Uintah Framework

Justin Luitjens, Qingyu Meng, John Schmidt, Martin Berzins, Todd Harman, Chuch Wight, Steven Parker, et al
Uintah Parallel Computing Framework

- **Uintah - far-sighted design by Steve Parker**:  
  - Component based design
    - Separated development
    - Swap components in and out
    - Code reuse
  - Automated parallelism
    - Engineer only writes “serial” code for a hexahedral patch
    - Complete separation of user code and parallelism
    - Asynchronous communication, message coalescing
    - Hybrid MPI/Threading
  - AMR Support
    - Automated load balancing & regridding
  - Multiple Simulation Components
    - ICE, MPM, Arches, MPMICE, et al.
  - Simulation of a broad class of fluid-structure interaction problems
Uintah Applications

- Plume Fires
- Angiogenesis
- Industrial Flares
- Explosions
- Micropin Flow
- Sandstone Compaction
- Virtual Soldier
- Shaped Charges
- Foam Compaction
How Does Uintah Work?

Task-Graph Specification
• Computes & Requires

Patch-Based Domain Decomposition
How Does Uintah Work?

- **Regridder**
- **Simulation Controller**
  - *Simulation (Arches, ICE, MPM, MPMICE, MPMArches, ...)*
  - **Scheduler**
  - **Load Balancer**
  - **Data Archiver**
  - **Models (EoS, Constitutive, ...)**
  - **Tasks**
  - **MPI**
  - **Checkpoints Data I/O**
  - **Callbacks**

- **Problem Specification**
- **XML**
- **Domain Expert**
- **Tuning Expert**
1) **Static**: Predetermined order
   • Tasks are Synchronized
   • Higher waiting times
Task Graph Execution

1) **Static**: Predetermined order
   • Tasks are Synchronized
   • Higher waiting times

2) **Dynamic**: Execute when ready
   • Tasks are Asynchronous
   • Lower waiting times (up to 25%)
Task Graph Execution

1) **Static**: Predetermined order
   - Tasks are Synchronized
   - Higher waiting times

2) **Dynamic**: Execute when ready
   - Tasks are Asynchronous
   - Lower waiting times (up to 25%)

3) **Dynamic Multi-threaded**:
   - Task-Level Parallelism
   - Decreases Communication
   - Decreases Load Imbalance
Tiled Regridding Algorithm

- Use fixed sized tiles
  - Occur at regular intervals
  - Can exploit regularity
    - Neighbor finding
    - Grid Comparisons

\[
\text{FOR each tile} \\
\quad \text{FOR each cell in tile} \\
\quad \quad \text{IF cell has refinement flag} \\
\quad \quad \quad \text{patches.add(tile)} \\
\quad \quad \quad \text{BREAK} \\
\quad \quad \text{END IF} \\
\quad \text{END FOR} \\
\text{END FOR}
\]

Trivial to paralleize
- Computation: $O(C/P)$
- Communication: None!
- Faster than creating the flags list!
Regridder Comparison

Berger-Rigoutsos
• Global algorithm
• Computation will not weak scale
• Communication will not weak or strong scale
• $O(\text{Patches})$ All reduces!
• Irregular patches
• Complex implementation

Tiled
• Local Algorithm
• Computation will weak & strong scale
• No communication
• Simple implementation
• Regular patches
• More Patches
• Over-refines
Uintah Load Balancing

• Assign Patches to Processors
  – Minimize Load Imbalance
  – Minimize Communication
  – Run Quickly in Parallel

• Uintah Default: Space-Filling Curves
  – $O((N \log N)/P + (N \log^2 P)/P$

• Support for Zoltan

In order to assign work evenly we must know how much work a patch requires
Cost Estimation: Performance Models

\[ E_{r,t} = c_1 G_r + c_2 P_r + c_3 \]

- \( E_{r,t} \): Estimated Time
- \( G_r \): Number of Grid Cells
- \( P_r \): Number of Particles
- \( c_1, c_2, c_3 \): Model Constants

- Need to be proportionally accurate
- Vary with simulation component, sub models, compiler, material, physical state, etc.

Can estimate constants using least squares at runtime

\[
\begin{bmatrix}
G_0 & P_0 & 1 \\
... & ... & ... \\
G_n & P_n & 1
\end{bmatrix}
\begin{bmatrix}
c_1 \\
c_2 \\
c_3
\end{bmatrix}
= 
\begin{bmatrix}
O_{0,t} \\
... \\
O_{n,t}
\end{bmatrix}
\]

- \( O_{r,t} \): Observed Time

What if the constants are not constant?
Cost Estimation: Fading Memory Filter

\[ E_{r,t+1} = \alpha O_{r,t} + (1 - \alpha) E_{r,t} \]

- \( E_{r,t} \): Estimated Time
- \( O_{r,t} \): Observed Time
- \( \alpha \): Decay Rate

Error in last prediction

- No model necessary
- Can track changing phenomena
- May react to system noise
- Also known as:
  - Simple Exponential Smoothing
  - Exponential Weighted Average

Compute per patch
Cost Estimation: Kalman Filter, $0^{th}$ Order

$E_{r,t}$: Estimated Time  \hspace{1cm}  $O_{r,t}$: Observed Time

Update Equation:  \hspace{1cm} $E_{r,t+1} = E_{r,t} + K_{r,t} (O_{r,t} - E_{r,t})$

Gain:  \hspace{1cm} $K_{r,t} = M_{r,t} / (M_{r,t} + \sigma^2)$

a priori cov: \hspace{1cm} $M_{r,t} = P_{r,t-1} + \phi$

a posteri cov: \hspace{1cm} $P_{r,t} = (1 - K_{r,t}) M_{r,t}$ \hspace{1cm} $P_0 = \infty$

- Accounts for uncertainty in the model: $\phi$
- Accounts for uncertainty in the measurement: $\sigma^2$
- No model necessary
- Can track changing phenomena
- May react to system noise
- Faster convergence than fading memory filter
Cost Estimation Comparison

Mean Absolute Percent Error

Exploding Container

Material Transport

- Filters provide best estimate
- Filters can spike with system noise

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Model LS</td>
<td>6.08</td>
<td>7.63</td>
</tr>
<tr>
<td>Memory</td>
<td>3.95</td>
<td>3.10</td>
</tr>
<tr>
<td>Kalman</td>
<td>3.44</td>
<td>2.01</td>
</tr>
</tbody>
</table>

Justin Luitjens and Martin Berzins, Improving the Performance of Uintah: A Large-Scale Adaptive Meshing Computational Framework, Accepted in IPDPS 2010.
AMR ICE Scalability

AMR-ICE Scaling

Mean Time Per Timestep [sec.]

Processors

12 24 48 96 192 384 768 1536 3072 6144 12288 24576 49152 98304

Strong
Weak
AMR MPMICE Scalability

Decent MPMICE scaling

More work is needed

One $8^3$ patch per processor

Problem: Exploding Container