Data Structures for Parallel Spatial Algorithms on Large Datasets

W. Randolph Franklin *(RPI, USA)*, Salles V. G. de Magalhães & Marcus V. A. Andrade *(UF Viçosa, Brasil)*

BIGSPATIAL, 2018-11-06
Abstract

- Efficient parallel data structures are different:
  - Suboptimal: trees, recursion, pointers, sweep lines, global topologies.
- 2D GIS and 3D CAD share a lot—learn from each other.
  - In additive manufacturing (3D printing), easier to build than to analyze.
- New way of looking at geometry is useful.
- This talk:
  - Local geometric data structures for map–reduce.
  - Local parallel computing.
- Big example: overlay two triangulated polyhedra, total 5.7M triangles in 5.5 real seconds on 16 core Xeon workstation.
  - That used rational numbers to prevent roundoff, and Simulation of Simplicity to handle geometric degeneracies, or it would have been even faster.
Our prior parallel geometry implementations...

on multicore Intel Xeon, with OpenMP

- Volume of union of 100M identical cubes (2003)
- 2D planar graph overlay (BIGSPATIAL 2015)
- 3D point location (Berlin Geometry Summit 2016)
- Triangulated polyhedra overlay (IMR 2018)

on Nvidia GPUs, with Thrust

- Find all pairs of 3D points closer than given $\delta$ (BIGSPATIAL 2017)
- Preprocess points in 2D to 6D for nearest point query (CCCG 2016)
Background

- Philosophically a Computer Scientist.
- PhD officially in Applied Math.
- Working in Electrical, Computer, and Systems Engineering Dept.
- Students are from Computer Science.
- Teaching Engineering Parallel Computing.
- Collaborating with Geographers for a long time.
- Enjoy applying computer science and engineering to GIS.
Historical analogy

- Roebling, builder of Brooklyn Bridge, graduated from RPI.
- 15 year project.
- after spending money for 2 years, there was no visible progress.
- Roebling was building the foundations.
- None of his bridges ever fell down.
- In contrast: In last few decades, three interstate highway bridges have collapsed from design errors compounding maintenance lack.

*Spend some time on the foundations.*
Massive shared memory

- An underappreciated resource.
- External memory often not needed.
- Paging virtual memory is obsolete.
- Inexpensive servers have 1TB of memory.
- Even for Nvidia GPUs:
  - up to 48GB,
  - several can be ganged together with hi-speed bus.
- Many problems don’t require the overhead of—
  - MPI,
  - supercomputers,
  - distributed cloud computing.
Parallel computing

- Multicore Intel Xeon underappreciated.
  - *Dual 20 core: 80 hyperthreads.*
- One Xeon core is $20 \times$ more powerful than one CUDA core.
- Nvidia GPUs: up to 5000 cores, 48GB memory.
- Lower clock speed 750MHz vs 3.4GHz
- Hierarchy of memory: small/fast $\leftrightarrow$ big/slow
- Communication cost $\gg$ computation cost
- Preferred: blocks of threads execute SIMT.
- Top 500 OS: never $\text{Windows}$ always some variant of
Why parallel HW?

- More processing $\rightarrow$ faster clock speed $\rightarrow$ more electrical power. *Each bit flip (dis)charges a capacitor through a resistance.*
- Faster $\rightarrow$ requires smaller features on chip
- Smaller $\rightarrow$ *greater* electrical resistance!
- Serial processors have hit a wall.
Some parallel programming tools

- **OpenMP—**
  - Shared memory, multiple CPU core model.
  - Good for moderate parallelism.
  - Easy to get started.
  - Options for protecting parallel writes:
    - Sum reduction: no overhead.
    - Atomic add and capture: small overhead.
    - Critical block: perhaps 100K instruction overhead.
  - Valid cost metric: real time used.
  - 2-thread programs perhaps slower than 1-thread.

- **CUDA/Thrust—**
  - Nvidia C++ template library for CUDA based on STL.
  - Functional paradigm: easier algorithm expression.
  - Hides many CUDA details: good and bad.
  - Powerful operators all parallelize: scatter/gather, reduction by key, permutation, sort, prefix sum.
  - Surprisingly efficient algorithms like bucket sort.
  - Possible back ends: CUDA, OpenMP, sequential on host.
Geometric Databases

- Hundreds of millions of primitive components.
- Some foundational operations—
  - nearest point
  - boolean intersection and union
  - planar graph overlay
  - mass property computation of the results of some boolean operation
- Higher applications—
  - Volume and moments of an object defined as the union of many overlapping primitives.
  - Two object interfere iff volume of intersection $> 0$.
  - Interpolate population from census tracts to flood zones.
  - Interpolate properties between two triangulations of same polyhedron.
  - ⋅⋅⋅ and many higher-level problems.
How few types of info does a polyhedron rep need?

A design is not complete until everything possible has been removed.

- Why?
  - fewer special cases ⇒ less code ⇒ less debugging
  - less space ⇒ faster

- Operations:
  - point location
  - area, center of gravity, high-order moments

- Ambiguous rep: set of vertices.
- Sufficient rep: set of faces.
- Above operations are now map-reductions.
Point Location on a Set of Faces

- "Jordan curve” method
- Extend a semi-infinite ray from query point.
- Count intersections with faces.
- Odd number \(\equiv\) inside.
- *Obvious but bad alternative:* sum subtended signed volumes. Implementing w/o arctan, and handling special cases wrapping around is tricky and reduces to Jordan curve.
Moment Computation on a Set of Faces

- Each face, with the origin, defines a tetrahedron.
- Compute its moment; sum them.
- Extends to any mass property, including (using a characteristic function) point location.
- Extends to functionally graded properties, e.g., 3D printer extruding a varying-density material.
The Advantages of Set of Faces Data Structure

- Simple enough to debug.
- *SW can be simple enough that there are obviously no errors, or complex enough that there are no obvious errors.*
- Less storage.
- Easy parallelization: reduction operations.

∃ Other reps (on the following slides).
Augmented vertices: another minimal polyhedron representation

- Augmented vertices: add a little to each vertex.
- These examples use rectilinear polygons, but all this works on general polygons and polyhedra.
- 8 types of vertices;
- Each gets a sign, $s = \pm 1$.
- Now, each vertex defined as $v_i = (x_i, y_i, s_i)$
- Area of polygon: $A = \sum s_i x_i y_i$
- Volume of polyhedron: $V = \sum s_i x_i y_i z_i$
- Moment of inertia about z-axis: $I = \sum s_i x_i^2 y_i^2$
Vertex incidences: YAMPR

Another minimal data structure, resembles half edges.

- Only data type is the incidence of an edge and a vertex, plus its neighborhood. For each such:
  - $\vec{V} = \text{coord of vertex}$
  - $\hat{T} = \text{unit tangent vector along the edge}$
  - $\hat{N} = \text{unit vector normal to } \hat{T} \text{ pointing into the polygon.}$

- Polygon (2 tuples per vertex): $\{(\vec{V}, \hat{T}, \hat{N})\}$
- Perimeter $= -\sum (\vec{V} \cdot \hat{T})$.
- Area $= \frac{1}{2} \sum (\vec{V} \cdot \hat{T})(\vec{V} \cdot \hat{N})$
- Multiple nested components ok.

- Mass properties are map-reductions.
What’s the point of this?

- Don’t we always know the edges?
- No, not easily for Boolean combinations.
- We know the input polyhedra’s faces.
- However finding the output polyhedron’s faces is much harder than merely finding the augmented vertices.
  - That requires finding more global topology.
- Three types of output vertices—
  - Some input vertices,
  - Some intersections of three input faces.
  - Some intersections of and input face with an edge.
- Filter them: an output vertex must be—
  - for intersection: *inside* all input polyhedra.
  - for union: *outside* all input polyhedra.
- Apply reduction equation to surviving vertices.
- *Next:* several examples.
Volume of Union of Many Cubes

- Illustrates power of these ideas.
- A prototype on an easy subcase (congruent axis-aligned cubes).
- Extends to general polyhedra.
- *Not* statistical sampling—this is exact output, apart from roundoff.
- *Not* subdivision-into-voxel method — the cubes’ coordinates can be any representable numbers.
Application: Cutting Tool Path

- Represent path of a tool as piecewise line.
- Each piece sweeps a polyhedron.
- Volume of material removed is (approx) volume of union of those polyhedra.
- Image is from Surfware Inc’s Surfcam website.
Traditional N-Polyhedron Union

- Construct pairwise unions of primitives.
- Iterate.

Time depends on intermediate swell, and elementary intersection time.

- Let \( P = \) size of union of an M-gon and an N-gon. Then \( P = O(MN) \).
- Time for union (using line sweep) \( T = \Theta(P \lg P) \).
- Total \( T = O(N^2 \lg N) \).

Hard to parallelize upper levels of computation tree.
Problems With Traditional Method

- $\text{lg } N$ levels in computation tree cause $\text{lg } N$ factor in execution time. Consider $N > 20$.
- *Intermediate swell*: worse as overlap is worse. Intermediate computations may be much larger than final result.
- The explicit output polyhedron has complicated topology: unknown genus, loops of edges, shells of faces, nonmanifold adjacencies.
- Tricky to get all this right.
- *However* explicit output not needed for computing mass properties.
- Set of vertices with neighborhoods suffices.
Fast parallel volume of union

- Find the intersections in one flat intersection test.
- Filter them.
- Map-reduce them.
- Processing 100M cubes with $L = 0.005$ using $1000^3$ grid took 5800 secs.
- Computed 3M face–edge and 3M face–face–face intersections.
- Optimization: many grid cells were completely inside an input cube.
- Note that this is not simply streaming the data—these are triple-object incidences.
2D and 3D overlay

2D planar graph

- Input: two planar graphs containing sets of polygons (aka faces).
- Output: all the nonempty intersections of one polygon from each map.
- Example: Census tracts with watershed polygons, to estimate population in each watershed.

3D triangulated polyhedra

- presented at International Meshing Roundtable 2018.

Important data structure

*uniform grid.*
Uniform grid

Summary
▶ Overlay a uniform 3D grid on the input.
▶ For each input primitive—face, edge, vertex—find overlapping cells.
▶ In each cell, store set of overlapping primitives.

Properties
▶ Simple, sparse, uses little memory if well programmed.
▶ Parallelizable.
▶ Robust against moderate data nonuniformities.
▶ Bad worst-case performance on extremely nonuniform data.
  ▶ As do octree and all hierarchical methods.

How it works
▶ Intersecting primitives must occupy the same cell.
▶ The grid filters the set of possible intersections.
Uniform Grid Qualities

- Major disadvantages: It’s so simple that it apparently cannot work, especially for nonuniform data. Efficient implementing takes care.

- Major advantage: For the operations I want to do (intersection, containment, etc), it works very well for any real data I’ve ever tried.

- Outside validation: used in our 2nd place finish in ACM 2016 SIGSPATIAL GIS Cup award.

USGS Digital Line Graph; VLSI Design; Mesh
2D Uniform Grid Time Analysis

For i.i.d. edges (line segments), time to find edge–edge intersections in $E^2$ is linear in size(input+output) regardless of varying number of edges per cell.

- N edges, length $1/L$, $G \times G$ grid.
- Expected # intersections = $\Theta(N^2L^{-2})$.
- Each edge overlaps $\leq 2(G/L + 1)$ cells.
- $\eta \triangleq$ # edges per cell, is Poisson; $\bar{\eta} = \Theta(N/G^2(G/L + 1))$.
- Expected total # xsect tests: $G^2\bar{\eta}^2 = N^2/G^2(G/L + 1)^2$.
- Total time: insert edges into cells + test for intersections. $T = \Theta(N(G/L + 1) + N^2/G^2(G/L + 1)^2)$.
- Minimized when $G = \Theta(L)$, giving $T = \Theta(N + N^2L^{-2})$.
- $T = \Theta$ (size of input + size of output).
EPUG-Overlay: 2D planar graph overlay

- Previous step, presented at 2015 ACM SIGSPATIAL
- Time (w/o I/O):
  - 1342 secs (1 thread);
  - 149 secs (16 cores, 32 threads).
  - 9X parallel speedup.
PINMESH: 3D point location

- Previous step, presented at 2016 Berlin Geometry Summit
- Uses rational numbers, Simulation of Simplicity, uniform grid, parallelism, simple data structures
- Biggest example: sample dataset with 50 million triangles.
  - Preprocessing: 14 elapsed seconds on 16-core Xeon.
  - Query time: 0.6 s per point.
Exact fast parallel intersection of large 3-D triangular meshes

- Intersect 3D meshes while
- Handling geometric degeneracies, including
  - Mesh with itself,
  - Mesh with its translation,
  - Mesh with its rotation.
- With no roundoff errors.
- Fast in parallel.
- Economical of memory.
- Extensively tested on hard cases.
- Compared to competing implementations.
- Example: Intersection of two big meshes from AIM@SHAPE: Ramesses: 1.7 million triangles x Neptune: 4 million triangles. 5.5 seconds on multicore Xeon.
Five key techniques

▶ Arbitrary precision rational numbers: for exactness.
▶ Simulation of Simplicity: for ensuring all the special cases are properly handled.
▶ Simple data representation and local information: parallelization and correctness.
▶ Uniform grid: accelerate computation; quickly constructed in parallel.
▶ Parallel programming

Hard part: making everything fit together.
Summary

The following techniques don’t solve all the world’s problems, but handle some foundational geometric ones nicely:

- deprecate hierarchies—
  - simple geometric representation,
  - bucket sort objects with uniform grid.
- local server HW processes large datasets in parallel.
- handles inter-object coincidences (not just streaming processing).
- exploits synergy between CAD and GIS.