Catalyst Documentation

Catalyst Development Team

Oct 14, 2022

Contents:

This document refers to the Catalyst API which was first introduced in ParaView 5.9. For earlier versions of Catalyst, please refer to earlier docs [TODO: add link]

CHAPTER 1

What is Catalyst?

Catalyst is an API specification developed for simulations (and other scientific data producers) to analyze and visualize data in situ.

It also includes the following:

- A light-weight implementation of the Catalyst API. This implementation is called **stub**. [TODO: need a better name]
- An SDK for developers to develop implementations of the Catalyst API to perform custom data processing and visualization tasks.

The Catalyst API uses 'C' and is binary compatible with different implementations of the API making it easier to change the implementation at runtime.

1.1 Relationship with ParaView

Starting with 5.9, ParaView releases come with an implementation of the Catalyst API. This implementation can be used in lieu of the **stub** to analyze and visualize simulation results using ParaView's data-processing and visualization capabilities.

1.2 Relationship with Conduit

The Catalyst API uses Conduit for describing data and other parameters which can be communicated between a simulation and Catalyst.

Conduit provides a standard way to describe computational simulation meshes. This is called the Mesh Blueprint. ParaView's implementation of the Catalyst API supports a subset of the Mesh Blueprint. Simulations that can use the Mesh Blueprint to describe their data can directly use ParaView's Catalyst implementation for in situ analysis and visualization.

1.3 ParaView Catalyst

ParaView Catalyst is the name now used to refer to ParaView's implementation of the Catalyst API. Prior to this API separation (i.e. ParaView 5.8 and earlier), ParaView Catalyst or simply Catalyst was used to denote the in situ API together with the data analysis and visualization capabilities it provided. In other words, the in-situ capabilities of ParaView were collectively called Catalyst.

With ParaView 5.9, while legacy uses will still be supported for a few more releases, we use the names to refer to specific components:

- **Catalyst**: the API and SDK described here.
- **ParaView**: the parallel data analysis and visualization application and framework.
- **ParaView Catalyst**: the implementation of the Catalyst API that uses ParaView for in situ data analysis and visualization.

ParaView Catalyst supports several ways for simulations to describe computational meshes and fields. One way is to use Conduit's Mesh Blueprint. Another way is to use Fides. Furthermore, developers can develop their own implementations of the Catalyst API and still use ParaView's capabilities for in situ data processing and visualization. ParaView provides API that such developers can use to initialize and invoke ParaView in situ. [TODO: link to ParaView docs for API to use in custom Catalyst implementations that use ParaView].

CHAPTER 2

Build and Install

2.1 Obtaining the source

To obtain the Catalyst source locally, clone the official code repository using Git.

```
git clone https://gitlab.kitware.com/paraview/catalyst.git
```

2.2 Building

Catalyst uses CMake to generate build system scripts and projects, such as Makefiles or Ninja build files. While IDE generators (Xcode and Visual Studio) are supported, Ninja is highly recommended.

To do a fresh build, start with an empty directory as follows:

```
mkdir catalyst-build
cd catalyst-build
ccmake -G Ninja [path to catalyst source directory]

# do the build
ninja

# optionally, run tests
ctest

# do the install
ninja install
```

`ccmake` is a graphical GUI that lets you specify various options for CMake. Alternately, those options can be specified on command line to `cmake` using `-Doption:type=value` (or `-Doption=value`) parameters as follows:

```
cmake -G Ninja -DCATALYST_BUILD_TESTING:BOOL=ON ... [path to catalyst src dir]
```

Using `-G Ninja` results in CMake generating build files for Ninja. You can switch to using any other supported generator of your choice. See CMake Docs for details. In that case, `ninja` will be replaced by the appropriate build tool in the steps above.

2.2.1 Supported CMake Options

Important CMake options that affect how Catalyst is built are:

- `CATALYST_BUILD_SHARED_LIBS` (default: `ON`): choose whether to build static or shared libraries for Catalyst. To support switching of Catalyst implementation at runtime, you must build with `CATALYST_BUILD_SHARED_LIBS` set to `ON` (default).
- `CATALYST_BUILD_STUB_IMPLEMENTATION` (default: `ON`): choose whether to build the stub Catalyst implementation. When building Catalyst only to develop another Catalyst API implementation, you may turn this option to `OFF`. If `OFF`, no `catalyst` library will be built.
- `CATALYST_BUILD_TESTING` (default: `ON`): enable/disable testing. Running the tests using `ctest` after a build has succeeded is a good way to verify that your build is functional.
- `CMAKE_BUILD_TYPE` (default: `Debug`): this is used to choose whether to add debugging symbols to the build. Supported values are `Debug`, `Release`, `MinSizeRel`, and `RelWithDebInfo`.
- `CMAKE_INSTALL_PREFIX`: path where to install the libraries and headers when requested.

CHAPTER 3

Catalyst for Simulation Developers

This section describes how simulation (and other computational codes) can use Catalyst.

3.1 Building with Catalyst

To use the Catalyst API in any code, the code must be built against an implementation of the Catalyst API. While one can use any implementation of the Catalyst API, the stub implementation is probably the easiest to build against since it doesn't have any external dependencies besides compiler tools.

There are two ways codes can build with Catalyst: using CMake, or using any build tool like make.

3.1.1 Using CMake

If your code already uses CMake as the build system generator, then to use Catalyst APIs, you simply need to find the Catalyst install using `find_package` and the link against the `catalyst::catalyst` target. This is done as follows:

```
# Find the Catalyst install.
#
# The version is optional but recommended since it lets you choose
# the compatibility version. The only supported value currently is 2.0
#
# REQUIRED ensures that CMake raises errors if Catalyst is not found
# properly.

find_package(catalyst 2.0 REQUIRED)


# Your simulation will have an executable (or a library) that
# houses the main-loop in which you'll make the Catalyst API falls.
# You need to link that executable (or the library) target with Catalyst.
# This is done as follows (where simulation_target must be replaced by the
```

(continues on next page)

(continued from previous page)

```
# name of the correct executable (or library) target.

target_link_library(simulation_target
  PRIVATE catalyst::catalyst)
```

Now, when you run `cmake` on your simulation code, a new cache variable `catalyst_DIR` can be set to the directory containing the file `catalyst-config.cmake` to help CMake find where you built Catalyst. That file can be found in either the Catalyst build directory or the Catalyst install directory.

3.1.2 Using *make* (or similar)

If not using CMake as the build system generator for your simulation code, it is still easy to make it aware of Catalyst. You simply need to pass the include path i.e. the location where the Catalyst headers are available, and the location and library to link against.

In a typical Catalyst install at location, `CATALYST_INSTALL_PREFIX`, these are:

- Include path: `<CATALYST_INSTALL_PREFIX>/include/catalyst-2.0`
- Library path: `<CATALYST_INSTALL_PREFIX>/lib`
- Library: `<CATALYST_INSTALL_PREFIX>/lib/libcatalyst.so`

Using `gcc`, for example, this translates to the following command-line:

[TODO: fix me]

```
> TODO
```

3.2 Catalyst API

Catalyst API is used by simulations to invoke Catalyst for co-processing. To use the Catalyst API, one must include the `catalyst.h` header file.

3.2.1 catalyst_initialize

```
enum catalyst_status catalyst_initialize(const conduit_node* params);
```

This function must be called once to initialize Catalyst. Metadata that can be used to configure the initialize is provided using a `params` pointer.

The catalyst will attempt to load the implementation named using `params["catalyst_load/implementation"]`. If not specified, but the `CATALYST_IMPLEMENTATION_NAME` environment variable is, it will be used. If no implementation is named, a default implementation using the stub functions will be used.

If an implementation is named, it will be loaded at runtime using `dlopen` (or the platform equivalent) by searching the nodes specified under the `params["catalyst_load/search_paths"]` node. Next, the paths specified by the `CATALYST_IMPLEMENTATION_PATHS` (using `;` as a separator on Windows and `:` otherwise) will be searched. Finally, the `catalyst` directory beside `libcatalyst` will be searched. Once found, it will be loaded and inspected for compatibility. If it is compatible, the implementation will be loaded and made available. The return code indicates the error received, if any.

The search priority of the `CATALYST_IMPLEMENTATION_` environment variables may be made first by setting teh `CATALYST_IMPLEMENTATION_PREFER_ENV` environment variable to a non-empty value.

3.2.2 catalyst_finalize

```
enum catalyst_status catalyst_finalize(const conduit_node* params);
```

This function must be called once to finalize Catalyst. Metadata is passed using `params` pointer.

3.2.3 catalyst_execute

```
enum catalyst_status catalyst_execute(const conduit_node* params);
```

This function is called for every time step as the simulation advances. This is the call in which the analysis may execute. `params` provides metadata as well as the data generated by the simulation for that time-step.

3.2.4 catalyst_about

```
enum catalyst_status catalyst_about(conduit_node* params);
```

This function fills up the `params` instance with metadata about the Catalyst library being used.

3.2.5 catalyst_results

```
enum catalyst_status catalyst_results(conduit_node* params);
```

This function fills up the `params` instance with updated parameters values from the Catalyst implementation side.

All the above functions use a `params` object which is a conduit_node. It is simply a hierarchical mechanism for describing data and/or metadata including simulation meshes and fields. Essentially, think of it as a map where keys are strings called paths and values are either data or pointers to data. What these keys can be and what they mean is totally up to the Catalyst API implementation being used.

To create and populate the `conduit_node` instance, you use the Conduit `C` API. e.g.

```
conduit_node* node = conduit_node_create();
conduit_node_set_path_int(node, "sim/timestep", 0);
conduit_node_set_path_double(node, "sim/time", 1.212);
...
conduit_node_destroy(node);
```

Refer to Conduit documentation for details of the `C` API. [TODO: there are no docs for Conduit C API upstream].

CHAPTER 4

Catalyst for Implementation Developers

Developers can develop custom implementations for the Catalyst API to support a wide variety of use-cases. In most cases, however, if your goal is to use ParaView for in situ data processing, it may be easier to simply use **ParaView Catalyst**. It support several ways for describing computational meshes and field arrays including Mesh Blueprint and Fides.

This section describes the workflow for those who want to implement a custom implementation for the Catalyst API.

4.1 Prerequisites

- To build a custom Catalyst implementation, your project needs to be a CMake-based project i.e. use CMake as the build system generator. While it is technically feasible to use a non-CMake based project, it is highly recommended to prefer to use CMake.

4.2 CMake Setup

The following sample CMakeLists.txt shows how to build a Catalyst implementation.

```
# When implementing the Catalyst API, as against using
# it to invoke Catalyst, one must use the component
# ``SDK`` in ``find_package`` call. This ensures that all necessary
# libraries etc. are available.
find_package(catalyst
             REQUIRED
             COMPONENTS SDK)

# use this function call to create a Catalyst API implementation.
catalyst_implementation(
  TARGET  MyCustomCatalystImpl
  NAME    MyImplName
  SOURCES MyCustomCatalystImpl.cxx)
```

That is it! `catalyst_implementation` creates the library with the appropriate CMake target-properties on the library including setting its name and version number. This function is only available when the `SDK` component is explicitly requested in the `find_package(catalyst .. )` call.

For more advanced usage, the following arguments are also supported:

- `EXPORT <export>`: Add the target to the named export set.
- `LIBRARY_DESTINATION <destination>`: Where to place the implementation underneath the build and install trees (with reasonable defaults if not provided).
- `CATALYST_TARGET <target>`: The name of the target which provides the Catalyst API (defaults to `catalyst::catalyst`).

4.3 Implementing Catalyst API

Providing an implementation for the Catalyst API implies providing code for the five `catalyst_` functions that are part of the Catalyst API: * `catalyst_initialize_MyImplName`, * `catalyst_finalize_MyImplName`, * `catalyst_execute_MyImplName`, * `catalyst_about_MyImplName`, * `catalyst_results_MyImplName`

To do that, simply include `catalyst.h` and `catalyst_impl_MyImplName.h` headers in your implementation file and add definitions for these functions. Definitions for all the five functions must be provided. You can choose to invoke the default stub implementation for any of the functions by including the `catalyst_stub.h` header and then calling `catalyst_stub_initialize`, `catalyst_stub_finalize`, `catalyst_stub_execute`, `catalyst_stub_about` or `catalyst_stub_results` in your implementations for the corresponding methods.

If your custom implementation is using C++, you can include `c/conduit_cpp_to_c.hpp` headers to convert the `conduit_node` pointer to a `conduit::Node` instance pointer using `conduit::cpp_node()`. Then you can use the `conduit::Node` API which is generally friendlier than the C API.

```
#include <catalyst.h>
#include <conduit.hpp>            // for conduit::Node
#include <conduit_cpp_to_c.hpp>   // for conduit::cpp_node()

...

enum catalyst_status catalyst_about_MyImplName(conduit_node* params)
{
  // convert to conduit::Node
  conduit::Node &cpp_params = (*conduit::cpp_node(params));

  // now, use conduit::Node API.
  cpp_params["catalyst"]["capabilities"].append().set("adaptor0");
}
```

On successful build of your project, you should get a shared library named `libcatalyst-ImplName.so`, `libcatalyst-ImplName.so`, or `catalyst-ImplName.dll` on Linux, macOS, and Windows respectively.

4.4 Using your Catalyst implementation

Now, to use your implementation with any simulation or code built with the stub Catalyst implementation, all you need to do is to make sure your Catalyst library is found and loaded by `catalyst_initialize`.

CHAPTER 5

Debugging and Catalyst Replay

To simplify the process of debugging in-situ pipelines, catalyst now supports the serialization of conduit_nodes. During each API call, users can write the `params` argument of each API call out to disk. Then, using `catalyst_replay`, the nodes will be read back in, and each API call will be invoked again. This prevents users from needing to re-run their simulation when debugging.

5.1 Serializing Nodes and Writing to Disk

To use the `catalyst_replay` command, nodes must first be written to disk. The steps to do this are simple:

1. Set the environment variable `CATALYST_DATA_DUMP_DIRECTORY` to the directory where the node data for each API invocation should be saved.
2. Invoke the **stub** implementation in your custom API implementation.

This will write the `conduit_node` passed into the API call out to `CATALYST_DATA_DUMP_DIRECTORY`. The `conduit_nodes` are written out as `.conduit_bin` files. They will follow the general pattern `<stage>_params.conduit_bin.<num_ranks>.<rank>`, where:

1. `<stage>` is one of `initialize`, `execute` or `finalize`.
2. `<num_ranks>` is the number of MPI ranks that the simulation was run with.
3. `<rank>` is the 0 based index of the rank used to generate this file.

Files for the `execute` stage will also include the invocation number, since `catalyst_execute` can be called multiple times. For example, `execute_invc0_params.conduit_bin.2.1` would contain the `params` passed into the 0th invocation of `catalyst_execute`, which was called by 2nd of two ranks (since rank indices are 0-indexed).

5.2 Replaying API Calls with *catalyst_replay*

After the node data has been written out to disk, the `catalyst_replay` command can be used to read the node data back into memory and execute the same API calls. Find the `catalyst_replay` executable in the `RUNTIME_OUTPUT_DIRECTORY` generated by CMake (this is usually `bin/`). Run `catalyst_replay` with the same number of MPI ranks as the simulation used to generate the data, and pass the value of `CATALYST_DATA_DUMP_DIRECTORY` as a command-line argument. This invoke each API method with the corresponding node data. For an example, see the `examples/replay` directory.

CHAPTER 6

Debugging Catalyst

Catalyst supports some facilities to debug its loading procedures.

6.1 *CATALYST_DEBUG*

The `CATALYST_DEBUG` environment variable may be set to a non-empty value to log the search and loading procedures for catalyst implementations.