Evaluation and Optimization of a Titanium Adaptive Mesh Refinement

Amir Kamil
Ben Schwarz
Jimmy Su
Adaptive Mesh Refinement

- AMR uses hierarchy of grids, with fine grids nested inside coarse ones.
- Grids are refined only where extra resolution is needed.

- Grid-based V-cycle code written in Titanium
- Severely limited by communication overheads
  - Communication is between levels on the V-cycle: Interpolation
  - Communication within levels of a V-cycle: Boundary Exchange
Unix Processes vs. Pthreads

- Pthreads expected to perform better, due to shared memory communication.
- However, some parts of the code perform worse with Pthreads.

Time in seconds (processes/Pthreads per process)

<table>
<thead>
<tr>
<th></th>
<th>1/1</th>
<th>2/1</th>
<th>1/2</th>
<th>4/1</th>
<th>1/4</th>
<th>8/1</th>
<th>1/8</th>
</tr>
</thead>
<tbody>
<tr>
<td>SolveAMR</td>
<td>235.4</td>
<td>164.0</td>
<td>140.3</td>
<td>147.3</td>
<td>91.05</td>
<td>128.3</td>
<td>87.47</td>
</tr>
<tr>
<td>Exchange</td>
<td>84.16</td>
<td>84.68</td>
<td>48.10</td>
<td>97.46</td>
<td>31.64</td>
<td>99.05</td>
<td>41.63</td>
</tr>
<tr>
<td>GSRB</td>
<td>59.97</td>
<td>29.69</td>
<td>30.58</td>
<td>14.69</td>
<td>15.29</td>
<td>7.22</td>
<td>7.12</td>
</tr>
<tr>
<td>CFInterp</td>
<td>45.4</td>
<td>24.28</td>
<td>35.02</td>
<td>16.03</td>
<td>29.61</td>
<td>10.10</td>
<td>30.61</td>
</tr>
<tr>
<td>Initialization</td>
<td>41.68</td>
<td>31.60</td>
<td>53.82</td>
<td>27.04</td>
<td>67.61</td>
<td>29.91</td>
<td>92.17</td>
</tr>
</tbody>
</table>
Static vs. Instance Accesses

• Static variable accesses are slow in Titanium.
• This slowdown is much greater with Pthreads (~7 times slower than with processes).
• Indirect accesses using wrappers can eliminate much of the cost.

Time in seconds (processes/Pthreads per process)

<table>
<thead>
<tr>
<th></th>
<th>10M accesses</th>
<th>100M accesses</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8/1</td>
<td>1/8</td>
</tr>
<tr>
<td>Instance</td>
<td>0.0342</td>
<td>0.0344</td>
</tr>
<tr>
<td>Static</td>
<td>1.68</td>
<td>11.6</td>
</tr>
<tr>
<td>Indirect</td>
<td>0.241</td>
<td>0.242</td>
</tr>
</tbody>
</table>

• This only accounts for a small fraction of the slowdown in AMR.
Cache Misses

- Since Pthreads share the heap segment of memory, false sharing can occur, resulting in many more cache misses.

- Slowest two lines of code in initialization show many more data cache misses with Pthreads.

Time and cache misses (processes/Pthreads per process)

<table>
<thead>
<tr>
<th></th>
<th>Time (seconds)</th>
<th>Cache misses (millions)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8/1</td>
<td>1/8</td>
</tr>
<tr>
<td>Line 1</td>
<td>5.67</td>
<td>16.72</td>
</tr>
<tr>
<td>Line 2</td>
<td>6.49</td>
<td>21.54</td>
</tr>
</tbody>
</table>
Load Balancing

• Conflicts with optimizing to reduce communication
  - Communication between coarse grids and fine grids for interpolation. Favors allocation of overlapping grids to a single processor so interpolation requires no communication.

• Rough estimate: Time spent waiting at barriers
  Different data sets for each configuration

![Bar Chart](chart.png)
Optimizing with Communication Primitives

- Titanium *exchange* over *broadcast* during initialization

**Broadcast**
Broadcast each data block (D) to everyone else. Broadcasts execute serially.

**Exchange**
Exchanges can happen in parallel.

* Results Pending
Exchange

• The exchange operation copies the elements in the overlap sections between boxes.
• The operation accounts for 30% of total running time for the sequential backend.
• The original code has two implementations of the exchange operations.
  – Array copy
  – Foreach loop over the intersection domain
Exchange Optimizations

• Domain intersection calculation accounts for 46% of the running time of exchange.
• Amortize the intersection cost by caching the calculation
• About 20% of the intersections only has a single element in it.
  – Most resulted from the intersections of corners of two boxes, which are not needed in the calculation.
  – 44% speedup in exchange by checking for this case
Box Overlap Representation

- Both the boxes and box intersections are static throughout the execution of the program.
- Box overlaps are currently represented as intersections of Titanium arrays.
- Even with caching of intersection calculations, some overheads are unavoidable.
  - Loop overhead for small intersections
  - Translation of points in a domain to virtual addresses
Experimental Representation

• Represent all intersections as an array of source/dest C pointer pairs
• 40% speedup over the intersection caching version
• Improving on the initial version to reduce the size of the representation
  – Represent small intersections pointer pairs
  – Represent large intersections as a base pointer and strides
Conclusion

• Faster than C++/MPI implementation of Chombo

• More speedups pending due to work in progress