In Situ Visualization using VisIt

Brad Whitlock  
Lawrence Livermore National Laboratory

Jean M. Favre  
Swiss National Supercomputing Centre

Jeremy S. Meredith  
Oak Ridge National Laboratory

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.
What We’re Doing

- We have created a library that lets simulations interface to a fully featured parallel visualization system
  - One goal is to avoid the high costs of I/O associated with writing and then reading the data
  - Another goal is to interactively explore data
- We have used our library to instrument a parallel cosmology code to investigate some of its performance aspects compared to doing I/O
Case For Using In Situ

- I/O in supercomputers has not kept pace with compute power
- Some applications report 90% of time spent in I/O [Peterka et al.]
- Post processing simulation files requires write then read, paying for I/O twice in different application
- In Situ may let us avoid some I/O

<table>
<thead>
<tr>
<th>Machine</th>
<th>Year</th>
<th>Writable FLOPS</th>
<th>Whole-System Checkpoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASCI Red</td>
<td>1997</td>
<td>0.075%</td>
<td>300 sec</td>
</tr>
<tr>
<td>ASCI Blue Pacific</td>
<td>1998</td>
<td>0.041%</td>
<td>400 sec</td>
</tr>
<tr>
<td>ASCI White</td>
<td>2001</td>
<td>0.026%</td>
<td>480 sec</td>
</tr>
<tr>
<td>ASCI Red Storm</td>
<td>2005</td>
<td>0.035%</td>
<td>660 sec</td>
</tr>
<tr>
<td>ASCI Purple</td>
<td>2005</td>
<td>0.025%</td>
<td>500 sec</td>
</tr>
<tr>
<td>NCCS XT4</td>
<td>2007</td>
<td>0.004%</td>
<td>1400 sec</td>
</tr>
<tr>
<td>Roadrunner</td>
<td>2008</td>
<td>0.005%</td>
<td>480 sec</td>
</tr>
<tr>
<td>NCCS XT5</td>
<td>2008</td>
<td>0.005%</td>
<td>1250 sec</td>
</tr>
<tr>
<td>ASC Sequoia</td>
<td>2012 (planned)</td>
<td>0.001%</td>
<td>3200 sec</td>
</tr>
</tbody>
</table>
A Marriage Between Two Fairly Inflexible Partners…

Our library enables a general purpose visualization tool to be flexibly coupled with a simulation.
In Situ Processing Strategies

We find 3 main strategies for in situ processing:

<table>
<thead>
<tr>
<th>In Situ Strategy</th>
<th>Description</th>
<th>Negative Aspects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loosely coupled</td>
<td>Visualization and analysis run on concurrent resources and access data over network</td>
<td>1) Data movement costs 2) Requires separate resources</td>
</tr>
<tr>
<td>Tightly coupled</td>
<td>Visualization and analysis have direct access to memory of simulation code</td>
<td>1) Very memory constrained 2) Large potential impact (performance, crashes)</td>
</tr>
<tr>
<td>Hybrid</td>
<td>Data is reduced in a tightly coupled setting and sent to a concurrent resource</td>
<td>1) Complex 2) Shares negative aspects (to a lesser extent) of others</td>
</tr>
</tbody>
</table>
Loosely Coupled In Situ Processing

- I/O layer stages data into secondary memory buffers, possibly on other compute nodes
- Visualization applications access the buffers and obtain data
- Separates visualization processing from simulation processing
- Copies and moves data
Tightly Coupled Custom In Situ Processing

- Custom visualization routines are developed specifically for the simulation and are called as subroutines
  - Create best visual representation
  - Optimized for data layout
- Tendency to concentrate on very specific visualization scenarios
- *Write once, use once*
Tightly Coupled General In Situ Processing

- Simulation uses data adapter layer to make data suitable for general purpose visualization library
- Rich feature set can be called by the simulation
- Operate directly on the simulation’s data arrays when possible
- *Write once, use many times*
Which Strategy is Appropriate?

There have been many excellent solutions in this space. Different circumstances often merit different solutions.

<table>
<thead>
<tr>
<th></th>
<th>Tightly Coupled</th>
<th>Loosely Coupled</th>
<th>Hybrid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Custom</td>
<td>✗</td>
<td></td>
<td></td>
</tr>
<tr>
<td>General</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>
Design Philosophy

- Visualization and analysis will be done in the same memory space as the simulation on native data to avoid duplication
- Maximize features and capabilities
- Minimize code modifications to simulations
- Minimize impact to simulation codes
- Allow users to start an in situ session on demand instead of deciding before running a simulation
  - Emphasis on interactive exploration
  - Debugging
  - Computational steering
Selecting an In Situ Strategy

- Our strategy is tightly coupled, yet general
- Fully featured visualization code connects interactively to running simulation
  - Allows live exploration of data for when we don’t know visualization setup a priori
  - Opportunities for steering
- We chose VisIt as the visualization code
  - VisIt runs on several HPC platforms
  - VisIt has been used at many levels of concurrency
  - We know how to develop for VisIt
Visualization Tool Architecture

- Clients run locally and display results computed on the server.
- Server runs remotely in parallel, handling data processing for clients.

- Data processed in data flow networks.
- Filters in data flow networks can be implemented as plug-ins.
Coordination Among Filters Using Contracts

**Diagram:**
- **Source**
- **Filter**
- **Contract\_0**
- **Contract\_1**
- Data flow:
  - **Update**
  - **Execute**

**Text:**

Coordination Among Filters Using Contracts.
We created Libsim, a library that simulations use to let VisIt connect and access their data.
Libsim Implements Tight Coupling

- Front end library controls access and plotting
- Data requested on demand through user-supplied *Data Access Code* callback functions
- Data shared via pointers
- Ported to Linux, Windows, and MacOS X
- Distributed in every version of VisIt

**Local VisIt Clients**

**Parallel Cluster**

- Simulation Code
  - Front end
  - Libsim Runtime
  - Data Access Code
- Data

Save image
In Situ Processing Workflow

1. The simulation code launches and starts execution
2. The simulation regularly checks for connection attempts from visualization tool
3. The visualization tool may connect to the simulation or the simulation may save images by itself
4. The simulation provides a description of its meshes and data types
5. Visualization operations are handled via Libsim and result in data requests to the simulation
Additions to the source code are usually minimal, and follow three incremental steps:

- Initialize Libsim and alter the simulation’s main iterative loop to listen for connections from VisIt.
- Create *data access callback* functions so simulation can share data with Libsim.
- Add control functions that let VisIt steer the simulation.
Adapting the Main Loop

- Libsim opens a socket and writes out connection parameters
- VisItDetectInput checks for:
  - Connection request
  - VisIt commands
  - Console input
Sharing Data

- **VisIt** requests data on demand through *Data Access Callback Functions*
  - The **best-suited** simulations allocate large contiguous memory arrays
  - Return actual pointers to your simulation’s data
  - Return alternate representation that Libsim can free
  - Written in C, C++, Fortran, Python

![Diagram showing data flow for C/C++, Fortran, and Python simulations](image)
Sharing Data Example

// Example Data Access Callback
visit_handle GetVariable(int domain, char *name, void *cbdata)
{
    visit_handle h = VISIT_INVALID_HANDLE;
    SimData_t *sim = (SimData_t *)cbdata;
    if(strcmp(name, "pressure") == 0)
    {
        VisIt_VariableData_alloc(&h);
        VisIt_VariableData_setDataD(h, VISIT_OWNER_SIM, 
                                     1, sim->nx*sim->ny,
                                     sim->pressure);
    }
    return h;
}
Data Access Callbacks

- GetMetaData
- GetMesh
- GetVariable
- GetMaterial
- GetSpecies
- GetDomainList
- GetDomainBoundaries
- GetDomainNesting

Most simulations implement these

- Simulations that use materials
- Parallel simulations
- AMR simulations

There are also write callbacks that can be used to “export” the processed data back to the simulation
Supported Data Model

- **Mesh Types**
  - Structured meshes
  - Point meshes
  - CSG meshes
  - AMR meshes
  - Unstructured & Polyhedral meshes

- **Variables**
  - 1 to N components
  - Zonal and Nodal

- **Materials**
- **Species**
Adding Control Functions

- The simulation provides commands to which it will respond.
- Commands generate user interface controls in Simulations Window.
Custom User Interfaces

- Simulation can provide UI description to generate custom simulation window in VisIt
Libsim in Practice

- We instrumented GADGET-2, a popular cosmology code, with Libsim and measured performance
- We conducted our experiments on a 216 node visualization cluster
  - Two, 6 core 2.8GHz Intel Xeon 5660 processors
  - 96Gb of memory per node
  - InfiniBandQDR high-speed interconnect
  - Lustre parallel file system
- We measured the impact of Libsim on a simulation’s main loop, without connecting to VisIt
- We measured memory usage after loading VisIt
Impact on the Main Loop

- Measure cost of calling Libsim in the main loop
- Instrumenting the main loop for a parallel simulation requires calling `VisItDetectInput` and `MPI_Bcast`
  - We timed how long it took to call both using 512 cores
  - 10K main loop iterations

<table>
<thead>
<tr>
<th>Cores</th>
<th><code>VisItDetectInput</code> overhead</th>
<th><code>MPI_Bcast</code> overhead</th>
<th>Overhead loading VisIt runtime libraries</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>2µs</td>
<td>8µs</td>
<td>1s (1 time cost)</td>
</tr>
</tbody>
</table>

Can now avoid penalty by static linking
## Impact on Memory Usage

- Measure memory used before and after VisIt is connected
- Measured our updateplots example program
- Read values from /proc/<pid>/smaps

<table>
<thead>
<tr>
<th>Event</th>
<th>Size</th>
<th>Resident Set Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulation startup</td>
<td>8.75 Mb</td>
<td>512 Kb</td>
</tr>
<tr>
<td>After Libsim Initialization</td>
<td>8.75 Mb</td>
<td>614 Kb</td>
</tr>
<tr>
<td>After Loading VisIt</td>
<td>222 Mb</td>
<td>43.5 Mb</td>
</tr>
</tbody>
</table>
Libsim/GADGET-2 Timing Results

- In situ competitive or faster than single file I/O with increasing cores
- It should be possible to do several in situ operations in the time needed for I/O
- Time savings compared to simulation followed by post processing

<table>
<thead>
<tr>
<th>16M particles</th>
<th>32 cores</th>
<th>256 cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>I/O 1 file</td>
<td>2.76s</td>
<td>4.72s</td>
</tr>
<tr>
<td>I/O N files</td>
<td>0.74s</td>
<td>0.31s</td>
</tr>
<tr>
<td>In situ</td>
<td>0.77s</td>
<td>0.34s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>100M particles</th>
<th>32 cores</th>
<th>256 cores</th>
<th>512 cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>I/O 1 file</td>
<td>24.45s</td>
<td>26.7s</td>
<td>25.27s</td>
</tr>
<tr>
<td>I/O N files</td>
<td>0.69s</td>
<td>1.43s</td>
<td>2.29s</td>
</tr>
<tr>
<td>In situ</td>
<td>1.70s</td>
<td>0.46s</td>
<td>0.64s</td>
</tr>
</tbody>
</table>

- I/O results are the average of 5 runs per test case
- In Situ results are averaged from timing logs for multiple cores
VisIt runs well at massive scale on diverse architectures

- 8K-32K cores
- Weak scaling study ~62.5M cells/core

Table 2. Performance across diverse architectures.

<table>
<thead>
<tr>
<th>Machine</th>
<th>No. of cores</th>
<th>Data set size (TCells)</th>
<th>Total I/O time (sec.)</th>
<th>Contour time (sec.)</th>
<th>Total pipeline execution time (sec.)</th>
<th>Rendering time (sec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purple</td>
<td>8,000</td>
<td>0.5</td>
<td>53.4</td>
<td>10.0</td>
<td>63.7</td>
<td>2.9</td>
</tr>
<tr>
<td>Dawn</td>
<td>16,384*</td>
<td>1.0</td>
<td>240.9</td>
<td>32.4</td>
<td>277.6</td>
<td>10.6</td>
</tr>
<tr>
<td>Juno</td>
<td>16,000</td>
<td>1.0</td>
<td>102.9</td>
<td>7.2</td>
<td>110.4</td>
<td>10.4</td>
</tr>
<tr>
<td>Ranger</td>
<td>16,000</td>
<td>1.0</td>
<td>251.2</td>
<td>8.3</td>
<td>259.7</td>
<td>4.4</td>
</tr>
<tr>
<td>Franklin</td>
<td>16,000</td>
<td>1.0</td>
<td>129.3</td>
<td>7.9</td>
<td>137.3</td>
<td>1.6</td>
</tr>
<tr>
<td>JaguarPF</td>
<td>16,000</td>
<td>1.0</td>
<td>236.1</td>
<td>10.4</td>
<td>246.7</td>
<td>1.5</td>
</tr>
<tr>
<td>Franklin</td>
<td>32,000</td>
<td>2.0</td>
<td>292.4</td>
<td>8.0</td>
<td>300.6</td>
<td>9.7</td>
</tr>
<tr>
<td>JaguarPF</td>
<td>32,000</td>
<td>2.0</td>
<td>707.2</td>
<td>7.7</td>
<td>715.2</td>
<td>1.5</td>
</tr>
</tbody>
</table>

* Dawn requires that the number of cores be a power of two.
† This measure indicates the time to produce the surface.
Additional Results

- We have recently instrumented additional simulations to investigate Libsim’s scaling properties on a Cray XE6 using up to 4224 cores.
- We identified and corrected a performance bottleneck in Libsim’s environment detection functions.

![Graph showing time to detect environment before and after fix.](Image)
Additional Results

- Simulation was run on 11, 22, 44, 88 and 176 nodes (24 cores/node)
- Each MPI task had a 512x512x100 block of data to isocontour at 10 different thresholds
- Parallel I/O to disk was done with netCDF-4, in files of size 27, 55, 110, 221, and 442 Gb per iteration

![Graph showing time per iteration vs number of cores](image-url)
Work in progress

- Adding functions to create plots and operators so simulation can set up visualization without user intervention
- Libsim runtime loaded automatically; no VisIt session needed

```c
int pcId = -1, sliceId = -1;
VisItAddPlot("Pseudocolor", "pressure", &pcId);
VisItAddOperator(pcId, "Slice");
VisItSetPlotOptionsS(pcId, "colorTable", "rainbow");
VisItDrawPlots(pcId);
VisItSaveWindow("image0000.png", 800, 800);
```
Limitations of Implementation

- Memory intensive
  - Runtime library cost is larger than with static-linking since we use the whole feature set
  - Filters may use intermediate memory
  - Zero-copy is not fully implemented
- Works best with an interactive session
Future Work

- Continue adding functions for setting up visualization so in situ processing can be less user-driven
- Further limit resources consumed by the VisIt runtime libraries in order to lessen the impact that in situ analysis has on the simulation
- Characterize performance costs of using shared libraries on larger scale runs
- Simplify static linking
Conclusion

- We have implemented Libsim, an easy to use library that enables in situ computations
  - Provides access to a fully featured, parallel visualization and analysis tool that excels at scale
  - Minimizes impact to simulation performance
  - Minimizes the amount of new code that must be written
  - Fully integrated with the open-source distribution of VisIt