



ParaView Catalyst: Scalable *In Situ* processing



Andrew Bauer (Kitware, Inc.)

Jeffrey Mauldin (Sandia National Laboratories)

June 19th, 2014

Agenda

- Introduction to ParaView Catalyst
- Catalyst for Developers
- Catalyst for Users
- Building ParaView Catalyst

Goals

- Understand the flow of control and data when using ParaView Catalyst with a simulation code
 - Understand efficiency trade-offs
- Hands-on exercises for:
 - Interfacing a simulation code with Catalyst
 - Creating VTK data structures to represent grid and field information from simulation data structures
 - Creating a Python script to extract data and screenshots through ParaView Catalyst
- Running a simulation with all of the parts put together

Online Materials

- This presentation and VM virtual appliance at:
 - www.paraview.org/Wiki/ParaView/Catalyst/PETTT_Tutorial
- Install ParaView 4.1
 - www.paraview.org/download
- Install VirtualBox
 - www.virtualbox.org

VirtualBox

- Run VirtualBox and go to “File→Import Appliance...”
- Click on “Open appliance...” and select catalyستutorial1.ova, click on “Next >”
- Can adjust appliance settings as needed and click on “Import”
 - Suggest keeping defaults
- Click on “Start” to run VirtualBox with Catalyst tutorial examples
 - Can click on “OK” for warning messages

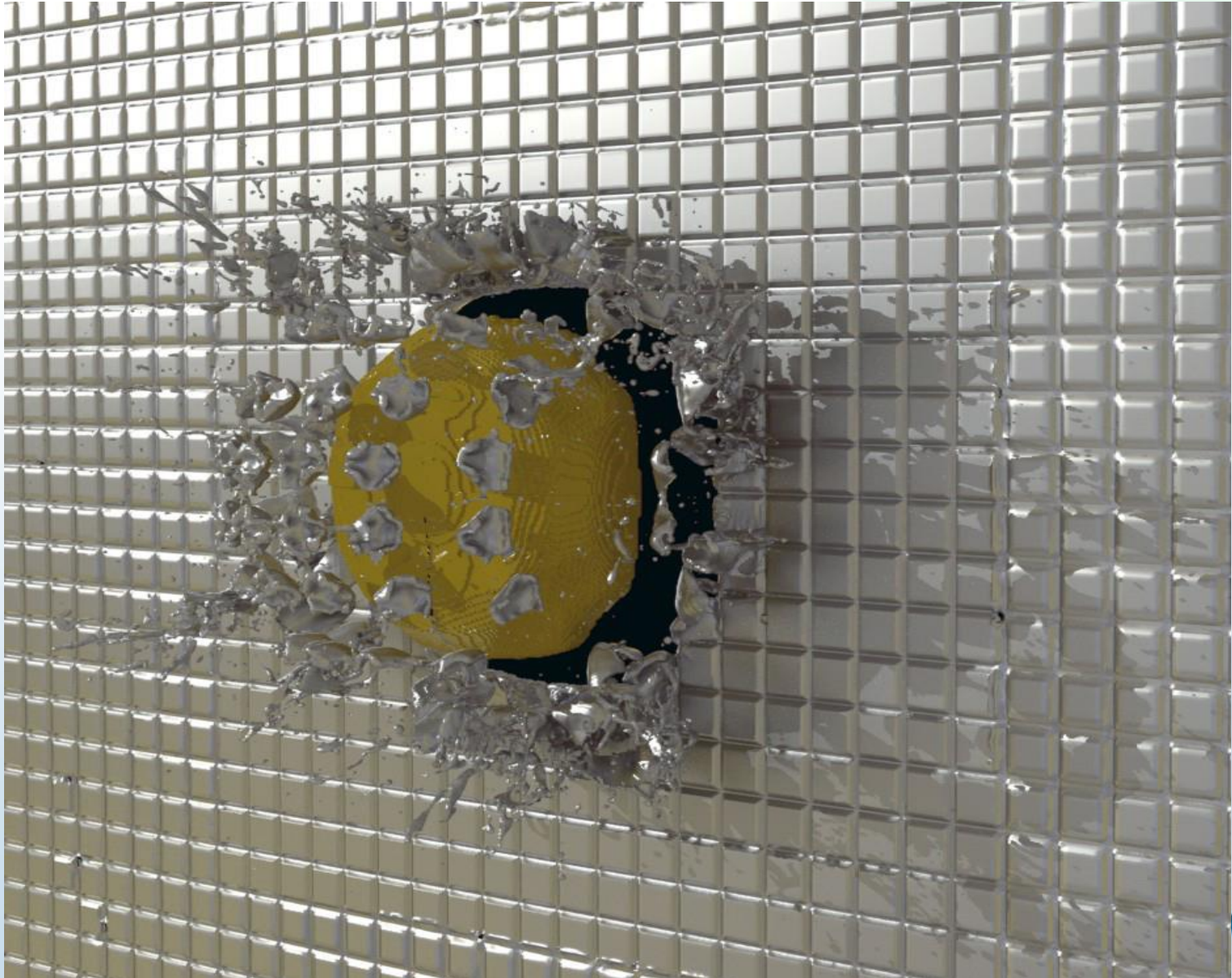
VirtualBox

- Use Jeff Mauldin account
 - Password is “l7oleW-r” (el seven oh el ee double-u dash are)
- Get terminal by clicking in upper-left and typing terminal
- Get examples with “wget https://github.com/acbauer/miniFE-2.0_ref_Catalyst/archive/tutorial.zip”
- May want to install emacs/vim
 - `sudo apt-get install [emacs23,vim]`

Online Help

- Catalyst User's Guide:
 - <http://paraview.org/Wiki/images/4/48/CatalystUsersGuide.pdf>
- Email list:
 - paraview@paraview.org
- Doxygen:
 - <http://www.vtk.org/doc/nightly/html/classes.html>
 - <http://www.paraview.org/ParaView3/Doc/Nightly/html/classes.html>
- Sphinx:
 - <http://www.paraview.org/ParaView3/Doc/Nightly/www/py-doc/index.html>
- Websites:
 - <http://www.paraview.org>
 - <http://catalyst.paraview.org>
- Examples:
 - <https://github.com/acbauer/CatalystExampleCode>

What is Catalyst?



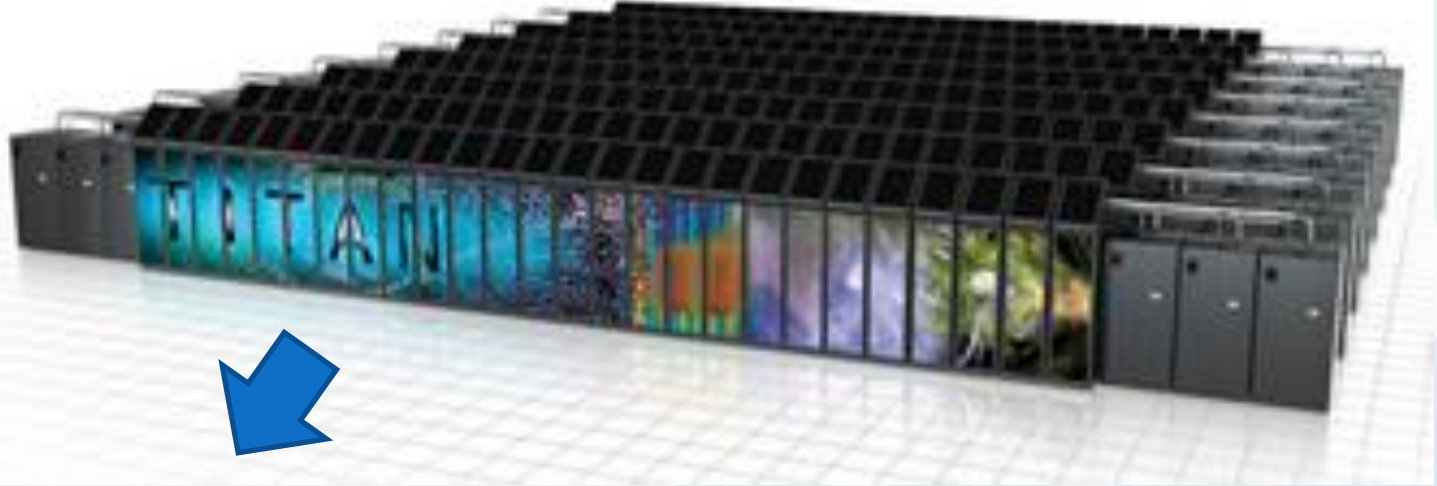
Why *In Situ*?

	2010	2018	Factor Change
System peak	2 Pf/s	1 Ef/s	500
Power	6 MW	20 MW	3
System Memory	0.3 PB	10 PB	33
Node Performance	0.125 Tf/s	10 Tf/s	80
Node Memory BW	25 GB/s	400 GB/s	16
Node Concurrency	12 cpus	1,000 cpus	83
Interconnect BW	1.5 GB/s	50 GB/s	33
System size (nodes)	20 K nodes	1 M nodes	50
Total Concurrency	225 K	1 B	4,444
Storage	15 PB	300 PB	20
Input/Output Bandwidth	0.2 TB/s	20 TB/s	100

DOE Exascale Initiative Roadmap, Architecture and Technology Workshop, San Diego, December, 2009.

Why *In Situ*?

Need a supercomputer to analyze results from a hero run



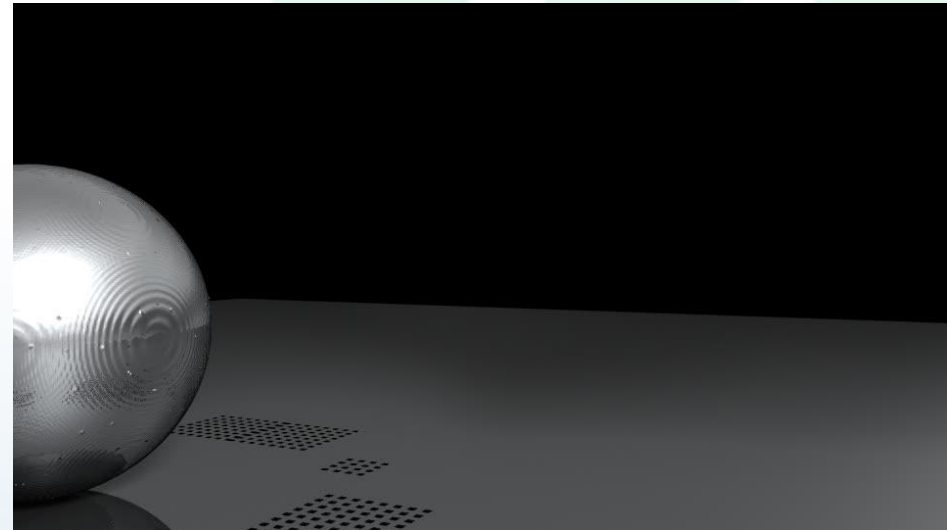
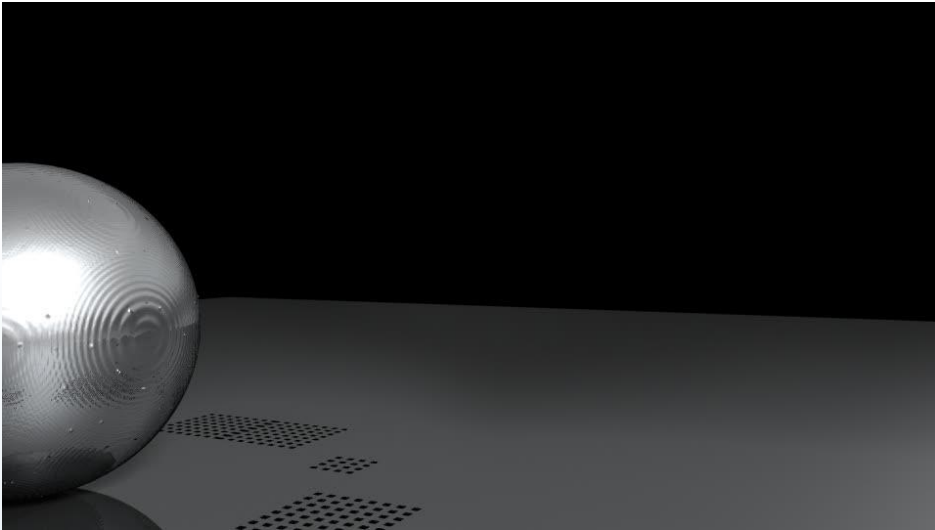
RHEA



2 orders of magnitude difference
between each level



Access to More Data



Dump
Times



Post-processing



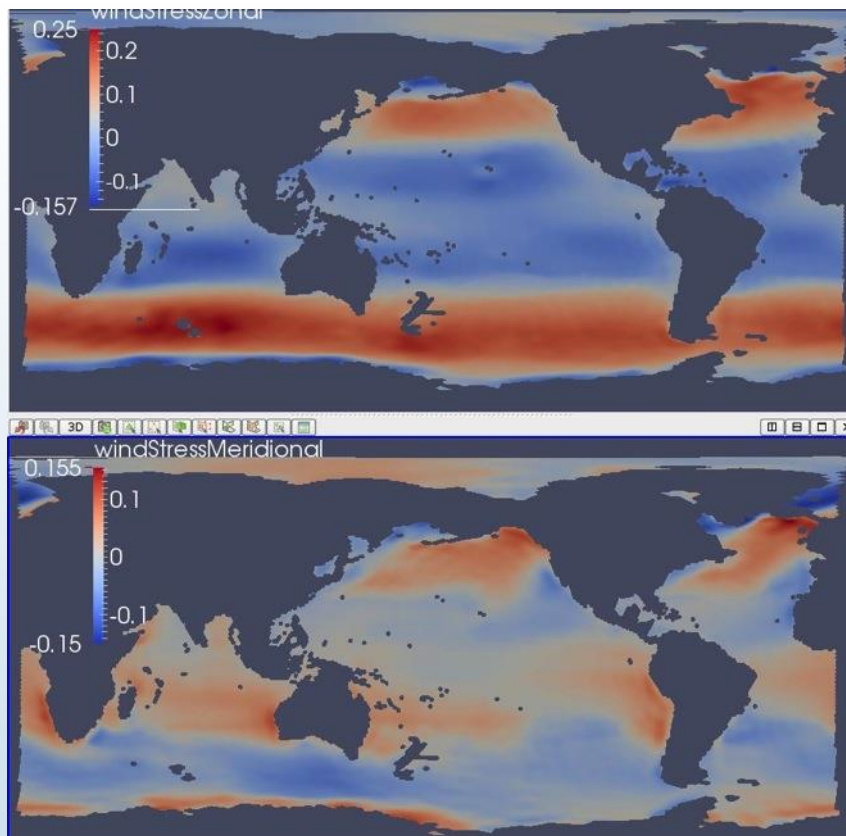
In situ processing

CTH (Sandia) simulation with roughly equal data stored at simulation time

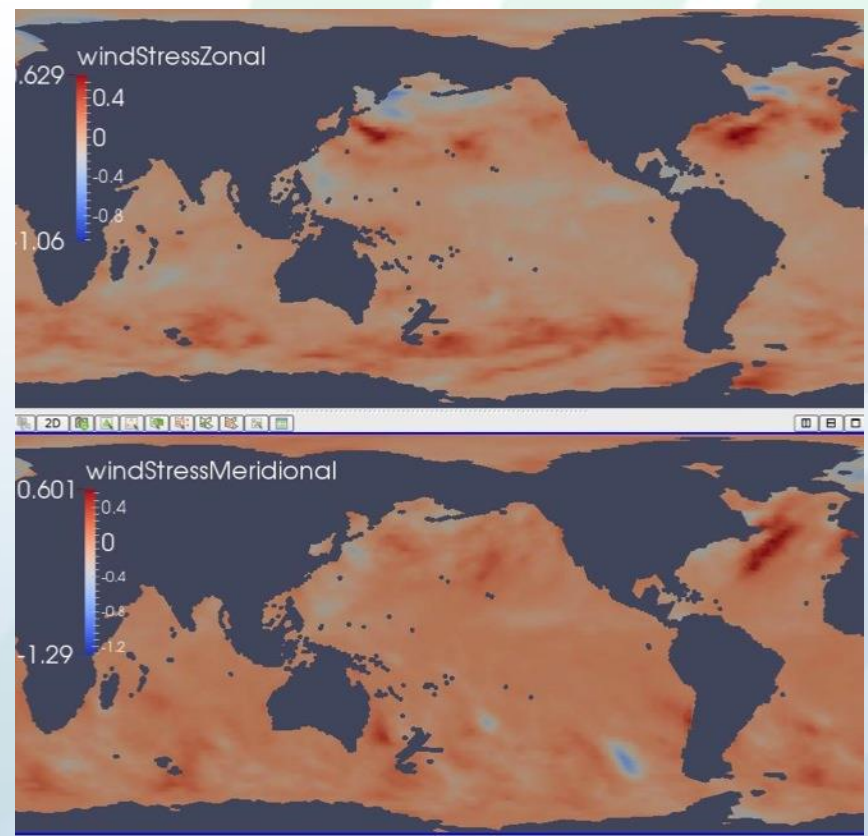
Reflections and shadows added in post-processing for both examples

Quick and Easy Run-Time Checks

Expected wind stress field at the surface of the ocean

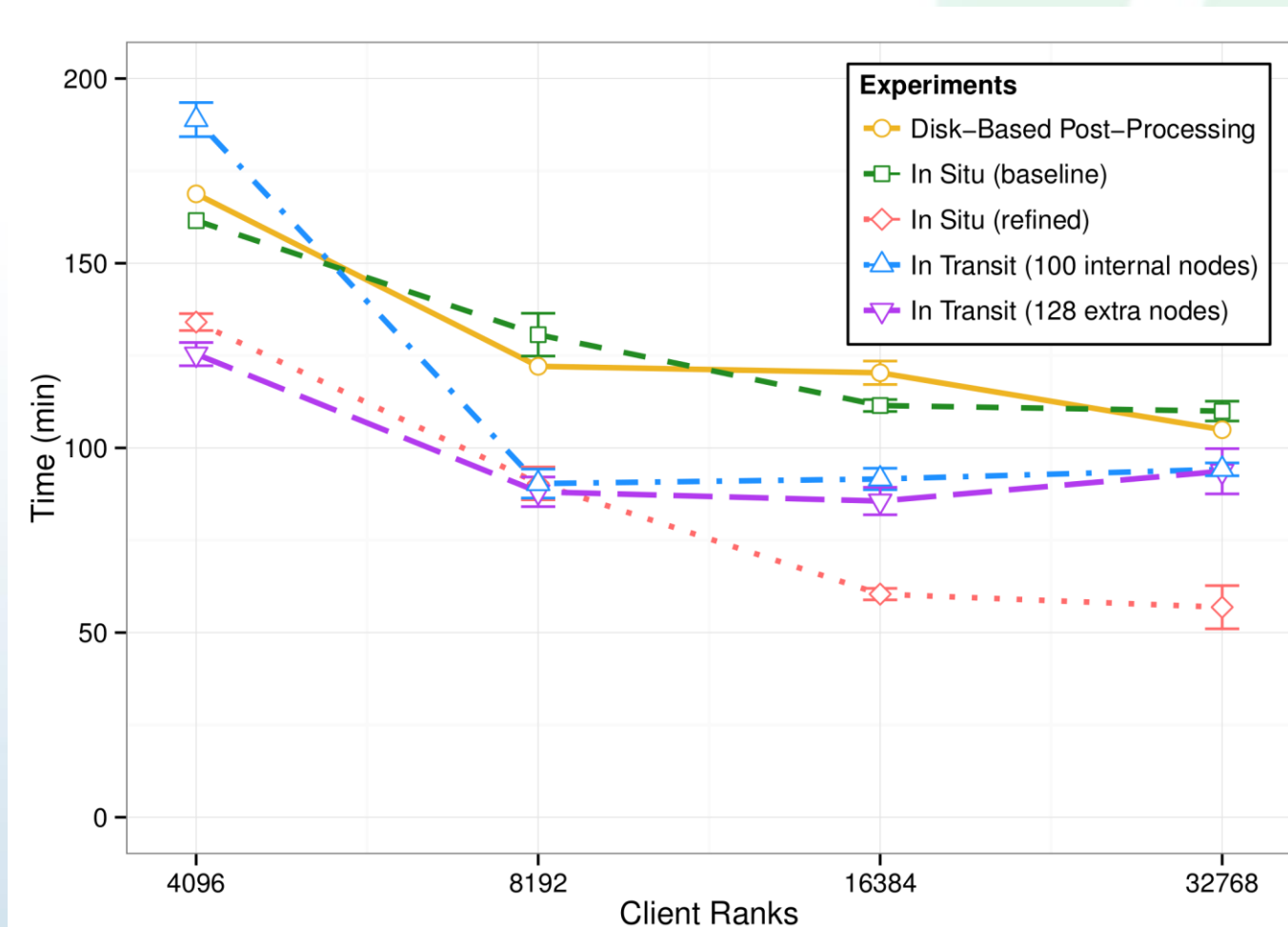


Wind stress in new run, quick glance indicates we are using wrong wind stress



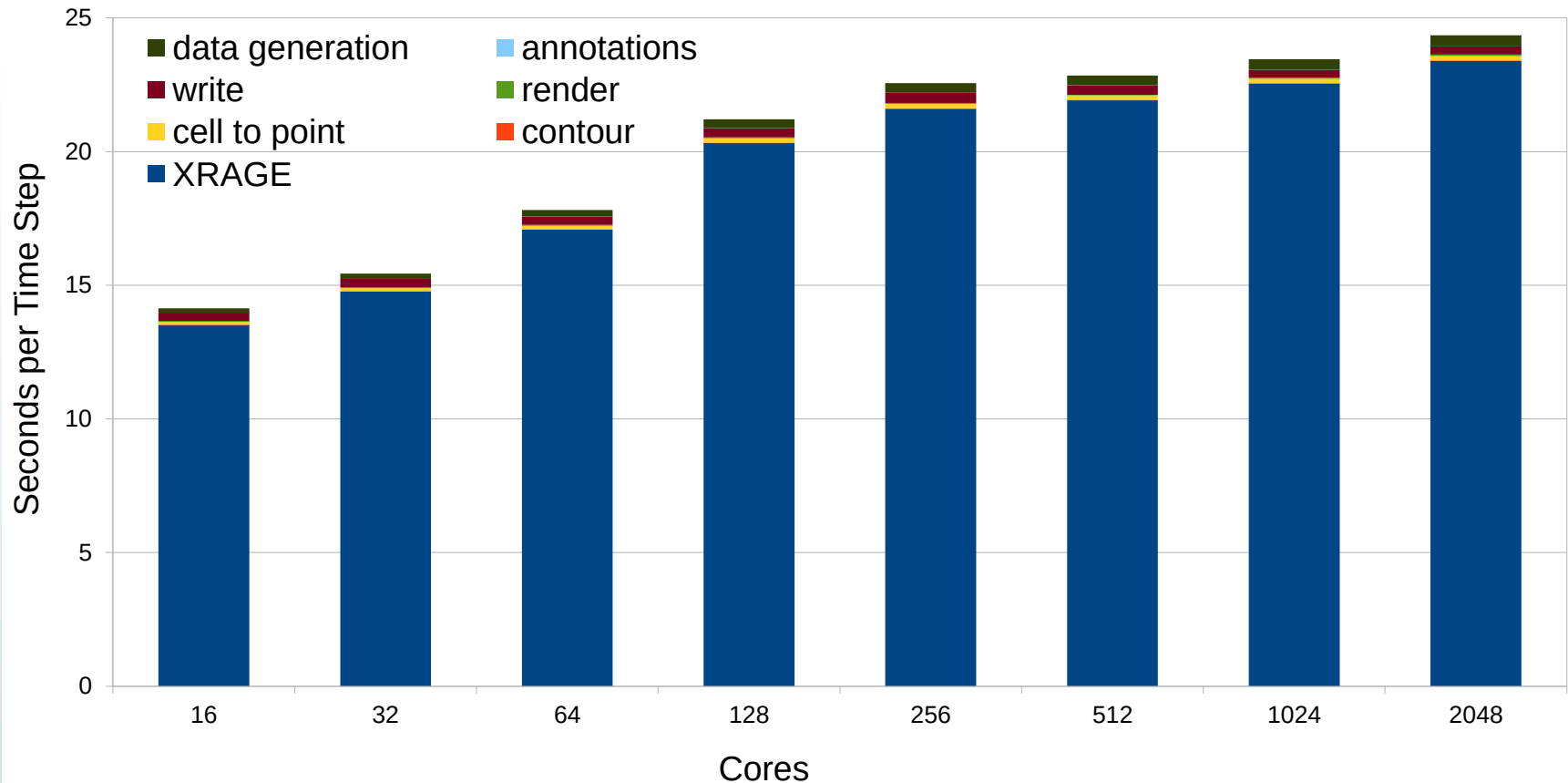
MPAS-O (LANL) simulation

Faster Time to Solution



CTH (Sandia) simulations comparing different workflows

Small Run-Time Overhead

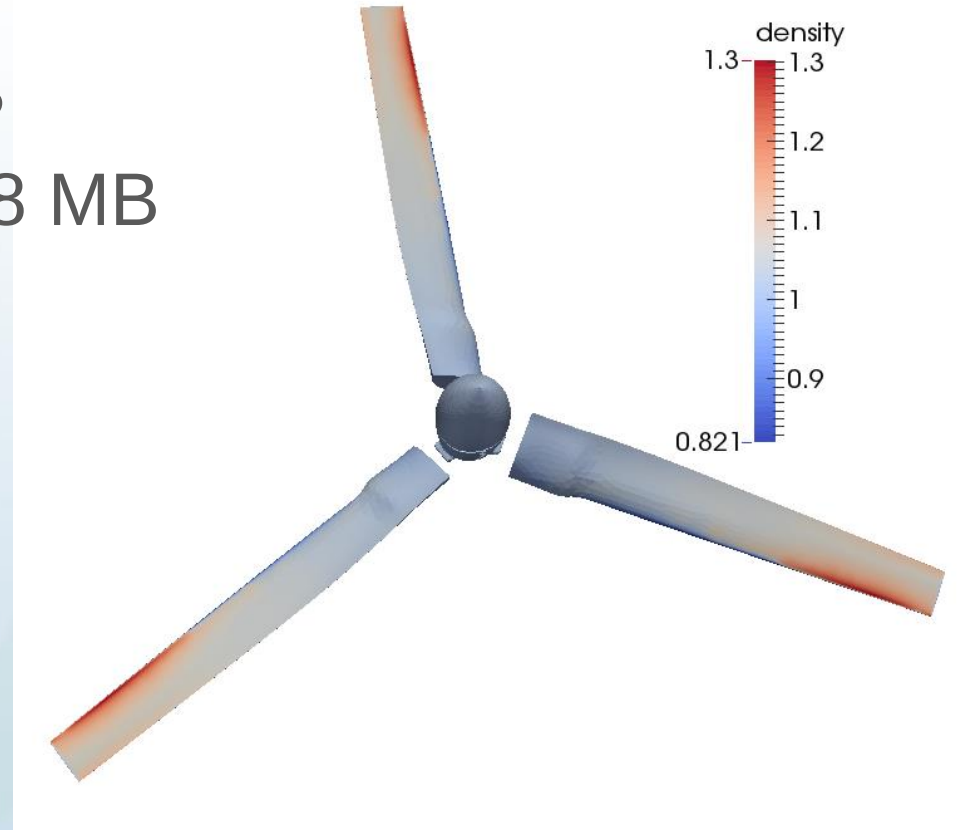


XRAGE (LANL) simulation

Reduced File Size

Rotorcraft simulation output size for a single time step

- Full data set – 448 MB
- Surface of blades – 2.8 MB
- Image – 71 KB

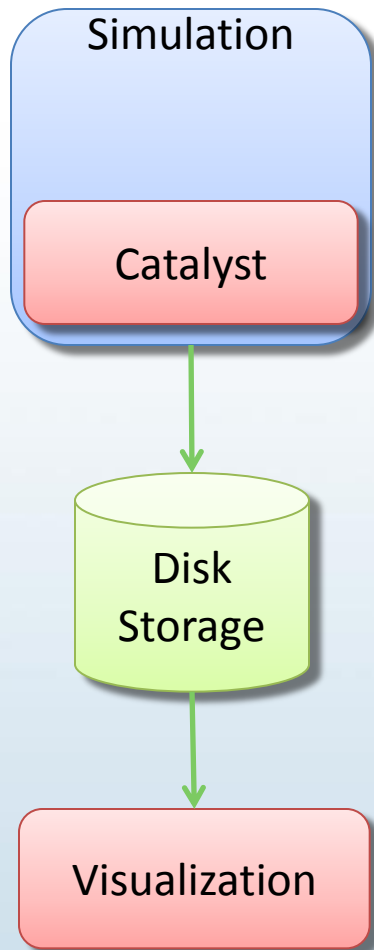


Reduced File IO Costs

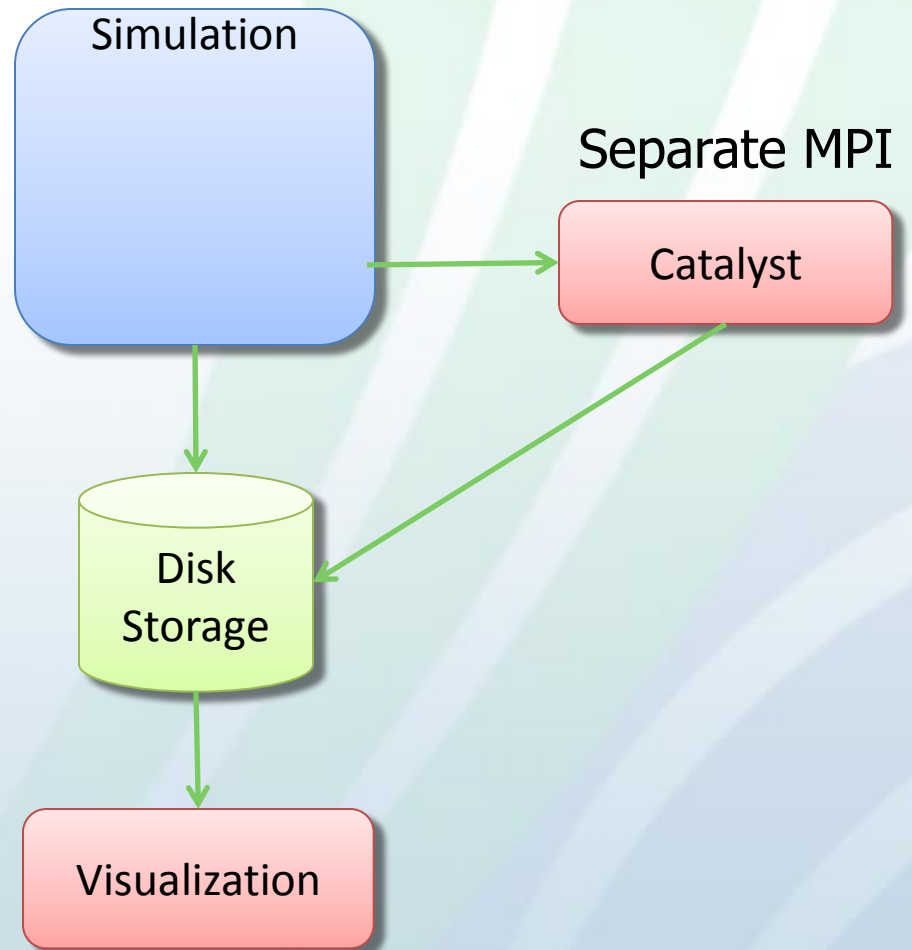
Time of Processing	Type of File	Size per File	Size per 1000 time steps	Time per File to Write at Simulation
Post	Restart	1,300 MB	1,300,000 MB	1-20 seconds
Post	Ensign Dump	200 MB	200,000 MB	> 10 seconds
<i>In Situ</i>	PNG	.25 MB	250 MB	< 1 second

XRAGE (LANL) simulation

Two Ways to Run in Batch



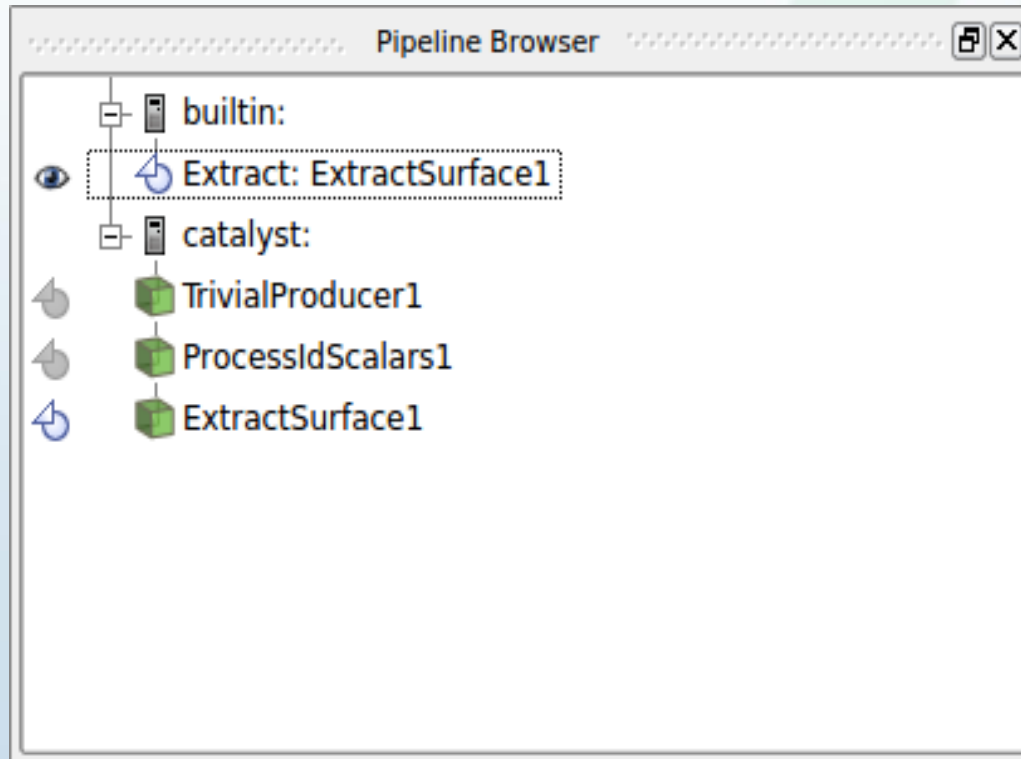
In Situ



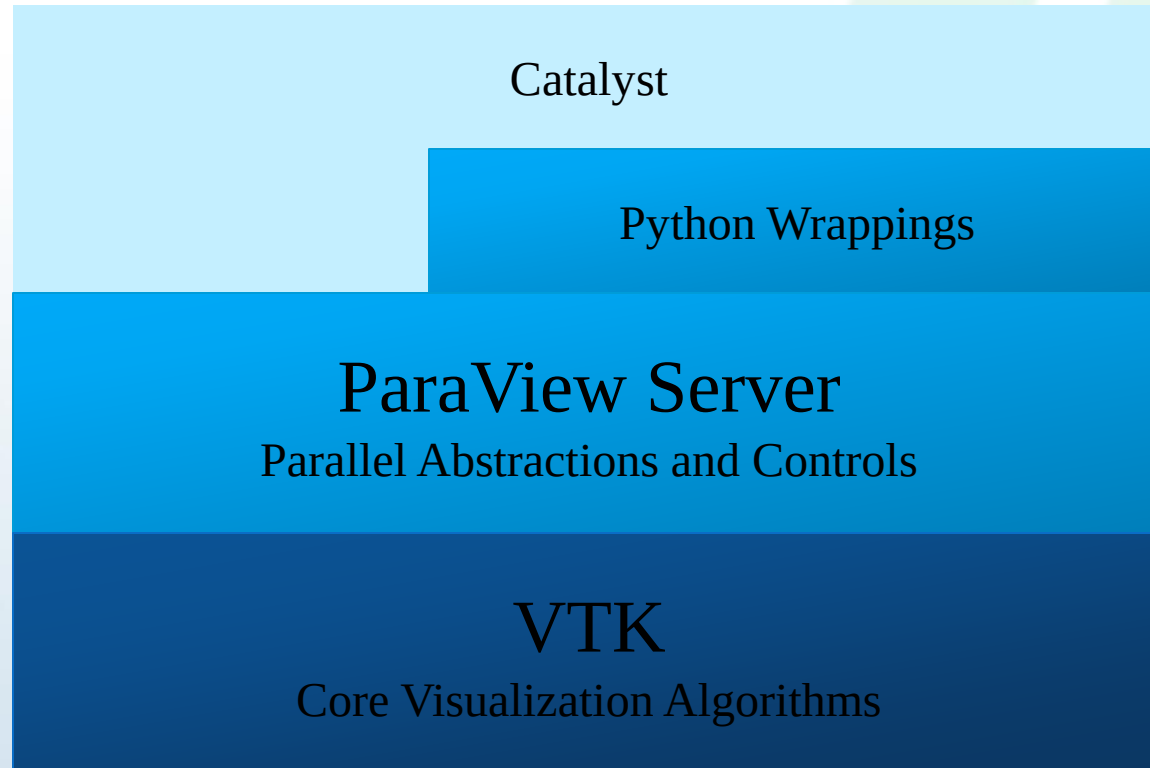
In Transit

Run Interactively

- Interactive to see current results from simulation
- Access to pipeline on Catalyst server and client



Catalyst Architecture



High Level View

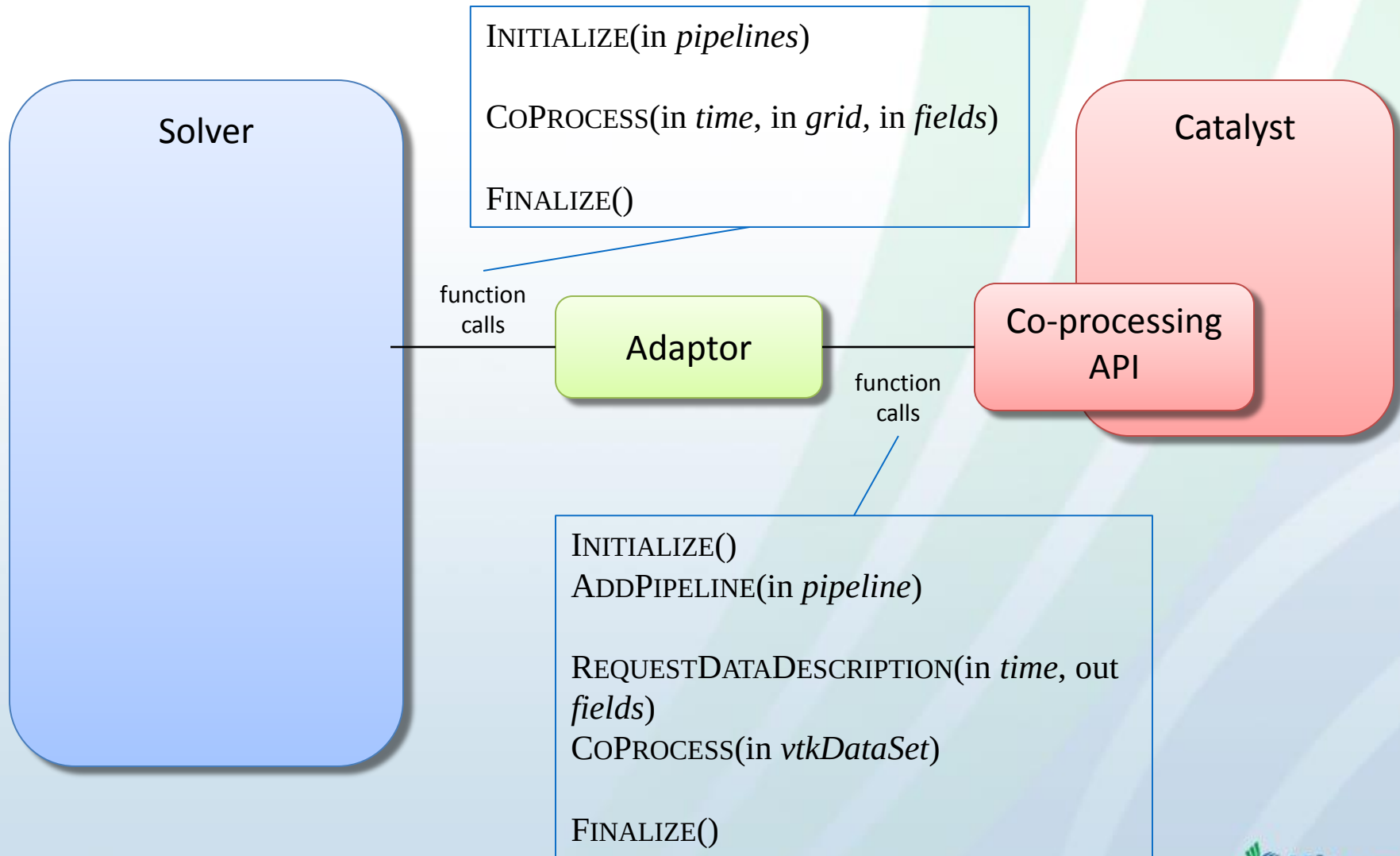
Simulation Developers

- Pass necessary simulation data to ParaView
- Need sufficient knowledge of both codes
 - VTK for grids and field data
 - ParaView Catalyst libraries
- Transparent to simulation users
- Extensible

Simulation Users

- Knowledge of ParaView as a post-processing/analysis tool
 - Basic interaction with GUI co-processing script generator plugin
 - Incremental knowledge increase to use the co-processing tools from basic ParaView use
- Programming knowledge can be useful to extend the tools

Developer Perspective

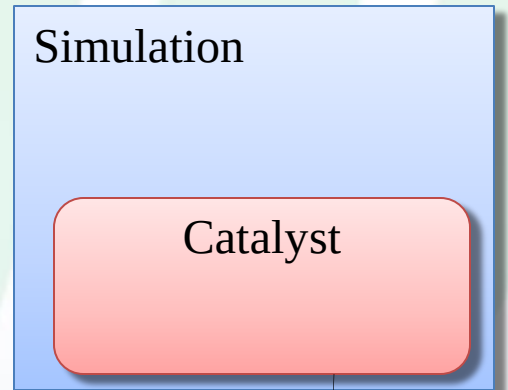


User Perspective

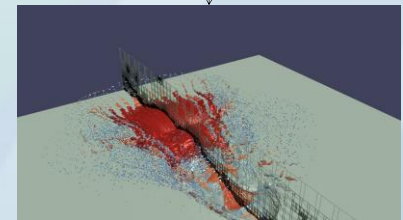
Simulation

Catalyst

User Perspective

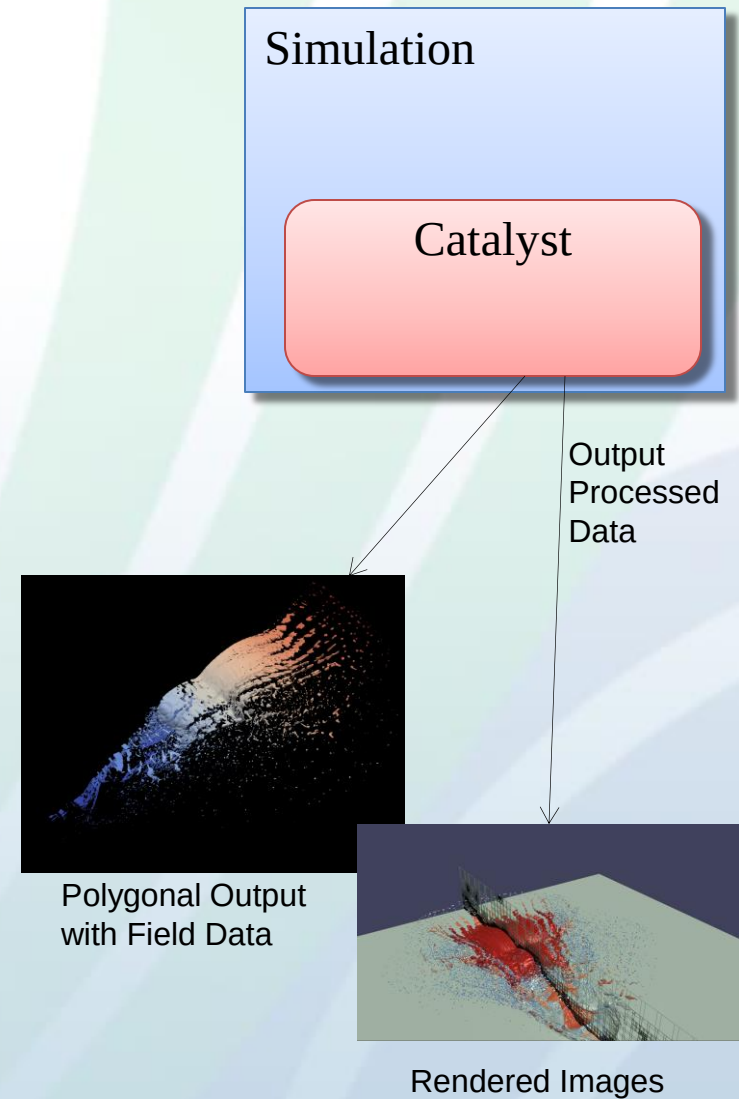


Output
Processed
Data

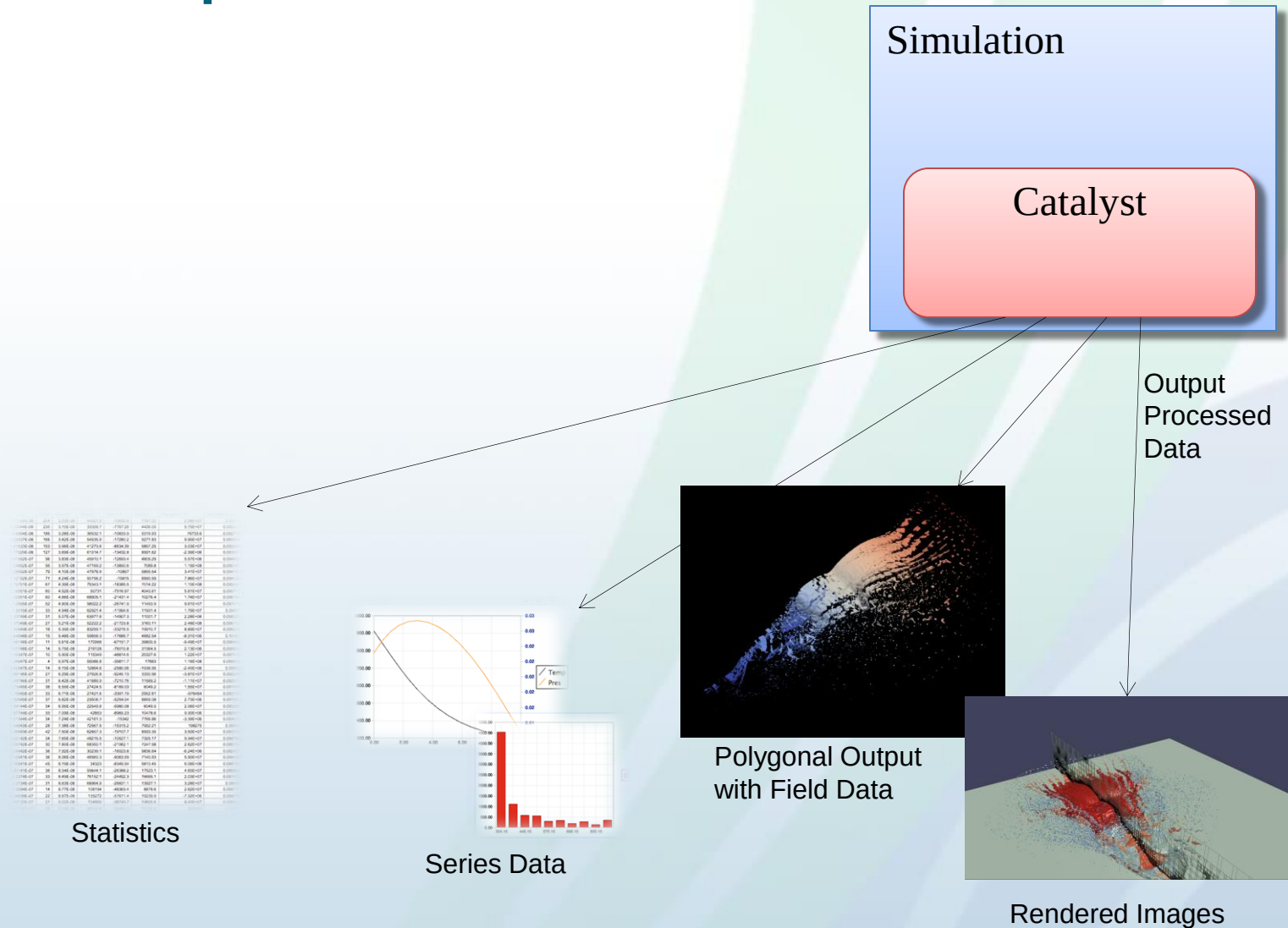


Rendered Images

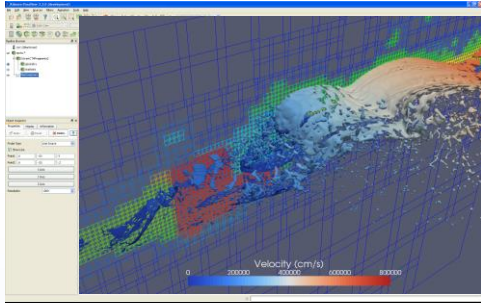
User Perspective



User Perspective



User Perspective



Script Export

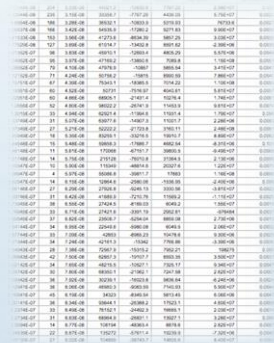
```
# Create the reader and set the filename.
reader = servermanager.sources.Reader(FileNames=path)
view = servermanager.CreateRenderView()
repr = servermanager.CreateRepresentation(reader, view)
reader.UpdatePipeline()
dataInfo = reader.GetDataInformation()
pDInfo = dataInfo.GetDataInformation()
arrayInfo = pDInfo.GetArrayInformation("displacements")
if arrayInfo:
    # get the range for the magnitude of displacement9
    range = arrayInfo.GetComponentRange(-1)
    lut = servermanager.rendering.PVLookupTable()
    lut.RGBPoints = [range[0], 0.0, 0.0, 1.0,
                    range[1], 1.0, 0.0, 0.0]
    lut.VectorMode = "Magnitude"
    repr.LookupTable = lut
    repr.ColorAttributeName = "displacement9"
    repr.ColorAttributeType = "POINT_DATA"
```

Augmented
script in
input deck.

Simulation

Catalyst

Output
Processed
Data



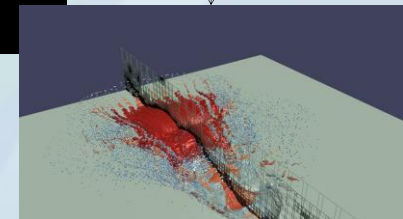
Statistics



Series Data

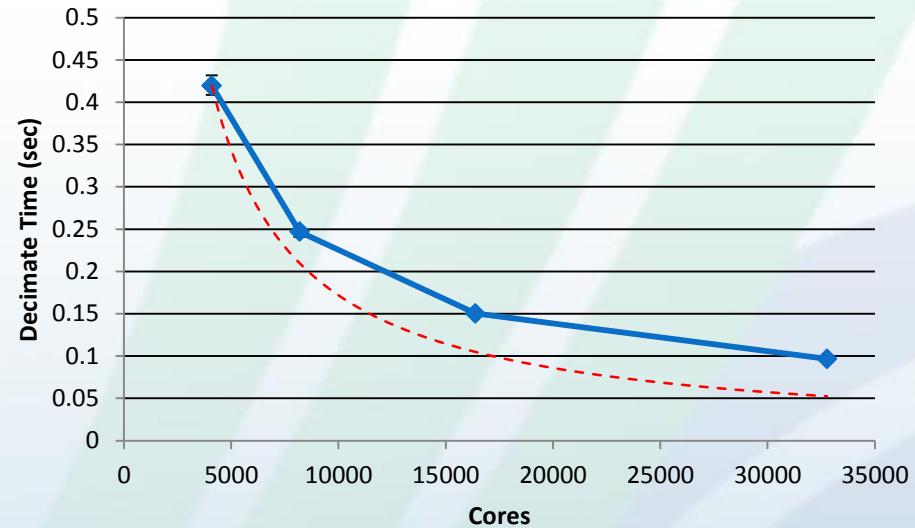
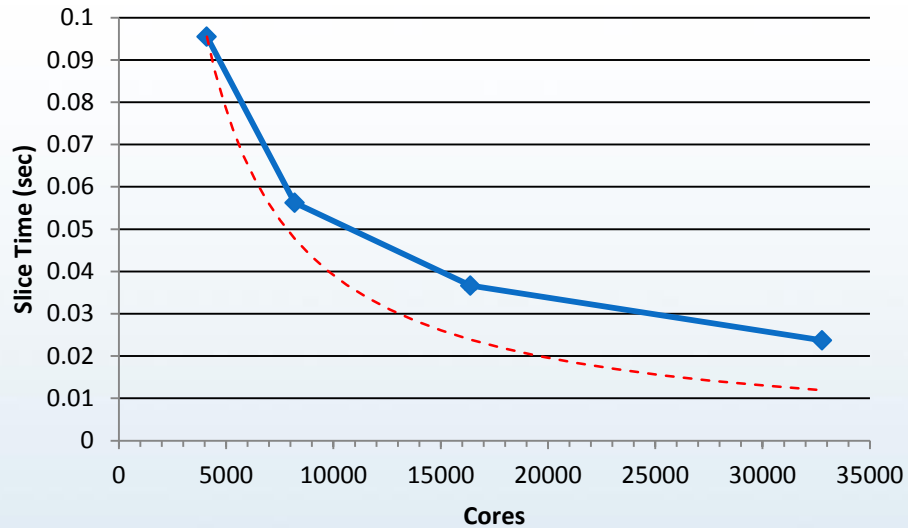


Polygonal Output with Field Data



Rendered Images

User & Developer Perspective





ParaView Catalyst for Simulation Developers

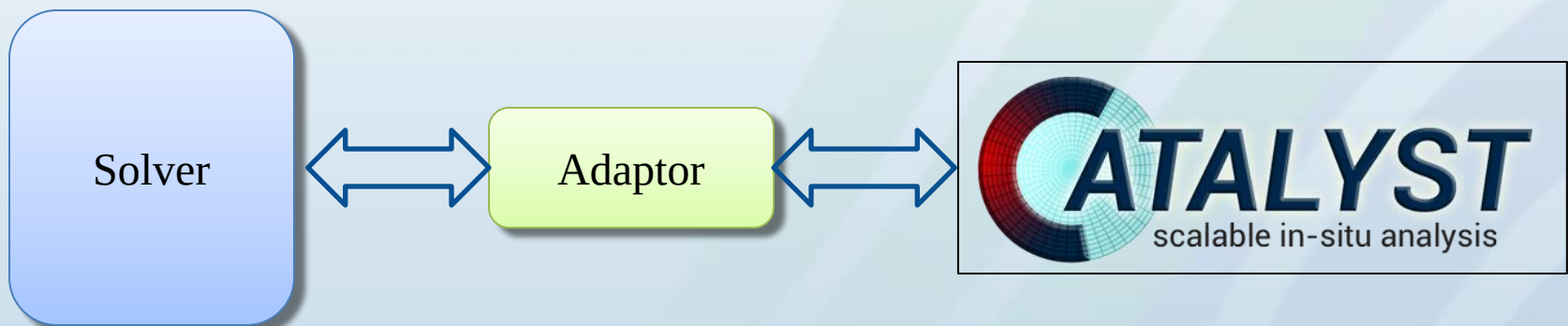


Developers Section Overview

- Most work is done in creating the adaptor between the simulation code and Catalyst
- Small footprint in main simulation code base
- Needs to be efficient both in memory and computationally
- Will work with examples based on miniFE-2.0 from Mantevo's set of mini-apps (mantevo.org)
 - Steady-state Poisson equation solver – instead of time stepping we'll examine CG iterative convergence of the field
 - Uses a Cartesian grid with nodal partitioning of the point data field

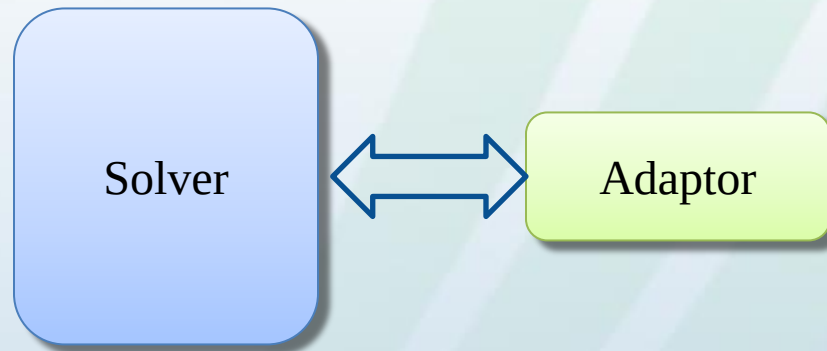
Interaction Overview

- Simulation has separate data structures from VTK data structures
- Use an adaptor to bridge the gap
 - Try to reuse existing memory
 - Also responsible for other interactions between simulation code and Catalyst



Interaction Overview

- Functional interface – typically 3 calls between simulation code and adaptor
 - Initialize
 - CoProcess
 - Finalize



Adaptor Overview

- Creates VTK data objects representing simulation data
- Creates Catalyst pipelines
 - Information on how to process VTK data objects to get desired output

Adaptor

Finalize Overview

- Last chance to do something
- Clean up any allocated memory

Information Needed By Adaptor During Initialization

- What pipelines to use?
 - Canned pipelines – pre-built pipelines that take in a few parameters (e.g. slice output with output file name, frequency and slice normal and origin)
 - Python pipelines – generated in the ParaView GUI and potentially edited
- Optionally
 - Information to construct grids and fields

Information Needed By Adaptor During CoProcess

- Time, time step
- Optionally
 - Information to construct grids and fields if not passed in during initialization step
 - Force all pipelines to execute (simulation may know something important has happened)
 - Other information to add in extra logic to pipelines

VTK Preliminaries

- Many objects in VTK are reused so we use reference counting to keep track of how many other objects are keeping using them
 - New() – creates a new object in dynamic memory and sets reference count to 1
 - Delete() – reduces reference count by 1 and if reference count goes to 0 then object gets deleted
 - a->Set*(b), a->Add*(b) – b's reference count gets increased by 1

```
vtkDoubleArray* a = vtkDoubleArray::New(); // a's ref count = 1
vtkPointData* pd = vtkPointData::New() ; // pd's ref count = 1
pd->AddArray(a); // a's ref count = 2
a->Delete(); // a's ref count = 1
a->SetName("an array"); // valid as a hasn't been deleted
pd->Delete(); // deletes both pd and a
```

Helpful VTK Objects

- `vtkSmartPointer<vtkObject*>`
 - Deals with creating and deleting `vtkObjects`
 - Increases reference count when set
 - Reduces reference count when it gets deleted

```
vtkPointData* pd = vtkPointData::New() ; // pd's ref count = 1
{
    vtkSmartPointer<vtkDoubleArray> a =
        vtkSmartPointer<vtkDoubleArray>::New();
    pd->AddArray(a);
}
// a no longer exists but the generated vtkDoubleArray does
vtkSmartPointer<vtkPointData> pd2 = pd; // pd's ref count = 2
```

Helpful VTK Objects (2)

- `vtkNew<vtkObject*>`
 - Deals with creating and deleting `vtkObjects`
 - Reduces reference count when it gets deleted
 - Need `GetPointer()` when a method needs a `vtkObject`

```
vtkPointData* pd = vtkPointData::New() ; // pd's ref count = 1
{
    vtkNew<vtkDoubleArray> a;
    pd->AddArray(a.GetPointer()); // need GetPointer() here and not just a
}
// a no longer exists but the generated vtkDoubleArray does
```


Writing the Initialization Method

- Assuming we will pass in data to create VTK grid and field data objects during CoProcess call
- ParaView classes to look at:
 - vtkCPPProcessor – main manager object
 - vtkCPPipeline/vtkCPPythonScriptPipeline – object to control Catalyst pipeline (typically one object per pipeline)

<http://www.paraview.org/ParaView3/Doc/Nightly/html/classvtkCPProcessor.html>

<http://www.paraview.org/ParaView3/Doc/Nightly/html/classvtkCPPythonScriptPipeline.html>

vtkC++Processor Initialization Step

- vtkC++Processor
 - vtkC++Processor* New()
 - Static constructor method that sets reference count to 1
 - void Delete()
 - Reduce reference count by 1 (not necessarily delete though)
 - int Initialize()
 - Returns 1 for success and 0 for failure
 - int AddPipeline(vtkC++Pipeline* pipeline)
 - Pass in a vtkC++Pipeline object, vtkC++Processor will keep reference to pipeline so it won't get deleted
 - Returns 1 for success and 0 for failure
- Will go through other important vtkC++Processor methods later

vtkCPPProcessor Initialization Step

For **ParaView 4.2**, can initialize Catalyst with a separate MPI communicator

- Use “int Initialize(vtkMPICommunicatorOpaqueComm*)” instead of “int Initialize()”
- vtkMPICommunicatorOpaqueComm defined in vtkMPI.h
- Use constructor
vtkMPICommunicatorOpaqueComm(MPI_Comm*)

vtkCPPipeline Initialization Step

- vtkCPPipeline
 - Abstract class – methods are usually only called by vtkCPProcessor
- vtkCPPythonScriptPipeline
 - Concrete class that derives from vtkCPPipeline
 - vtkCPPythonScriptPipeline* New()
 - Static constructor method that sets reference count to 1
 - int Initialize(const char *fileName)
 - fileName is the Python script pipeline which can include a relative or absolute path
 - Returns 1 for success and 0 for failure

vtkCPProcessor Finalize Step

- vtkCPProcessor
 - int Finalize()
 - Delete the vtkCPProcessor object
- If ParaView was built with DEBUG_LEAKS_ON, will get messages that VTK objects were not deleted before exiting.
 - Most likely from not deleting vtkCPProcessor, vtkCPipeline or VTK objects created in the adaptor.

Configuring with CMake

- Configuring the build is simplified by using CMake
 - Determines dependencies automatically for compiling (location of header files) and linking (location of libraries)
- Typically use ccmake for terminal-based interface
 - Often building on a remote machine
 - Command: `ccmake <path to source dir> <preset configuration options>`
- Create build directory at `<miniFE-2.0_ref>/adaptor_build`

Example 1 – Write It!

- Assume all Python scripts for pipeline specifications
- Starting file:
 - <miniFE-2.0_ref>/adaptor/catalyst_adapter.cpp
 - Function signatures:
 - `int initialize(miniFE::Parameters& params)`
 - Python script names in `params.script_names` which is type `std::vector<std::string>`
 - Processor in anonymous namespace
 - `int finalize()`
- CMake configuration step in `adapter_build`:
 - `ccmake ../adapter -DParaView_DIR:PATH=<location of ParaView build>`

Example 1 – A Solution

```
void initialize(miniFE::Parameters& params)
{
    Processor = vtkCPPProcessor::New();
    Processor->Initialize();
    for(std::vector<std::string>::const iterator
        it=params.script_names.begin();
        it!=params.script_names.end();it++) {
        vtkCPPythonScriptPipeline* pipeline =
            vtkCPPythonScriptPipeline::New();
        pipeline->Initialize(it->c_str());
        Processor->AddPipeline(pipeline);
        pipeline->Delete();
    }
}
```

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


Writing the CoProcess Method

- Assuming we will pass in data to create VTK grid and field data objects during CoProcess call
- ParaView classes to look at:
 - vtkCPPProcessor – main manager object
 - vtkCPDataDescription

Writing the CoProcess Method

Main stage with 2 stages

- First stage
 - Check with Catalyst pipelines to see if there is anything to do
 - Should check after every time step
 - Should be very fast
- Second stage
 - Only done if at least one Catalyst pipeline wants to do something this time step
 - Construct needed input (i.e. VTK grids and fields)
 - Execute the pipelines that need to do something

Writing the CoProcess Method: First Stage

- ParaView classes to look at:
 - vtkCPPProcessor – main manager object
 - vtkCPDataDescription – meta-description of what simulation is able to provide

vtkCPProcessor CoProcess Step: First Stage

- vtkCPProcessor
 - int RequestDataDescription(vtkCPDataDescription*)
 - Have the processor check if any Catalyst pipelines need to execute this time step
 - Returns 1 for success and 0 for failure
- vtkCPDataDescription
 - void AddInput(const char* name)
 - Specify source of pipelines
 - Convention for single input is name="input"
 - void SetTimeData(double time, vtkIdType timeStep)
 - Set the current simulation time and time step
 - void ForceOutputOn()/SetForceOutput() – optional
 - Specify that all Catalyst pipelines should execute

Example 2 – Write It!

- Function signature:
 - `void coprocess(const double spacing[3], const Box& global_box, const Box& local_box, std::vector<double>& minifepointdata, int time_step, double time, bool force_output);`
- Starting file:
 - `<miniFE-2.0_ref>/adaptor/catalyst_adapter.cpp`
 - Only need to use `time_step`, `time` and `force_output` for this part

Example 2 – A Solution

```
void coprocess(const double spacing[3], const Box& global_box,
    const Box& local_box, std::vector<double>& minifepointdata,
    int time_step, double time, bool force_output)
{
    vtkSmartPointer<vtkCPDataDescription> dataDescription =
        vtkSmartPointer<vtkCPDataDescription>::New();
    dataDescription->AddInput("input");
    dataDescription->SetTimeData(time, time_step);
    dataDescription->SetForceOutput(force_output);
    if(Processor->RequestDataDescription(dataDescription) == 0)
    {
        return;
    }
    ...
}
```

Writing the CoProcess Method: Second Stage

- ParaView classes to look at:
 - vtkCPPProcessor – main manager object
 - vtkCPDataDescription – meta-description of what simulation is able to provide
 - vtkCPInputDataDescription – Catalyst description of what it needs in order to execute a pipeline

vtkCPProcessor CoProcess Step: Stage Two

- vtkCPProcessor
 - int CoProcess(vtkCPDataDescription*)
 - Have the coprocessor execute the desired Catalyst pipelines
 - Returns 1 for success and 0 for failure
- vtkCPDataDescription
 - vtkCPInputDataDescription*
GetInputDescriptionByName(const char* name)
 - Given an input name (“input”), get the input description object to determine what is needed
- vtkCPInputDataDescription
 - void SetGrid(vtkDataObject*)
 - Set the VTK data source object for a Catalyst pipeline

Example 3 – Write It!

- Still using function signature:
 - `void coprocess(const double spacing[3], const Box& global_box, const Box& local_box, std::vector<double>& minifepointdata, int time_step, double time, bool force_output);`
 - Create a `vtkImageData` object that we'll set up later

Example 3 – A Solution

```
void coprocess(const double spacing[3], const Box& global_box,
    const Box& local_box, std::vector<double>& minifepointdata,
    int time_step, double time, bool force_output)
{
    vtkSmartPointer<vtkCPDataDescription> dataDescription =
        vtkSmartPointer<vtkCPDataDescription>::New();
    dataDescription->AddInput("input");
    dataDescription->SetTimeData(time, time_step);
    dataDescription->SetForceOutput(force_output);
    if(Processor->RequestDataDescription(dataDescription) == 0)
    {
        return;
    }
    vtkNew<vtkImageData> grid;
    dataDescription->GetInputDescriptionByName("input")->SetGrid(grid.GetPointer());
    ...
}
```

Writing the CoProcess Method: Second Stage

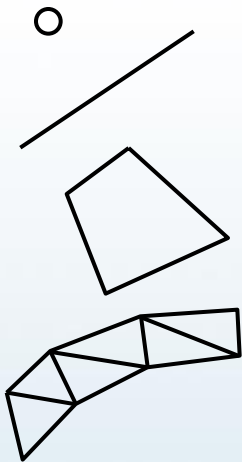
- VTK classes to look at:
 - Depends on simulation's grid type
 - Classes that derive from:
 - vtkDataObject for grids
 - vtkAbstractArray for fields
 - vtkFieldData for managing vtkAbstractArrays
 - Other useful classes:
 - vtkCellType – types of cells for unstructured grids (vtkPolyData and vtkUnstructuredGrid)
- Can look at readers or sources that provide the same type of VTK data object

Getting Data Into Catalyst

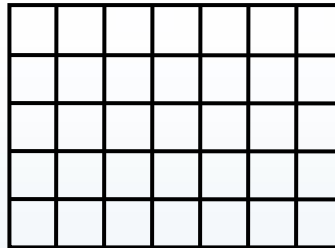
- Main object will derive from vtkDataObject
 - Grids that derive from vtkDataSet
 - Multiblock grids that contain multiple vtkDataSets
- Field (aka attribute) information
 - Point data – information specified for each point in a grid
 - Cell data – information specified for each cell in a grid
 - Field data – meta-data not associated with either points or cells
- All object groups are 0-based/C/C++ indexing

vtkDataSet Subclasses

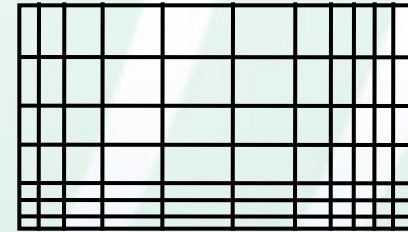
vtkPolyData



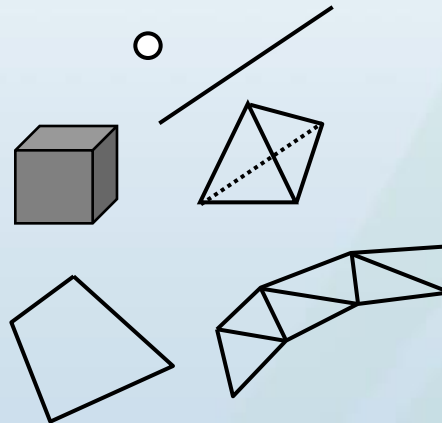
vtkImageData
vtkUniformGrid



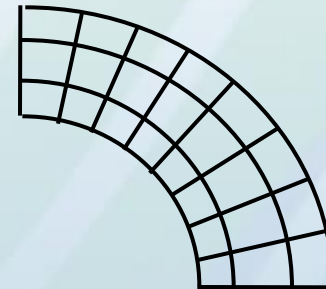
vtkRectilinearGrid



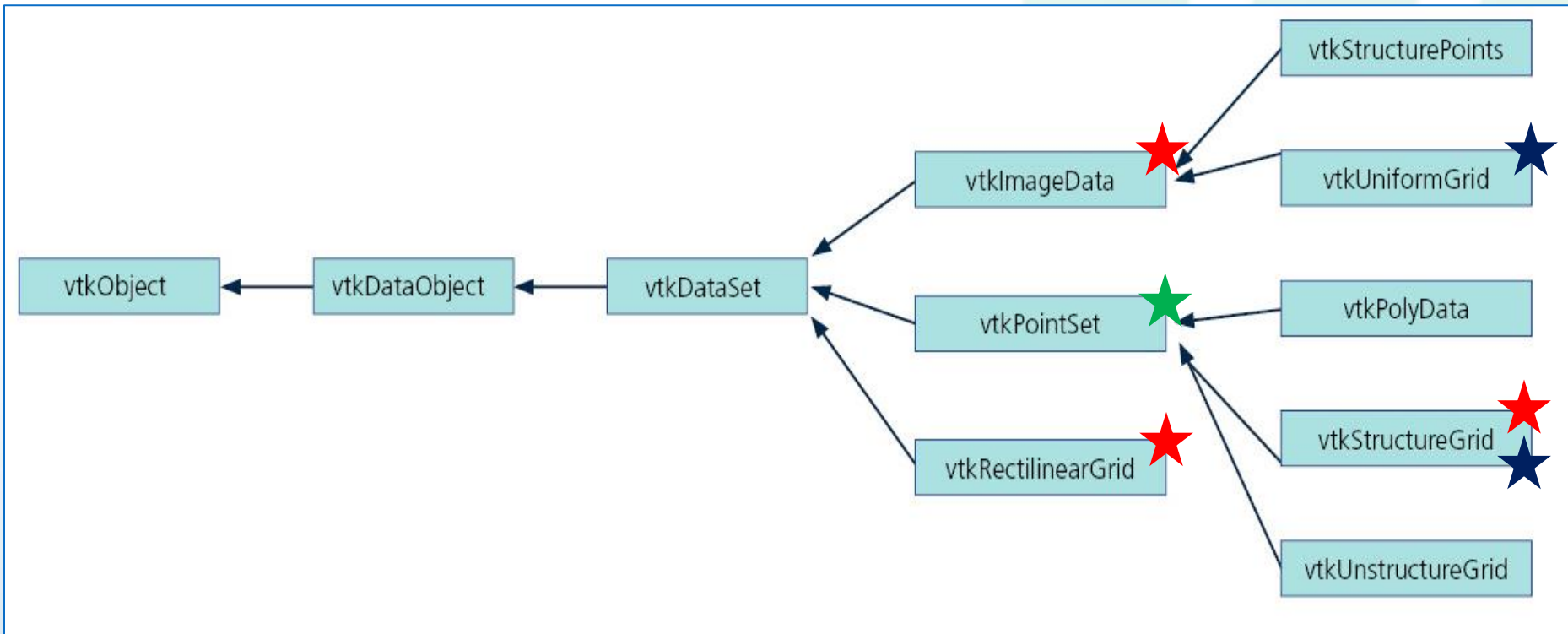
vtkUnstructuredGrid



vtkStructuredGrid



vtkDataSet Class Hierarchy



★ Topologically regular grid

★ Irregular geometry

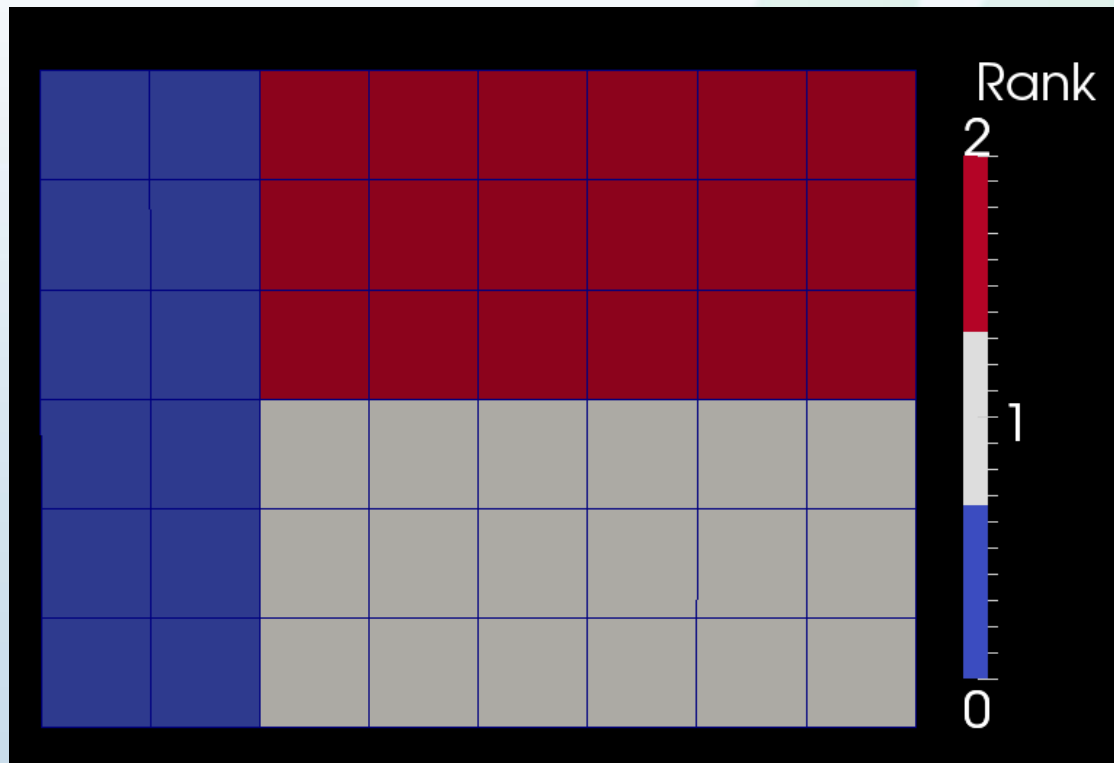
★ Supports blanking

Topologically Regular Grids

- `vtkImageData/vtkUniformGrid`, `vtkRectilinearGrid` and `vtkStructuredGrid`
- Topological structure defined by whole extent
 - Gives first and last point in each logical direction
 - Not required to start at 0
- Partitioning defined by extents
 - First and last point in each logical direction of part of the entire grid (`xmin`, `xmax`, `ymin`, `ymax`, `zmin`, `zmax`)
- Ordering of points and cells is fastest in logical x-direction, then y-direction and slowest in z-direction

Extents

- Whole extent for all processes (0, 8, 0, 6, 0, 0)
- Extents different for each process
 - Rank 0: (0, 2, 0, 6, 0, 0), 21 points, 12 cells
 - Rank 1: (2, 8, 0, 3, 0, 0), 28 points, 18 cells
 - Rank 2: (2, 8, 3, 6, 0, 0), 28 points, 18 cells



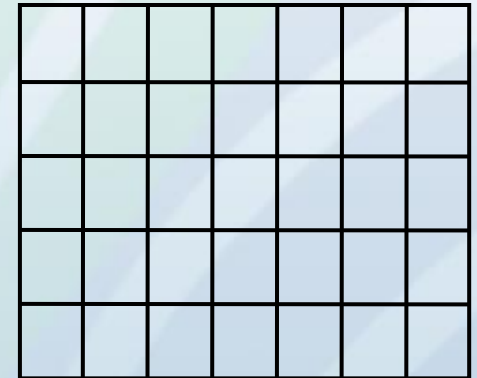
Extents

- Point and cell indices for extent of (2, 8, 3, 6, 0, 0)
 - From rank 2 in previous slide
 - 28 points and 18 cells

21 (2,6,0)						27 (8,6,0)
12 (2,5,0)	13 (3,5,0)	14 (4,5,0)	15 (5,5,0)	16 (6,5,0)	17 (7,5,0)	
6 (2,4,0)	7 (3,4,0)	8 (4,4,0)	9 (5,4,0)	10 (6,4,0)	11 (7,4,0)	
0 (2,3,0)	1 (3,3,0)	2 (4,3,0)	3 (5,3,0)	4 (6,3,0)	5 (7,3,0)	
0 (2,3,0)						6 (8,3,0)

vtkImageData/vtkUniformGrid

- `vtkCPInputDataDescription::SetWholeExtent()` – total number of points in each direction
- `SetExtent()` – a process's part of the whole extent
- `SetSpacing()` – cell lengths in each direction (same for all processes)
- `SetOrigin()` – location of point at $i=0, j=0, k=0$ (same for all processes)
- `vtkUniformGrid`
 - Supports cell blanking
 - Currently written to file as `vtkImageData`



Example 4 – Write It!

- Still using function signature:
 - `void coprocess(const double spacing[3], const Box& global_box, const Box& local_box, std::vector<double>& minifepointdata, int time_step, double time, bool force_output);`
 - Local extent is contained in `local_box`:
 - Points range from `local_box[0][0]` to `local_box[0][1]` in X-direction, `local_box[1][0]` to `local_box[1][1]` in Y-direction, `local_box[2][0]` to `local_box[2][1]` in Z-direction
 - Global extent is contained in `global_box`:
 - Points range from `global_box[0][0]` to `global_box[0][1]` in X-direction, `global_box[1][0]` to `global_box[1][1]` in Y-direction, `global_box[2][0]` to `global_box[2][1]` in Z-direction

Example 4 – A Solution

```
...  
vtkNew<vtkImageData> grid;  
int extent[6] = {local_box[0][0], local_box[0][1], local_box[1][0],  
                local_box[1][1], local_box[2][0], local_box[2][1]};  
grid->SetExtent(extent);  
grid->SetSpacing(spacing[0], spacing[1], spacing[2]);  
grid->SetOrigin(0, 0, 0);  
dataDescription->GetInputDescriptionByName("input")->SetGrid(grid.GetPointer());  
int wholeExtent[6] = {global_box[0][0], global_box[0][1], global_box[1][0],  
                    global_box[1][1], global_box[2][0], global_box[2][1]};  
dataDescription->GetInputDescriptionByName("input")->  
    SetWholeExtent(wholeExtent);  
...
```

vtkRectilinearGrid

- `vtkCPInputDataDescription::SetWholeExtent()` – total number of points in each direction
- `SetExtents()` – a process's part of the whole extent
- `Set<X,Y,Z>Coordinates()` – point coordinates in each direction
 - Only values for process's extents
 - Index starting at 0



vtkPointSet

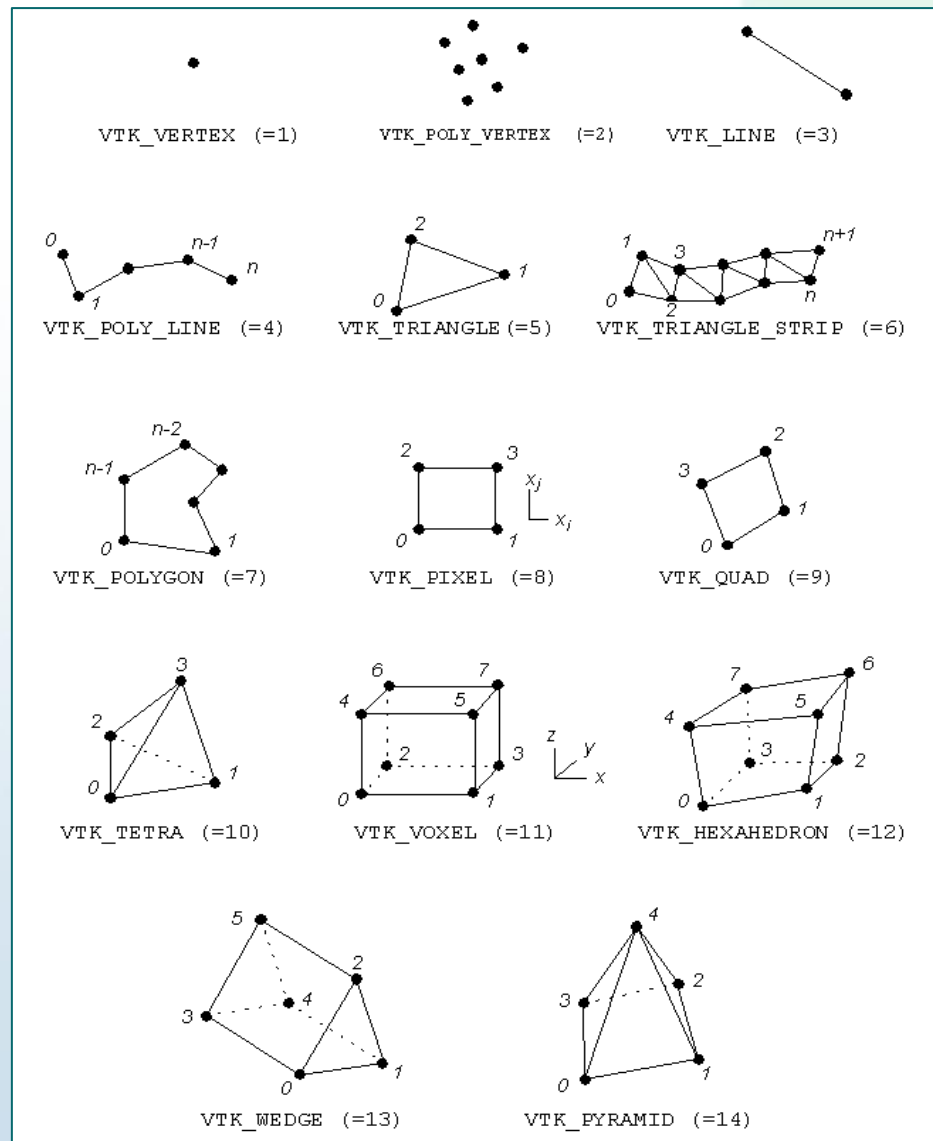
- Abstract class for grids with arbitrary point locations
 - Derived classes: vtkStructuredGrid, vtkPolyData, vtkUnstructuredGrid
- SetPoints(vtkPoints*) – set the number and coordinates of a process's points
- vtkPoints
 - Class used to store actual point location array
 - C/C++ indexing
 - SetDataTypeTo<Int,Float,Double>()
 - SetNumberOfPoints() – allocation
 - SetPoint() – fast but doesn't do bounds checking
 - InsertPoint()/InsertNextPoint() – safe but does bounds checking and reallocation as needed

vtkStructuredGrid

- `vtkCPInputDataDescription::SetWholeExtent()` – total number of points in each direction
- `SetExtents()` – a process's part of the whole extent
- Ordering for points and cells
 - Fastest in x-direction
 - Next in y-direction
 - Slowest in z-direction
 - `vtkPoints` must be ordered in same manner



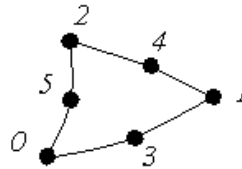
vtkCell Straight-Edged Types



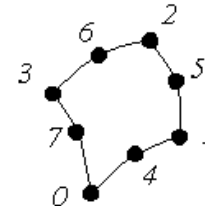
vtkCell Curvilinear-Edge Types



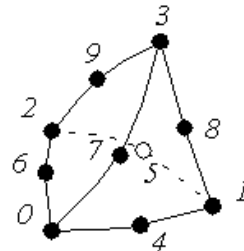
(a) Quadratic Edge



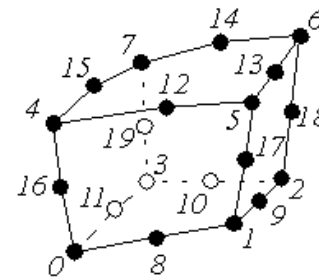
(b) Quadratic Triangle



(c) Quadratic Quadrilateral



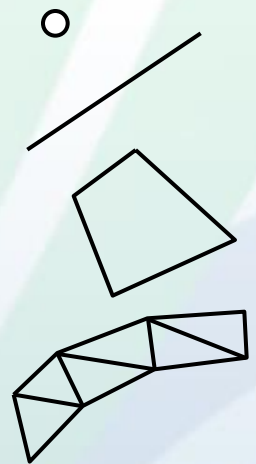
(d) Quadratic Tetrahedron



(e) Quadratic Hexahedron

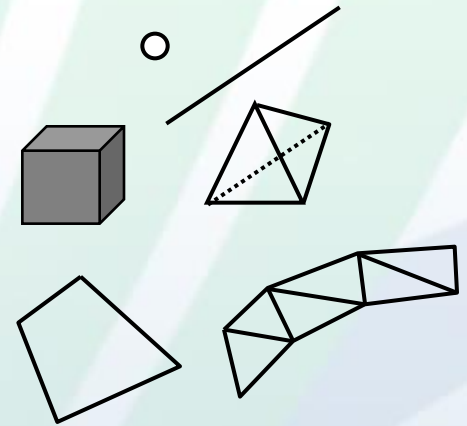
vtkPolyData Cells

- Efficient way – build through vtkCellArray
 - `vtkCellArray::SetNumberOfCells()`
 - Done for verts, lines, polys, triangle strips
- Simple way – build through poly data API
 - `Allocate()`
 - `InsertNextCell()`
 - `Squeeze()` – reclaim unused memory
- WARNING – Order that cells are added may be different than they are accessed
 - Ordered by verts, lines, polys, triangle strips
 - Need to match with cell data



vtkUnstructuredGrid Cells

- Order of cells is order of insertion
 - 0-based indexing
- Allocate()
- InsertNextCell()
- SetCells()
- Squeeze()
- Inserted points of cells are 0-based indexing



Example 5 – Write It!

- Still using function signature:
 - `void coprocess(const double spacing[3], const Box& global_box, const Box& local_box, std::vector<double>& minifepointdata, int time_step, double time, bool force_output);`
- Hint: `example5_catalyst_adaptor.cpp`
 - `void getpointcoordinate(const int indices[3], const double spacing[3], double coord[3]);`
 - `void getcellpointids(const Box& local_box, const int indices[3], vtkIdType pointids[8]);`

Example 5 – A Solution

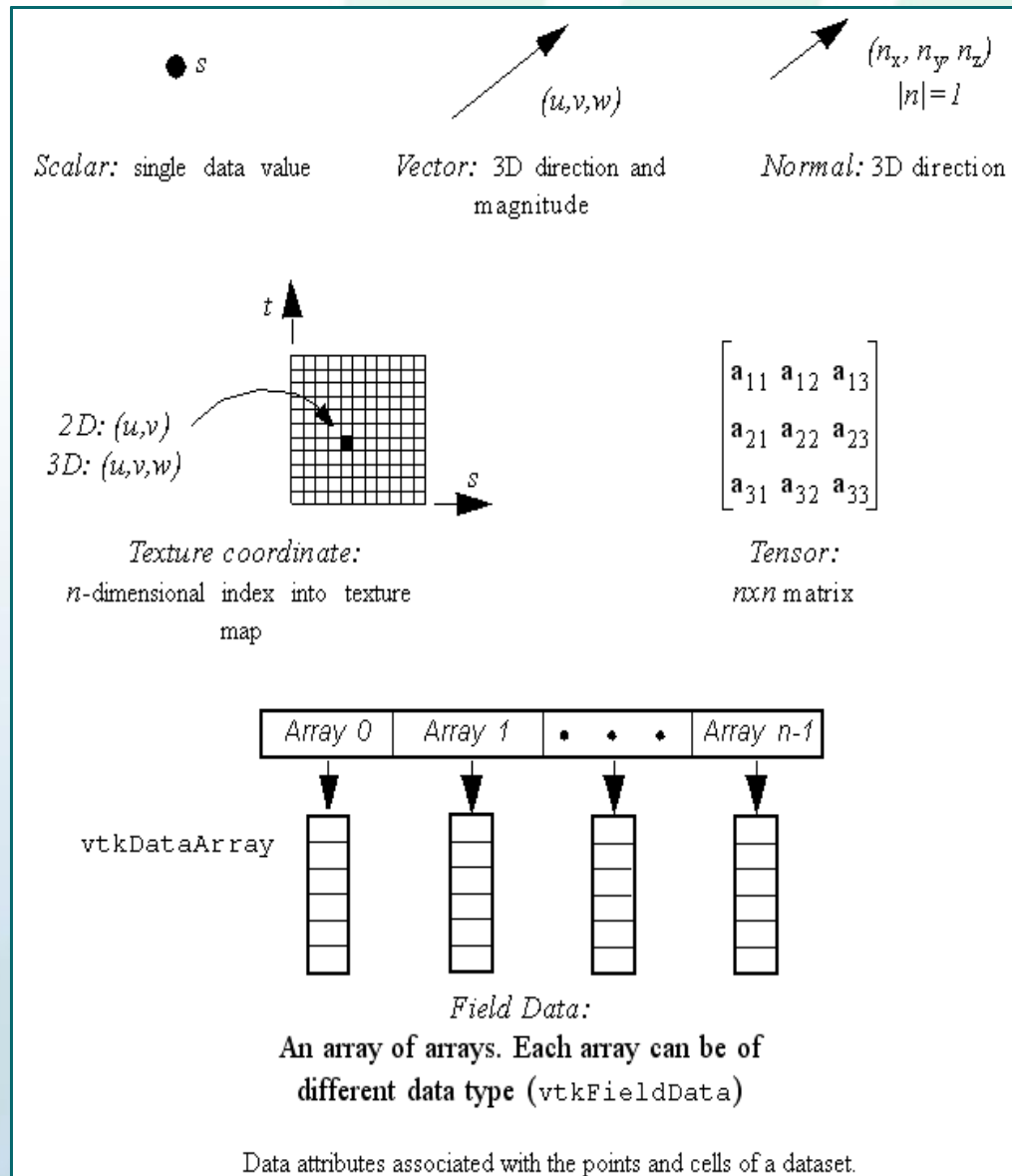
```
...  
vtkNew<vtkUnstructuredGrid> grid;  
grid->Initialize();  
vtkNew<vtkPoints> points;  
points->SetNumberOfPoints( (local_box[0][1]-local_box[0][0]+1) *  
    (local_box[1][1]-local_box[1][0]+1)*(local_box[2][1]-local_box[2][0]+1) );  
double coord[3]; int indices[3]; vtkIdType id=0;  
for(int iz=local_box[2][0]; iz<=local_box[2][1]; ++iz)  
    for(int iy=local_box[1][0]; iy<=local_box[1][1]; ++iy)  
        for(int ix=local_box[0][0]; ix<=local_box[0][1]; ++ix) {  
            int indices[3] = {ix, iy, iz};  
            getpointcoordinate(indices, spacing, coord);  
            points->SetPoint(id++, coord);  
        }  
grid->SetPoints(points.GetPointer());  
...
```

Example 5 – A Solution (2)

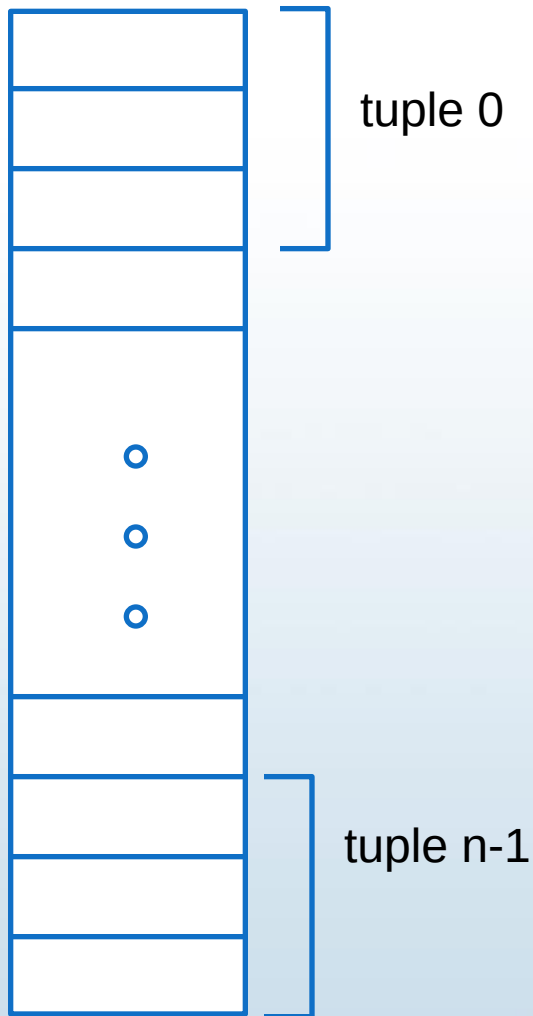
```
...
grid->Allocate( (local_box[0][1]-local_box[0][0]) *
    (local_box[1][1]-local_box[1][0])*(local_box[2][1]-local_box[2][0]) );
vtkIdType pointids[8];
for(int iz=local_box[2][0]; iz<local_box[2][1]; ++iz)
    for(int iy=local_box[1][0]; iy<local_box[1][1]; ++iy)
        for(int ix=local_box[0][0]; ix<local_box[0][1]; ++ix) {
            int indices[3] = {ix, iy, iz};
            getcellpointids(local_box, indices, pointids);
            grid->InsertNextCell(VTK_HEXAHEDRON, 8, pointids);
        }
...
```

Field Information

- Store information defined over grids
- Stored in concrete classes that derive from `vtkDataArray`
 - `vtkFloatArray`
 - `vtkIntArray`
 - `vtkDoubleArray`
 - `vtkUnsignedCharArray`
 - ...



vtkDataArray – Basis for vtkDataObject Contents



- An array of n tuples
- Each tuple has m components which are logically grouped together
- Internal implementation is a pointer to an $n \times m$ block of memory
- Data type determined by class
- Two APIs : generic and data type specific

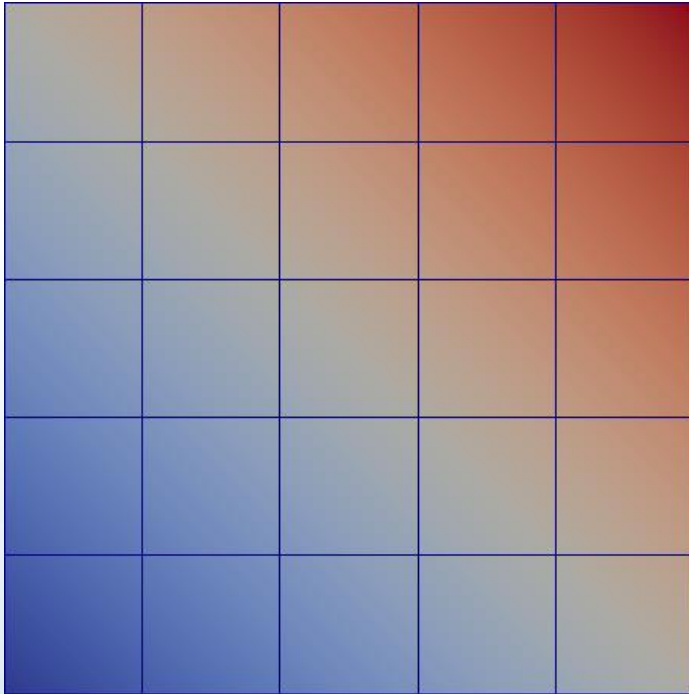
vtkDataArray

- SetNumberOfComponents() – call first
- SetNumberOfTuples()
 - For point data must be set to number of points
 - For cell data must be set to number of cells
- **SetArray()**
 - If flat array has proper component & tuple ordering use existing simulation memory – most efficient
 - Can specify who should delete
 - VTK uses pipeline architecture so Catalyst libraries will NOT modify array values
- SetTupleValue() – uses native data type
- SetValue() – uses native data type
- SetName() – array descriptor, e.g. velocity, density

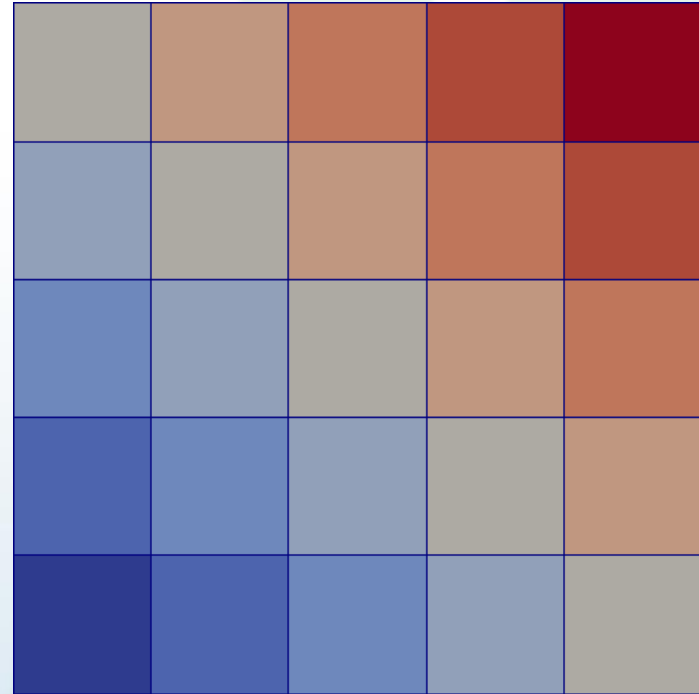
vtkFieldData

- Object for storing vtkDataArrays
- vtkDataSet::GetFieldData() – arrays not associated with points or cells
- Derived classes
 - vtkPointData – vtkDataSet::GetPointData()
 - vtkCellData – vtkDataSet::GetCellData()
- vtkFieldData::AddArray(vtkDataArray*)
- Specific arrays are normally retrieved by name from vtkFieldData
 - Uniqueness is not required but can cause unexpected results

Point Data and Cell Data



Point data – 36 values



Cell data – 25 values

Example 6 – Write It!

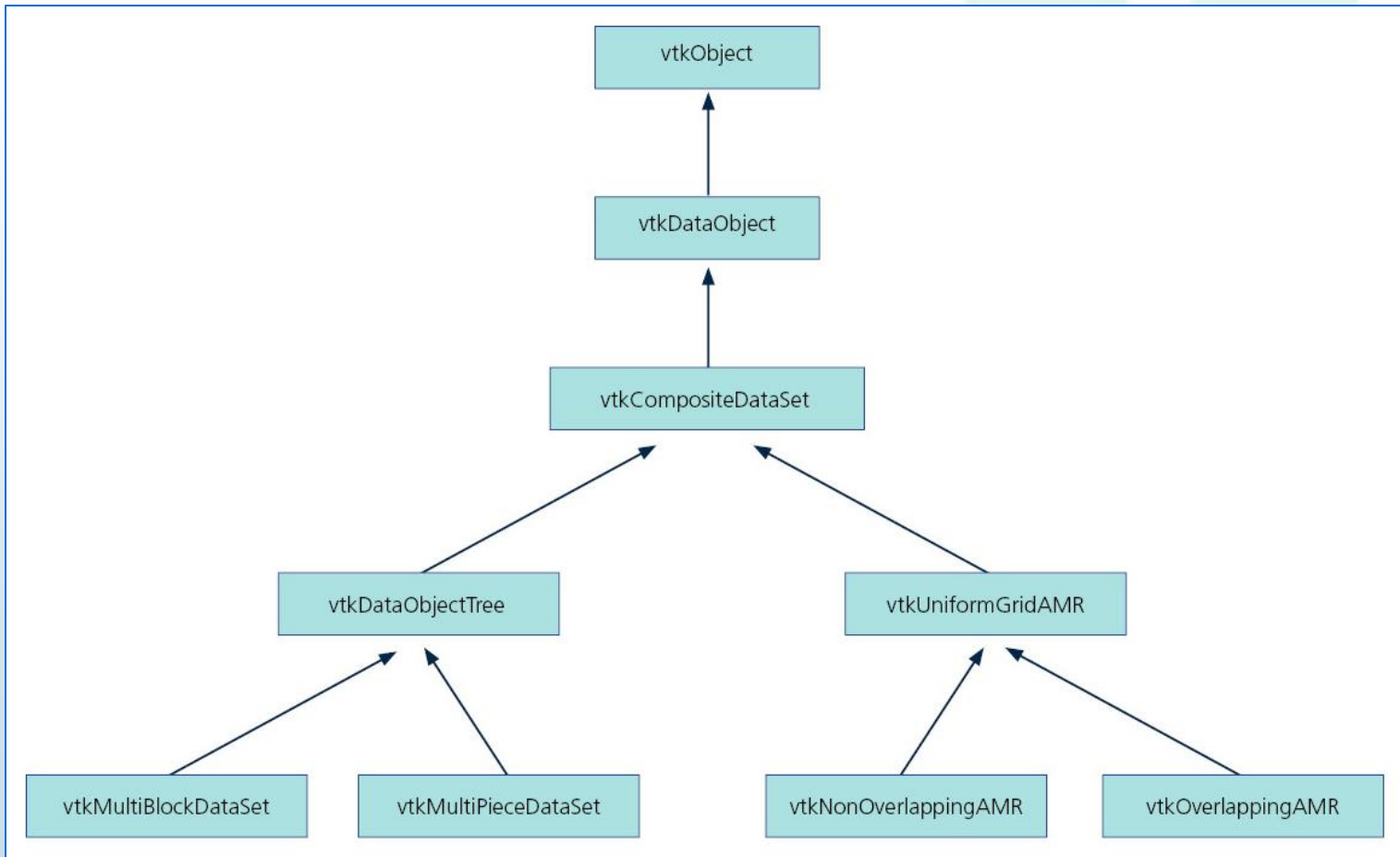
- Still using function signature:
 - `void coprocess(const double spacing[3], const Box& global_box, const Box& local_box, std::vector<double>& minifepointdata, int time_step, double time, bool force_output);`
- Use `vtkpointdata` vector for source
 - Already ordered for our grid's points

Example 6 – A Solution

```
...  
std::vector<double> vtkpointdata;  
getlocalpointarray(global_box, local_box, minifepointdata, vtkpointdata);  
vtkSmartPointer<vtkDoubleArray> myDataArray =  
    vtkSmartPointer<vtkDoubleArray>::New();  
myDataArray->SetNumberOfComponents(1);  
myDataArray->SetName("myData");  
myDataArray->SetArray(&(vtkpointdata[0]), vtkpointdata.size(), 1);  
grid->GetPointData()->AddArray(myDataArray);  
...
```

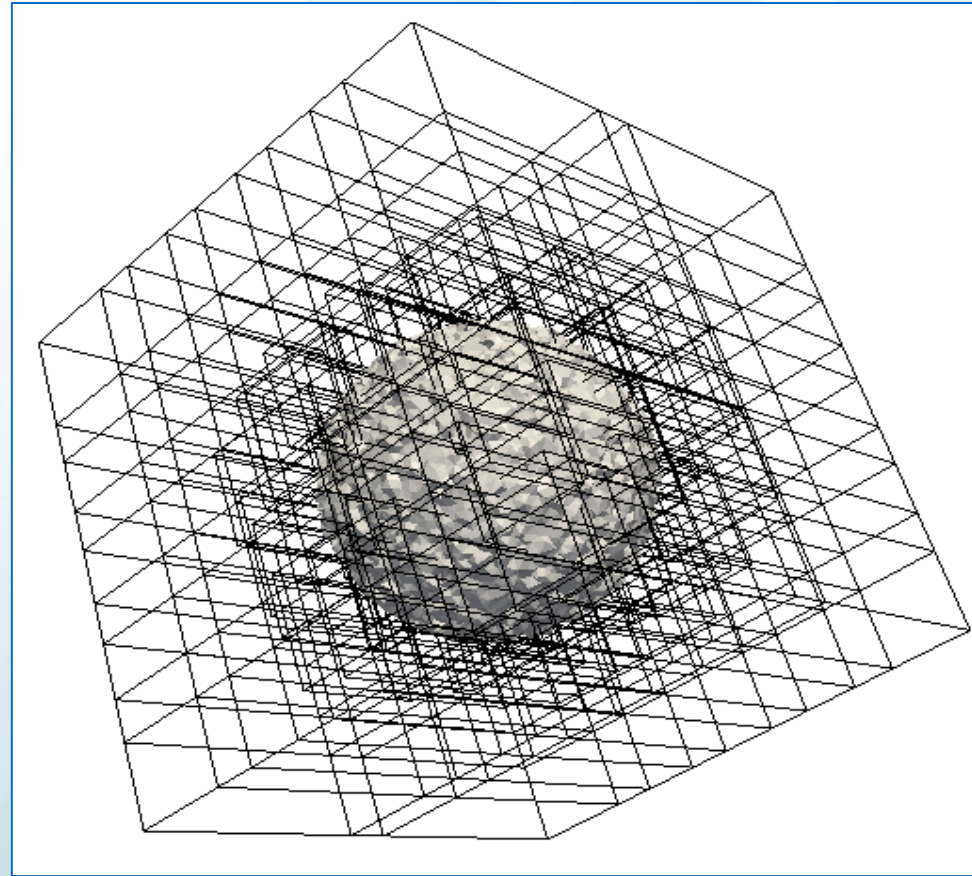
Multiple Grids

- Use something that derives from `vtkCompositeDataSet` to group multiple `vtkDataSets` together



vtkMultiBlockDataSet

- Most general way of storing vtkDataSets and other vtkCompositeDataSets
- Block structure must be the same on each process
- Block is null if data set is on a different process
- SetNumberOfBlocks()
- SetBlock()



vtkMultiPieceDataSet

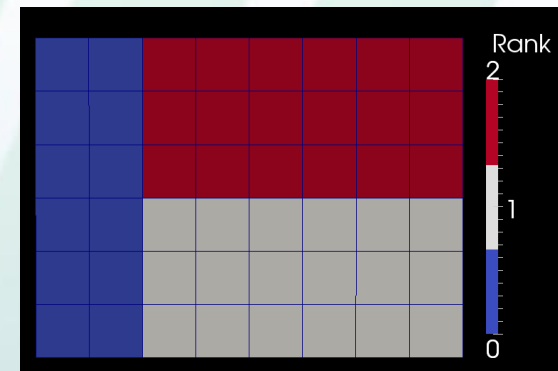
- Way to combine multiple vtkDataSets of the same type into a single logical object
 - Useful when the amount of pieces is unknown *a priori*
- Must be contained in a vtkMultiBlockDataSet
- SetNumberOfPieces()
- SetPiece()

AMR Data Sets

- Classes derive from vtkUniformGridAMR
- Concept of grid levels for changing cell size locally
- Uses only vtkUniformGrids
 - vtkNonOverlappingAMR – no blanking used since no data sets overlap
 - vtkOverlappingAMR – vtkUniformGrids overlap and use blanking

Grid Partitioning

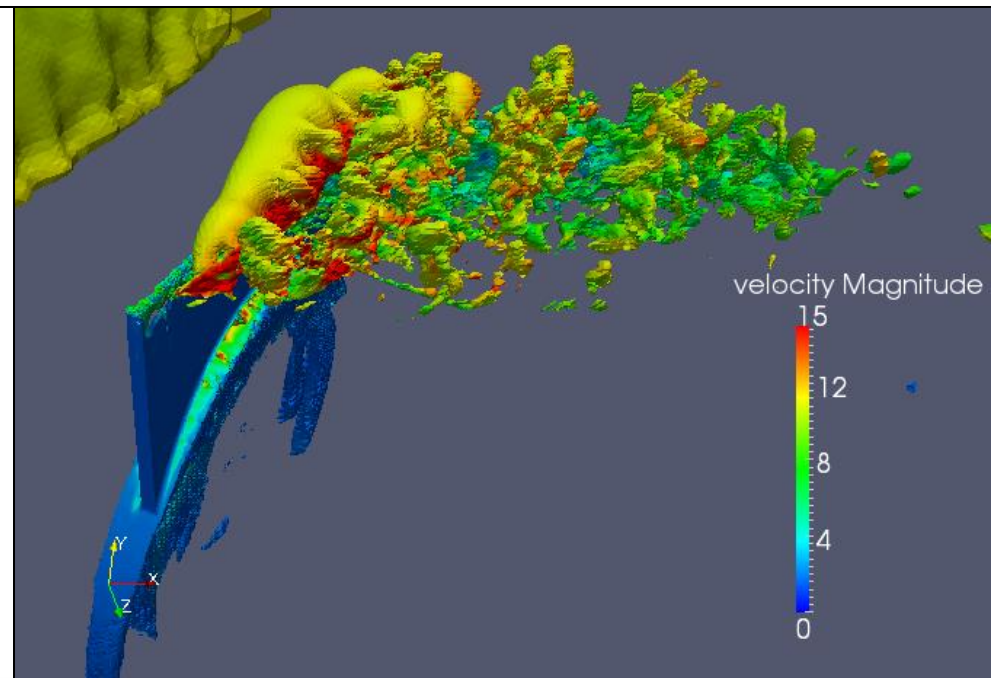
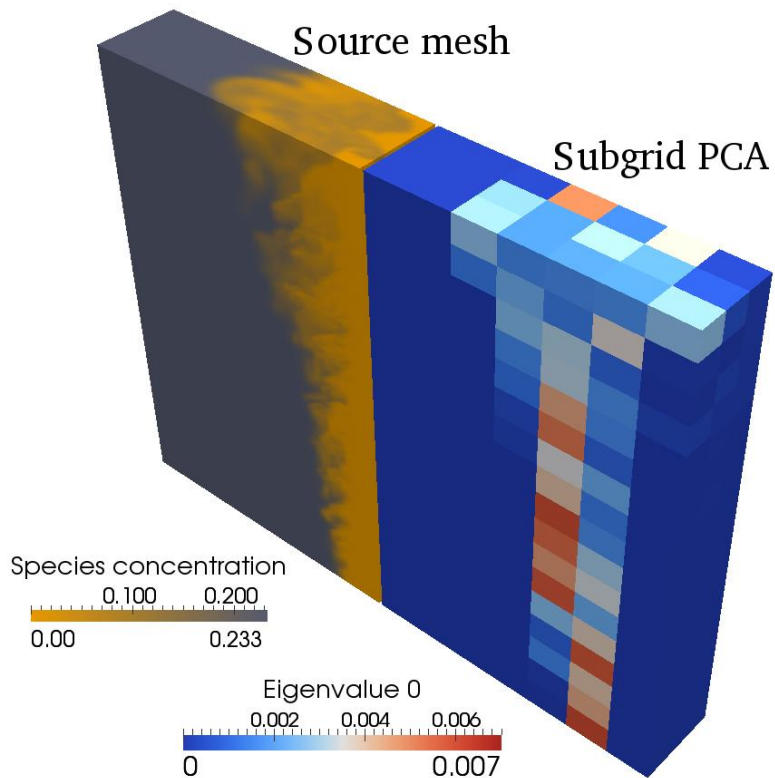
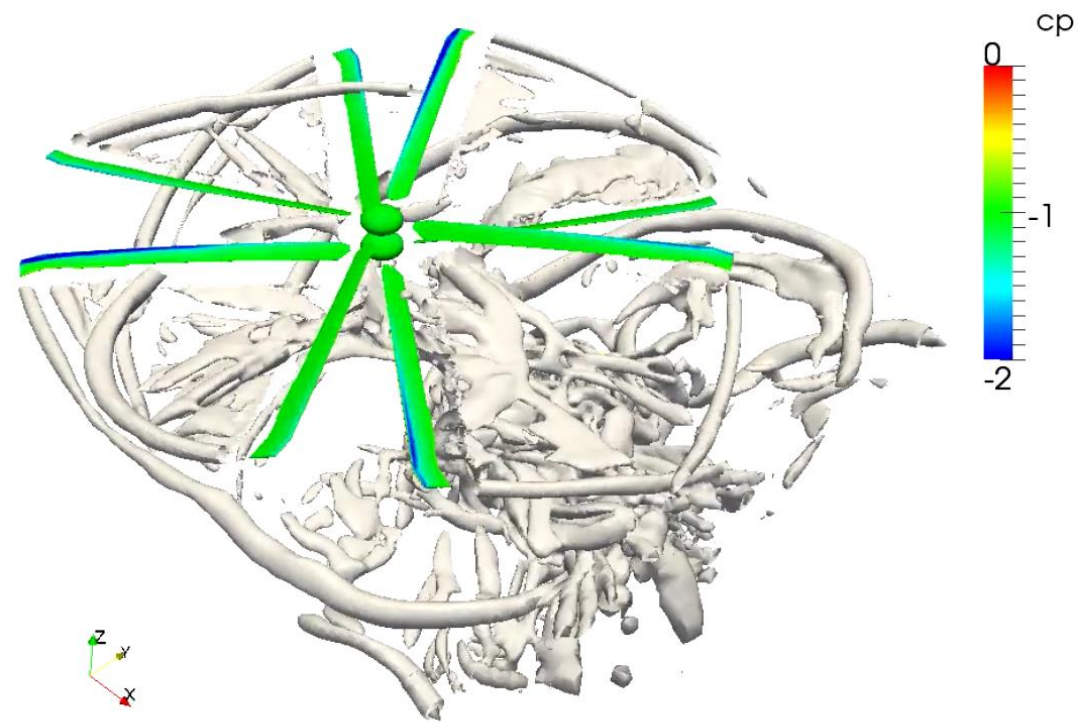
- For unstructured grids and polydatas
 - Single data set per process – use as is
 - Multiple data sets per process – choice of combining or not depends on memory layout
- For topologically structured grids
 - Must be partitioned into logical blocks
 - SetExtent()
 - Index of first and last point in each direction
 - Will be different on each process
 - vtkCPInputDataDescription::SetWholeExtent() – same on each process



Advanced Topics

- Zero-copy for VTK arrays
 - Reuse existing memory in layout different than standard VTK arrays
- Hard-coded C++ pipelines
 - Removes dependency on Python
 - Use simulation input parameters to remove requirement that users create pipelines in ParaView
- Ghost information
- CMake for automatic C++ and Fortran function name mangling

Gratuitous Catalyst Images





ParaView Catalyst for Simulation Users



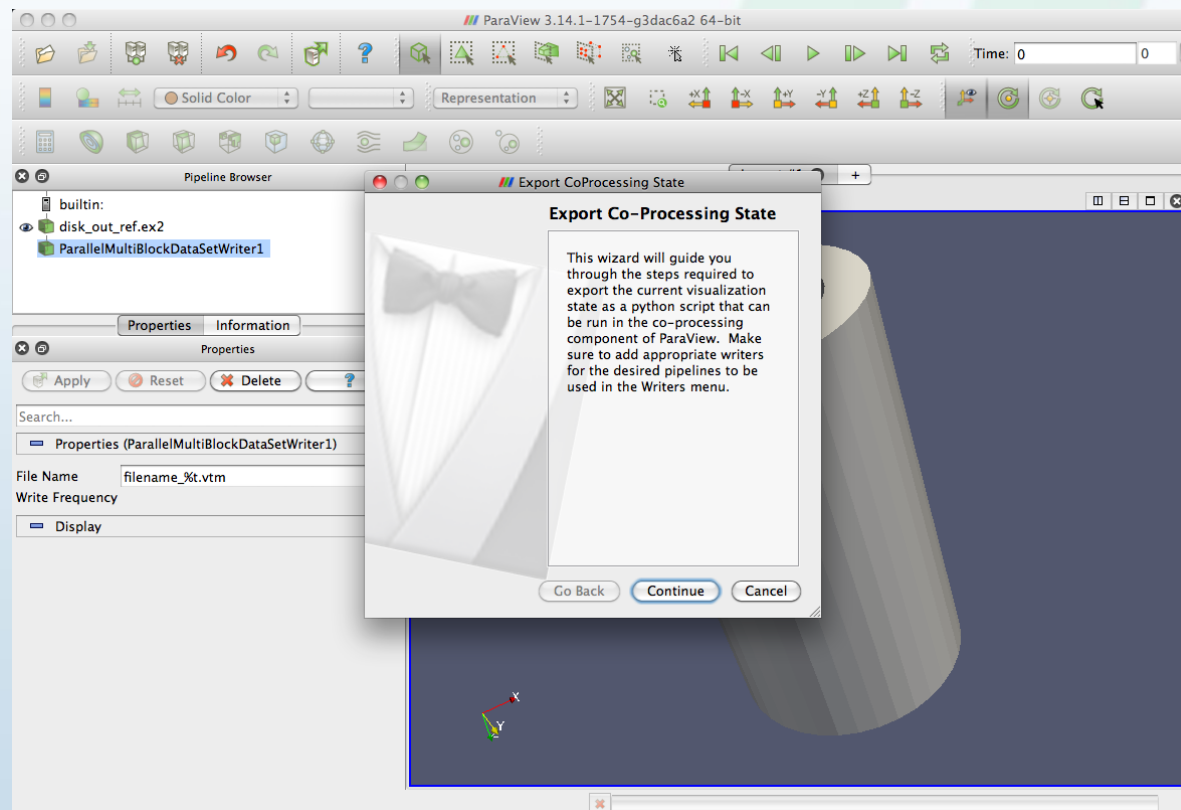
Creating Catalyst Output

Two main ways:

- Create and/or modify Python scripts
 - ParaView GUI plugin to create Python scripts
 - Modification with knowledge of ParaView Python API
- Developer generated “canned” scripts
 - User provides parameters for already created Catalyst pipelines
 - User may not even need to know ParaView
 - See ParaView Catalyst User’s Guide

Create Python Scripts from ParaView

- Interact with ParaView normally
- Export a script that mimics that interaction
- Queries during each co-processing step
 - (one frame at a time)



ParaView GUI Plugin

- Similar to using ParaView interactively
 - Setup desired pipelines
 - Ideally, start with a representative data set from the simulation
- Extra pipeline information to tell what to output during simulation run
 - Add in data extract writers
 - Create screenshots to output
 - Both require file name and write frequency

In Situ Demo

- Create a ParaView Catalyst Python pipeline script
 - Specify desired outputs from run
 - Export the script
- Run the script with miniFE
 - `./miniFE.x script_names=<file name>`
- Examine results

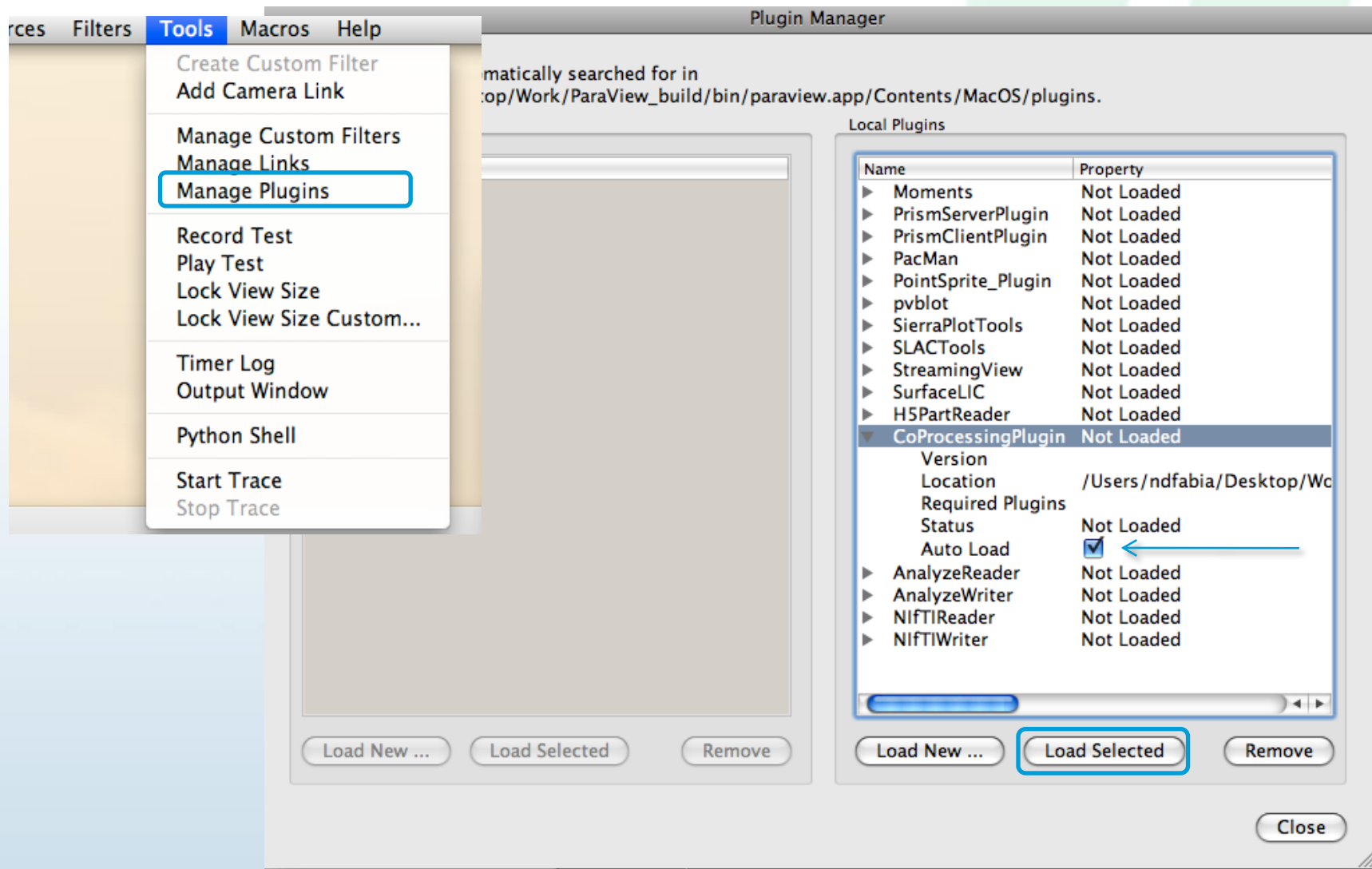
Use Catalyst to Write Out Grid

- Need something to start with
- Grid of the same type
- Fields of the same type
- The closer to the actual simulation the better

Run It!

- Go to <miniFE-2.0_ref>/src and finish building miniFE, i.e. “make”
- Make sure gridwriter.py is there
 - Simple Catalyst Python pipeline to write out the full grid and fields every time step
 - Run with “./miniFE.x script_names=gridwriter.py”
 - Output file will be:
 - filename_*.pvtu for unstructured grid case
 - filename_*.pvti for image data (i.e. Cartesian grid) case

In Situ Demo – Load Plugin Step



In Situ Demo – New Plugin Menus

CoProcessor Writers

Export State

Parallel Hierarchical Box Data Writer

Parallel MultiBlockDataSet Writer

Parallel Image Data Writer

Parallel PolyData Writer

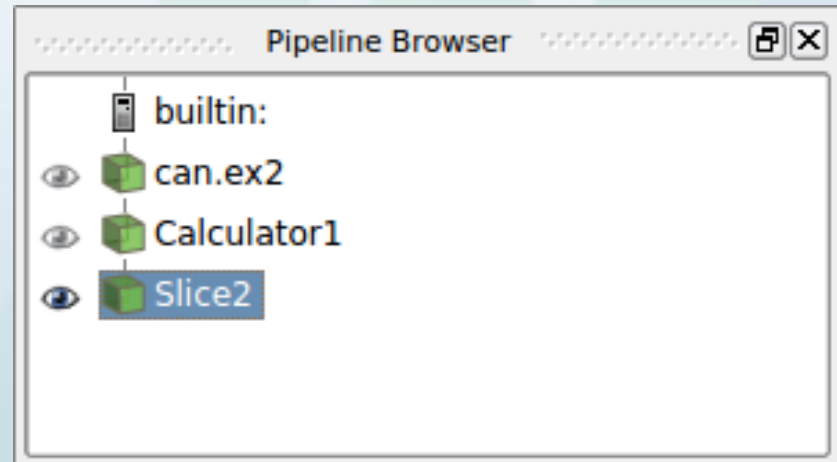
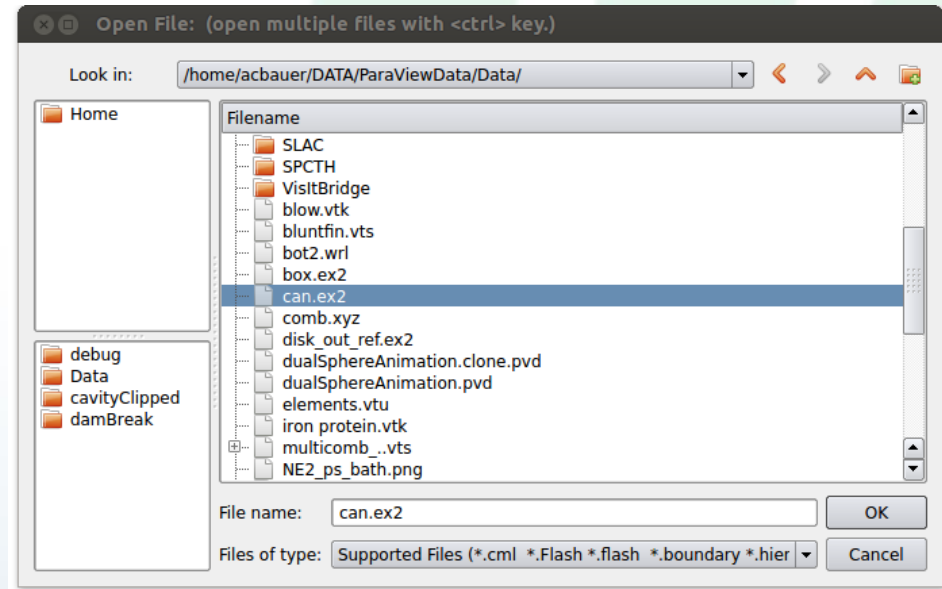
Parallel Rectilinear Grid Writer

Parallel Structured Grid Writer

Parallel UnstructuredGrid Writer

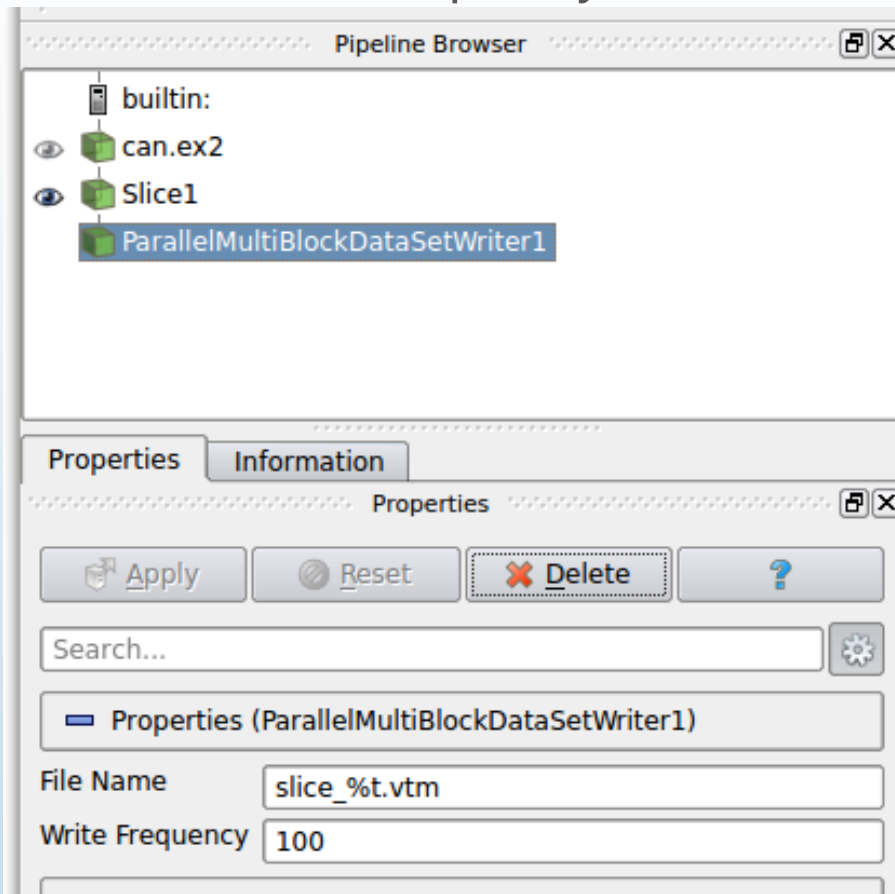
In Situ Demo – Creating a Catalyst Python Script

- Load output file
filename_*. {pvti,pvtu}
from previous run
- Create desired pipeline



In Situ Demo – Adding in Writers

- Parameters:
 - File Name – %t gets replaced with time step
 - Write Frequency



Writers

Parallel Hierarchical Box Data Writer

Parallel MultiBlockDataSet Writer

Parallel Image Data Writer

Parallel PolyData Writer

Parallel Rectilinear Grid Writer

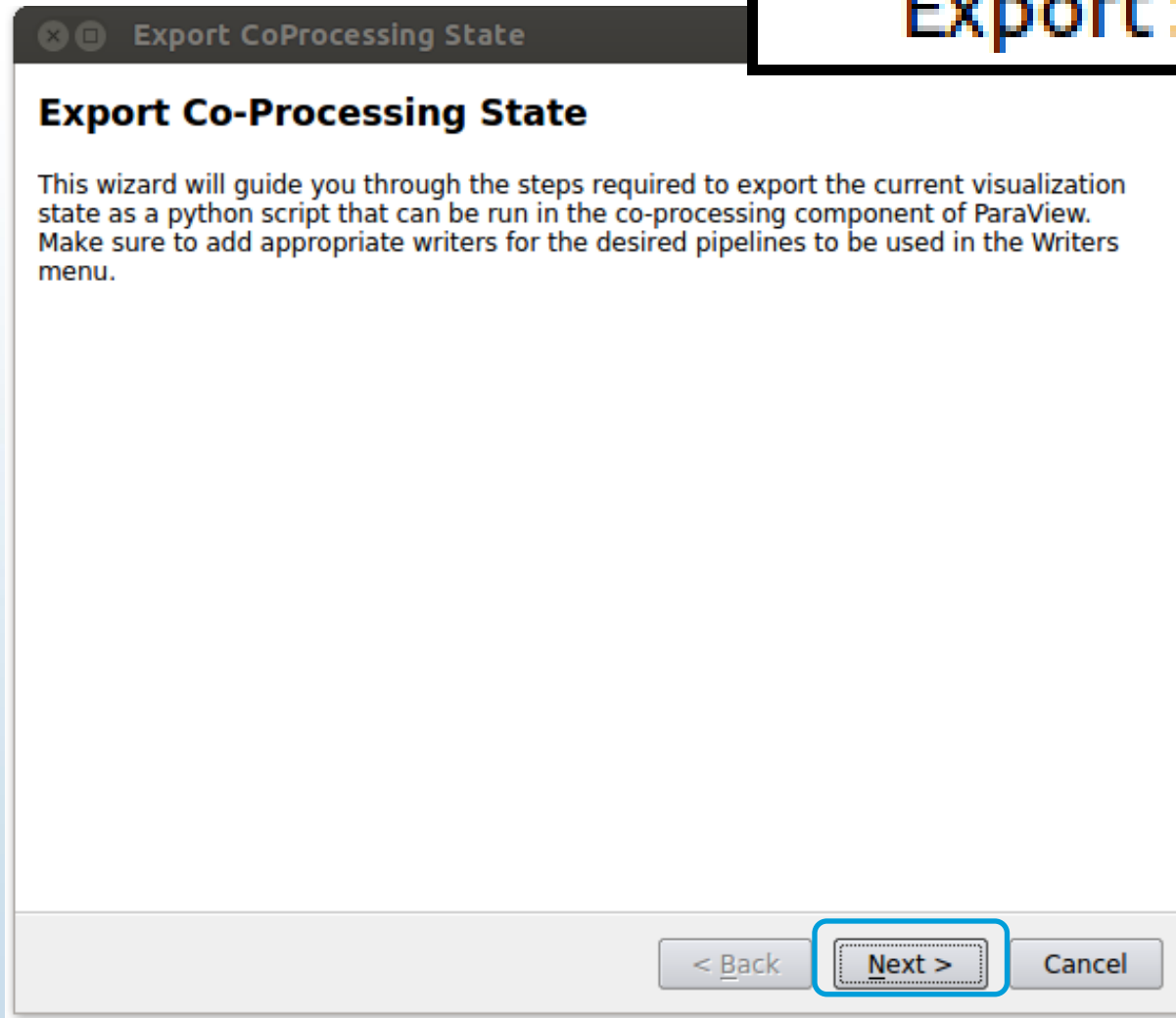
Parallel Structured Grid Writer

Parallel UnstructuredGrid Writer

In Situ Demo – Exporting the Script

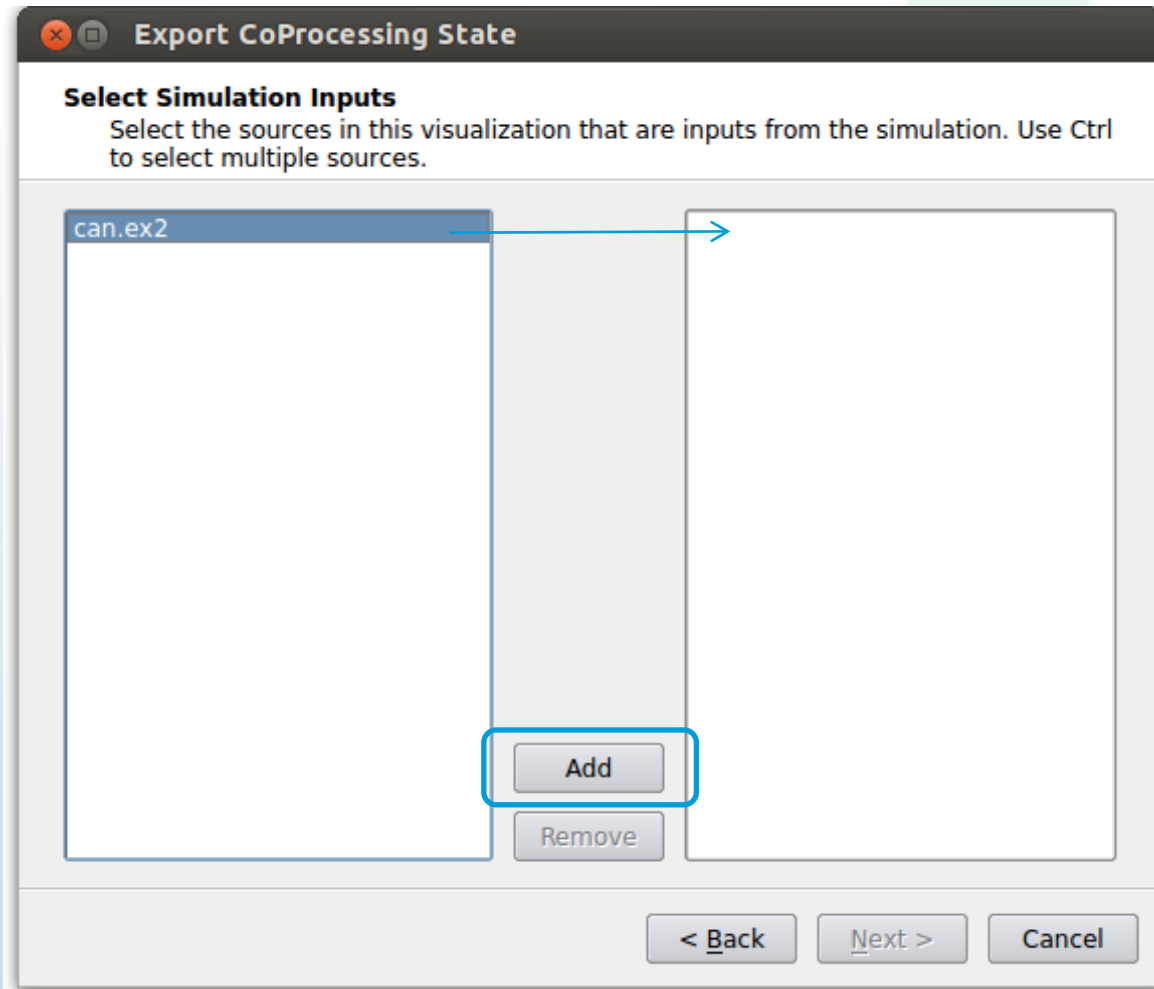
CoProcessor

Export State



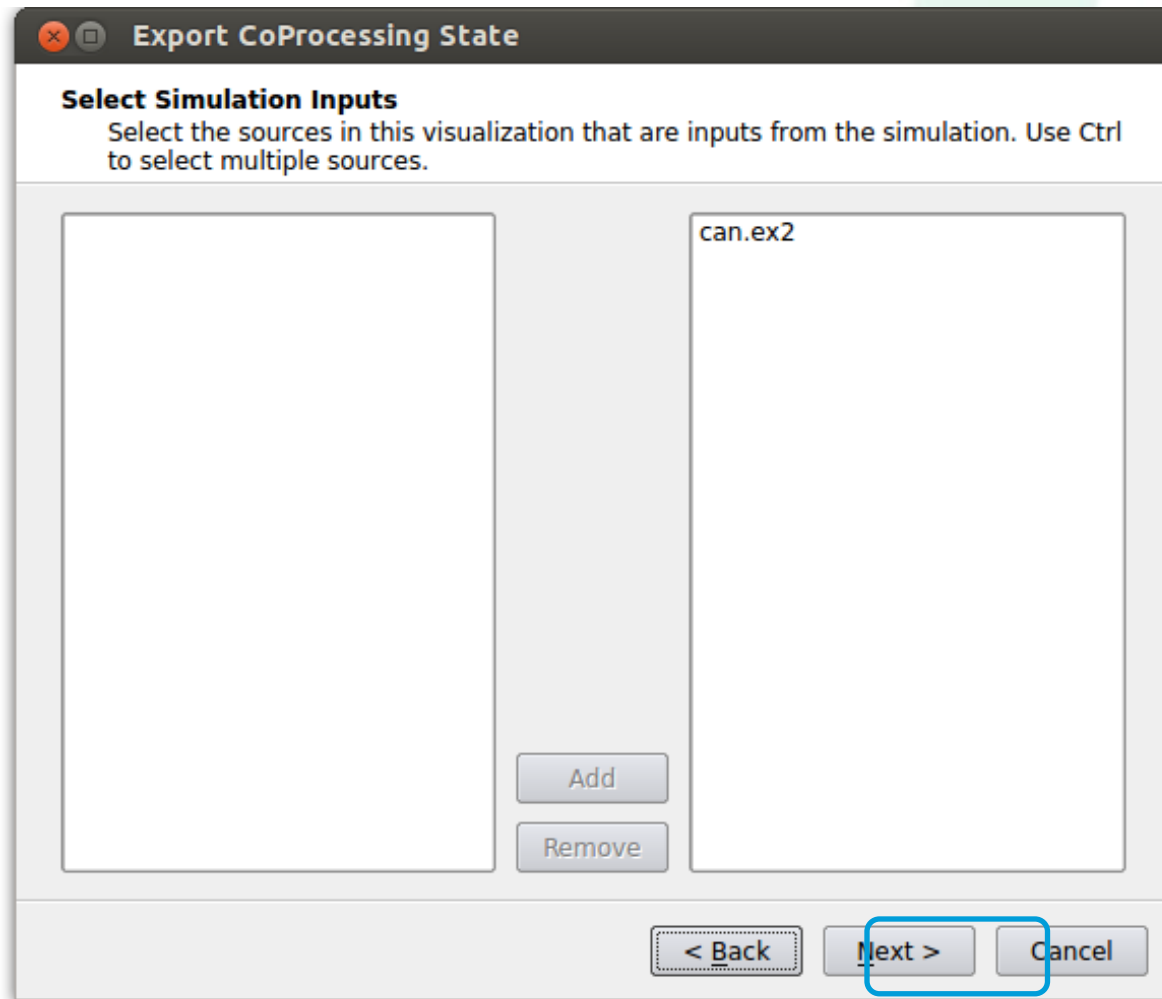
In Situ Demo – Select Inputs

- Usually only a single input but can have multiple inputs



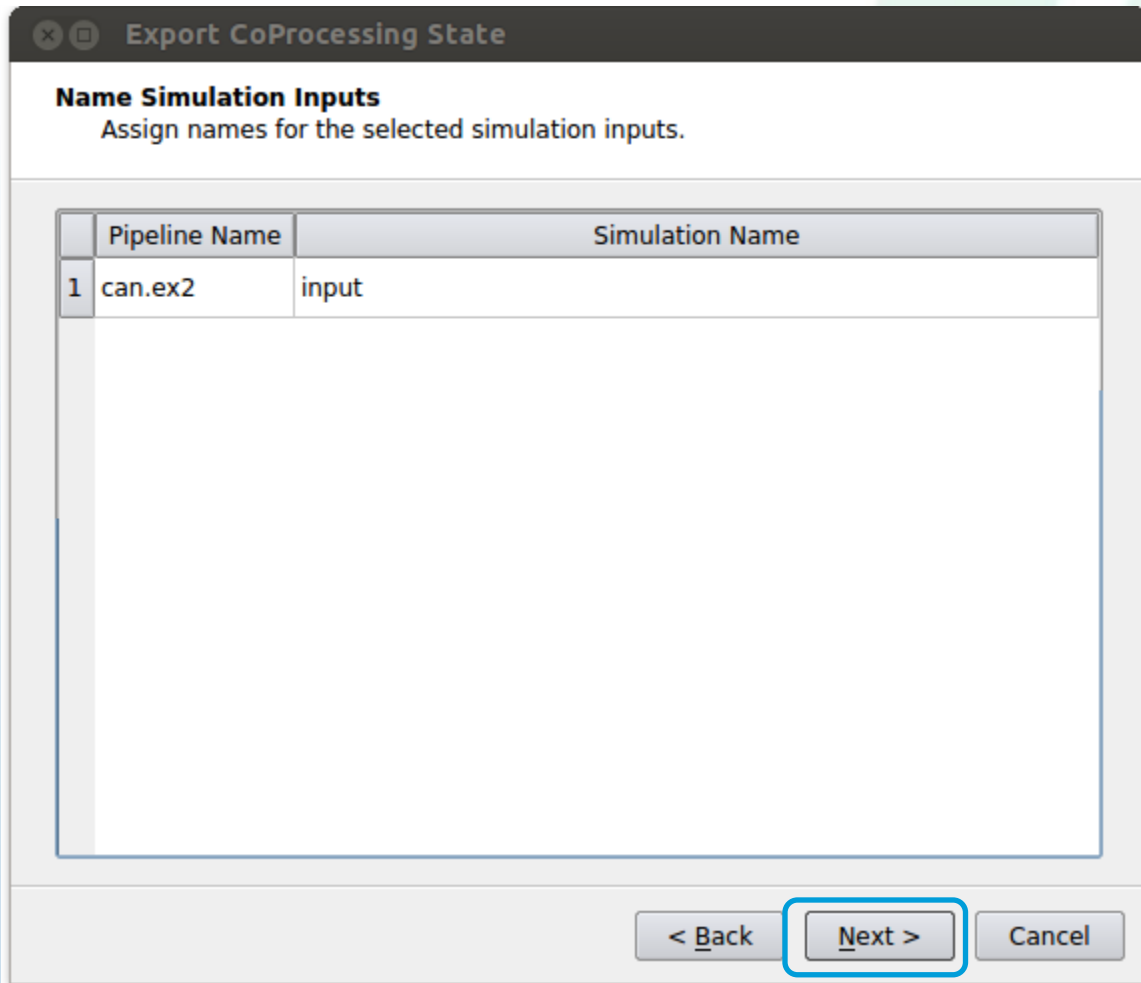
In Situ Demo – Select Inputs

- Each pipeline source is a potential input



In Situ Demo – Match Up Inputs

- Source name (e.g. “can.ex2”) needs to be matched with string key in adaptor (e.g. “input”)



Export CoProcessing State

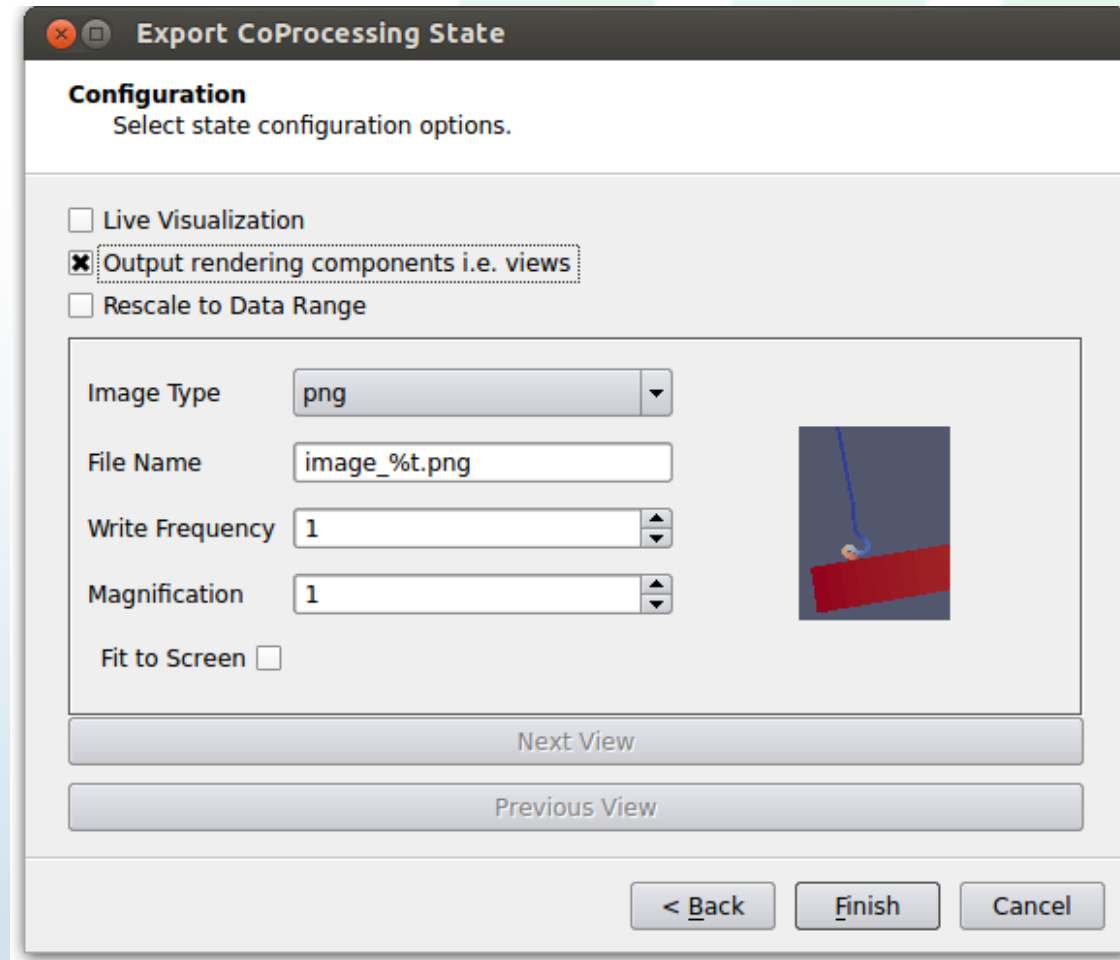
Name Simulation Inputs
Assign names for the selected simulation inputs.

	Pipeline Name	Simulation Name
1	can.ex2	input

< Back **Next >** Cancel

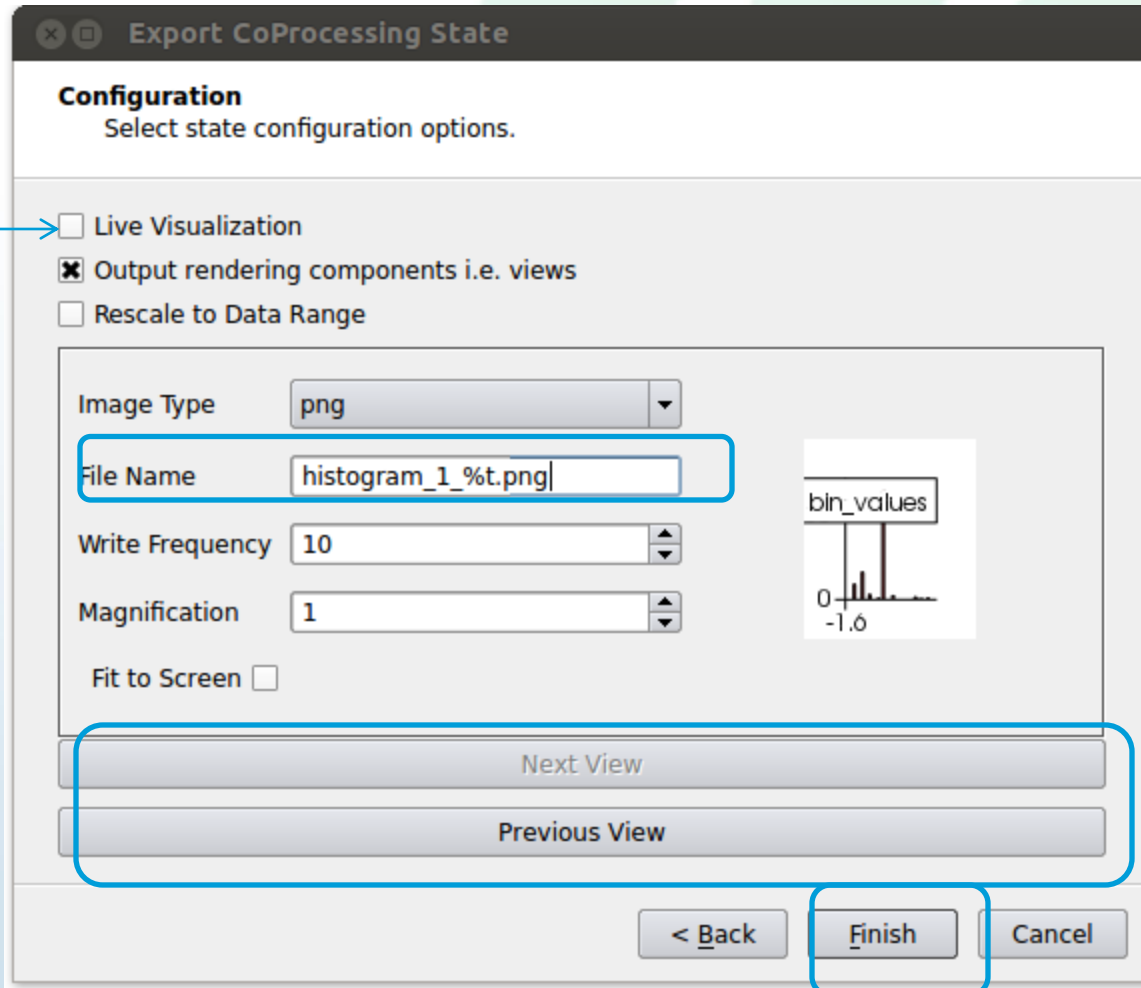
In Situ Demo – Generating an Image

- Parameters/Options:
 - Live visualization
 - Rescale to Data Range (all images)
 - Individual images
 - Image Type
 - File Name
 - Write Frequency
 - Magnification
 - Fit to Screen
- %t gets replaced with time step



In Situ Demo – Generating Two Images

- Parameters/Options:
 - Live visualization
 - Check this as we will use this later
 - Rescale to Data Range (all images)
 - Individual images
 - Image Type
 - File Name
 - Write Frequency
 - Magnification
 - Fit to Screen



In Situ Demo – Write Out the Script

- Generated script will look something like this

```
def DoCoProcessing (datadescription)

    input = CreateProducer ( datadescription "input"

    ParallelMultiBlockDataSetWriter1 = CreateWriter (
XMLMultiBlockDataWriter "filename_%t.vtm" 1
```

```
try: paraview.simple import *
except: from paraview.simple import *

cp_writers = []

def RequestDataDescription(datadescription):
    "Callback to populate the request for current timestep"
    timestep = datadescription.GetTimeStep()

    input_name = 'input'
    if (timestep % 1 == 0) :
        datadescription.GetInputDescriptionByName(input_name).AllFieldsOn()
        datadescription.GetInputDescriptionByName(input_name).GenerateMeshOn()
    else:
        datadescription.GetInputDescriptionByName(input_name).AllFieldsOff()
        datadescription.GetInputDescriptionByName(input_name).GenerateMeshOff()

def DoCoProcessing(datadescription):
    "Callback to do co-processing for current timestep"
    global cp_writers
    cp_writers = []
    timestep = datadescription.GetTimeStep()

    input = CreateProducer( datadescription, "input" )

    ParallelMultiBlockDataSetWriter1 = CreateWriter( XMLMultiBlockDataWriter, "filename_%t.vtm", 1 )

    for writer in cp_writers:
        if timestep % writer.cpFrequency == 0:
            writer.FileName = writer.cpFileName.replace("%t", str(timestep))
            tne()

            rager.GetRenderViews()

            renderviews)):
            ame.replace("%v", str(view))
            ace("%t", str(timestep))
            renderviews[view])

            oxies -- we do it this way to avoid problems with prototypes
            Delete()
            :

            esToDelete()

    iter = servermanager.vtkSHPProxyIterator()
    iter.Begin()
    tobedeleted = []
    while not iter.IsAtEnd():
        if iter.GetGroup().find("prototypes") != -1:
            iter.Next()
            continue
        proxy = servermanager._getPyProxy(iter.GetProxy())
        proxygroup = iter.GetGroup()
        iter.Next()
        if proxygroup != 'timekeeper' and proxy != None and proxygroup.find("pq_helper_proxies") == -1 :
            tobedeleted.append(proxy)

    return tobedeleted

def CreateProducer(datadescription, gridname):
    "Creates a producer proxy for the grid"
    if not datadescription.GetInputDescriptionByName(gridname):
        raise RuntimeError, "Simulation input name '%s' does not exist" % gridname
    grid = datadescription.GetInputDescriptionByName(gridname).GetGrid()
    producer = TrivialProducer()
    producer.GetClientSideObject().SetOutput(grid)
    producer.UpdatePipeline()
    return producer

def CreateWriter(proxy_ctor, filename, freq):
    global cp_writers
    writer = proxy_ctor()
    writer.FileName = filename
    writer.add_attribute("cpFrequency", freq)
    writer.add_attribute("cpFileName", filename)
    cp_writers.append(writer)
    return writer
```

In Situ Demo – Run the Script

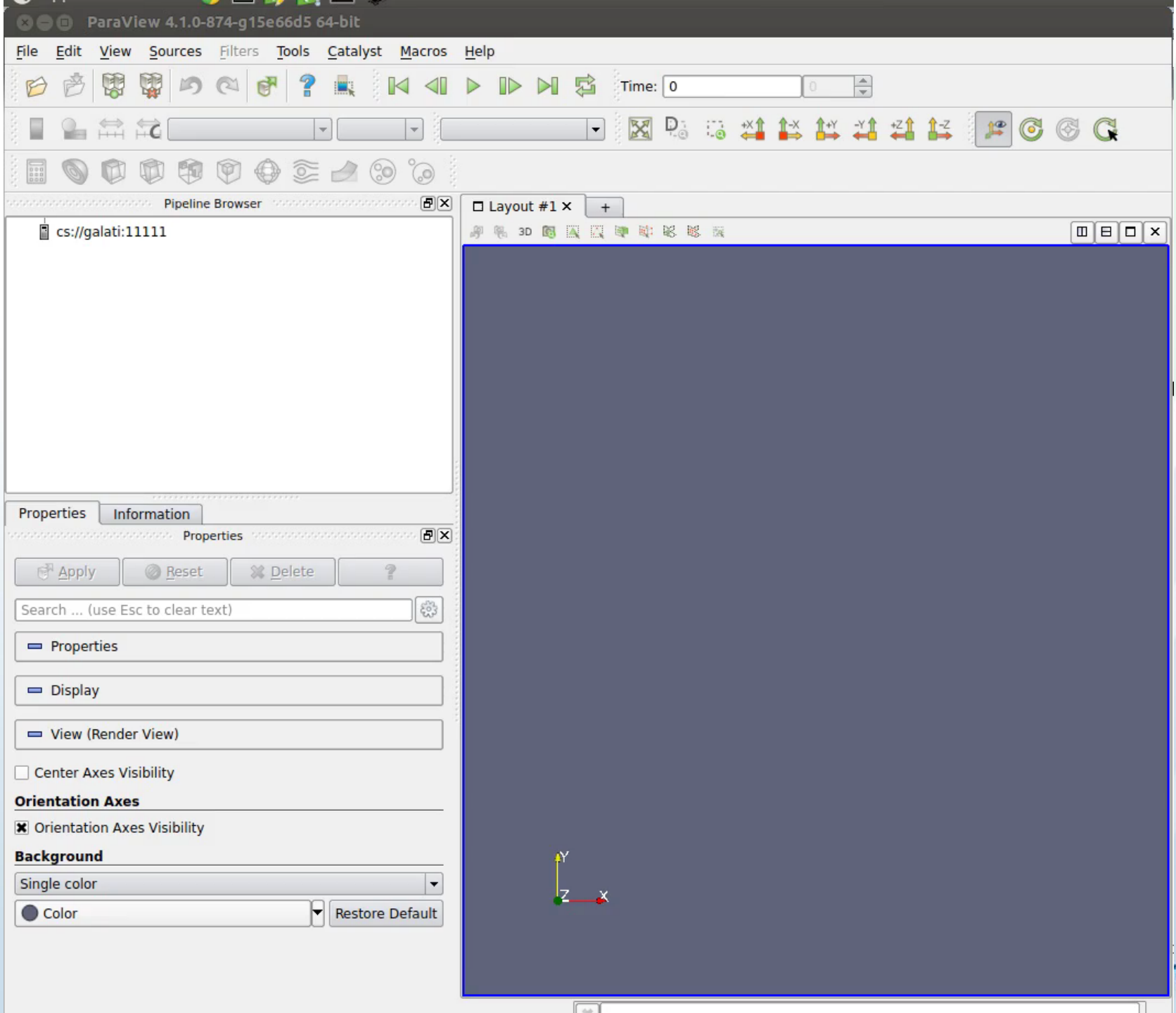
- Put the generated Python script in <miniFE-2.0_ref>/src directory
- Run with “./miniFE.x
script_names=<outputscript>”
 - Can use MPI to run with more processes
 - Change grid size with nx=, ny=, nz= command line arguments
 - Run multiple scripts with comma separated list
script_names=<script1>,<script2>,<script3>

Live *In Situ* Analysis and Visualization

- Everything before this was “batch”
 - Preset information with possible logic
- Functionality for interacting with simulation data during simulation run
 - When exporting a Python script, select “Live Visualization”
 - During simulation run choose the “Tools→Connect to Catalyst” GUI menu item

Two Ways for Live *In Situ* Analysis and Visualization

- Without blocking
 - Simulation proceeds while the user interacts with the data from a specific time step
 - “At scale” mode
- With blocking
 - Simulation is blocked while the user interacts with the data
 - “Debugging” mode
 - Work in progress
- Can switch between interactive and batch during run

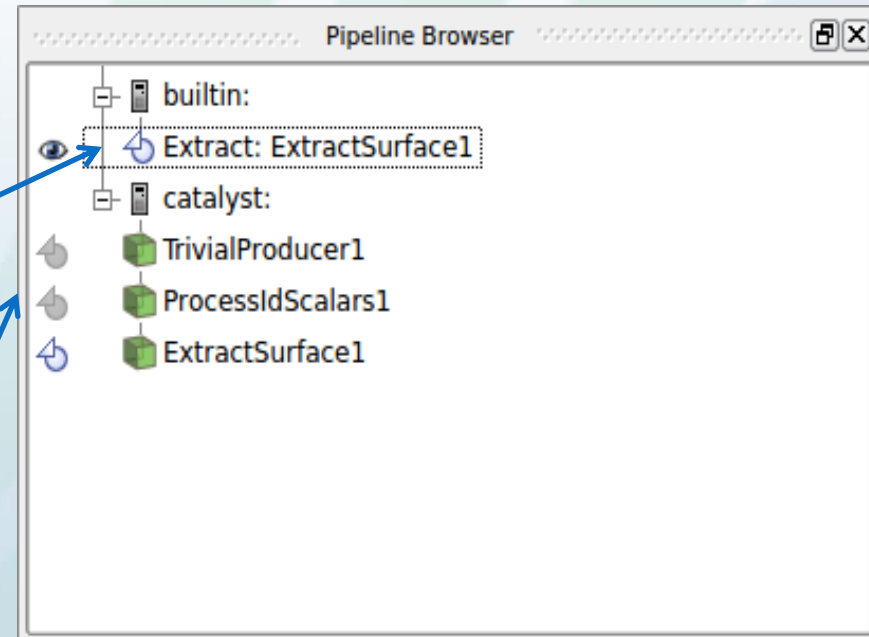


Live *In Situ* Example

- Need to “lengthen” the simulation time
 - Increase nx, ny and nz
 - Add in “sleep(5);” at the end of coprocess() method in catalyst_adapter.cpp
- Run as before with “./miniFE.x script_names=<outputscript>”
 - Must have Catalyst “Live Visualization” enabled
- Start ParaView and select Tools→Connect to Catalyst
 - Select port (22222 is default)

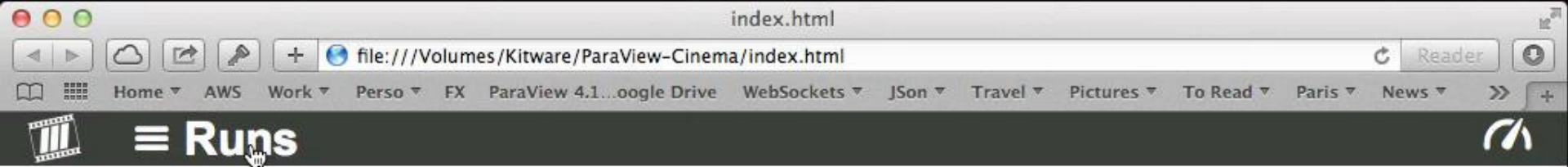
Live *In Situ* Example

- Only transfer requested data from server (simulation run) to client
 - ExtractSurface1 is already getting extracted
- Use  Delete on client  to stop transferring to client
- Click on  to transfer to client from Catalyst



ParaView Cinema + Catalyst

- Image output is orders magnitude less than an entire data set
 - Helios: full data set – 448 MB, surface of blades – 2.8 MB, image – 71 KB
- Use Catalyst to make a set of images at each time step with small output changes (rotations, slice planes locations, iso-surfaced variables & values, etc.)
- Use a web-browser to view images later
- **Should be in ParaView 4.2 as beta functionality**



An Image-based approach to Extreme Scale In Situ Visualization and Analysis

James Ahrens*, Sebastien Jourdain**, Patrick O'Leary**,
John Patchett*, David H. Rogers*

* Los Alamos National Laboratory

** Kitware Inc.

ADIOS Writer

- Improved IO for at scale HPC runs
 - <https://www.olcf.ornl.gov/center-projects/adios/>
- Still in development but should be in ParaView 4.2
- Separate PETTT project to study scaling IO on Garnet & Copper DSRCs



Building ParaView Catalyst



Build Considerations Overview

- Shared library build
 - Easier for development but can kill the IO system on HPC machines for large scale runs
- Static library build
 - Freezing Python allows it to be built statically into the executable instead of having it behave similar to shared libraries
- Mesa
 - When window manager isn't available or don't want render windows popping up for image generation

Catalyst Editions

- Doing a full ParaView build gives full Catalyst functionality
 - Also can add 150 MB or more to the executable size depending on ParaView configuration
- Most people use a small subset of VTK and ParaView filters for analysis and visualization
- Catalyst editions allow simpler ways to remove unneeded functionality
 - Can get Catalyst library size down to 15 to 30 MB depending on configuration

Preconfigured Catalyst Editions

- Base – a minimal set of functionality to build other editions from but not meant for use on its own
- Essentials – base + several of the most popular filters
- Extras – base + essentials + parallel XML writers and a couple more filters
- Rendering – coming soon...
- Python version of each available as well
- Available at www.paraview.org/download

Acknowledgements

- Funding through PETTT project PP-ACE-KY05-009-P3
- Tutorial facilities setup by Michael Stephens & John Mason

Online Help

- Catalyst User's Guide:
 - <http://paraview.org/Wiki/images/4/48/CatalystUsersGuide.pdf>
- Email list:
 - paraview@paraview.org
- Doxygen:
 - <http://www.vtk.org/doc/nightly/html/classes.html>
 - <http://www.paraview.org/ParaView3/Doc/Nightly/html/classes.html>
- Sphinx:
 - <http://www.paraview.org/ParaView3/Doc/Nightly/www/py-doc/index.html>
- Websites:
 - <http://www.paraview.org>
 - <http://catalyst.paraview.org>
- Examples:
 - <https://github.com/acbauer/CatalystExampleCode>