Using Transactions in Delaunay Mesh Generation

Milind Kulkarni, L. Paul Chew and Keshav Pingali
Cornell University

Workshop on Transactional Memory Workloads, 2006
Introduction

- New benchmark: Delaunay mesh generation
  - Algorithm well suited for optimistic parallelization/transactions
  - Useful for evaluating transactional memory
- Problems with naïve TM implementations
  - How to mitigate while using standard TM
Mesh Generation

- Mesh generation useful for
  - Finite element method for solving PDEs
  - Graphics
    - Tessellation of surface into polygons for rendering
- Delaunay mesh generation
  - Mesh generation algorithm which provides guaranteed quality meshes
Delaunay Mesh Generation

- What is a Delaunay mesh?
  - Tessellation of a surface given vertices
Delaunay Mesh Generation

- What is a Delaunay mesh?
  - Tessellation of a surface given vertices
What is a Delaunay mesh?

- Tessellation of a surface given vertices
- Satisfies the Delaunay property
  - Circumcircle of any triangle does not contain another point in the mesh
Delaunay Mesh Generation

- Want all triangles in mesh to meet quality constraints
  - No angle < 30°
- Fix bad triangles through iterative refinement
  - Add new vertices to mesh and retriangulate
Mesh Refinement

- Choose “bad” triangle
Mesh Refinement

- Choose “bad” triangle
- Add new vertex at center of circumcircle
Mesh Refinement

- Choose “bad” triangle
- Add new vertex at center of circumcircle
- Gather all triangles that no longer satisfy Delaunay property into cavity
Mesh Refinement

- Choose “bad” triangle
- Add new vertex at center of circumcircle
- Gather all triangles that no longer satisfy Delaunay property into cavity
- **Re-triangulate** affected region, including *new point*
Mesh Refinement

- Choose “bad” triangle
- Add new vertex at center of circumcircle
- Gather all triangles that no longer satisfy Delaunay property into cavity
- Re-triangulate affected region, including new point
- Continue until all bad triangles processed
Refinement Example

Original Mesh

Refined Mesh
Mesh m = /* read in mesh */
WorkList wl;
wl.insert(mesh.badTriangles());

while (!wl.empty()) {
    Element e = wl.removeAny(); //non-deterministic choice
    if (e no longer in mesh) continue;

    Cavity c = new Cavity(e); //determine new vertex
    c.expand(); //determine affected triangles
    c.retriangulate(); //re-triangulate region

    m.update(c); //update mesh

    wl.insert(c.badTriangles()); //add new bad triangles to queue
}
Parallelization Opportunities

Before

After
Parallelization Strategy

- Process bad triangles in parallel
  - One bad triangle per iteration
  - Only safe if iterations are independent
    - Cavities must not overlap

- “Pessimistic” solutions infeasible
  - Static Analysis
    - Dependences are highly data dependent
  - Inspector-Executor
    - Inspector stage must perform entire computation
Optimistic Parallelism

- Only feasible solution
  - Process bad triangles in parallel regardless of dependences
  - Detect cavity overlap at run time
  - If interference detected, roll back execution

- How do we ensure no conflicts?

```c
void main() {
    /* initialization */
    for (/*number of threads*/)
        spawn_thread(process)
}

void process() {
    while (!wl.empty()) {
        Element e = wl.removeAny();
        if (e no longer in mesh) continue;

        Cavity c = new Cavity(e);
        c.expand();
        c.retriangulate();

        m.update(c);
        wl.insert(c.badTriangles());
    }
}
```
Lock-based Implementation

- Uses two-phase locking protocol (with rollback) to provide optimistic parallelism
- 40K triangles in mesh
  - 19K initial bad triangles
  - 58K triangles processed
  - 32% of iterations aborted
- Implementation is very complex

![Graph showing efficiency vs. number of processors]

System details:
- 4 processors
- Itanium 2 @ 1.6GHz
- pthreads
Transactional Approach

- Processing each bad triangle is a transaction
- Transactional memory detects conflicts between transactions
  - Safety: conflicts must occur if cavities overlap
  - Optimality: conflicts if and only if cavities overlap
- Transactional memory automatically performs rollbacks when conflicts detected
Naïve use of transactional memory

Satisfies safety conditions, but does not exhibit parallelism

```java
startTransaction();

Element e = wl.removeAny();
if (e no longer in mesh)
    endTransaction();
    continue;

Cavity c = new Cavity(e);
c.expand();
c.retriangulate();
m.update(c);
wl.insert(c.badTriangles());
endTransaction();
```
TM Limitations

- **Problem:** Worklist manipulations must be visible to other transactions before commit.

- **Solutions**
  - Move worklist operations outside transaction
  - **Use open nesting**
    - Must specify undo action for nested operations

```java
startTransaction();
Element e = wl.removeAny();
if (e no longer in mesh)
  endTransaction();
  continue;

Cavity c = new Cavity(e);
c.expand();
c.retriangulate();
m.update(c);
wl.insert(c.badTriangles());
endTransaction();
```
TM Limitations

- Problem: Worklist manipulations must be visible to other transactions before commit

- Solutions
  - Move worklist operations outside transaction
  - Use open nesting
    - Must specify undo action for nested operations
TM Limitations

- Problem: **Worklist manipulations** must be visible to other transactions before commit

- Solutions
  - Move worklist operations outside transaction
  - **Use open nesting**
    - Must specify undo action for nested operations

```java
startTransaction();
beginOpen();
Element e = wl.removeAny();
endOpen();
if (e no longer in mesh)
    endTransaction();
    continue;
Cavity c = new Cavity(e);
c.expand();
c.retriangulate();
m.update(c);
beginOpen();
w1.insert(c.badTriangles());
endOpen();
endTransaction();
```
Mesh data structure

- Basic implementation: adjacency list using STL map
- Rebalancing red-black trees causes conflicts

Solution

- Carefully choose data structures to avoid unwanted conflicts
  - e.g. Use hash tables

```java
startTransaction();
beginOpen();
Element e = wl.removeAny();
endOpen();

if (e no longer in mesh)
    endTransaction();
    continue;

Cavity c = new Cavity(e);
c.expand();
c.retriangulate();
m.update(c);
beginOpen();
wI.insert(c.badTriangles());
endOpen();
endTransaction();
```
Scheduling issues

- Transactions are long running, so rollbacks expensive
- Empirical results: 30% of transactions abort with random scheduling
  - Can partition mesh and worklist to eliminate virtually all rollbacks
- Transaction scheduling important

![Bar chart showing efficiency comparison between random and partitioned scheduling across different numbers of processors.](chart.png)
Related work

- Original Delaunay refinement algorithm by Chew and Ruppert
  - L. Paul Chew, *Guaranteed Quality Mesh Generation for Curved Surfaces*
  - Jim Ruppert, *A New and Simple Algorithm for Quality 2-dimensional Mesh Generation*

- Previous implementations
  - Sequential implementation
    - J. Shewchuck, *Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator*
  - Parallel, distributed memory implementation
    - Antonopoulos et al, *Multigrain Parallel Delaunay Mesh Generation: Challenges and Opportunities for Multithreaded Architectures*

- Scale to real world problems
Conclusions

‣ Delaunay mesh generation is ideal transactional memory benchmark
  ▪ Real-world application
  ▪ Optimistic parallelization only approach
    ▪ Well suited for TM
‣ Platform to explore several TM issues
  ▪ Careful demarcation of transactions and use of open nesting
  ▪ Choosing data structures with understanding of TM limitations
  ▪ Managing scheduling policy to reduce rollbacks
‣ New style of TM to encompass all these issues?
Questions/Comments?