Parallel SAH k-D Tree Construction

Byn Choi, Rakesh Komuravelli, Victor Lu, Hyojin Sung, Robert L. Bocchino, Sarita V. Adve, John C. Hart
Motivation

• Real-time Dynamic Ray Tracing
  – Efficient rendering with the right spatial data structure
• SAH-based k-D tree proven very effective
  – Dynamic content requires rebuilding tree every frame
    ⇒ Tree build becomes bottleneck in rendering pipeline
• Prior parallelization efforts abandon SAH
  – Sacrifice tree quality, increase rendering time
Contributions

• First to parallelize k-D tree construction with precise SAH
  – High quality AND high performance
• Two parallel algorithms: **Nested** and **In-place**
  – Different performance/scalability characteristics
• Up to 8x speedup on 32 cores
Straightforward || Construction

• Top-down – Recursive subdivision
• Divide-n-Conquer style
  – Recursive parallelism
  – Each node == a task
• Problem
  – Not enough parallelism at top
  – More work per node at top
  – Serial for top nodes:
    • Benthin PhD, 2006
    • Popov et al. IRT 2006
    • Hunt et al. IRT 2006
k-D Tree Parallel Pattern

**Geometry Parallel**
Process triangles in parallel

**Challenge:** Compute precise SAH here

**Moore’s Law**
Every 18 months the transition line descends a level

**Node Parallel**
Subtrees built in parallel
Previous Approaches

Shevtsov et al. CGF 07
- 4-core CPU, LRB
- Δ count median for upper-tree nodes

Zhou et al. SA 08
- Streaming GPU
- Spatial median for upper-tree nodes

As core counts increase, median (non-SAH) constructions degrade rendering performance

Frame Rate (frames per second)

Median→SAH transition depth (# processors)

Always SAH

Existing parallel k-D tree construction

all SAH

all median
Calculating SAH

• Prob. of hitting triangles $\propto$ surface area of bounding box
• Largest number of triangles in least surface area
• Need to find out…
  – $\text{Area}_L, \text{Area}_R$ – Surface area
  – $\#_L, \#_R$ – # of triangles

$\text{SAH} \propto \#_L \times \text{Area}_L + \#_R \times \text{Area}_R$

• Linear dependency between events
• Events need to be sorted!
Seq. k-D Tree Construction & Nested

(Wald and Havran, 2006)

- Recursive tree-building algorithm $O(n \log n)$
- Sorted list of events as input
- 3 Major phases within a node
  - FindBestPlane (41%)
  - ClassifyTriangles (4%)
  - FilterGeom (55%)

- Parallelization
  - FindBestPlane
    Linear dependence $\rightarrow$ Parallel Prefix
  - FilterGeom
    Sorted output $\rightarrow$ Parallel Prefix
Issues

• Two extra full-scans introduced (Parallel Prefix)
  – FindBestPlane
  – FilterGeom

• Data movement
  – Events moved from one container to another

• ClassifyTriangles hard to parallelize
  – Arbitrary bit writes by multiple threads into a shared bit vector
    • Synchronization overhead
    • False-sharing
  – 4% execution time == 25x maximum theoretical speedup
In-place Algorithm

• Events are kept “in-place” – no need to preserve ordering
  – Eliminates FilterGeom phase
  – Does less work
• Events responsible for tracking own membership(s)

<table>
<thead>
<tr>
<th>Node</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
</tr>
<tr>
<td></td>
<td>A</td>
</tr>
<tr>
<td></td>
<td>B</td>
</tr>
</tbody>
</table>

• Change of membership is an update, not a move/copy
In-place Algorithm

- Iterative tree-building algorithm
- 3 Major phases within an iteration
  - FindBestPlane (85%)
  - Newgen (0.04%)
  - ClassifyTriangle (14%)
- Fill phase (0.52%)
- Parallelization
  - FindBestPlane → Parallel Prefix
  - ClassifyTriangles → Fully Parallel
Methodology

- Both algorithms implemented using Intel TBB
- Five 3D models from
  - Stanford 3D Scanning Repository
  - Georgia Tech’s Large Geometric Models Archive
  - The Utah 3D Animation Repository
- Machine configurations

<table>
<thead>
<tr>
<th>Processor</th>
<th>Xeon E7450 (“Dunnington”)</th>
<th>Xeon X7550 (“Beckton”)</th>
</tr>
</thead>
<tbody>
<tr>
<td>μarch</td>
<td>Core</td>
<td>Nehalem</td>
</tr>
<tr>
<td>Core Count</td>
<td>24</td>
<td>32</td>
</tr>
<tr>
<td>Cache</td>
<td>12 MB (L2)</td>
<td>18 MB (L3)</td>
</tr>
<tr>
<td>Memory b/w</td>
<td>1x</td>
<td>9x</td>
</tr>
<tr>
<td>Memory</td>
<td>48 GB</td>
<td>64 GB</td>
</tr>
</tbody>
</table>
The main bottleneck to scalability for nested is its sequential traversal of the scene, which can lead to a decrease in performance as the number of cores increases. Nested saturates sooner than in-place and in-place performs better than nested on larger, uniform-sized triangles. However, nested outperforms in-place for three scanned inputs, indicating that in-place is more scalable.

Thus, although nested outperforms in-place for three inputs, its performance advantage reduces on the new machine. Conversely, for the cases where nested starts outperforming it for all cases, its performance advantage increases further on the new machine. The figure clearly shows that for the two inputs where in-place starts out better on the older machine, it remains so on both machines.

Absolute speedup is shown on the y-axis, with the number of threads ranging from 8 to 32. The graph illustrates the performance of both algorithms in exploiting additional resources of the newer machine, with in-place better able to exploit the resources of the newer machine. Although both algorithms perform better effective at exploiting parallelism than nested for all inputs, nested is potentially outperforming it for all cases.

The figure immediately shows that in-place is more efficient than nested in newer machines, to show higher scalability than nested in newer machines. This is evident in the performance advantage it provides, which increases on the new machine. For the cases where nested starts outperforming it for all cases, its performance advantage increases further on the new machine.

The absolute speedup plot shows that for smaller Bunny inputs, nested outperforms in-place for all cases, indicating that in-place starts out better on the older machine. However, its performance advantage reduces on the new machine. Conversely, for the cases where nested starts outperforming it for all cases, its performance advantage increases further on the new machine.

The graph also clarifies the difference between best-serial and parallel algorithm performance and one-core parallel algorithm performance, which is useful for understanding the potential for future larger machines. The figure shows the ratio of the best execution time of the parallelized phases, to show higher scalability than nested in newer machines.

The figure further quantifies this with the speedup on state-of-the-art machine, which also clarifies the difference between best-serial and parallel algorithm performance and one-core parallel algorithm performance, which is useful for understanding the potential for future larger machines.

The diagram includes a table listing running times, in seconds, on Beckton for the best-serial running times, including the unnecessary runs of the parallel nested and in-place implementations as a reference. The table also includes memory size, memory bandwidth, frequency, last-level shared cache size, socket count, and core count for different models.
Scalability Analysis

- Future hardware
  - More cores
  - Larger caches
  - More memory b/w

- Nested has sequential portion => Amdahl’s Law

![Graph: Performance of in-place/nested]

<table>
<thead>
<tr>
<th>Input</th>
<th>32 threads</th>
<th>∞ threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>bunny</td>
<td>12.9</td>
<td>21.0</td>
</tr>
<tr>
<td>fairy</td>
<td>13.1</td>
<td>21.6</td>
</tr>
<tr>
<td>angel</td>
<td>11.2</td>
<td>16.7</td>
</tr>
<tr>
<td>dragon</td>
<td>10.3</td>
<td>14.7</td>
</tr>
<tr>
<td>happy</td>
<td>10.3</td>
<td>14.8</td>
</tr>
</tbody>
</table>
Conclusion / Future Work

• Parallel build of high-quality k-D tree critical for ray tracing
  – Prior work trades quality for performance
• We show parallel build with high quality AND performance
  – Two algorithms with up to 8x speedup
  – Different performance/scalability characteristics
• Future work
  – GPU implementation of in-place
    Streaming nature more amenable to SIMD-fication
Thank You!

• Questions?