Catalyst Revised: Rethinking the ParaView In Situ Analysis and Visualization API

Utkarsh Ayachit[1], Andrew C. Bauer[3], Ben Boeckel[1], Berk Geveci[1], Kenneth Moreland[2], Patrick O'Leary[1], and Tom Osika[1]

[1] Kitware, Inc., Clifton Park, NY
[2] Oak Ridge National Lab, Oak Ridge, TN
[3] U.S. Army Engineer Research and Development Center, Vicksburg, MS

Abstract. As in situ analysis goes mainstream, ease of development, deployment, and maintenance becomes essential, perhaps more so than raw capabilities. In this paper, we present the design and implementation of *Catalyst*, an API for in situ analysis using ParaView, which we refactored with these objectives in mind. Our implementation combines design ideas from in situ frameworks and HPC tools, like Ascent and MPICH.

Keywords: in situ analysis and visualization · high-performance computing · software engineering

1 Introduction

Since the first release of ParaView Catalyst [4,6], the in situ data analysis landscape has evolved considerably. A wide array of libraries and frameworks are now available, each targeting different use-cases or environments [1, 5, 11, 18]. Leveraging these in situ frameworks for data analysis typically involves modifying the simulation code to pass and convert the simulation data structures to something that the in situ data analysis frameworks can interpret and then request processing. This development step is called instrumentation. With ParaView Catalyst, this instrumentation process has involved developing an adaptor that converts simulation data structures to VTK data objects. The VTK data model defines various ways of representing computation meshes and variables. The VTK data objects are then passed to the ParaView Catalyst engine for processing using the provided API. Except for a few Python-based simulations, i.e., simulations that use Python as the primary programming language or those that use Python to pass data (and control) to ParaView Catalyst, this adaptor is invariably a C++ codebase that requires a ParaView software-development-kit (SDK) to build. The simulation build and deployment workflow thus involves building the adaptor with a ParaView SDK and linking that with the simulation. This workflow, which seemed fairly reasonable in the early years, exposed significant maintenance challenges as the simulation codes and ParaView progressed through multiple versions.

Challenges developing the adaptor: Writing a new adaptor requires an intimate understanding of the VTK data model. The choice of the type of data object often has implications on both memory overhead and performance. Furthermore, VTK data APIs often support multiple ways of initialization. We expect developers to be aware of which APIs result in deep-copies and which ones do not. It is not uncommon for developers to pick an incorrect variant without realizing that it results in deep copies. The VTK data model itself keeps evolving. Since the first release of ParaView Catalyst, there have been changes to array layouts (array-of-structures and structure-of-arrays) [17], ghost-cell definitions [14], unstructured grid cell connectivity layout, and even addition of new data types for better representing composite datasets and AMR datasets. Several of these changes were motivated by the need to support various standard simulation memory layouts to avoid deep copies. Leveraging these changes, however, requires updating the adaptors to use the new APIs. This update adds another burden on the adaptor developers – to stay up-to-date with such data model changes and then update the adaptor to leverage them.

Challenges with build and deployment: For Fortran, C, and C++ simulators, the standard adaptor is written in C++ and uses ParaView and VTK C++ APIs. This process requires the adaptor is built using a ParaView sotware-development-kit (SDK). The SDK provides all the necessary ParaView headers and libraries needed to compile the adaptor. These are not available in standard ParaView package. Thus, developers have to build ParaView and all its dependencies from source and cannot use readily available binary distributions. The large set of dependencies and the diversity of supported compilers and platforms make providing a redistributable universal SDK challenging. Building ParaView and all its dependencies from source can be quite challenging, too.

There are several ways to build ParaView and its dependencies, including superbuild [16] and Spack [8]; however, with each release, there are inevitable new issues since the platforms, the dependencies, and the ParaView codebase are continually changing. Furthermore, the adaptor's build-system often needs to be updated to reflect changes to ParaView's build system. Over the years, ParaView library names have changed as have the ways of linking against them. These changes require that in addition to keeping abreast with the API changes, adaptor maintainers need to update the build system with each ParaView release. Since the adaptor links with the simulation executable, the simulation has a transitive dependency on the ParaView build. Thus, for every new release of ParaView, the maintainer needs to update and build the adaptor and rebuild the simulation. Each simulation build is tightly coupled with a specific ParaView build. Short of having multiple builds, it is not easy to switch between different adaptor or ParaView versions. Consequently, if there is a regression in a newer version of ParaView, the users cannot quickly test with an older version without having a separate complete build for the simulation.

These challenges apply to all of the in situ frameworks mentioned earlier to varying degrees, primarily based on the complexity, capability, and flexibility of the codebase and its dependencies.

In this paper, we propose an approach to the in situ API design that alleviates these maintenance difficulties for production in situ analysis and visualization. Our proposal defines a stable API that we call the Catalyst API and the mechanisms to provide ABI-compatible API implementation. ParaView-Catalyst is simply implementation of the Catalyst API that uses ParaView for data processing and visualization and is ABI-compatible with any other Catalyst API implementations. ABI-compatibility implies that a simulation built using any Catalyst API implementation can swap the implementation at runtime without rebuilding the simulation code.

2 Design

Our design takes a multi-pronged approach to address the challenges we encountered as we started using ParaView for in situ data analysis and visualization in production workflows. We liberally leverage design ideas and implementation from other in situ frameworks and HPC tools during the design process whenever possible.

2.1 Simplifying the adaptor

One of the challenges with the original Catalyst adaptor design is that it requires the developer to have a reasonable understanding of the VTK data model. Our original thinking was that it would be VTK and ParaView developers who would be the ones developing such adaptors – which, it turns out, does not reflect the reality. More often than not, it is the members of the simulation development community taking on the adaptor's development. This development requires a deep understanding of the VTK data model to make critical design choices when mapping simulation data structures to the VTK data model. The requirement is unreasonable and burdensome, impeding adoption and potentially resulting in implementations that leave room for memory and performance improvements.

Strawman [10], which later evolved into Ascent [12], presents an unique solution to this challenge. Instead of making the adaptor developers do the mapping to the target data model, it provides an API for describing and passing arrays of data. This API is called Conduit [10], which allows simulation developers to describe the simulation data, such as computational meshes and fields. By standardizing one (or several) schemas that support a diverse collection of computational meshes and field arrays encountered, we can provide standard implementations for converting a Conduit mesh description to an appropriate VTK data object. These converter implementations can be part of the ParaView distribution and hence evolve as the ParaView/VTK APIs evolve to represent the data optimally without requiring any effort on the part of simulation developers. Thus, the adaptor no longer requires converting simulation data structures to VTK data structures, but rather simply describes what they are using Conduit.

2.2 Simplifying build and deployment

Our design takes inspiration from the MPICH ABI compatibility initiative [15] to simplify the build and deployment process. The initiative aims to enable developers to build their code using any compatible MPI implementation and swap with another compatible implementation at runtime. This makes it easier to distribute executables (or libraries) without binding to a specific MPI implementation. We extend this concept for in situ APIs. We explicitly define functions that form the in situ API, which includes the API to describe simulation data structures (Section 2.1) and a few other function calls to initialize, update and finalize the in situ analysis. This API specification is what we now refer to as *Catalyst*. By enabling implementations of this API that are runtime compatible with one another, simulations can be compiled with one implementation and executed with a different implementation. This decoupling has several advantages.

First, developers no longer need a ParaView SDK to compile instrumented simulation codes. They can use any Catalyst API implementation, including the *stub* implementation that we provide. This stub implementation is lightweight, minimal-capability, easy-to-build, and has insignificant overhead at runtime. Developers can use this implementation during the compilation stage to compile the adaptor. At execution time, they can easily swap the Catalyst implementation using environment modules or other platform-specific loader configuration mechanisms. The ParaView distribution includes an implementation of the Catalyst API that we now refer to as *ParaView-Catalyst*. Thus, ParaView-Catalyst is simply one specific implementation of the Catalyst API that uses ParaView for data processing.

Second, switching between multiple versions of Catalyst API implementations does not require any recompilation. The Catalyst API version and the ParaView versions are independent. Thus, several versions of ParaView can provide Catalyst implementations that are runtime compatible with each other.

Third, if, in the future, non-backward compatible API changes are introduced to the Catalyst API causing its version number to change, implementations can continue to support earlier versions. Thus, ParaView distributions can continue to provide multiple versions of the ParaView-Catalyst library, each compatible with a particular version of the Catalyst API.

3 Implementation

The Catalyst API, together with the *stub* implementation, is now a separate project [7]. The project intentionally does not have any external dependencies. This separation keeps the project simple to build and easy to deploy on any platform with a C++ compiler and standard build tools. Simulation developers can use this *stub* implementation of the API when instrumenting simulations and do not need a full ParaView SDK.

Figure 1 shows the directory structure and relevant files in a Catalyst install on a Linux-based system. The Catalyst API's current version number is

```
# Catalyst install directory contents
..[install prefix]/
|-- include/
|   |-- catalyst-2.0/
|       |-- catalyst.h
|       |-- (other headers)
|   |-- lib/
|       |-- libcatalyst.so -> libcatalyst.so.2
|       |-- libcatalyst.so.2
|       |-- ...
```

Fig. 1. Contents of a Catalyst install

2.0 to help distinguish it from the earlier implementation of ParaView Catalyst. `catalyst.h` is the header with all function declarations that are part of the public API. It internally includes other headers installed under the `include/` directory. The `libcatalyst.so`, which is a symbolic link to `libcatalyst.so.2`, is the single shared library for the *stub* implementation. The library name includes the Catalyst API version number. Whenever the API changes in a non-ABI compatible fashion, the number will be incremented. `libcatalyst.so` does not have any runtime dependencies except C and C++ language runtimes.

The Catalyst API comprises C functions and data-structures alone. We do not expose any C++ interface as part of this public API to avoid ABI compatibility issues when using different C++ compilers for compiling the simulation and the Catalyst implementation. Using C also makes interfacing with other languages such as Python and FORTRAN trivial. The API comprises four `catalyst_..` functions that act as entry points to the Catalyst framework (Figure 2) and several `conduit_...` functions that are part of the Conduit C API used to communicate data and other parameters.

```
1  /** initialize catalyst */
2  void catalyst_initialize(const conduit_node* params);
3
4  /** execute catalyst per cycle */
5  void catalyst_execute(const conduit_node* params);
6
7  /** finalize catalyst **/
8  void catalyst_finalize(const conduit_node* params);
9
10 /** query information about catalyst implementation and capabilities **/
11 void catalyst_about(conduit_node* params);
```

Fig. 2. Catalyst API

The `conduit_node` object provides a JSON-inspired hierarchical description of parameters and in-core data, which communicates both the control parameters to configure the Catalyst implementation and the simulation meshes. A Catalyst implementation is free to define an arbitrary schema for exchanging data and control parameters with the simulation through this API. Simulations tar-

geting a particular implementation use the appropriate schema when generating the hierarchical description. ParaView-Catalyst, which is now a ParaView-based implementation of the Catalyst API, supports the schema described in Figure 3. The schema borrows heavily from the schema supported by Ascent [12] with simple extensions to support ParaView-Catalyst concepts such as channels and scripts.

```
{ /* schema for 'catalyst_initialize' */
  "catalyst": {
    // all catalyst params/data are under this root
    "scripts": {
        // collection of Python scripts for analysis pipelines
        "name0": "path/scriptname.py",
        ...
    }
  }
}

{ /* schema for 'catalyst_execute' */
  "catalyst": {
    "state": {
        "cycle": [integer] /* cycle/timestep number */,
        "time" : [number]  /* time */
    },
    "channels": {
       // named data channels.
       "channel-name0" : {
          "type": [string] /* type of the channel data */
          "data": {} /* data description node based on chosen 'type' */
       }
    }
  }
}
```

Fig. 3. Schema supported by ParaView-Catalyst

The *channels* can be used to pass multiple meshes for in situ analysis. The "type" attribute selects the mesh schema. Currently, ParaView-Catalyst supports the Conduit Mesh Blueprint [13], which covers a wide range of computation meshes and memory layouts. The "type" attribute lets us support additional mesh description schema in future releases.

Instrumenting a simulation involves populating the `conduit_node` object with appropriate values based on the schema and invoking the `catalyst_...` functions at appropriate times. There is no explicit mapping of simulation data structures to VTK data objects anymore. Instead, developers simply provide the data description. Converting that to VTK data objects is handled by the ParaView-Catalyst library itself. The following snippet highlights modifications necessary to a typical simulation to use Catalyst.

```
// this header is needed for all the conduit_ and catalyst_ functions
#include <catalyst.h>

```

```
// ** initialize catalyst **
conduit_node* catalyst_init_params = conduit_node_create();
// pass initialization parameters e.g. scripts to load.
conduit_node_set_path_char8_str(catalyst_init_params,
  "catalyst/scripts/script0", ".../script0.py");
...
catalyst_initialize(catalyst_init_params);
conduit_node_destroy(catalyst_init_params);

...
for (cycle=0; ..., ++cycle) // simulation loop
{
  // ..... advance simulation ...

  // ** execute catalyst per timestep/cycle**
  conduit_node* catalyst_exec_params = conduit_node_create();
  // 'state' is used to pass time/cycle information.
  conduit_node_set_path_int64(catalyst_exec_params, "catalyst/state/cycle",
    cycle);
  conduit_node_set_path_float64(catalyst_exec_params, "catalyst/state/time",
    time);

  // the data must be provided on a named channel. the name is determined by
    the
  // simulation. for this one, we're calling it "grid".

  // declare the type of the channel; we're using Conduit Mesh Blueprint
  // to describe the mesh and fields, chosen using the type "mesh"; in future
  // other types can be supported.
  conduit_node_set_path_char8_str(catalyst_exec_params,
    "catalyst/channels/grid/type", "mesh");

  // now, create the mesh;
  conduit_node* mesh = conduit_node_create();

  // The 'mesh' node is populated as per Conduit Mesh Blueprint applicable
  // for the specific simulation. For example.  a uniform grid is defined as
    follows
  conduit_node_set_path_char8_str(mesh, "coordsets/coords/type", "uniform");
  conduit_node_set_path_int64(mesh, "coordsets/coords/dims/i", i_dim);
  conduit_node_set_path_int64(mesh, "coordsets/coords/dims/j", j_dim);
  conduit_node_set_path_int64(mesh, "coordsets/coords/dims/k", k_dim);
  // .. and so on. Refer to Conduit Mesh Blueprint for details.

  ...
  // the mesh is passed on the named channel as "../data"
  conduit_node_set_path_external_node(catalyst_exec_params,
     "catalyst/channels/grid/data", mesh);
  catalyst_execute(catalyst_exec_params);
  conduit_node_destroy(catalyst_exec_params);
  conduit_node_destroy(mesh);
}
...

// ** finalize catalyst**
conduit_node* catalyst_fini_params = conduit_node_create();
catalyst_finalize(catalyst_fini_params);
conduit_node_destroy(catalyst_fini_params);
```

The instrumented simulation can be compiled with any implementation of the Catalyst API. The *stub* implementation is preferred since it is easy to build and has no dependencies. To compile this adaptor, one only needs to add the path to the `catalyst.h` header to the include path and link with the single catalyst shared library. With gcc, for example, this can be done as follows:

```
$ gcc -I<catalyst-install-prefix>/include/catalyst-2.0
      -L<catalyst-install-prefix>/lib
      -lcatalyst
      [source files] -o [output]
```

At execution time, the operating system loader will look to load the library `libcatalyst.so.2`. Since all Catalyst implementations for a specific version are compatible with one another, the end-user can make the loader load a different Catalyst implementation than the one compiled using standard mechanisms. On Linux systems, one can set the `LD_LIBRARY_PATH` environment variable to point to the `libcatalyst.so.2` in a ParaView binary distribution to use that ParaView-Catalyst implementation. Switching to another version solely requires changing the environment and re-executing, no need to recompile or require a ParaView SDK. On most HPC systems, this can be easily handled by environment modules.

4 Evaluation

To evaluate the design, we instrumented LULESH [9] – a mini-application (mini-app) that represents a typical hydrodynamics code, like ALE3D – to use the Catalyst API. Incidentally, LULESH was previously instrumented with the legacy ParaView/Catalyst framework [2], which allowed us to compare and contrast the two implementations.

Build system: LULESH uses a simple Makefile to build the code. The first thing that the legacy adaptor implementation did was convert the build-system to use CMake instead because adding a build dependency to the legacy ParaView/Catalyst implementation was much easier in a CMake-based system; the library dependency chain can be quite long and cumbersome to resolve outside of CMake. The new implementation [3] did not require us to perform similar work. We simply extended the Makefile to define two new variables, `CATALYST_CXXFLAGS` and `CATALYST_LDFLAGS`, and subsequently add them to the compile and link lines, respectively.

```
# for Catalyst
CATALYST_ROOT=...
CATALYST_CXXFLAGS = -DVIZ_CATALYST=1 -I$(CATALYST_ROOT)/include/catalyst-2.0/
CATALYST_LDFLAGS = -L$(CATALYST_ROOT)/lib -lcatalyst

.cc.o: lulesh.h
  @echo "Building $<"
  $(CXX) -c $(CXXFLAGS) $(CATALYST_CXXFLAGS) -o $@  $<

lulesh2.0: $(OBJECTS2.0)
  @echo "Linking"
  $(CXX) $(OBJECTS2.0) $(LDFLAGS) $(CATALYST_LDFLAGS) -lm -o $@
```

Adaptor complexity: The legacy adaptor included approximately 13 header files. While this by itself is not necessarily a detriment, it is a good indication of the number of different classes the adaptor used and the different building blocks that the developer had to be aware of when developing the adaptor. In contrast, the new adaptor code only includes one header, `catalyst.h`. The bulk of the new adaptor code is simply populating the `conduit_node` object according to the schema described in Section 3. When deciding on the VTK dataset type to use,

the legacy adaptor implementation for LULESH used a `vtkUnstructuredGrid`. While that is acceptable, in retrospect, that is not the best choice in this case since it requires that the cell connectivity is explicitly specified when the grid is actually topologically regular. Not only does that result in memory overhead, but it also impacts the performance of in situ analysis pipelines. This poor design choice underscores one of the primary motivations for this effort to revise the API. Without in-depth knowledge of the VTK data model, it was not easy to make the best choice of which VTK dataset type to use, and the choice often has a significant impact on the performance of the processing pipelines. The new Catalyst API does not suffer from the same issue since the VTK dataset choice is deferred to the Catalyst implementation, specifically, ParaView-Catalyst.

Code changes: Comparing the code changes between the two implementations of the Catalyst Adaptor using git, we get the following:

```
# legacy version
git diff --stat master..catalyst_adaptor
 CMakeLists.txt     |  38
 lulesh-catalyst.cc | 209
 lulesh-catalyst.h  |  17
 lulesh-util.cc     |  13
 lulesh.cc          |  18
 lulesh.h           |  33
 6 files changed, 325 insertions(+), 3 deletions(-)

# new version
git diff --compact-summary master..catalyst-2.0
 Makefile                 | 13
 lulesh-catalyst.cc (new) | 52
 lulesh-init.cc           | 56
 lulesh-util.cc           | 10
 lulesh.cc                |  5
 lulesh.h                 | 22
 6 files changed, 155 insertions(+), 3 deletions(-)
```

As expected, the new code is more compact with only 155 lines of implementation compared to 325 lines of implementation with the legacy API. This reduction is mainly due to not including any code to create VTK datasets.

4.1 Debugging and Regression Testing

To make it easier to develop and debug, the Catalyst stub implementation supports generating binary data dumps for the conduit nodes passed to each `catalyst_` call. For each API call, it can write the `conduit_node` argument out to disk. Using another executable, `catalyst_replay`, these dumps can be read back in and each API call invoked again in the same order. This avoids the need for rerunning the simulation for debugging. To generate the data dumps, one uses the stub implementation with an environment variable `CATALYST_DATA_DUMP_DIRECTORY` set to point a directory where the node data for each API invocation should be saved. `catalyst_replay` can then be used to read these data dumps while using any Catalyst implementation.

Besides assisting in development and debugging, this also helps avoid regressions. Data dumps can be generated for validation and verification setups for codes of interest and then used by Catalyst implementations for regression testing to ensure newer versions continue to work for supported codes.

5 Conclusion and Future Work

As in situ analysis and visualization become widely adopted in production, ease of development, deployment, and maintenance become just as important as the framework capabilities.

The Catalyst API enables the development of implementations that use different libraries underneath for the actual in situ data processing instead of ParaView. It is conceivable that frameworks like SENSEI [1] and Ascent themselves can be provided as implementations of the Catalyst API. Thus, a simulation, once instrumented, can switch between any framework at runtime by merely switching runtime modules.

Our current design relies on shared libraries for runtime swapping of implementations. There may be cases where a fully static build is required. Our ongoing work is to ensure that such cases can be supported, albeit with limited runtime flexibility.

Our implementation currently only supports C/C++ codes. Fortran compatibility and Python simulation support is pending development.

Acknowledgments

This work was supported by the following funding sources.

This material is based in part upon work supported by the US Army's Engineer Research And Development Center.

This work was partially 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.

This material is based in part upon work supported by the US Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under contract DE-AC05-00OR22725.

This material is based upon work supported by the U.S. Department of Energy, Office of Science, ASCR program under Award Number DE-SC-0021343.

References

1. Ayachit, U., Whitlock, B., Wolf, M., Loring, B., Geveci, B., Lonie, D., Bethel, E.W.: The sensei generic in situ interface. In: 2016 Second Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization (ISAV). pp. 40–44 (2016)
2. Ayachit, U.: Paraview catalyst enabled lulesh (Feb 2018). https://doi.org/10.5281/zenodo.4013875, https://doi.org/10.5281/zenodo.4013875
3. Ayachit, U.: Catalyst2.0-enabled lulesh (Jul 2021). https://doi.org/10.5281/zenodo.5143793, https://doi.org/10.5281/zenodo.5143793

4. Ayachit, U., Bauer, A., Geveci, B., O'Leary, P., Moreland, K., Fabian, N., Mauldin, J.: 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. p. 25–29. ISAV2015, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2828612.2828624, https://doi.org/10.1145/2828612.2828624
5. Bauer, A.C., Abbasi, H., Ahrens, J., Childs, H., Geveci, B., Klasky, S., Moreland, K., O'Leary, P., Vishwanath, V., Whitlock, B., Bethel, E.W.: *In Situ* Methods, Infrastructures, and Applications on High Performance Computing Platforms, a State-of-the-art (STAR) Report. Computer Graphics Forum (Special Issue: Proceedings of EuroVis 2016) **35**(3) (Jun 2016), lBNL-1005709
6. Bauer, A.C., Geveci, B., Schroeder, W.: The ParaView Catalyst Users Guide. Clifton Park, NY: Kitware (2013)
7. Catalyst Developers: Catalyst (2020), https://gitlab.kitware.com/paraview/catalyst/
8. Gamblin, T., LeGendre, M.P., Collette, M.R., Lee, G.L., Moody, A., de Supinski, B.R., Futral, W.S.: The Spack Package Manager: Bringing order to HPC software chaos. In: Supercomputing 2015 (SC'15). Austin, Texas (November 15-20 2015), http://tgamblin.github.io/pubs/spack-sc15.pdf
9. Karlin, I., Keasler, J., Neely, R.: Lulesh 2.0 updates and changes. Tech. Rep. LLNL-TR-641973 (August 2013)
10. Larsen, M., Harrison, C., Brugger, E., Griffin, K., Elliot, J.: Strawman: A batch in situ visualization and analysis infrastructure for multi-physics simulation codes (4 2016). https://doi.org/10.1145/2828612.2828625
11. Larsen, M., Ahrens, J., Ayachit, U., Brugger, E., Childs, H., Geveci, B., Harrison, C.: 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. p. 42–46. ISAV'17, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3144769.3144778, https://doi.org/10.1145/3144769.3144778
12. Larsen, M., Ahrens, J., Ayachit, U., Brugger, E., Childs, H., Geveci, B., Harrison, C.: The alpine in situ infrastructure: Ascending from the ashes of strawman. pp. 42–46 (11 2017). https://doi.org/10.1145/3144769.3144778
13. Lawrence Livermore National Laboratory: Conduit: Simplified Data Exchange for HPC Simulations - Conduit Blueprint (2019), https://llnl-conduit.readthedocs.io/en/v0.5.1/blueprint_mesh.html
14. Lipsa, Dan and Geveci, Berk: Ghost and Blanking (Visibility) Changes (2015), https://blog.kitware.com/ghost-and-blanking-visibility-changes/
15. MPICH Developers: MPICH ABI Compatibility Initiative (2014), https://wiki.mpich.org/mpich/index.php/ABI_Compatibility_Initiative
16. ParaView Developers: ParaView-Superbuild (2020), https://gitlab.kitware.com/paraview/paraview-superbuild/
17. Vacanti, Allison and Rose, Libby: New Data Array Layouts in VTK 7.1 (2016), https://blog.kitware.com/new-data-array-layouts-in-vtk-7-1
18. Whitlock, B., Favre, J.M., Meredith, J.S.: Parallel in situ coupling of simulation with a fully featured visualization system. In: Proceedings of the 11th Eurographics Conference on Parallel Graphics and Visualization. p. 101–109. EGPGV '11, Eurographics Association, Goslar, DEU (2011)