Delaunay Mesh Generation and Parallelization

January 31st, 2006
Context

- Mesh generation useful for
  - Finite element method
    - Determines basis functions
  - Graphics
    - Tessellation of surface into polygons for rendering
Mesh Quality

- Why is mesh quality important?
  - Can impact solution of linear systems
    - Angles too large can lead to errors
    - Angles too small lead to ill-conditioned systems
  - Tradeoffs regarding number of elements
    - More elements for better accuracy
    - Fewer elements for better speed
Structured vs. Unstructured

Structured Mesh

Unstructured Mesh

Some diagrams taken from Jonathan Shewchuk’s lecture notes: http://www.cs.berkeley.edu/~jrs/mesh/
Different Types of Meshing

- **Advancing Front**
  - Starts at boundary of mesh
  - Uses heuristics to “grow” mesh inwards

From JS lecture notes
Different Types of Meshing

- **Quadtree based meshes**
  - Subdivide plane using quadtree structure
  - Warp quadtree to match vertices
  - Triangulate resulting grid

From JS lecture notes
Delaunay 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
Why Delaunay Meshes?

- Provides specific guarantees
  - “The Delaunay triangulation of a point set minimizes the maximum angle over all possible triangulations”
  - Fewer skinny triangles

- Additional points can be inserted to meet certain quality constraints
  - Angle constraints
  - Triangle size constraints
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
Mesh m = /* read in mesh */
WorkQueue wq;
wq.enqueue(mesh.badTriangles());

while (!wq.empty()) {
    Triangle t = wq.dequeue();  // choose bad triangle
    Cavity c = new Cavity(t);    // determine new vertex
    c.expand();                 // determine affected triangles
    c.retriangulate();         // re-triangulate region
    m.update(c);               // update mesh
    wq.enqueue(c.badTriangles());  // add new bad triangles to queue
}
Refinement Example

Original Mesh

Refined Mesh
Parallelization Opportunities

- Multiple bad triangles to be processed
- Algorithm inherently non-deterministic
  - Order of processing irrelevant
- Effects of re-triangulation localized
  - Update operations mostly independent
- Can process multiple triangles in parallel
  - Triangles must be sufficiently far apart
Parallelization Opportunities

- Estimated available parallelism for mesh of 1M triangles
- Actual ability to exploit parallelism dependent on scheduling of processing
  - C. Antonopolous, X. Ding, A. Chernikov, F. Blagojevic, D. Nikolopolous and N. Chrisochoides *Multigrain parallel Delaunay Mesh generation*, ICS05
Mesh m = /* read in mesh */
WorkQueue wq;
wq.enqueue(mesh.badTriangles());

while (!wq.empty()) {
    Triangle t = wq.dequeue();  // choose bad triangle
    Cavity c = new Cavity(t);  // determine new vertex
    c.expand();  // determine affected triangles
    c.retriangulate();  // re-triangulate region

    m.update(c);  // update mesh

    wq.enqueue(c.badTriangles());  // add new bad triangles to queue
}
Abstractions

- **WorkSet abstraction**
  - Replaces queue of bad triangles
  - Queue introduces loop carried dependence

- **getAny operation**
  - Does not make ordering guarantees
    - Removes dependence between iterations
    - In the absence of interference, iterations can execute in parallel
Abstractions

- **Graph abstraction**
  - Mesh can be viewed as an undirected graph
    - Nodes in Graph represent triangles in mesh
    - Edges in Graph capture triangle adjacency

- **Subgraph abstraction**
  - Collection of affected triangles is a Subgraph of overall Graph

- **replaceSubgraph operation**
  - Updating mesh after re-triangulation is replacing one Subgraph with another
Graph g = /* read in mesh */
WorkSet ws;
ws.add(g.badNodes());

while (!ws.empty()) {
    Node n = ws.getAny(); // choose bad node
    Subgraph s1 = expandCavity(n); // determine affected nodes
    Subgraph s2 = reTriangulate(s1); // re-triangulate region
    g.replaceSubgraph(s1, s2); // update graph
    ws.add(s2.badNodes()); // add new bad nodes to set
}
Automatic Parallelization

- Can now exploit “getAny” to parallelize loop
  - Can try to expand cavities in parallel
  - Expansions can still conflict

- How do we deal with this?
  - Ideally, automatically
Most automatic parallelization focuses on “regular” data structures

- arrays, matrices

Parallelization for irregular data structures much trickier

- Why?
Automatic Parallelization

- Must ensure no dependences between parallel code
  - Static detection of possible dependences
  - If dependence exists, parallel execution not possible
Automatic Parallelization

- Must ensure no dependences between parallel code
  - Static detection of possible dependences
  - If dependence exists, parallel execution not possible

- Analysis harder for irregular data structures
- Static determination of dependences intractable
In practice, most cavities can be expanded in parallel safely
  - No way to know this \textit{a priori}
  - Only guaranteed safe approach is serialization

What if we perform parallelization without prior guarantee of safety?
Optimistic Parallelization

- Expand cavities in parallel
- Perform run time checks to ensure that expansions do not conflict
  - If check fails, roll back expansion process, try again
Parallelization Issues

- Must ensure that run-time checks are efficient
- How to perform roll backs
- Would like to minimize conflicts
  - **Scheduling** becomes important
    - Number of available cavities for expansion exceeds computational resources
    - Choose cavities to expand to minimize conflicts
      - Empirical testing: ~30% of cavity expansions conflict
Distributed Memory

- Elements of mesh now distributed among processors
  - Accessing elements on different processors high latency
    - Expanding cavity whose elements are on multiple processors is slow
    - Want to hide latency
  - Perform multiple expansions *per processor*.
Mesh generation is an important algorithm
  - Delaunay mesh generation is a particularly useful variant of this algorithm

Delaunay mesh generation intuitively parallelizable
  - In practice (especially automatically) not so easy
    - Must determine that loop can be parallelized
    - Must ensure that cavities do not conflict

Several challenges to effective parallelization
  - Dynamic dependence checking
  - Scheduling
  - Distributed memory model