Domain Decomposition of a Combinatorial Geometry Monte Carlo Transport Code

Matthew O’Brien, Greg Greenman and Spike Procassini
NECDC
October 20, 2008
LLNL-PRES-407916

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.
Outline

• Overview of the *Mercury* Monte Carlo Transport Code.
• Domain Decomposition of Combinatorial Geometry.
  — What is distributed across processors?
• Example Problem: LIFE Engine with 5.6 Million Cells.
• **Mercury** is LLNL’s modern Monte Carlo Transport Code.
• Neutron, gamma and light charged particle transport.
• Solves various criticality, criticality probability and source transport problems.
• 1D, 2D, 3D structured and unstructured meshes, 3D Combinatorial Geometry.
• Massively parallel, scalable, supports domain decomposition and domain replication, with dynamic load balancing.
• Written in C, being converted to C++.
• Define a list of *Quadric Surfaces* (at most 2\textsuperscript{nd} order surfaces), such as planes, spheres, ellipsoids, cylinders, cones, etc.

\[ \sum_{0 \leq i + j + k \leq 2} a_{ijk} x^i y^j z^k = 0 \]

• Combine the surfaces using logical operations to form cells.
• Example: cell1 = insideOf(sphere1) AND outsideOf(sphere2)
The Need For Domain Decomposition of CG

- Traditionally, all of the geometry information in a CG Monte Carlo code has been redundantly stored on all of the processors.
- The particle workload is then divided among all of the processors, for an embarrassingly parallel solution. (i.e. no coupling between processors as the calculation is running, only a final reduction to get the total answer).
- As the geometric description of the problem gets larger and more complex, this model breaks down. With millions or billions of CG cells, there is not enough memory on one processor to store all of the geometry information. The geometry must be decomposed across processors.
- This creates coupling between processors as particles stream from one processor to another.
## Domain Decomposition: Mesh vs. CG

<table>
<thead>
<tr>
<th></th>
<th>Mesh</th>
<th>CG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cell Boundary Crossing</td>
<td>Adjacent cells know.</td>
<td>Must check adjacent candidate cells, don’t explicitly know adjacency.</td>
</tr>
<tr>
<td>Domain Boundary Crossing</td>
<td>Adjacent domains known.</td>
<td>Adjacent domains known. (new)</td>
</tr>
<tr>
<td>Input</td>
<td>Input description is already domain decomposed.</td>
<td>Must decide if each surface/cell should be assigned to each processor. (Need to domain decompose user input.) (new)</td>
</tr>
<tr>
<td>Output (graphics)</td>
<td>Each processor writes its domains. A master file describes how to assemble the pieces.</td>
<td>Each processor writes the portion of space it owns, explicitly introducing domain boundary surfaces for cells on domain boundaries. A master file describes how to assemble the pieces. (new)</td>
</tr>
</tbody>
</table>
User defines global CG problem.

User defines Cartesian Domain Decomposition.

Domain 0

Domain 1
CG Can Be Used To Create Complex Geometries

NIF Target Chamber and Support Structure.
What Data is Distributed?

• The following lists of data can grow arbitrarily long, so we need a way to distribute the data across processors.
  — List of surfaces.
  — List of surfaces that define a cell.
  — List of cells.
  — List of *templated* (cloned) surfaces and cells.
User defines global CG problem.

Domain 0

Domain 1

If a cell’s bounding box intersects a domain, the cell is assigned to the domain. Some cells are assigned to multiple domains.
Simple Examples

- 3 by 3 by 3 lattice of small spheres surrounded by a big sphere.
- Divided into 2 domains.
- Decomposition surfaces are written out as part of the cell definition for visualization.
- Visualized using VisIt’s CG Adaptive visualizer.

16 Domains, colored by domain.
Scalability Issues

- The entire CG input deck must be read into memory at once.
- The entire list of surfaces/cells is read in, then a surface/cell is kept on a domain only if the surface’s/cell’s bounding box intersects the domain’s bounding box.
- The entire list of surfaces that define a cell are read in, then only the surfaces that intersect the domain that the cell is on our kept.
Scalability Solutions

• After initialization, each domain only stores *local* information; hence the algorithm is scalable.
• We could treat CG input similar to how mesh geometry is treated: the geometry is decomposed into separate files and each processor only deals with the domains that are assigned to it. (parse time solution).
• If the large cell count arises due to repeated hierarchical structures, we achieve scalability through the input "template" mechanism.
Cell Parsing

```c
foreach(input deck cell) {
    foreach (domain on this processor) {
        temp_cell = MCIO_Input_Cell_3D_H(input deck cell)
        Compute_Bounding_Box( &temp_cell );
        int on_domain = IsCellOnDomain(temp_cell, domain_index )
        if ( on_domain ) {
            MCIO_Input_Cells_3D_H_Insert_Cell(domain, &temp_cell);
        }
    }
}
```
During the *Is-Point-In-Cell* routine, if there is more than 1 domain, then the algorithm ensures that the input particle is inside of the input domain. 

If that test passes, continue as before.

Otherwise the particle is definitely not on the input domain.
• At the end of the **Find-Nearest-Facet** tracking routine, the algorithm calculates the distance to the next implicit domain boundary crossing.
• If the domain boundary is closer than the nearest facet, then:
  We have a **Domain Boundary Crossing** Event.
Otherwise:
  We have a **Standard Facet Crossing** Event.
• The domain boundary crossing is handled as a special case of a facet crossing.

\[ d_1 = \text{distance to domain boundary} \]
\[ d_2 = \text{distance to nearest facet.} \]
\[ d_1 < d_2 \text{ so we have a Domain Boundary Crossing Event.} \]

1° x 1° Wedge of Pebbles. \( r_1 = 423 \text{cm}, \ r_2 = 504 \text{cm} \)

569 Pebbles. 1 Pebble \(( r = 1 \text{cm})\) contains 2445 Triso pellets.

569 pebbles \times 2445 \text{trisos} \times 4 \text{layers} = 5.6 \text{Million CG cells.}

- Flibe Coolant: Li, Be, F
- Pebble Filler: C
- Triso Layer 1: \( \text{U}^{238}, \text{O}, \text{C} \)
- Triso Layer 2, 3: C
- Triso Layer 4: C, Si

Each Triso pellet has 4 Layers.

\( r = 497 \mu \text{m} \)
Preliminary Results

Pebble with 2447 CG cells, homogenized trisos.

Seconds spent doing particle transport:

<table>
<thead>
<tr>
<th></th>
<th>1proc</th>
<th>2procs</th>
<th>4procs</th>
<th>8procs</th>
<th>16procs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1domain</td>
<td>848</td>
<td>427</td>
<td>226</td>
<td>131</td>
<td>74</td>
</tr>
<tr>
<td>2domains</td>
<td>736</td>
<td>235</td>
<td>148</td>
<td>82</td>
<td>52</td>
</tr>
<tr>
<td>4domains</td>
<td>668</td>
<td>190</td>
<td>65</td>
<td>34</td>
<td>20</td>
</tr>
<tr>
<td>8domains</td>
<td>659</td>
<td>162</td>
<td>57</td>
<td>20</td>
<td>12</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
<td>---------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16domains</td>
<td>686</td>
<td>214</td>
<td>113</td>
<td>32</td>
<td>12</td>
</tr>
<tr>
<td>64domains</td>
<td>732</td>
<td>207</td>
<td>116</td>
<td>37</td>
<td>18</td>
</tr>
</tbody>
</table>

---------------more domains hurts-------------
Without domain decomposition, a particle in the filler must calculate the distance to 2445 surfaces, which is very expensive.

With domain decomposition, a particle in the filler must calculate the distance to only local surfaces on this domain, which is significantly faster.
Existing Dynamic Load Balancing Works with CG Domain Decomposition.

- 64 processors, 16 domains.
- The number of processors assigned to each domain is proportional to the domain’s workload.
- Pseudocolor plot of the number of processors working on each domain.
- Red = 17 processors, Blue = 1 processor.

**Cycle 0:**
uniform assignment of 4 processors to each domain.
Changing Material Colors Allows for BabyRuth Modeling
Memory Requirements of the LIFE Problem

- Memory requirements for LIFE geometry, with 175 energy groups per cell (for scalar flux) is about 36GB + particle memory.
- This is more memory than any one processor has, so it must be domain decomposed.
- Using input templates, we are able to run this problem.
Python Input Deck Scripting

<table>
<thead>
<tr>
<th>Input Deck/Python Script</th>
<th>Evaluated Input Deck</th>
</tr>
</thead>
<tbody>
<tr>
<td>python pyDeltaT = 1e-8 end_python</td>
<td>time time_stop 1.0e-7 delta_t {pyDeltaT} end_time</td>
</tr>
<tr>
<td>def definePebble(trisoFile):</td>
<td>cell ... end_cell</td>
</tr>
<tr>
<td>file = open(trisoFile, 'r') ...</td>
<td>... source category Point center_coords 0 0 0 end_source</td>
</tr>
</tbody>
</table>
|   output = "cell ... end_cell" return output end_python | python ... end_python defines your python variables and functions. Replaces \{curly braces\} with the python value of the expression. This is very useful for generating complex input such as the LIFE engine.
Conclusions

• CG Domain Decomposition is necessary to run Monte Carlo transport problems with extremely large cell counts.
• Domain Decomposition can actually speed up a calculation, since only local geometry information is stored, automatically ruling out non-local geometry.
• Future Work
  — CG Domain Decomposition gives you the correct answer for integrated tallies. But for tallies of an individual cell, we need to add code to do a reduction for cells that are split across multiple domains to get the total answer.