In-Situ Visualization with ParaView Catalyst

For ParaView Catalyst 1.0
Agenda

• Introduction to ParaView Catalyst
• Catalyst for Users
• Catalyst for Developers
Online Help

- Email list: paraview@paraview.org
- Doxygen:
- Sphinx:
- Websites:
  - http://www.paraview.org
  - http://catalyst.paraview.org
- Examples:
  - https://github.com/acbauer/CatalystExampleCode
What is Catalyst
Access to More Data

Post-processing

In situ processing

Roughly equal data stored at simulation time

Reflections and shadows added in post-processing for both examples
## Why *In Situ*?

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

Two ways to run

Simulation

Catalyst

Disk Storage

Visualization

Simulation

Disk Storage

Visualization

Separate MPI
Catalyst Architecture

Catalyst

Python Wrappings

ParaView Server
Parallel Abstractions and Controls

VTK
Core Visualization Algorithms
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
User Perspective
User Perspective
User Perspective

Simulation

Catalyst

with Field Data

Output Processed Data

Rendered Images

Kitware
User Perspective

Simulation Catalyst

Output Processed Data

with Field Data

Rendered Images

Statistics

Series Data
# 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.GetPointDataInformation()
arrayInfo = pDinfo.GetArrayInformation("displacement9")
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.ColorArrayName = "displacement9"
    repr.ColorAttributeType = "POINT_DATA"

Statistics
Series Data
Rendered Images
with Field Data
Output Processed Data
Augmented script in input deck.

Simulation
Catalyst
User Perspective
Developer Perspective

Solver

Adaptor

Catalyst

function calls

function calls

INITIALIZE()
ADDPIPELINE(in pipeline)

REQUESTDATALDESCRIPTION(in time, out fields)
COPROCESS(in vtkDataSet)

FINALIZE()
Developer Perspective

![Graphs showing the relationship between cores and slice time (left) and decimate time (right). The graphs demonstrate a decrease in time as the number of cores increases.]
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 a fictitious input
  – Time dependent grid and field data come from a file instead of from an actual simulation

• Examine results
### In Situ Demo – Build Step

CMake 2.8.2 – /Users/ndfabia/Desktop/Work/ParaView_build

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUILD_COPROCESSING_ADAPTERS</td>
<td>✓</td>
</tr>
<tr>
<td>BUILD_FORTRAN_COPROCESSING_ADAPTERS</td>
<td>✓</td>
</tr>
<tr>
<td>BUILD_PARTICLE_COPROCESSING_ADAPTERS</td>
<td></td>
</tr>
<tr>
<td>PARAVIEW_BUILD_PLUGIN_CoProcessingScriptGenerator</td>
<td>✓</td>
</tr>
<tr>
<td>PARAVIEW_ENABLE_COPROCESSING</td>
<td>✓</td>
</tr>
</tbody>
</table>

Press Configure to update and display new values in red, then press Generate to generate selected build files.

Configure  Generate  Current Generator: Unix Makefiles
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 Unstructured Grid Writer
*In Situ* Demo – Creating a Catalyst Python Script

- Load `can.ex2`
  - Note that there are 44 time steps
- Create desired pipeline
In Situ Demo – Adding in Writers

- Parameters:
  - File Name – \%t gets replaced with time step
  - Write Frequency
In Situ Demo – Exporting the Script

Export Co-Processing State

This wizard will guide you through the steps required to export the current visualization state as a python script that can be run in the co-processing component of ParaView. Make sure to add appropriate writers for the desired pipelines to be used in the Writers menu.
**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”)
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
  - 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

```python
def DoCoProcessing (datadescription):
    input = CreateProducer ( datadescription, "input"

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

```python
def CreateProducer (datadescription, gridname):
    *Creates a producer proxy for the grid*
    if not datadescription.GetInputDescriptions().GetName(gridname):
        raise RuntimeWarning, "Must provide a grid name. It does not exist": gridname
    grid = datadescription.GetInputDescriptions().GetName(gridname).GetGrid()
    producer = grid.GetProducer()
    producer.GetInputDescriptions().SetName(gridname)
    producer.UpdateLine(
```

```python
def CreateWriter (XMLMultiBlockDataWriter, filename, line):
    global opwriters
    writer = opwriter()
    writer.GetFileName = filename
    writer.add_distributed_output_file(filename)"op_frequency", free)
    opwriter.append(writer)
    return writer
```
In Situ Demo – Run the Script

- Put candriver.py and the generated Python script in the same directory

- Linux and Mac from a terminal
  - `<path>/pvpython candriver.py <generated script> <path>/can.ex2`

- Windows from a command prompt
  - `start “simple example” <path>/pvpython.exe candriver.py <generated script> <path>/can.ex2`
Live *In Situ* Analysis and Visualization

- Everything before this was “batch”
  - Preset information with possible logic

- “Beta” 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
Live *In Situ* Example

- Linux and Mac from a terminal
  - `<path>/pvpython livecoprocessing.py <path>/can.ex2`

- Windows from a command prompt
  - `start "simple example" <path>/pvpython.exe livecoprocessing.py <path>/can.ex2`

- 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 button](image) on client to stop transferring to client
- Click on ![send button](image) to transfer to client from Catalyst
Gratuitous Catalyst Images
ParaView Catalyst for Developers
ParaView Pipeline Concept

- Fundamental concept in ParaView
- Directed acyclic graph specifying how to process information
- Filters are nodes in the graph
  - Perform a certain action on a data set (grid and fields)
    - Contours, streamlines, file IO, etc.
  - Do not modify input data set
- Catalyst executes user pipelines at specified times
Catalyst Pipelines

- User generated Python scripts from the ParaView plugin
  - Executed with vtkCPPythonScriptPipeline
- Hard-coded pipelines ("canned" output)
  - Executed with a class that derives from vtkCPPipeline
vtkCPPythonScriptPipeline

- Typically from ParaView script generator plugin
- Initialize from a path to a ParaView Catalyst Python script
  - `vtkCPPythonScriptPipeline::Initialize(const char* fileName)`

Advantages

- Easily created through ParaView plugin
  - Output should be moderately readable
- Encourage advanced users to modify
  - Can use ParaView’s Python trace utility to see options
- View/screenshot settings can be difficult to set
  - Camera angle, zoom, lighting, data representations, etc.
- Takes care of parallel image compositing
- Can modify without recompiling
Disadvantages

- Slight overhead compared to C++ hard-coded pipeline
  - Roughly $10^{-5}$ seconds per time step
- Simulation code must be linked with Python
  - Static build issues
  - More complex to minimize executable size
Hard-coded Pipelines

• Derives from vtkCPPipeline
• Generally C++ but could be Python code
• Most are done directly creating VTK filters and connecting them together
  – Creating screen shots from rendering pipeline with compositing can be daunting
• Possible to do using ParaView C++ proxies
  – Low level access is more complex
  – Simpler set up for rendering pipeline
• Less dependencies for compiling and linking

Catalyst

- Catalyst’s job is to create and execute pipelines
Data Structures

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

- Initialization
  - Information for creating pipelines
• After simulation completes time step update
  – Time, time step, force output flag
  – Information for creating grid and field information
Information Flow

- After simulation completes time step update
  - Time, time step, force output flag passed to each pipeline
Information Flow

- After simulation completes time step update
  - Flag indicating which pipelines need to be executed/updated
Information Flow

- After simulation completes time step update
  - If any pipeline needs to be executed
    - Adaptor creates VTK objects to represent grids and fields
After simulation completes time step update
- Pass VTK data object representing grids and fields to pipelines that need to execute/update
Information Flow

- After simulation completes time step update
  - Pipelines execute and output desired information
What is Catalyst?

- Communication/synchronization routines
- Pipeline mechanics
- Data processing through filters
- Writers
- Compositing and rendering
Interfacing with Catalyst

- Catalyst calls should have minimal footprint in simulation code
  - Initialization call
  - Co-processing call
  - Finalize call
- VTK data structures usually not appropriate for simulation code data structures
- Adaptor’s job is to provide interface between simulation code and Catalyst
Adaptor Architectural View

**Adaptor:**
- `vtkCPProcessor* Processor;`
- `vtkDataObject* CreateGrid();`
- `vtkCPPipeline* CreatePipeline();`

**Initialization**
- Creates `vtkCPProcessor` and initializes it
  - `vtkCPProcessor` manages pipelines
  - `vtkCPProcessor::Initialize()`
- Creates `vtkCPPipeline` objects and adds them to `vtkCPProcessor`
  - `vtkCPProcessor::AddPipeline(vtkCPPipeline*)`
Adaptor Architectural View

Adaptor:
vtkCPProcessor* Processor;
vtkDataObject* CreateGrid();
vtkCPPipeline* CreatePipeline();

• When called after update time step is completed
  – Creates a vtkCPDataDescription
    • Information needed by Catalyst to determine what pipelines need to execute
    • vtkCPDataDescription::SetTimeData(double time, vtkIdType timeStep)
    • vtkCPDataDescription::SetForceOutput(bool) optional
  – Queries pipelines to determine if co-processing is necessary
    • Processor->RequestDataDescription(vtkCPDataDescription*)
    • Returns 0 if no co-processing is needed that time step
Adaptor Architectural View

Adaptor:
vtkCPPProcessor* Processor;
vtkDataObject* CreateGrid();
vtkCPPPipeline* CreatePipeline();

• If Processor->RequestDataDescription() != 0
  – CreateGrid()
    • Creates grids and fields
    • Biggest part of the adaptor
  – vtkCPDataDescription::AddInput(const char* name)
    • Convention is that name := “input” for a single input
  – vtkCPDataDescription::GetInputDataDescriptionByName(const char* name)->SetGrid( CreateGrid() )
  – Processor->CoProcess(vtkCPDataDescription*)
Adaptor Architectural View

Adaptor:
vtkCPPProcessor* Processor;
vtkDataObject* CreateGrid();
vtkCPPPipeline* CreatePipeline();

• Finalize
  – Processor->Finalize()
Main Co-Processing Parts

• Initialize
• Check with each Catalyst pipeline if execution is necessary
  – Input is simulation time and time step
  – Output is if co-processing is needed along with needed grids and fields
• If co-processing is needed at specified time and time step:
  – Input the grid and field information
  – Execute the proper Catalyst pipelines
• Finalize
```cpp
int main(int argc, char* argv[]) {
  MPI_Init(&argc, &argv);
  std::string cpPythonFile = argv[1];
  int nSteps = atoi(argv[2]);
  vtkCPProcessor* processor = vtkCPProcessor::New();
  processor->Initialize();
  vtkCPPythonScriptPipeline* pipeline =
      vtkCPPythonScriptPipeline::New();

  // read the coprocessing python file
  if (pipeline->Initialize(cpPythonFile.c_str()) == 0) {
    cout << "Problem reading the python script.\n";
    return 1;
  }
  processor->AddPipeline(pipeline);
  pipeline->Delete();

  if (nSteps == 0) {
    return 0;
  }
...```
```cpp
... 
double tStart = 0.0;
double tEnd = 1.0;
double stepSize = (tEnd - tStart)/nSteps;

vtkCPDataDescription* dataDesc = vtkCPDataDescription::New();
dataDesc->AddInput("input");

for (int i = 0; i < nSteps; ++i)
{
    double currentTime = tStart + stepSize*i;
    // set the current time and time step
    dataDesc->SetTimeData(currentTime, i);

    // check if the script says we should do coprocessing now
    if(processor->RequestDataDescription(dataDesc) != 0)
    {
        // create our vtkDataObject with the grids and fields
        vtkDataObject* dataObject = <generate grid>;
        dataDesc->GetInputDescriptionByName("input")->SetGrid(dataObject);
        processor->CoProcess(dataDesc);
    }
}

dataDesc->Delete();
processor->Finalize();
processor->Delete();

MPI_Finalize();

return 0;
```
Creating VTK Objects
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/vtkUnstructuredGrid`, `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

- 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 (3, 8, 3, 6, 0, 0)
  - From rank 2 in previous slide
  - 28 points and 18 cells
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
- SetOrigin() – location of point 0 (i=0, j=0, k=0)

- vtkUniformGrid
  - Supports cell blanking
  - Currently written to file as vtkImageData

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

Irregular Geometry

- Data sets that derive from vtkPointSet
  - vtkStructuredGrid – still topologically regular
  - vtkPolyData – topologically irregular with 0D, 1D and 2D cell types
  - vtkUnstructuredGrid – topologically irregular with 0D, 1D, 2D and 3D cell types

- Uses vtkPoints to specify grid’s point locations
  - SetPoints(vtkPoints*) – set the number and coordinates of a process’s points
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() – non-grid associated arrays
• 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
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()

http://www.vtk.org/doc/nightly/html/classvtkMultiBlockDataSet.html
vtkMultiPieceDataSet

• Way to combine multiple vtkDataSets of the same type into a single logical object
  – Useful when the amount of pieces is unknown \textit{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
  - vtkCPIInputDataDescription::SetWholeExtent() – same on each process
void CreateGrid(vtkCPDataDescription* dataDescription, int numPoints, double* coordsArray, int numCells, int* cellConnectivity, double* dofArray)
{
    vtkUnstructuredGrid* grid = vtkUnstructuredGrid::New();
    vtkPoints* nodePoints = vtkPoints::New();
    vtkDoubleArray* coords = vtkDoubleArray::New();
    coords->SetNumberOfComponents(3);
    coords->SetNumberOfTuples(numPoints);
    for(int i=0; i<*numPoints; i++)
    {
        double tuple[3] = {coordsArray[i],
                            coordsArray[i+numPoints],
                            coordsArray[i+numPoints*2]};
        coords->SetTupleValue(i, tuple);
    }
    nodePoints->SetData(coords);
    coords->Delete();
    grid->SetPoints(nodePoints);
    nodePoints->Delete();
...
vtkUnstructuredGrid Example (2)

- Only tetrahedra in this example
- Canonical ordering of tetrahedra is same in simulation data structures and VTK

```c
vtkIdType pts[4];
for(int iCell=0;iCell<numCells;iCell++)
{
    for(int i=0;i<4;i++)
    {
        pts[i] = cellConnectivity[iCell+i*numCells];
    }
    grid->InsertNextCell(VTK_TETRA, 4, pts);
}
dataDescription->GetInputDescriptionByName("input")
    ->SetGrid(grid);
grid->Delete();
...
vtkUnstructuredGrid Example (3)

...  
vtkCPInputDataDescription* idd =
dataDescription->GetInputDescriptionByName("input");
if(idd->IsFieldNeeded("velocity")) {
  vtkDoubleArray* velocity = vtkDoubleArray::New();
  velocity->SetName("velocity");
  velocity->SetNumberOfComponents(3);
  velocity->SetNumberOfTuples(numPoints);
  for (vtkIdType idx=0; idx<numPoints; idx++) {
    velocity->SetTuple3(idx, dofArray[idx],
                       dofArray[idx+ numPoints], dofArray[idx+ 2*numPoints]);
  }
  grid->GetPointData()->AddArray(velocity);
  velocity->Delete();
}
if(idd->IsFieldNeeded("pressure")) {
  vtkDoubleArray* pressure = vtkDoubleArray::New();
  pressure->SetName("pressure");
  pressure->SetArray(dofArray+3*numPoints, numPoints, 1);
  grid->GetPointData()->AddArray(pressure);
  pressure->Delete();
}