Damaris/Viz: a Nonintrusive, Adaptable and User-Friendly In Situ Visualization Framework

Matthieu Dorier, Robert Sisneros, Tom Peterka, Gabriel Antoniu, Dave Semeraro
Outline

• In situ visualization
• Motivation/Damaris
• Damaris/Viz
  – Instrumentation details
  – Results
In Situ Terminology

• In situ visualization: Analysis and vis tasks run concurrent to a simulation
  – Bypasses storage system
  – Generates results while simulation runs

• In situ flavors:
  – Tightly-coupled: analysis+vis share simulation’s resources
    • Time-partitioning: sim stops, vis code takes over
    • Space-partitioning: sim resources are dedicated to vis
  – Loosely-coupled: sim and vis resources are separated
Damaris/Vis

- In situ visualization: Analysis and vis tasks run concurrent to a simulation
  - Bypasses storage system
  - Generates results while simulation runs

- In situ flavors:
  - **Tightly-coupled**: analysis+vis share simulation’s resources
    - Time-partitioning: sim stops, vis code takes over
    - **Space-partitioning**: sim resources are dedicated to vis
  - Loosely-coupled: sim and vis resources are separated
Deployment of In Situ Techniques

• Who has the simulations? Domain scientists.

• What do they want? An in situ framework that has…
  – Low impact on the code
  – High adaptability
  – Low impact on run time
  – Optimized resource utilization
Damaris/Viz

• Reduce code impact while maximizing capabilities
  – Provide unified data management interface
  – Incorporate existing functionality through this interface

• Hide performance impact of in situ tasks
Damaris: I/O Middleware

Damaris -- Dedicated Adaptable Middleware for Application Resources Inline Steering

• Started in 2010 by Matthieu Dorier during an internship at NCSA

• The purpose: decouple I/O and computation to enable scalable asynchronous I/O

• The approach: dedicated I/O core(s) on each node
  – Limits OST contention to the node level
  – Leverages shared memory for efficient interaction
  – When simulation “writes” data, Damaris utilizes shared memory to effectively aggregate writes to the “right” size
Damaris at a Glance
Application: Reducing I/O Jitter in CM1

- I/O Jitter is the variability in I/O operations that arise from any number of common interferences.

- CM1
  - Atmospheric simulation
  - Uses serial and parallel HDF5
  - Tunable I/O
How Damaris Helps with I/O Jitter

- Jitter is “moved” to the dedicated core
- Even with the reduction in number of cores performing computation, performance is not adversely affected, in fact….
Results: Damaris+CM1

- In these runs, Damaris spent at least 75% its time waiting!
- Plugins are intended to use this time

Weak scalability factor: \[ S = N \frac{T_{base}}{T} \]

- \( N \): number of cores
- \( T_{base} \): time of an iteration on one core w/ write
- \( T \): time of an iteration + a write
The In Situ Framework

DAMARIS/VIZ
Damaris/Viz: The Components

- Damaris client: the simulation
- Damaris server
  - Manages shared memory
  - Calls plugin to perform in situ tasks
In Situ Plugins: The State of Some of the State of the Art

- Add functions to your simulation that let VisIt connect
- Add functions to your simulation that expose your arrays as data VisIt will process
- Link your simulation with libsim

Run the simulation and connect with VisIt
- You will be able to perform any of VisIt’s operations on your simulation data
- Advance the simulation and watch your plots update

- **libsim features**
  - Species
  - Vector, Tensor data
  - AMR meshes
  - CSG meshes
  - Users don’t allocate memory
  - Additional error checking
  - Write in C, Fortran, or Python
  - Windows support
libsim, Informally

- Time-partitioning
- VisIt treats running simulation as a VisIt engine
  - Fully functioning VisIt is at your control
  - Instrumentation requires enough to convince VisIt there is a fully functioning engine available
Instrumenting libsim

<table>
<thead>
<tr>
<th>Simulation</th>
<th>libsim Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>curve.c</td>
<td>C 144 lines</td>
</tr>
<tr>
<td>mesh.c</td>
<td>167 lines</td>
</tr>
<tr>
<td>var.c</td>
<td>271 lines</td>
</tr>
<tr>
<td>life.c</td>
<td>305 lines</td>
</tr>
</tbody>
</table>

Damaris plugin: “VisIt”
Current Plugins

• VisIt
• ParaView (under development)
• Python

```python
var = damaris.open("temperature")
for chunks in var.select( iteration = 1 )
    print numpy.average(chunks.data)
```
Using Damaris/Viz

- Create XML description of data
- Instrument simulation with Damaris code
  - Simulation uses provided Damaris API
  - Allows Damaris to correctly manage shared memory
- Call in situ plugin
  - Set up necessary data structures, etc.
XML Data Files

• Generic description of structures to visualize
  – Meshes, points, curves, scalars, etc.

• Data model based on survey of existing in situ interfaces
  – ParaView, VisIt, ezViz, VTK
Example Data File

```xml
<variable name="mesh_x" .../>
<variable name="mesh_y" .../>
<variable name="mesh_z" .../>

<mesh type="rectilinear" name="my mesh">
  <coord name="mesh_x" unit="cm" label="width" />
  <coord name="mesh_y" unit="cm" label="height" />
  <coord name="mesh_z" unit="cm" label="depth" />
</mesh>

<variable name="temperature" mesh="my_mesh" ../>
```
## Describing Data

<table>
<thead>
<tr>
<th>Simulation</th>
<th>libsim Code</th>
<th>Damaris Code</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>C</td>
<td>XML</td>
</tr>
<tr>
<td>curve.c</td>
<td>144 lines</td>
<td>31 lines</td>
</tr>
<tr>
<td>mesh.c</td>
<td>167 lines</td>
<td>39 lines</td>
</tr>
<tr>
<td>var.c</td>
<td>271 lines</td>
<td>53 lines</td>
</tr>
<tr>
<td>life.c</td>
<td>305 lines</td>
<td>39 lines</td>
</tr>
</tbody>
</table>
Instrumentation

```c
float* mesh_x = dc_alloc("mesh x");
float* mesh_y = dc_alloc("mesh y");
float* mesh_z = dc_alloc("mesh z");
double* temp = dc_alloc("temperature");
...
dc_commit("temperature");
...
dc_end_iteration();
```
## Code Impact

<table>
<thead>
<tr>
<th>Simulation</th>
<th>libsim Code</th>
<th>Damaris Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>curve.c</td>
<td>144 lines</td>
<td>31 lines</td>
</tr>
<tr>
<td>mesh.c</td>
<td>167 lines</td>
<td>39 lines</td>
</tr>
<tr>
<td>var.c</td>
<td>271 lines</td>
<td>53 lines</td>
</tr>
<tr>
<td>life.c</td>
<td>305 lines</td>
<td>39 lines</td>
</tr>
</tbody>
</table>

These code changes are compatible with any Damaris in situ plugin!
More on Instrumentation

- \texttt{dc\_alloc("variable")} – allocate shared memory, still only visible to simulation
- \texttt{dc\_commit("variable")} – simulation is finished writing data, Damaris can now read
- \texttt{dc\_end\_iteration("variable")} – lets Damaris know that variable may be processed, stored, removed, etc.

```

Variable  Variable  Variable

Shared Memory  Shared Memory  Shared Memory
```

(a) After \texttt{dc\_alloc}
(b) After \texttt{dc\_commit}
(c) After \texttt{dc\_end\_iteration}
Blocking, Non-blocking, and Revisiting In Situ Terminology

• Blocking
  – If shared memory is full, \texttt{dc\_alloc} waits
  – Intuitive space-partitioning, nothing is lost

• Non-Blocking
  – If shared memory is full, \texttt{dc\_alloc} uses “local memory”
  – \texttt{dc\_commit} has no effect
  – \texttt{dc\_end\_iteration} frees memory
  – In situ plugin never runs on this data
  – Our default space-partitioning
Results

PERFORMANCE EVALUATION
Time vs. Space

• Test applicability of space-partitioning (dependent on vis algorithm scalability)

• Setup:
  – CM1 I/O kernel
    • Avoid low probability of something interesting happening
    • Avoid wait in the event it does
    • Data grid size: 3840x3840x400
    • Run on Blue Waters from 16 to 6400 cores (ppn=16)
  – Two visualization scenarios using VisIt to view dbz field
    • Ray cast volume rendering
    • 10-level contour plot
Volume Rendering

Space-partitioning faster!

Contours

Space-partitioning 16X more efficient!
Space in Practice

- Nek5000
  - CFD
  - Fortran 77
  - Curvilinear mesh
- This experiment:
  - TurbChannel configuration (designed for 32 to 64 cores)
  - Run on Reims stremi cluster of French Grid’5000 48 cores (ppn = 24)
Time-partitioning

Negative performance impact

Space-partitioning

In situ completely transparent!
Providing a (Low-calorie) Free Lunch

- Nek5000 MATiS configuration
  - Designed for 512 to 2048 cores
  - Run on 816 cores

No Vis:
Average iteration: 
~1 minute

Lots of Vis:
Average iteration: 
~3.5 minutes

Some Vis:
Average iteration: 
~1 minute
How Much Vis?

- 1 image per ~25 iterations
  - Acceptable to Nek5000 users
  - Free! Free! Free!

No Vis:
Average iteration: ~1 minute

Lots of Vis:
Average iteration: ~3.5 minutes

Some Vis:
Average iteration: ~1 minute
The Last Slide

• Regardless of in situ’s issues – it presents an import direction for research (as doom approaches)
• The core concepts in this work represent target areas for refinement
  – These double as directives for adoption
  – More free-er the better

• Check out Damaris:
  – http://damaris.gforge.inria.fr/doku.php