In-situ visualization with ParaView Catalyst2

ALEX RAZOUMOV
alex.razoumov@westdri.ca

Zoom controls

- Please mute your microphone and camera unless you have a question
- To ask questions at any time, type in Chat, or Unmute to ask via audio
  - please address chat questions to "Everyone" (not direct chat!)
- Raise your hand in Participants

- Email training@westdri.ca
- Our fall training schedule `https://bit.ly/wg2022b`
  - webinars, local workshops, autumn school

Why Catalyst

- Modern parallel simulations can produce huge amounts of data $\Rightarrow$ I/O bottleneck
  - large HPC codes cannot afford to write full data to disk every time step

Why Catalyst

- Modern parallel simulations can produce huge amounts of data ⇒ I/O bottleneck
  - large HPC codes cannot afford to write full data to disk every time step
- In-situ visualization idea: instead of writing full 3D variables to disk, only write either:
  1. some derived data needed for visualization (e.g. 2D polygonal isosurfaces), or
  2. images directly from the simulation code

  ⇒ much smaller disk output, no need to write full 3D datasets every time step
  - sometimes refereed to as "co-processing"

Why Catalyst

- Modern parallel simulations can produce huge amounts of data $\Rightarrow$ I/O bottleneck
  - large HPC codes cannot afford to write full data to disk every time step
- In-situ visualization idea: instead of writing full 3D variables to disk, only write either:
  1. some derived data needed for visualization (e.g. 2D polygonal isosurfaces), or
  2. images directly from the simulation code

  $\Rightarrow$ much smaller disk output, no need to write full 3D datasets every time step
  - sometimes refereed to as "co-processing"
- Wait ... writing images or smaller datasets to disk directly from simulation codes is nothing new: we've been doing it for the last 50 years

Why Catalyst

- Modern parallel simulations can produce huge amounts of data ⇒ I/O bottleneck
  - large HPC codes cannot afford to write full data to disk every time step
- In-situ visualization idea: instead of writing full 3D variables to disk, only write either:
  1. some derived data needed for visualization (e.g. 2D polygonal isosurfaces), or
  2. images directly from the simulation code

  ⇒ much smaller disk output, no need to write full 3D datasets every time step
  - sometimes refereed to as "co-processing"
- Wait ... writing images or smaller datasets to disk directly from simulation codes is nothing new: we've been doing it for the last 50 years
- Let's rephrase: in-situ in a ParaView Catalyst / VisIt LibSim context refers to exposing your in-memory simulation data arrays to Catalyst (at runtime, without writing any data to disk) and then using familiar interactive Python pipelines to process and visualize data
  1. instrument your simulation code once
     - Catalyst is extremely diligent about not duplicating any data arrays in memory, passing only pointers and data description via its API
  2. modify and apply your analysis/visualization pipelines without recompiling your code

Catalyst history

- Circa 2012-2013 - development of the original Catalyst library
  - relied heavily on users mapping their simulation code's data structures to VTK data objects; required good knowledge of the VTK data model
  - it was relatively easy to make mistakes duplicating data arrays in memory, as there are different ways to initialize VTK data APIs depending on the data object type
  - ParaView / VTK codebase is continually changing $\Rightarrow$ simulation code maintainers had to continuously update their Catalyst adaptor, or even maintain multiple code builds to work with different ParaView versions

Catalyst history

- Circa 2012-2013 - development of the original Catalyst library
  - relied heavily on users mapping their simulation code's data structures to VTK data objects; required good knowledge of the VTK data model
  - it was relatively easy to make mistakes duplicating data arrays in memory, as there are different ways to initialize VTK data APIs depending on the data object type
  - ParaView / VTK codebase is continually changing $\Rightarrow$ simulation code maintainers had to continuously update their Catalyst adaptor, or even maintain multiple code builds to work with different ParaView versions
- 2020 - first release of Catalyst2 (the focus of this webinar)
  - for user-provided simulation data (meshes and fields) description switched to Conduit API `https://llnl-conduit.readthedocs.io`
  - converting of a Conduit mesh description to an appropriate VTK data object is done inside the Catalyst2 library (not exposed to user) $\Rightarrow$ the Catalyst adaptor inside the simulation code should be relatively stable across many versions of ParaView / VTK

Core Catalyst API: converting data structures to VTK, few functions to initialize / update / finalize the in-situ analysis (standalone GitHub repo)

Combined ParaView-Catalyst API: the ParaView-based implementation of the Catalyst API = core Catalyst API library + collection of Catalyst-related libraries from a specific ParaView build

We will work with Conduit mesh description (exposing data to Catalyst)
`https://llnl-conduit.readthedocs.io/en/latest/blueprint_mesh.html`
in the second example

---

1. Start with a much simpler, pre-built Python miniapp acting as a simulation code
2. Continue with a local or remote C simulation code generating UnstructuredGrid data
3. If we have time: ParaView Live (primarily for debugging)

- Let's start with a basic ParaView Python script:

```
from paraview.simple import *

wavelet1 = Wavelet()

slice1 = Slice(registrationName='Slice1', Input=wavelet1)
slice1.SliceType = 'Plane'
slice1.SliceType.Normal = [0, 0, 1]

sliceDisplay = Show(slice1)

ResetCamera()    # fit the object into the view
```

```
paraview --script=sample3a.py
```

- No file output for now
- Here we generate a VTK source `Wavelet()` on the fly

- This same code can be used in a Catalyst Python script to set up a visualization pipeline
  - should switch our data source from `Wavelet()` to take in data generated by a simulation
- We'll use a miniapp `paraview.demos.wavelet_miniapp` that acts as a simulation producing a time-value `vtkImageData` dataset
  - run this miniapp for 3 timesteps alongside a ParaView Catalyst script
  - in Catalyst, the simulation data is available on *named channels*
  - use `--channel` to name the output of `paraview.demos.wavelet_miniapp` (inside the miniapp the output is not named)
  - inside the Catalyst script use `registrationName` to specify the data channel
  - this argument to `Wavelet()` is an optional argument to any source/filter in ParaView; if specified, it'll override the function itself $\Rightarrow$ now `Wavelet()` does not produce a wavelet but the simulation data as its output

```
< wavelet1 = Wavelet()
---
> wavelet1 = Wavelet(registrationName="Wavelet1")
```

```
pvbatch -m paraview.demos.wavelet_miniapp --script sample3b.py \
        --script-version 2 --timesteps 3 --channel Wavelet1
```

- Let's add some output with 10 time steps:

```
> from paraview import print_info
> view = GetActiveView()
>
> def catalyst_execute(info):
>     fname = "output-%d.png" % info.timestep
>     print_info("time=%f, saving file: %s", info.time, fname)
>     SaveScreenshot(fname, view, ImageResolution=[1000, 1000])
```

```
pvbatch -m paraview.demos.wavelet_miniapp --script sample3c.py \
        --script-version 2 --timesteps 10 --channel Wavelet1
```

- Try replacing `Wavelet(...)` with `Sphere(...)` or any other source ... the result won't change
- `catalyst_execute()` is executed every time step

- Optionally in Catalyst scripts you define `catalyst_initialize()` and `catalyst_finalize()` – these are executed once, at the start and the end of your pipeline, e.g.

```
def catalyst_initialize():
    print_info("in '%s::catalyst_initialize'", __name__)

def catalyst_finalize():
    print_info("in '%s::catalyst_finalize'", __name__)
```

Extractors

- There is an even better way to output images and data in Catalyst!
- In ParaView's GUI, starting with 5.9, you can use Extractors to create various Catalyst functions that get triggered when you run `catalyst_execute` from your simulation code (typically, once per timestep)

---

1. In ParaView GUI start from scratch and create Sources | Wavelet (could be anything actually; we just want to create a screenshot)
2. Optionally rename the input (Wavelet is good for the wavelet example; important step for actual simulation codes)
3. In the pipeline browser select apply Slice and then apply Extractors | Image | PNG
4. File | Save Catalyst State to save it as `extract-image.py`
5. In the saved script, look for `CreateExtractor()` function and all attributes of a handle produced by this function, and copy these to our Catalyst script replacing `catalyst_execute`

Image extractor

```
< def catalyst_execute(info):
<     fname = "output-%d.png" % info.timestep
<     print_info("time=%f, saving file: %s", info.time, fname)
<     SaveScreenshot(fname, view, ImageResolution=[1000, 1000])
---
> # create an image extractor
> png1 = CreateExtractor('PNG', view, registrationName='PNG1')
> png1.Trigger = 'TimeStep'
> png1.Writer.FileName = 'RenderView1_{timestep:06d}.png'
> png1.Writer.ImageResolution = [1000, 1000]
> png1.Writer.Format = 'PNG'
```

```
pvbatch -m paraview.demos.wavelet_miniapp --script sample4a.py \
        --script-version 2 --timesteps 3 --channel Wavelet1
```

- Now all output goes to `datasets/RenderView1_00000*`

Data extractor

1. In ParaView GUI create Sources | Wavelet and apply Filters | Slice or run `sample3a.py`
2. In the pipeline browser select the Slice, apply Extractors | Data | VTP (aka VTK Polygonal Data)
3. File | Save Catalyst State – save it as `extract-data.py`
4. In your Catalyst script use the relevant lines to replace the image extractor with a data extractor

```
< # create an image extractor
< png1 = CreateExtractor('PNG', view, registrationName='PNG1')
< png1.Trigger = 'TimeStep'
< png1.Writer.FileName = 'RenderView1_{timestep:06d}.png'
< png1.Writer.ImageResolution = [1000, 1000]
< png1.Writer.Format = 'PNG'
---
> # create a data extractor
> vtp1 = CreateExtractor('VTP', slice1, registrationName='VTP1')
> vtp1.Trigger = 'TimeStep'
> vtp1.Writer.FileName = 'Slice1_{timestep:06d}.Pvt'

pvbatch -m paraview.demos.wavelet_miniapp --script sample4b.py \
        --script-version 2 --timesteps 3 --channel Wavelet1
```

- Now in `datasets/` we also have 2D polygonal data, in `pvtp` file format (parallel VTK Polygonal)
  - for a partitioned dataset with parallel `pvbatch`, the extractor will write it into multiple files
  - you can also apply Extractors | Data to the 3D dataset in the pipeline browser – the result will be a 3D VTK Image Data sequence

Starting with Catalyst2 API

1. Download and compile core Catalyst2 library from `https://gitlab.kitware.com/paraview/catalyst.git`

2a. Optionally compile ParaView server from source `https://www.paraview.org/download`

- at runtime you'll need quite a few Catalyst-related libraries from a ParaView build of the same version you will use to create Catalyst scripts
- Catalyst instructions mention that you can use precompiled ParaView, however there are numerous online reports of problems that only go away when you compile ParaView from source (I ran into similar issues); likely related to the included MPI version

2b. Alternatively, on clusters you can use our precompiled `paraview-offscreen` module, but then you'll be tied to a specific ParaView version, and you might still run into problems

3. In ParaView's source code play with `ParaView-v5.10.1/Examples/Catalyst2/`

Modified C example

- For this demo on Cedar, I modified the C example from `ParaView-v5.10.1/Examples/Catalyst2/CFullExample/{*.c,*.h,catalyst_pipeline.py}`
  - file descriptions in the next slide
  - in `FEDataStructures.c` defined a 3D "sine envelope" wave function inside a unit cube ($x_i \in [0,1]$)

$$f(x_1,x_2,x_3) = \sum_{i=1}^{2}\left[\frac{\sin^2\left(\sqrt{\xi_{i+1}^2+\xi_i^2}\right)-0.5}{\left[0.001(\xi_{i+1}^2+\xi_i^2)+1\right]^2}+0.5\right], \text{ where } \xi_i \equiv 15(x_i-0.5)$$

    as a CellData 3D array called "density"
  - for time evolution I simply shift the function along the x-axis with periodic BCs
  - to `CatalystAdaptor.h` added an "IO" pipeline to write out data (more on this later)
  - modified `FEDriver.c` to run only one timestep
- Two versions of the same code: `standalone-mpi/{imageData,unstructuredData}`, both included in the ZIP file (link below)
- Compile and then run either serial or parallel code

```
cd standalone-mpi/unstructuredData && ./bin/CFullExampleV2 <flags_and_parameters>
mpirun -np <number_of_cores> ./bin/CFullExampleV2 <flags_and_parameters>
```

Code components

Let's take a look at these source files inside `standalone-mpi/unstructuredData`:

`FEDriver.c`	the main loop of the simulation
`FEDataStructures.c`	define simulation data: `InitializeGrid()`, `InitializeAttributes()`,`UpdateFields()`
`CatalystAdaptor.h`	interface with Catalyst library: `do_catalyst_initialization()`,`do_catalyst_execute()`,`do_catalyst_finalization()`
`catalyst_pipeline.py`	Catalyst script called at runtime

Compiling and running parallel C simulation code on a cluster

Details with self-hosted ParaView in `compilingOnCedar.txt` inside the ZIP file (link below)

- Via Slurm with manually compiled core Catalyst2 library and `paraview-offscreen` module
- For best performance, do not run on a parallel filesystem!
  1. compile core Catalyst2 library in `/tmp` on a login node
  2. tar it along with your simulation code into an archive in your `$HOME`
  3. unpack into `$SLURM_TMPDIR` at runtime

```
cd ~/scratch
module load gcc/9.3.0 openvkl/0.10.0 paraview-offscreen/5.10.0
salloc --ntasks=2 --time=0:30:0 --mem-per-cpu=3600 --account=def-razoumov-ac
cd $SLURM_TMPDIR
tar xvfz ~/catalyst-packed.tgz --strip-components=2 # unpack catalyst/ standalone-mpi/
cd standalone-mpi/unstructuredData
mpicc -O2 -c FEDriver.c FEDataStructures.c -DPARAVIEW_IMPL_DIR=\"\" -DUSE_CATALYST=1 \
      -I$SLURM_TMPDIR/catalyst/include/catalyst-2.0
mpicc FEDataStructures.o FEDriver.o -o bin/CFullExampleV2 -L../catalyst/lib64 \
      -lcatalyst -lm -Wl,-rpath,../catalyst/lib64 \
      -Wl,-rpath,$EBROOTOPENVKL/lib64,-rpath,$SLURM_TMPDIR/catalyst/lib64
export CATALYST_IMPLEMENTATION_PATHS=$EBROOTPARAVIEW/lib64/catalyst
export CATALYST_IMPLEMENTATION_NAME=paraview
mpirun -np 2 ./bin/CFullExampleV2 catalyst_pipeline.py   # each MPI rank will print
```

Compiling and running serial C simulation code in MacOS (laptop)

- With manually compiled core Catalyst2 library and manually compiled ParaView

```
cd $DEMO/standalone/unstructuredData
sed -i '' '/mpi.h/d' FEDriver.c FEDataStructures.c
sed -i '' '/MPI_/d' FEDriver.c FEDataStructures.c
gcc -O2 -c FEDataStructures.c FEDriver.c -DPARAVIEW_IMPL_DIR=\"\" -DUSE_CATALYST=1 \
    -I$DEMO/catalyst/include/catalyst-2.0
gcc FEDataStructures.o FEDriver.o -o bin/CFullExampleV2 -L$DEMO/catalyst/lib \
    -lcatalyst -Wl,-rpath,$DEMO/catalyst/lib
export CATALYST_IMPLEMENTATION_PATHS=$DEMO/paraview/lib/catalyst
export CATALYST_IMPLEMENTATION_NAME=paraview
./bin/CFullExampleV2 catalyst_pipeline.py
```

The default ParaView Catalyst script

- `catalyst_pipeline.py` that came with the example does not do much ...

```
from paraview.simple import *

print("executing catalyst_pipeline")

# registrationName must match the channel name used in Catalyst Adaptor
producer = TrivialProducer(registrationName="grid")

# familiar function, triggered when there is new data in the channel
def catalyst_execute(info):
    global producer

    print("-----------------------------------")
    print("executing (cycle={}, time={})".format(info.cycle, info.time))
    producer.UpdatePipeline()
    print("bounds:", producer.GetDataInformation().GetBounds())
    print("velocity-magnitude-range:", producer.PointData["velocity"].GetRange(-1))
    print("density-range:", producer.CellData["density"].GetRange(0))
```

Representative dataset

- We want to replace `catalyst_pipeline.py` with an actual visualization pipeline script processing our data in some way and then writing output with an image/data Extractor
  1. load a representative dataset and process it in the GUI
  2. apply an Extractor
  3. File | Save Catalyst State to save your script
- In the previous (miniapp) example we could create a representative dataset in the GUI via Sources | Wavelet, as it happened to produce exactly the same data type as the miniapp
- Our representative dataset must:
  1. be of the same dataset type, e.g. vtkUnstructuredGrid, vtkImageData, etc.
  2. have the same attributes defined over the grids as the simulation adaptor code will provide to Catalyst during simulation runs

  \+ to generate the script in the GUI, we must use the same ParaView version as the version of ParaView Catalyst that the simulation code will run with
- In Catalyst1 there used to be a script `gridwriter.py` one could always run with the simulation code, that would generate a representative dataset
- In Catalyst2 a generic `gridwriter.py` is currently being developed (the one you can find on GitHub is not yet functional)

Where do we get a representative dataset?

Two options:

1. Create via a combination of Sources (for grids) and Calculators (for 3D variables), perhaps using a Programmable Filter for Unstructured Grids – straightforward but slow manual process
2. Add a data writer to your Catalyst Adaptor following the C++ example in `ParaView-v5.10.1/Examples/Catalyst2/CxxImageDataExample`

Let's take a look at `CatalystAdaptor.h` in our modified C example:
here we create a Conduit node to describe our data and pass it to the Catalyst API

- node passed to `catalyst_initialize()` ☞ see next slide
- info in the node used *at every time step* to store all data in the named channel to disk
- array data never duplicated; nodes used merely for description

ParaView-Catalyst Blueprint based on Conduit Mesh Blueprint

`https://llnl-conduit.readthedocs.io/en/latest/blueprint_mesh.html`
`https://kitware.github.io/paraview-docs/latest/cxx/ParaViewCatalystBlueprint.html`

- Catalyst API provides 3 main functions to pass data and control from a simulation code to the ParaView-Catalyst implementation: `catalyst_initialize()`, `catalyst_execute()`, and `catalyst_finalize()`
- Each of the three functions below is passed a Conduit Node object
  - think of it as a light-weight container with hierarchical construction (nodes inside nodes)
  - a top-level node named catalyst_load is used to load a specific Catalyst implementation, e.g. ParaView-Catalyst
  - if PV-Catalyst, it looks for a top-level node named catalyst
  - `catalyst_initialize()` can process the following children of catalyst:
    - scripts/name is the path to the Python script to load for in-situ analysis
    - pipelines is an object node with child nodes that provide type and parameters for pipelines
  - `catalyst_execute()` can process the following children of catalyst:
    - state/timestep and state/time to specify the timestep and time
    - channels/[channel-name] to specify a named channel
    - channels/[channel-name]/type to specify the channel type (“mesh” or “multimesh”)
    - channels/[channel-name]/data to communicate the simulation data on this channel, takes its own separate Conduit node with a hierarchical structure to describe the mesh topology, point coordinates, maybe cell definitions, and fields
  - `catalyst_finalize()` takes an empty node

ParaView-Catalyst Blueprint (cont.)

Useful for debugging:

`conduit_node_print(<Conduit_Node>)` - shows what information was passed to Catalyst

e.g. in our example could use one of the following lines:

```
conduit_node_print(catalyst_exec_params);
conduit_node_print(mesh);
```

Representative dataset ⇒ Catalyst script ⇒ in-situ visualization

- Adjust the number of time steps, recompile and then run

```
./bin/CFullExampleV2 --output dataset-%04ts.vtpd
```

  - Catalyst always produces a partitioned dataset
  - if run in parallel, would produce multiple files per timestep

1. Load one of the timesteps into ParaView
2. Apply Resample To Image at 50^3
3. Apply Filters | Contour at $\rho = 0.15$
4. Apply Extractors | Image | PNG
5. File | Save Catalyst State to save it as `extract-image.py`
6. Make sure `registrationName` in the script matches the data channel name in the simulation code
   - alternatively, could do this by renaming the first data object in the pipeline browser
7. Run

```
./bin/CFullExampleV2 extract-image.py
```

8. Check `datasets/` directory

In-situ visualization with data output

Repeat steps 1-3 from last slide, but this time:

4. Apply Extractors | Data | VTP
5. File | Save Catalyst State to save it as `extract-data.py`
6. Make sure `registrationName` in the script matches the data channel name in the simulation code
   - alternatively, could do this by renaming the first data object in the pipeline browser
7. Run

   ```
   ./bin/CFullExampleV2 extract-data.py
   ```

8. Check the file sizes of the original 3D and the latest 2D data

ParaView Live

- Idea: load data from your live simulation on the cluster into a ParaView server (without writing this data to disk!) and view it in a ParaView client on your computer

---

- For simplicity assuming that both the ParaView server + client are running on your computer as part of the ParaView application
- In principle, it *might be tempting* to set up a ParaView server (talking to the simulation) on a remote machine with a higher bandwidth to the cluster, and the ParaView client on your computer, but it is actually not a good idea, since all heavy processing should be done in-situ by the simulation code, and you should be sending much smaller, derived datasets to the ParaView server
- In what follows, our ParaView application will act as a server (with a GUI on top) that your live simulation on the cluster is connecting to (avoiding the term "client" on purpose)
- This is not for production visualization, but rather for interactive work: either (1) creating/debugging Catalyst scripts or (2) debugging your simulation
- Ideally, all production visualization should be done non-interactively via Catalyst scripts

Create a ParaView Live Catalyst script

Repeat steps 1-3 from previous in-situ slides, but this time:

4. Apply Extractors | Image | PNG
5. File | Save Catalyst State to save it as `extract-live.py`, checking "Enable Catalyst Live"
   - this flag will expose (with an option to download) other data structures in the pipeline
   - make sure `registrationName` in the script matches the data channel name in the simulation code
6. Optionally, copy this script to the cluster (not needed for a local demo)
7. Before running the simulation, need to set up a ParaView server

ParaView Live with a simulation code on Cedar

1. On Cedar prepare your simulation

```
cd ~/scratch
module load gcc/9.3.0 openvkl/0.10.0 paraview-offscreen/5.10.0
salloc --time=0:30:0 --mem-per-cpu=3600 --account=def-razoumov-ac
cd $SLURM_TMPDIR
tar xvfz ~/catalyst-packed.tgz --strip-components=2 # unpack catalyst/ standalone-mpi/
cd standalone-mpi
>>> modify FEDriver.c to run many timesteps
mpicc -O2 -c FEDriver.c FEDataStructures.c -DPARAVIEW_IMPL_DIR=\"\" -DUSE_CATALYST=1 \
      -I$SLURM_TMPDIR/catalyst/include/catalyst-2.0
mpicc FEDataStructures.o FEDriver.o -o bin/CFullExampleV2 -L../catalyst/lib64 -lcatalyst -lm \
      -Wl,-rpath,../catalyst/lib64,-rpath,$EBROOTOPENVKL/lib64,-rpath,$EBROOTPARAVIEW/lib64
export CATALYST_IMPLEMENTATION_PATHS=$EBROOTPARAVIEW/lib64/catalyst
export CATALYST_IMPLEMENTATION_NAME=paraview
>>> write down the node's name
```

2. On your computer organize port forwarding; default ports are 22222 on both sides; here assuming local port 22223 and remote port 22225

```
ssh username@cedar.computecanada.ca -R 22225:cdr528:22223
```

Optionally, might need to clear local ports if port forwarding fails:

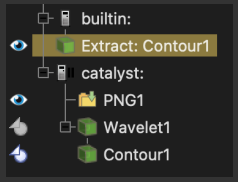
```
sudo lsof -i :22223   # check which processes occupy the port
sudo kill -9 <pid>
```

ParaView Live with a simulation code on Cedar (cont.)

3. Launch ParaView GUI and then Catalyst | Connect and confirm to accept connections from Catalyst on port 22223
4. Catalyst | Pause Simulation
5. Back inside the job on Cedar start your simulation

```
>>> optionally modify extract-live.py: options.Port = 22223   # if using non-default port 22223
./bin/CFullExampleV2 extract-live.py
```

6. Wait for the connection between the simulation and the server (your laptop's ParaView) to be established
7. Click on greyed-out *data* icons in the pipeline to send this data to the server
   - ☞ be super-super-careful: the data size could be huge for large datasets!!!
   - ☞ this is where the pipeline inside `extract-live.py` matters
8. Enable the corresponding visual layer in the pipeline browser
9. Do interactive visualizaton as usual

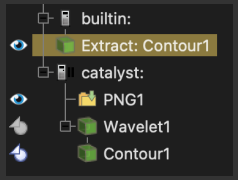


Demo: ParaView Live with a local simulation code

1. Launch ParaView GUI and then Catalyst | Connect and confirm to accept connections from Catalyst on port 22222
2. Catalyst | Pause Simulation
3. Start your simulation

```
export CATALYST_IMPLEMENTATION_PATHS=/Users/razoumov/tmp/joshua/paraview/lib/catalyst
export CATALYST_IMPLEMENTATION_NAME=paraview
./bin/CFullExampleV2 extract-live.py
```

4. Wait for the connection between the simulation and the server (your laptop's ParaView) to be established
5. Click on greyed-out *data* icons in the pipeline to send this data to the server
   - ☞ be super-super-careful: the data size could be huge for large datasets!!!
   - ☞ this is where the pipeline inside `extract-live.py` matters
   - ☞ in this specific demo we have three data objects to pull the data from: choose wisely
6. Enable the corresponding visual layer in the pipeline browser
7. Do interactive visualizaton as usual
8. Without a breakpoint Catalyst | Continue will run to the end of the simulation
9. Catalyst | Set Breakpoint (seems buggy at the moment ... no apparent breakpoint) followed by Catalyst | Continue



Summary

- Steep learning curve ... however, once implemented in a simulation code, you do not even need to recompile the code to switch to a new pipeline
- Catalyst Python pipelines can be created interactively in the GUI
- Catalyst can produce both images and derived ($\Rightarrow$ smaller) data
  - writes partitioned data, one file / MPI rank
- Can be used from a variety of languages: C, C++, Fortran, Python, Julia
- Catalyst has been shown to scale to millions of CPU cores
- Design philosophy: use Conduit nodes to pass pointers to existing arrays in memory
  - arrays will not be duplicated
  - you need to describe the mesh type / coordinates / topology
  - do not need to know the underlying VTK data model
  - internally, supports all VTK data types: ImageData, StructuredGrid, UnstructuredGrid, ...
- Make sure to use latest Catalyst2 (a significant rewrite of the original library)
- Be careful when enabling data in ParaView Live
- Official Catalyst User's Guide `https://www.paraview.org/download` is a good read but in places few years out of date
- Universal in-situ project `https://sensei-insitu.org` – data producers talking to generic in-situ endpoints via a unified API, can combine multiple endpoints in a single simulation (next slide)

Questions?

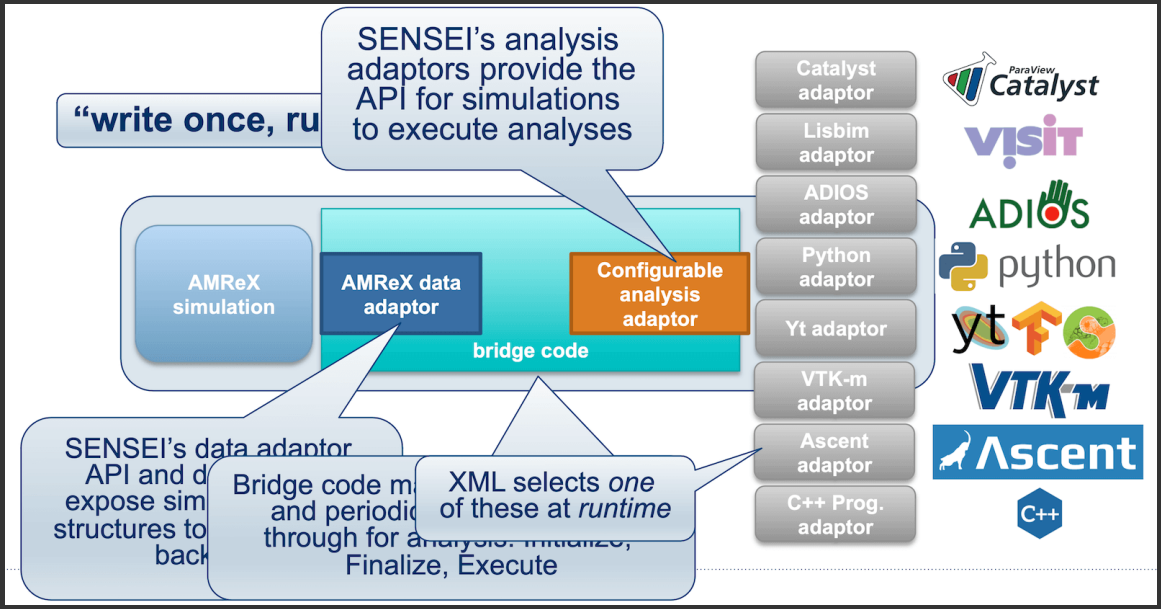


Figure borrowed from SENSEI tutorial at Supercomputing 2019
https://sensei-insitu.org/tutorials