of VisIt's viewer application to the simulation.

```
static void
libsim_bcast_cmd_cb(int *command, void *cbdata)
{
    MPI_Bcast(command, 1, MPI_INT, 0, MPI_COMM_WORLD);
}
static void
libsim_control_cb(
    const char *cmd, const char *args, void *cbdata)
{
    /* Optional: Respond to text commands */
}

/* MetaData and Mesh callbacks shown later...*/

static void libsim_setup_callbacks(void)
{
    void *cbdata = /* Point this at application data */;
    VisItSetCommandCallback(libsim_cmd_cb, cbdata);
    VisItSetSlaveProcessCallback2(libsim_bcast_cmd_cb, cbdata);
    VisItSetGetMetaData(libsim_metadata_cb, cbdata);
    VisItSetGetMesh(libsim_mesh_cb, cbdata);
}

/* Simplified example - invoked by the simulation. */
void libsim_interactive(void)
{
    switch(VisItDetectInput(blocking, -1))
    {
    case 0:
        /* No input from VisIt, return control to sim. */
        break;
```

```
    case 1:
        /* VisIt is trying to connect to sim. */
        if(VisItAttemptToCompleteConnection() == VISIT_OKAY)
            libsim_setup_callbacks();
        break;
    case 2:
        /* VisIt wants to tell the engine something. */
        if(!VisItProcessEngineCommand())
            VisItDisconnect();
        break;
    }
}
```

The deferred nature of Libsim data requests ensures that the simulation does not have to waste time computing results that might not be used, as when computing derived fields for visualization. Data requests are assembled inside of the VisIt runtime libraries from its execution contract, which includes a manifest of all of the data needed to create a visualization. Libsim's callback function design enables the VisIt runtime library to request data on demand from the Libsim adaptor in the simulation. Data are requested in stages, first metdata is obtained to inform the VisIt runtime about the meshes and variables provided by the simulation. Simulations can expose as little data or as much data as they like. The callback functions include a user-defined data argument that allows application data to be associated with callbacks when they are registered in order to make it easier to access application data from callbacks when they are invoked by the VisIt runtime library.

```
visit_handle
libsim_metadata_cb(void *cbdata)
{
    visit_handle md = VISIT_INVALID_HANDLE,
                 mmd = VISIT_INVALID_HANDLE;
    /* Create metadata. */
    if(VisIt_SimulationMetaData_alloc(&md) == VISIT_OKAY)
    {
        /* Access application data */
        application_data *app = (application_data *)cbdata;

        /* Set the simulation state. */
        VisIt_SimulationMetaData_setMode(md,
                                         VISIT_SIMMODE_RUNNING);
        VisIt_SimulationMetaData_setCycleTime(md, app->cycle,
                                              app->time);

        /* Add mesh metadata. */
        if(VisIt_MeshMetaData_alloc(&mmd) == VISIT_OKAY)
        {
            /* Set the mesh's properties.*/
            VisIt_MeshMetaData_setName(mmd, "mesh");
            VisIt_MeshMetaData_setMeshType(
                mmd, VISIT_MESHTYPE_RECTILINEAR);
            VisIt_MeshMetaData_setTopologicalDimension(mmd, 3);
```

```
            VisIt_MeshMetaData_setSpatialDimension(mmd, 3);
            VisIt_MeshMetaData_setNumDomains(
                mmd, app->total_num_domains);
            VisIt_SimulationMetaData_addMesh(md, mmd);
        }

        /* We could expose more meshes, variables, etc. */
    }
    return md;
}
```

Once the data requirements are determined for a visualization, Libsim invokes the registered mesh callback to obtain mesh data on a per-domain basis. Libsim is flexible and it can represent several mesh types. Meshes, as with most Libsim data constructs, are constructed from arrays. Libsim provides functions that enable simulation data arrays to be annotated with size, type, offset, and stride information so arrays can be passed back to VisIt to be used zero-copy as much as possible. In addition, simulation callback functions can wrap temporary memory that VisIt is allowed to free in case zero-copy representations are not feasible. Once the mesh callback has been executed, variables and then other types of data are requested, each from their respective callback function. As a simulation adaptor grows more complete, additional callbacks can be registered to support variables, materials, AMR nesting, mesh decompositions, etc.

```
visit_handle
libsim_mesh_cb(int domain, const char *name, void *cbdata)
{
    visit_handle h = VISIT_INVALID_HANDLE;
    if(VisIt_RectilinearMesh_alloc(&h) != VISIT_ERROR)
    {
        visit_handle hx, hy, hz;
        /* Access application data */
        application_data *app = (application_data *)cbdata;
        VisIt_VariableData_alloc(&hx);
        VisIt_VariableData_alloc(&hy);
        VisIt_VariableData_alloc(&hz);
        VisIt_VariableData_setDataD(hx, VISIT_OWNER_SIM, 1,
                                    app->dims[0], app->xc);
        VisIt_VariableData_setDataD(hy, VISIT_OWNER_SIM, 1,
                                    app->dims[1], app->yc);
        VisIt_VariableData_setDataD(hz, VISIT_OWNER_SIM, 1,
                                    app->dims[2], app->zc);
        VisIt_RectilinearMesh_setCoordsXYZ(h, hx, hy, hz);
    }
    return h;
}
```

With simulations able to produce an ever increasing amount of data, Libsim's emphasis gradually shifted from being a tool for debugging simulation codes towards production of data products without massive amounts of I/O. To generate data products, the simulation can call Libsim functions to set up VisIt plots and VisIt

operators and to set their attributes before saving images or exporting processed datasets. These operations can also be set up via a VisIt session file rather than relying on fixed sets of plots. This allows the user to connect using VisIt interactively to set up the desired visualization, save the configuration to a session file, and then apply the recipe in batch to produce movies and other data products.

```
/* Save plots designated by a session file. */
VisItRestoreSession("setup.session");
VisItSaveWindow("a0000.png", 1024, 1024, VISIT_IMAGEFORMAT_PNG);

/* Set up some plots directly */
VisItAddPlot("Mesh", "mesh");
VisItAddPlot("Pseudocolor", "pressure");
VisItDrawPlots();
VisItSaveWindow("a0001.png", 1048, 1024, VISIT_IMAGEFORMAT_PNG);
```

Libsim has been used increasingly with CFD codes with common needs for producing lightweight surface-based data extracts that can be explored using desktop visualization tools. Surface data extracts often consist of slices, isosurfaces, or boundary surfaces plus field data. Generating such extracts in situ results in a drastic reduction in saved data and time needed compared to extracting such data from bulk volume data during post-processing. To permit general surface extracts to be specified via an external configuration file and simplify multiple aspects of instrumenting codes using Libsim (particularly for parallel event loops), we have created a companion library called "extract control". Extract control enables multiple extract types (e.g. surfaces, images, or Cinema databases [3]) to be requested via a convenient YAML file that the user can change, as opposed to direct Libsim function calls or using VisIt session files. The extract control library also encapsulates some of the usual boilerplate code needed to support interactive event loops as well as batch-style Libsim integrations, resulting in fewer lines of code.

Interactive instrumentation using Libsim allows the VisIt GUI to use the simulation as a normal compute engine, making it possible to do most kinds of analysis or data interrogation with large file-based datasets. Interactive simulations benefit from other features provided by VisIt and Libsim. For instance, Libsim provides functions that let the simulation provide sample data that can be aggregated into strip charts that plot quantities of interest over time. Strip charts can display arbitrary sample data, though time and memory measurements are commonly plotted. The VisIt GUI displays strip charts and other simulation state in the Simulation window. The Simulation window also exposes controls published by the simulation. These controls take the form of command buttons in the simplest case that, when pressed, can invoke callback functions in the simulation adaptor. This allows the user to initiate actions in the simulation based on button clicks in the VisIt GUI. The VisIt GUI also allows for simulation-specific custom user interfaces. Custom user interfaces are designed using Qt Designer and the VisIt GUI can replicate such user interfaces as extensions within the Simulation window. Custom user interfaces enable the VisIt GUI to alter simulation parameters to affect more complicated steering actions. This feature

was successfully used by Sanderson et al [24] to create a customized simulation dashboard for the Uintah software suite.

2.2 Runtime Behavior

Libsim accepts control from the simulation and then enters an event loop or other batch-oriented code in the simulation adaptor to generate data extracts and immediately return. When Libsim's operations complete, control is returned to the simulation. Libsim's runtime behavior depends on how it was used to instrument the simulation, and its behavior varies between human-in-the-loop-blocking to nobody in the loop, non-blocking. The behavior for interactive use cases is determined by how the `VisItDetectInput()` function was called when instrumenting the simulation. The function can be used to implement blocking event loops or polling event loops that are invoked periodically from the simulation. Blocking calls return when commands have been received by the VisIt GUI and may result in additional calls that request user input. Blocking calls may also include a timeout that enables the function to return after a specified period of inactivity to return control to the simulation. Libsim includes other functions that can be called in conjunction with the event loop to notify VisIt's runtime library of new simulation data so it can be used to push data to the VisIt GUI. This feature allows the VisIt GUI to connect to the running simulation and recompute its plots in response to updates from the simulation so the user can watch the simulation evolve. Connecting to the running simulation, making plots, watching for a while, and then disconnecting is supported in simulations that use Libsim and this cycle can be repeated over the life of the simulation.

2.3 Underlying Implementation

VisIt functionality is divided into different processes, according to function. VisIt provides client programs such as the GUI so users can analyze data interactively. VisIt's viewer acts as a central hub, which manages state, communication with other programs, and rendering data. VisIt's compute "engine" reads data and executes any plot and operator pipelines to generate geometry or image data for the viewer. The compute engine can run locally or on other HPC systems via client/server mode. Libsim enables a simulation to act as a proxy for the VisIt compute engine. Libsim is actually separated into a front-end library and a runtime library. The front-end library is minimal and is linked to the simulation. The front-end library provides all of the user-facing functions such as event handling while providing an interface to runtime library functions that are loaded later once Libsim is actually told to do work. This separation allows simulations to link with Libsim once but dynamically change the version of VisIt used at runtime. An important function of the front end library is

to write a *sim2* file, which is a file containing networking information that the VisIt GUI can use to initiate a socket connection to the simulation. Since VisIt relies on the *sim2* file for connecting to simulations, from a user's point of view, accessing a running simulation is essentially the same as accessing any file-based dataset. Upon opening the *sim2* file, VisIt initiates a socket connection to the simulation and once a successful connection is made via the `VisItDetectInput()` function, the simulation dynamically loads the Libsim runtime library and calls additional data adaptor code to register data-producing callback functions with Libsim.

The Libsim runtime library incorporates parts of the VisIt engine and viewer so it can manage plots and execute visualization pipelines to deliver VisIt functionality. When plots are created, VisIt instantiates a visualization pipeline consisting of various data analysis filters. The pipeline uses a contract mechanism to build a list of data that is needed to produce the desired visualization. The list of data in the contract is used to request simulation data via a specialized database plug-in. Libsim's database plug-in invokes callback functions from the simulation data adaptor to obtain data rather than reading from files. A callback function is responsible for returning different types of data such as a mesh domain or a specific variable on that mesh domain. Data are returned by making Libsim function calls that package simulation data into Libsim objects. The Libsim objects ferry data from the simulation into the Libsim database reader where they are unpacked and used to assemble VTK datasets that VisIt can use internally in its visualization pipelines.

2.4 Use Case

Libsim has been used in a variety of domains and applications. Libsim automatically produces batch data extracts (such as geometry, rendered images, or Cinema databases) at scale and lets VisIt connect to running simulations for interactive exploration, monitoring, and steering. In the CFD domain, Libsim has been used to generate in situ data extracts consisting of reduced sets of geometry suitable for post hoc data analysis of engineering datasets [11, 14]. Forsythe et al [13] successfully used this approach in CREATE-AV Kestrel to generate geometric extracts to accelerate analysis of fully coupled high-fidelity simulations of a rotorcraft in a ship's airwake. Helicopters landing on sea-based landing platforms experience turbulent airflows resulting from their surroundings such as the ship airwake produced from air moving around the structures on naval ships. Airwakes produces nonlinear aerodynamic effects that must be taken into account in order to accurately simulate landing in such a setting, as in this flight simulator coupling. Libsim was integrated into Kestrel to produce geometric data extracts in FieldView XDB format representing 45 seconds in 2700 time steps, taken every 5 simulation time steps. The resulting extract-based simulation data are drastically smaller than a corresponding volume dataset, and the system *"rendered the animations in hours rather than the days it would have otherwise taken [23]."*

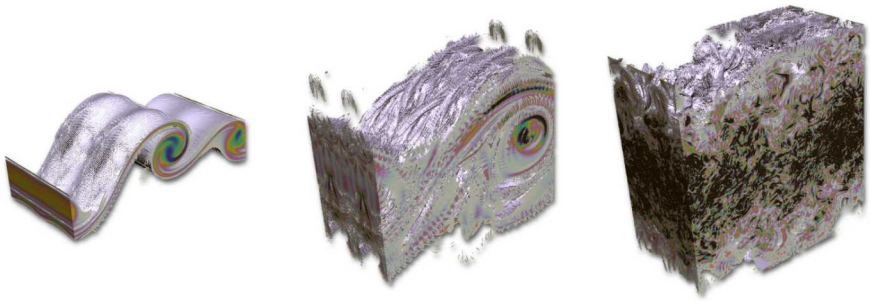

Fig. 3 The Evolution of Temporal Mixing Layer from Initial to Vortex Breakdown.

In a larger computation, Libsim was used as an in situ analysis infrastructure coupled to the AVF-LESLIE [28, 29] combustion code by SENSEI. AVF-LESLIE was configured to simulate unsteady dynamics of a temporally evolving planar mixing layer at the interface between two fluids. This interface results in a type of fundamental flow that mimics the dynamics encountered when two fluid layers slide past one another and is found in atmospheric and ocean fluid dynamics as well as combustion and chemical processing. Visualizations of the flowfield in Figure 3 show isosurfaces of the vorticity field, at 10,000, 100,000, and 200,000 time steps where the flow evolves from the initial flow field, vortex braids begin to form, wrap and then the flow breaks down leading to homogeneous turbulence, respectively.

AVF-LESLIE was statically linked to Libsim and VisIt and run on Titan at Oak Ridge Leadership Class Computing Facility on 131,072 cores. Static linking was selected because of an observation that Libsim's usual deferred loading of the VisIt runtime library incurred significant overhead when running at large scale on Titan. A SENSEI data adaptor was created for AVF-LESLIE. It passed structured mesh and field data from the main FORTRAN-based simulation through a C-language compatibility layer where data pointers were used to create VTK datasets that were passed into SENSEI. VTK datasets were exposed to Libsim inside SENSEI via an additional data adaptor that ultimately passed data to the VisIt runtime library to create data products. Two types of in situ computations were performed: a rendering workflow, and an extract-based workflow. The rendering workflow generated 1600×1600 pixel images of a vorticity isosurface and composited partial images into a single PNG image using tree-based compositing within VisIt's runtime library. The vorticity quantity was computed on demand in the SENSEI adaptor for AVF-LESLIE. The extract workflow saved the same isosurface to FieldView XDB files, aggregating geometry to smaller subgroups of 96 ranks to reduce file system contention. A typical extract from this dataset was approximately 200 times smaller than the full volume data, and this enabled the project to save data 20 times more frequently while remaining still 10 times smaller than when using volume data.

Libsim continues to provide in situ capabilities for frameworks and codes that need to run at large scale and want to leverage the capabilities in a fully-featured visualization tool such as VisIt.

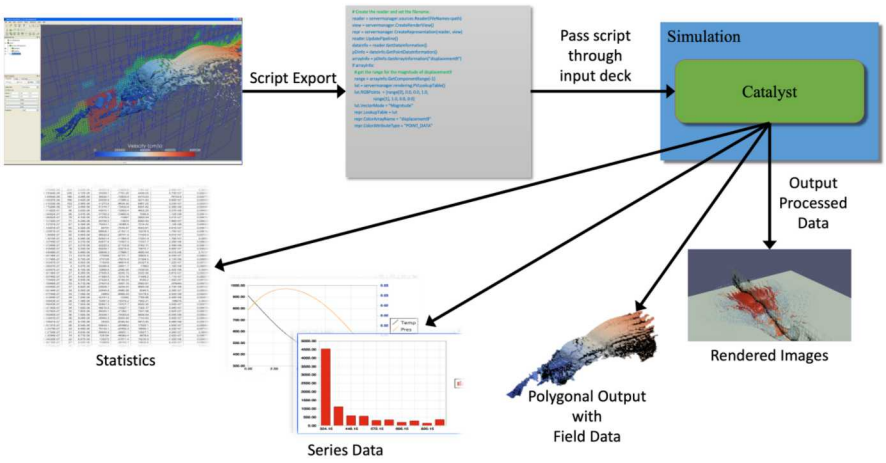

Fig. 4 *In situ* workflow with a variety of Catalyst outputs.

3 Catalyst

The ParaView Catalyst library is a system that addresses challenges of in situ visualization and is designed to be easily integrated directly into large-scale simulation codes. Built on and designed to interoperate with the standard visualization toolkit VTK and scalable ParaView application, it enables simulations to perform analysis intelligently, generate relevant output data, and visualize results concurrent with a running simulation. The ability to concurrently visualize and analyze data from simulations is synonymous with *in situ* processing, co-processing, co-analysis, concurrent visualization, and co-visualization. Thus ParaView Catalyst, or Catalyst, is often referred to as a co-processing, or *in situ*, library for high-performance computing (HPC).

Figure 4 demonstrates a typical workflow using Catalyst for *in situ* processing. In this figure, we assume a simulation code is integrated with Catalyst. The end-user initiates the workflow by creating a Python script using the ParaView application graphical user interface (GUI), which specifies the desired output from the simulation. Next, when the simulation starts, it loads the Python script; then, during execution, Catalyst generates synchronously (i.e. while the simulation is running) any analysis and visualization output. Catalyst can produce images (i.e. screenshots) and image databases [3], compute statistical quantities, generate plots, and extract derived information such as polygonal data, such as iso-surfaces, to visualize.

A variety of simulation codes have used Catalyst. A subset list of these codes instrumented to use Catalyst include PHASTA from the University of Colorado, Boulder [22]; MPAS-Atmosphere and MPAS-Ocean from the climate modeling group at Los Alamos National Laboratory (LANL) and the National Center for Atmospheric Research (NCAR) [31]; XRAGE, NPIC, and VPIC from LANL [20]; HPCMP CREATE-AV^{TM} Helios from the U.S. Army's CCDC AvMC Technology

Fig. 5 ParaView Catalyst interface architecture.

Development Directorate; CTH, Albany and the Sierra simulation framework from Sandia National Laboratories [18]; H3D from the University of California, San Diego (UCSD), and Code Saturne from Électricité de France (EDF) [16].

The most significant scale run to date used over 1 million MPI processes on Argonne National Laboratory's BlueGene/Q Mira machine [5]. The scaling studies utilized PHASTA, a highly scalable CFD code, developed by Kenneth Jansen at the University of Colorado, Boulder, for simulating active flow control on complex wing design.

3.1 Integration with Simulation

In this section, we describe how developers can interface a simulation code with the ParaView Catalyst libraries. The interface to the simulation code is called an *adaptor*. Its primary function is to adapt the information in internal data structures of the simulation code and transform these data structures into forms that Catalyst can process. We depict this process in Figure 5.

A developer creating an adaptor needs to know the simulation code data structures, the VTK data model, and the Catalyst application programming interface (API).

3.1.1 Simulation Codebase Footprint

Although interfacing Catalyst with a simulation code may require significant effort, the impact on the codebase is minimal. In most situations, the simulation code only calls three functions.

First, we must *initialize* Catalyst in order to place the environment in the proper state. For codes that depend on the message-passing interface (MPI), we place this method after the MPI_Init() call.

```
MPI_Init(argc, argv);
#ifdef CATALYST
CatalystInitialize(argc, argv);
#endif
```

Next, we call the *coprocess* method to check on any computations that Catalyst may need to perform. This call needs to provide the simulation mesh and field data structures to the adaptor as well as time and time step information. It may also

provide additional control information, but that is not required. Typically, we call the *coprocess* method at the end of every time step in the simulation code after updating the fields and possibly the mesh.

```
for (int timeStep=0; timeStep < numberOfTimeSteps; timeStep++) {
  // < simulation does its thing >
  // < update fields and possibly mesh after timeStep >
#ifdef CATALYST
  CatalystCoProcess(timeStep, time, <grid info>, <field info>);
#endif
}
```

Finally, we must *finalize* Catalyst state and adequately clean up. For codes that depend on MPI, we place this method before the MPI_Finalize() call.

```
#ifdef CATALYST
CatalystFinalize();
#endif
MPI_Finalize();
```

In general, we colocate the *initialize* and the *finalize* methods with the *coprocess* method in the adaptor and the developer implements the adaptor code in a separate source file, which simplifies the simulation code build process.

3.1.2 Instrumentation Details

As shown in Figure 5, the adaptor code is responsible for the interface between the simulation code and Catalyst.

Core to the adaptor is the *vtkCPProcessor* class, which manages the *in situ* analysis and visualization pipelines, which in turn automate the flow of data through a series of tasks. Given

```
vtkCPProcessor* Processor = NULL; // static data
```

we can define an example *initialize* method, *CatalystInitialize*, as

```
void CatalystInitialize(int numScripts, char* scripts[]) {
  if (Processor == NULL) {
    Processor = vtkCPProcessor::New();
    Processor->Initialize();
  }
  // scripts are passed in as command line arguments
  for (int i=0; i<numScripts; i++) {
    vtkCPPythonScriptPipeline* pipeline =
                               vtkCPPythonScriptPipeline::New();
    pipeline->Initialize(scripts[i]);
    Processor->AddPipeline(pipeline);
    pipeline->Delete();
  }
}
```

In this way we provide pipeline scripts passed in as command-line arguments to an instantiation of *vtkCPProcessor* to manage. For our example, the *finalize* method, *CatalystFinalize*, simply deletes the storage for any defined pipelines.

```
void CatalystFinalize() {
  if (Processor) {
    Processor->Delete();
    Processor = NULL;
  }
}
```

Besides being responsible for initializing and finalizing Catalyst, the other responsibilities of the adaptor are:

- Determining whether or not to perform co-processing.
- Mapping the simulation fields and mesh to VTK data objects for co-processing.

For this example, we specify the mesh as a uniform, rectilinear grid defined by the number of points in each direction and the uniform spacing between points. There is only one field associated with this mesh, which is called temperature and defined over the points (vertices or nodes) of the mesh. Thus, the *coprocess* method, *CatalystCoProcess*, performs the following commonly required tasks:

```
void CatalystCoProcess(
    int timeStep, double time, unsigned int numPoints[3],
    unsigned int numGlobalPoints[3], double spacing[3],
    double* field) {
  vtkCPDataDescription* dataDescription =
                                  vtkCPDataDescription::New();
  dataDescription->AddInput("input");
  // 1. Specify the current time and time step for Catalyst.
  dataDescription->SetTimeData(time, timeStep);
  // 2. Check whether Catalyst has anything to do at this time.
  if (Processor->RequestDataDescription(dataDescription) != 0) {
    // 3. Create the mapped VTK mesh.
    vtkImageData* grid = vtkImageData::New();
    grid->SetExtents(
        0, numPoints[0]-1, 0, numPoints[1]-1, 0, numPoints[2]-1);
    // 4. Identify the VTK mesh for Catalyst to use.
    dataDescription->GetInputDescriptionByName("input")->
                                                  SetGrid(grid);
    dataDescription->GetInputDescriptionByName("input")->
                        SetWholeExtent(0, numGlobalPoints[0]-1,
                                       0, numGlobalPoints[1]-1,
                                       0, numGlobalPoints[2]-1);
    grid->Delete();
    // 5. Associate mapped VTK fields with the mapped VTK mesh.
    vtkDoubleArray* array = vtkDoubleArray::New();
    array->SetName("temperature");
    array->SetArray(field, grid->GetNumberOfPoints(), 1);
    grid->GetPointData()->AddArray(array);
    array->Delete();
    // 6. Call CoProcess to execute pipelines.
    Processor->CoProcess(dataDescription);
```

```
  }
  dataDescription->Delete();
}
```

In Section 3.3 we'll discuss the details of the API to help solidify the understanding of the flow of information.

3.2 Runtime Behavior

The analysis and visualization methods can be implemented in C++ or Python and can run *in situ*, in transit, or a hybrid of the two methods. Python scripts can be crafted from scratch or using the ParaView GUI to set up prototypes and export as Catalyst scripts interactively.

We designed Catalyst to run synchronously (tightly coupled) with the simulation supporting *in situ* workflows, where we execute analysis methods and visualization pipelines alongside the simulation, leveraging the same address space.

Catalyst can support in transit workflows using two sub-groups of a global MPI communicator: one for simulation processes and one for analysis and visualization processes. However, the data movement from the simulation processes is not automatic and requires the writing of an additional communication routine during instrumentation.

Much more commonly, Catalyst enables hybrid workflows using either VTK's I/O capabilities or by leveraging additional middleware such as ADIOS [5]. For example, analysis methods and visualization pipelines could send intermediate results to burst buffers, and ParaView or another application would pull data from the burst buffers for interaction and further analysis.

Also, Catalyst can connect to a separately running ParaView Live session for exploring results on the fly. The Live method can facilitate a Monitoring/Steering workflow. This capability, in turn, enables subtly unique steering workflows, where the analysis methods and visualization pipelines are modified interactively through user feedback.

Finally, we aligned synchronous and asynchronous communication patterns with specific Catalyst workflows. Live supports both, and communications can be changed, as described above with hybrid workflows, utilizing third-party software.

3.3 Underlying Implementation

The core of our implementation is how the adaptor passes information back and forth between the simulation code and Catalyst. We need to exchange three types of information: VTK data objects, pipelines, and control information. The VTK data objects are the information containing the input to the pipelines. The pipelines specify what operations to perform on the data and how to output the results. The

control information specifies when each pipeline should execute, and the required information from the VTK data objects needed to execute the pipelines properly.

Before providing the details of the API, we want to describe the flow of information and its purpose. This information affords a higher level of understanding of how the pieces work together.

First, we initialize Catalyst, which sets up the environment and establishes the pipelines to execute. Next, we execute the pipelines as required. Finally, we finalize Catalyst.

The initialize and finalize steps are reasonably straightforward, but the intermediate step has a lot happening in the underlying implementation. Principally, the intermediate step queries the pipelines to see if any of the pipelines require processing. If not, then control returns immediately to the simulation code. This query is nearly instantaneous, where the expectation of many calls wastes negligible compute cycles. On the other hand, if one or more pipelines demand re-execution, then the adaptor needs to update the VTK data objects representing the mesh and fields from the simulation, and then execute the desired pipelines with Catalyst. The execution time can vary widely depending on the quantity and type of tasks. Once the re-executing pipelines finish, then control returns to the simulation code.

The main classes of interest for the Catalyst API are vtkCPProcessor, vtkCPPipeline, vtkCPDataDescription, vtkCPInputDataDescription, and the derived classes that are specialized for Python. When Catalyst is built with Python support, all of these classes are Python wrapped as well.

vtkCPProcessor is responsible for managing the pipelines. This management includes storing them, querying them, and executing them. Note that the AddPipeline method fundamentally adds a pipeline (vtkCPPipeline or vtkCPPythonScriptPipeline) for execution at requested times. This class mimics the structure of the simulation instrumentation.

First, the Initialize method initializes the object and sets up Catalyst. The initialization method uses either MPI_COMM_WORLD or an API supplied MPI communicator. Note that the Initialize method can depend on vtkMPICommunicatorOpaqueComm, defined in vtkMPI.h, and is used to avoid directly having to include the mpi.h header file. Next, the CoProcess method executes the proper pipelines based on information in the required argument description. When applying this method, we update and add the description to the vtkDataObject representing the mesh and fields. We use the helper method, RequestDataDescription, to determine, for a given description, if we desire execution of any pipelines. For this method, the description argument should have the current time and time step set and the identifier for available inputs. Finally, the Finalize method releases all resources used by Catalyst. If a Catalyst Python script includes a Finalize method, we execute this method at this point.

The vtkCPDataDescription class stores information passed between the adaptor and the pipelines. The provided information comes from either the adaptor for the pipeline or the pipeline for the adaptor. The adaptor needs to provide the pipelines with the current time, the current time step, and the names for input meshes produced by the simulation. For most use cases, the adaptor will provide a single input mesh to

Catalyst called input. Naming the inputs is needed for situations where the adaptor provides multiple input meshes with each mesh treated independently.

The vtkCPInputDataDescription class is similar to vtkCPDataDescription in that it passes information between the adaptor and the pipelines. The difference is that vtkCPInputDataDescription passes information about the meshes and fields.

Finally, there are a variety of other methods to increase the efficiency of the adaptor. For example, to streamline data preparation for coprocessing, other methods may inform the adaptor of the requested fields for the pipelines.

3.4 HPCMP CREATE-AV™ Helios Use Case

The U.S. Department of Defense's High-Performance Computing Modernization Program's (HPCMP) Computational Research and Engineering Acquisition Tools and Environments for Air Vehicles (CREATE-AV) project has overseen the development of a rotorcraft simulation tool called Helios [26, 27, 34], a high-fidelity, multi-disciplinary computational analysis platform for rotorcraft aeromechanics simulations. Used by academia, government, and industry, Helios handles the aerodynamics solutions using a dual-mesh paradigm: body-fitting meshes in the near-body region and adaptive mesh refinement (AMR) meshes in the off-body region.

The Helios package contains tools for a near-complete workflow, except geometry creation and post-processing tools. These tools include specifying rotor blade geometry and movement, mesh assembly and partitioning, a graphical user interface (GUI) for defining simulation input parameters, the parallel simulation environment, and management of simulation results. The default computational fluid dynamics (CFD) libraries include SAMCart for CFD solution in the off-body region and a choice of using kCFD and mStrand for the near-body region. Additionally, FUN3D and OVERFLOW can be used as plugins CFD solvers for the near-body region.

3.4.1 Specialized Workflow for Rotorcraft Analysis

Helios handles specialized complex high-fidelity simulations of coupled fluid-structure interaction for a variety of flight conditions, including rotorcraft flying in their turbulent wake. Likewise, the Helios development team tailored the *in situ* processing.

The first Helios release that used ParaView Catalyst for *in situ* capabilities was version 3, which only included streamlines, slices, and contours and required the end-user to hand-edit Python input files. December 2019 marked the latest Helios release, version 10. Since version 3, the developers enhanced the *in situ* operations with particle paths, Cartesian extracts, taps, and derived variable calculations, defined through Helios's pre-processing GUI, shown in Figure 6. Helios can utilize custom Catalyst scripts created with the ParaView GUI, but end-users seem to prefer using the pre-processing GUI due to the specialized nature of rotorcraft analysis.

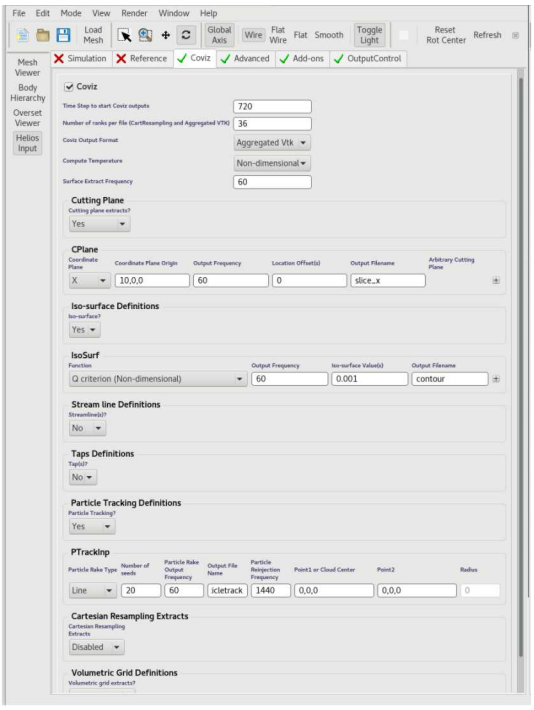

Fig. 6 Example set up of the *in situ* panel of Helios's pre-processing GUI.

The widely varying post-processing experience with specific tools by Helios end-users dictated the use of data extracts for the *in situ* outputs. Thus, enabling the end-user to manage their regular post-processing workflow with familiar tools.

3.4.2 Specialized Catalyst Edition

ParaView is a large software project with a variety of functionalities not required for the batch *in situ* processing done with Helios. Since Helios does not generate *in situ* images, all rendering components can be excluded from the Helios specific Catalyst *in situ* library. In addition, we can remove most data readers and writers except the readers for *in situ* restart and the writers for extracts (no requirement for input/output libraries like HDF5 or NetCDF). Customizing ParaView Catalyst specific to Helios provides the following benefits:

- Reduction in the number of source code files and associated compile time.
- Decrease in the number of third-party library dependencies and simplifying the build and install process.
- Reduction in the Catalyst library size and faster shared library load times due to smaller library size and a smaller number of linked libraries [7].

Table 1 Helios memory reporting on node 1 on a Cray XC40/50.

In Situ Build Type	Memory Usage (MB)
None	24,450
Standard ParaView Catalyst	29,798
Helios Catalyst Edition	26,958

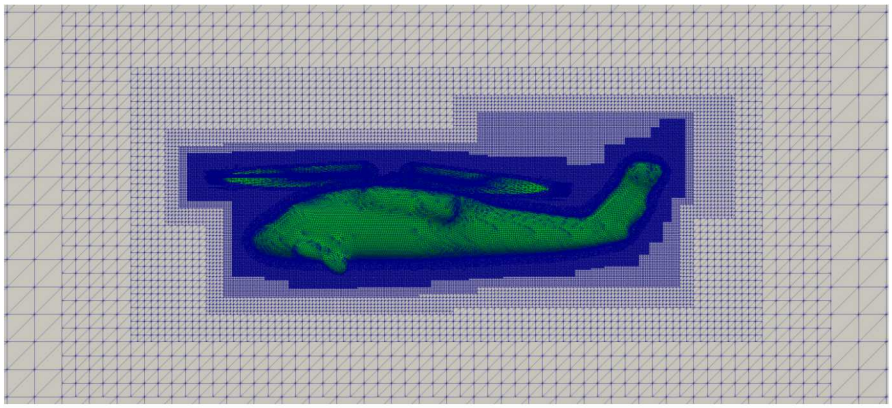

Fig. 7 *In situ* slice and surface extract showing the Helios near-body grids (green) and off-body grids (grey).

Version 10 of Helios uses a specialized Catalyst edition based off of the ParaView 5.6 release that includes Python wrapping.[1] Table 1 shows the memory load for three different Helios shared library build configurations: without Catalyst, with Catalyst 5.6, and the specialized Catalyst edition. The *in situ* extracts for these comparison runs were the internal surface and slice. The simulation runs used the Department of Defense's Onyx HPC machine at the Engineer Research and Development Center, which is a Cray XC40/50 with the memory reported from the first node by Helios's memory reporting routines.

3.4.3 Combination of Bespoke and VTK Functionality

Helios uses a dual-grid paradigm where an AMR grid is used sufficiently far from the rotorcraft body and a curve-fitting grid around the rotorcraft body, as shown in Figure 7. Depending on the setup, the end-user may select different solvers on the near-body grid and over the computational domain.

For example, kCFD could be used to compute the CFD solution on the grid for the rotorcraft fuselage, OVERFLOW could be used to compute the CFD solution for the rotor blades, and SAMCart could be used to compute the CFD solution over the off-body AMR grid. The rotor blade grids rotate, and their intersection with the

[1] This ParaView edition is available at `https://github.com/acbauer/ParaViewParticleTrackingCatalystEdition`.

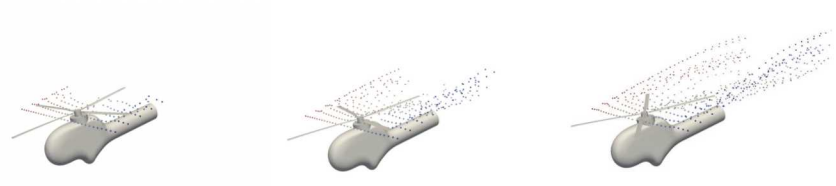

Fig. 8 *In situ* particle path and surface extract outputs for Higher-harmonic control Aeroacoustics Rotor Test II case. Images used with permission from Andrew Bauer courtesy of Kitware Source Quarterly Magazine.

other grids will change as the simulation proceeds. Thus, the grid overlap needs to be computed dynamically along with the blanking on the overlapping portion of the grids.

PUNDIT is a library that computes the overlap and blanking while transferring fields between grids. One of the Helios *in situ* outputs is an interpolated result onto a Cartesian grid where PUNDIT performs the interpolation computation. The Python-wrapped PUNDIT seamlessly operates within VTK's Python wrapping to interface with both Numpy and the VTK writers to output the desired information for a variety of post-processing tools. This combination of bespoke Helios and VTK functionality provides a convenient way to implement essential *in situ* functionality.

3.4.4 Temporal Analysis

Traditionally, modifying an *in situ* temporal analysis with an associated post-processing tool to work *in situ* has been difficult principally due to the pipeline architecture employed by these visualization tools. ParaView has a separate *in situ* particle path filter that works around this limitation. This filter is responsible for caching the dataset from previous time steps to relieve the visualization pipeline of this obligation. Additionally, it supports a simulation restarting the *in situ* particle path computation by reading specified particle locations from a file. The *in situ* particle path computation must behave the same regardless of whether the simulation was restarted or continuously computed from the initial conditions. Figure 8 demonstrates this functionality for the popular Higher-harmonic control Aeroacoustics Rotor Test II (HART-II) test case that maintains a good validation database.

Besides *in situ* particle paths, Helios supports temporal averaging of the interpolated output onto the Cartesian grid. The reason for implementing this functionality in a bespoke manner was the simplicity in computing the temporal average natively within Helios. As with the previous bespoke solution, it supports simulation restart using VTK writers and readers to dump out and read back in, respectively, restart information.

3.4.5 Zero-Copy Issues

For the *in situ* particle path filter to update the particle path location at a time step, it requires the full solution at both the current and previous time steps. This requirement prevents the adaptor from using a zero-copy of the simulation data arrays on the full dataset since the CFD solvers are not caching their meshes or fields for previous time steps. Also, because Helios uses multiple CFD solvers in a single CFD simulation, each of these CFD solvers will have a different non-dimensionalization scheme and store fields in the solver specific non-dimensionalized form.

Also, the Helios *in situ* output is always in the SAMCart, or off-body CFD solver, non-dimensionalized form. Thus, all near-body CFD fields require conversion to this non-dimensionalization form. This conversion prevents using zero-copying of the near-body fields, even without requesting *in situ* particle path output. In the future, after ParaView 5.6, the vtkScaledSOADataArrayTemplate class[2] will be used to alleviate this limitation.

3.4.6 Common Output Benefit

There are multiple reasons that the Helios tools and workflow support multiple CFD solvers for the near-body grids. A primary reason is the validation by comparing the results of different CFD solvers for the same simulation case. Comparing results through full data dumps in each CFD solver's native format is a complex and burdensome task. With *in situ* data extracts, both the near-body and off-body grids use a common data format, and all of the fields are in a consistent non-dimensionalization scheme, regardless of which near-body CFD solver used, enabling easy comparisons.

4 Conclusion

Libsim and Catalyst provide extensive tools for performing in situ visualization. They are likely the most feature-rich in situ libraries available to date.

That said, other similar in situ libraries exist. Lighter weight scripting libraries like Mayavi [8, 21] and yt [30] have been leveraged to perform in situ visualization. Other libraries like Ascent [15] are being designed from the ground up with in situ in mind. In contrast, some simulation frameworks, such as SCIRun [19], incorporate their own visualization functionality that can be used in situ.

Directly using the Libsim or Catalyst library requires what is often referred to as a "closely coupled" or "on-node proximity" in which the library is linked with the data producing program. However, they can be used with a general interface layer such as SENSEI [5] or Damaris/Viz [10] to decouple the in situ library from the data production. I/O libraries such as ADIOS [1, 17] can similarly be used for decoupling.

[2] `https://vtk.org/doc/nightly/html/classvtkScaledSOADataArrayTemplate.html`

See Bauer, et al. [6] for a broader literature review of current in situ tools.

We have seen in this chapter that Libsim and Catalyst share many features and design decisions. When they initially started, each had their own focus. Libsim got its early start as an interactive simulation debugging tool, but as file I/O became a major bottleneck on HPC, Libsim's main mode shifted to batch processing. Conversely, Catalyst got its start as a batch coprocessing library [12], but as use grew, interactive capabilities were added. Today, the functionality of the two tools overlap. The major difference is in the post-processing tool that each best interfaces with (VisIt versus ParaView), and simulation teams would do well to integrate the in situ library that works best with the other visualization tools used by the team.

Acknowledgements This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525. This chapter describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the chapter do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

Material presented in this chapter is a product of the CREATE (Computational Research and Engineering for Acquisition Tools and Environments) element of the U.S. Department of Defense HPC Modernization Program Office (HPCMO). Detailed input from the CREATE-AV^{TM} Helios development team was provided in order to properly customize the *in situ* workflow for rotorcraft analysis. Mark Potsdam of the U.S. Army's CCDC AvMC Technology Development Directorate was the main technical point of contact for Army SBIRs and has contributed significantly to the vision of Catalyst.

References

[1] Abbasi H, Lofstead J, Zheng F, Schwan K, Wolf M, Klasky S (2009) Extending I/O through high performance data services. In: IEEE International Conference on Cluster Computing and Workshops, DOI 10.1109/CLUSTR.2009.5289167

[2] Ahrens J, Geveci B, Law C (2005) ParaView: An end-user tool for large data visualization. In: Visualization Handbook, Elesvier, ISBN 978-0123875822

[3] Ahrens J, Jourdain S, O'Leary P, Patchett J, Rogers DH, Petersen M (2014) An image-based approach to extreme scale *in situ* visualization and analysis. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp 424–434, DOI 10.1109/SC.2014.40

[4] Ayachit U, Bauer A, Geveci B, O'Leary P, Moreland K, Fabian N, Mauldin J (2015) Paraview catalyst: Enabling in situ data analysis and visualization. In: Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization (ISAV 2015), pp 25–29, DOI 10.1145/2828612.2828624

[5] Ayachit U, Bauer A, Duque EPN, Eisenhauer G, Ferrier N, Gu J, Jansen KE, Loring B, Lukić Z, Menon S, Morozov D, O'Leary P, Ranjan R, Rasquin M, Stone CP, Vishwanath V, Weber GH, Whitlock B, Wolf M, Wu KJ, Bethel EW (2016) Performance analysis, design considerations, and applications of extreme-scale *in situ* infrastructures. In: SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, DOI 10.1109/SC.2016.78

[6] Bauer AC, Abbasi H, Ahrens J, Childs H, Geveci B, Klasky S, Moreland K, O'Leary P, Vishwanath V, Whitlock B, Bethel EW (2016) In situ methods, infrastructures, and applications on high performance computing platforms. Computer Graphics Forum 35(3):577–597, DOI 10.1111/cgf.12930

[7] Boeckel B, Ayachit U (2014) Why is paraview using all that memory? `https://blog.kitware.com/why-is-paraview-using-all-that-memory/`

[8] Buffat M, Cadiou A, Penven LL, Pera C (2017) In situ analysis and visualization of massively parallel computations. International Journal of High Performance Computing Applications 31(1):83–90, DOI 10.1177/1094342015597081

[9] Childs HR, Brugger E, Whitlock BJ, Meredith JS, Ahern S, Biagas K, Miller MC, Weber GH, Harrison C, Pugmire D, Fogal T, Garth C, Sanderson A, Bethel EW, Durant M, Camp D, Favre JM, Rubel O, Navratil P (2012) VisIt: An end-user tool for visualizing and analyzing very large data. In: High Performance Visualization: Enabling Extreme-Scale Scientific Insight, Chapman and Hall, pp 357–368

[10] Dorier M, Sisneros R, Peterka T, Antoniu G, Semeraro D (2013) Damaris/Viz: a nonintrusive, adaptable and user-friendlyin situ visualization framework. In: IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV), DOI 10.1109/LDAV.2013.6675160

[11] Duque EP, Whitlock BJ, Stone CP (2015) The impact of in situ data processing and analytics upon weak scaling of CFD solvers and workflows. In: ParCFD

[12] Fabian N, Moreland K, Thompson D, Bauer AC, Marion P, Geveci B, Rasquin M, Jansen KE (2011) The ParaView coprocessing library: A scalable, general purpose in situ visualization library. In: Proceedings of the IEEE Symposium on Large-Scale Data Analysis and Visualization, pp 89–96, DOI 10.1109/LDAV.2011.6092322

[13] Forsythe JR, Lynch E, Polsky S, Spalart P (2015) Coupled flight simulator and cfd calculations of ship airwake using kestrel. In: 53rd AIAA Aerospace Sciences Meeting, DOI 10.2514/6.2015-0556

[14] Kirby A, Yang Z, Mavriplis D, Duque E, Whitlock B (2018) Visualization and data analytics challenges of large-scale high-fidelity numerical simulations of wind energy applications. In: 2018 AIAA Aerospace Sciences Meeting, DOI 10.2514/6.2018-1171

[15] Larsen M, Ahrens J, Ayachit U, Brugger E, Childs H, Geveci B, Harrison C (2017) The ALPINE in situ infrastructure: Ascending from the ashes of strawman. In: Proceedings of the In Situ Infrastructures on Enabling Extreme-Scale Analysis and Visualization (ISAV '17), pp 42–46, DOI 10.1145/3144769.3144778

[16] Lorendeau B, Fournier Y, Ribes A (2013) In-situ visualization in fluid mechanics using catalyst: A case study for Code Saturne. In: IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV), DOI 10.1109/LDAV.2013.6675158

[17] Moreland K, Oldfield R, Marion P, Jourdain S, Podhorszki N, Vishwanath V, Fabian N, Docan C, Parashar M, Hereld M, Papka ME, Klasky S (2011) Examples of *in transit* visualization. In: Petascale Data Analytics: Challenges and Opportunities (PDAC-11)

[18] Oldfield RA, Moreland K, Fabian N, Rogers D (2014) Evaluation of methods to integrate analysis into a large-scale shock physics code. In: Proceedings of the 28th ACM international Conference on Supercomputing (ICS '14), pp 83–92, DOI 10.1145/2597652.2597668

[19] Parker SG, Johnson CR (1995) SCIRun: A scientific programming environment for computational steering. In: Proceedings ACM/IEEE Conference on Supercomputing

[20] Patchett J, Ahrens J, Nouanesengsy B, Fasel P, Oleary P, Sewell C, Woodring J, Mitchell C, Lo LT, Myers K, Wendelberger J, Canada C, Daniels M, Abhold H, Rockefeller G (2013) LANL CSSE L2: Case study of in situ data analysis in asc integrated codes. Tech. Rep. LA-UR-13-26599, Los Alamos National Laboratory

[21] Ramachandran P, Varoquaux G (2011) Mayavi: 3D visualization of scientific data. Computing in Science & Engineering 13(2):40–51, DOI 10.1109/MCSE.2011.35

[22] Rasquin M, Smith C, Chitale K, Seol ES, Matthews BA, Martin JL, Sahni O, Loy RM, Shephard MS, Jansen KE (2014) Scalable implicit flow solver for realistic wing simulations with flow control. Computing in Science & Engineering 16:13–21, DOI 10.1109/MCSE.2014.75

[23] Rintala R (2015) In situ XDB Workflow Allows Coupling of CFD to Flight Simulator for Ship Airwake/Helicopter Interaction. `http://www.ilight.com/en/news/in-situ-xdb-workflow-allows-coupling-of-cfd-to-flight-simulator-for-ship-airwake-helicopter-interaction` (accessed January 15, 2020)

[24] Sanderson A, Humphrey A, Schmidt J, Sisneros R (2018) Coupling the uintah framework and the visit toolkit for parallel in situ data analysis and visualization and computational steering. In: Weiland M, Alam S, Shalf J, Yokota R (eds) High Performance Computing - ISC High Performance 2018 International Workshops, Revised Selected Papers, Springer-Verlag, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), pp 201–214, DOI 10.1007/978-3-030-02465-9_14

[25] Schroeder W, Martin K, Lorensen B (2004) The Visualization Toolkit: An Object Oriented Approach to 3D Graphics, 4th edn. Kitware Inc., ISBN 1-930934-19-X

[26] Sitaraman J, Wissink A, Sankaran V, Jayaraman B, Datta A, Yang Z, Mavriplis D, Saberi H, Potsdam M, O'Brien D, Cheng R, Hariharan N, Strawn R (2010)

Application of the helios computational platform to rotorcraft flowfields. In: 48th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition, DOI 10.2514/6.2010-1230

[27] Sitaraman J, Potsdam M, Wissink A, Jayaraman B, Datta A, Mavriplis D, Saberi H (2013) Rotor loads prediction using helios: A multisolver framework for rotorcraft aeromechanics analysis. Journal of Aircraft 50(2):478–492, DOI 10.2514/1.C031897

[28] Smith TM, Menon S (1996) The structure of premixed flame in a spatially evolving turbulent flow. Combustion Science and Technology 119

[29] Stone CP, Menon S (2003) Open loop control of combustion instabilities in a model gas turbine combustor. Journal of Turbulence 4

[30] Turk MJ, Smith BD, Oishi JS, Skory S, Skillman SW, Abel T, Norman ML (2011) yt: a multi-code analysis toolkit for astrophysical simulation data. The Astrophysical Journal Supplement Series 192(9), DOI 10.1088/0067-0049/192/1/9

[31] Turuncoglu UU (2018) Towards in-situ visualization integrated earth system models: RegESM 1.1 regional modelling system. Geoscientific Model Development Discussions DOI 10.5194/gmd-2018-179

[32] Whitlock B, Favre JM, Meredith JS (2011) Parallel in situ coupling of simulation with a fully featured visualization system. In: Eurographics Symposium on Parallel Graphics and Visualization, DOI 10.2312/EGPGV/EGPGV11/101-109

[33] Whitlock BJ, Favre JM, Meredith JS (2011) Parallel in situ coupling of simulation with a fully featured visualization system. In: Proceedings of the 11th Eurographics conference on Parallel Graphics and Visualization, Eurographics Association, pp 101–109, DOI 10.2312/EGPGV/EGPGV11/101-109

[34] Wissink AM, Potsdam M, Sankaran V, Sitaraman J, Mavriplis D (2016) A dual-mesh unstructured adaptive cartesian computational fluid dynamics approach for hover prediction. Journal of the American Helicopter Society 61(1):1–19, DOI 10.4050/JAHS.61.012004