Efficient Parallel GIS and CAD Operations on Very Large Data Sets

W. Randolph Franklin, RPI

2016-10-31
What?

- design very fast computational geometry and GIS algorithms, and
- implement and test them on large data sets.
- terrain:
  - multiple observer siting,
  - viewsheds (24G point terrain),
  - hydrography (2.5G points)
  - incremental TINs (100M points).
- maps:
  - overlay (53M edges, 730K faces),
  - simplify (generalize) while preserving control points.
- geometry:
  - nearest point queries in 6D (10M points) or 3D (184M),
  - point location in 3D meshes (50M triangles),
  - compute the volume of the union of 100M cubes,
  - find connected components in 3D (1G voxels)
How?

- simple data structures
- minimal explicit topology
- parallel algorithms
- sometimes handle special cases with Simulation of Simplicity
- sometimes prevent roundoff errors with big rationals.
Why?

- goal: to do something
  - better,
  - faster,
  - in parallel,
  - on bigger datasets

- be useful to others:
  - Our algorithms are well documented in papers, and our code is freely available for nonprofit research and education.
  - This is not always true for competing algorithms.
My background

- Philosophically a Computer Scientist.
- PhD officially in Applied Math.
- Working in Electrical, Computer, and Systems Engineering Dept.
- Students in Computer Science
- Teaching Engineering Parallel Computing.
- Collaborating with Geographers for a long time.
- Enjoy applying computer science & engineering to geometry & GIS.
Structure of this talk

- describe the hard part, and our solution, for several algorithms
- then one big example: overlay two maps.
Collaborators

- Marcus V. A. Andrade
- Daniel Benedetti
- Chaulio Ferreira
- Thiago L. Gomes
- Mauríció G. Gruppi
- David Hedin
- Metin Inanc
- D Keane
- Eric Landis
- Eddie Lau
- Wenli Li
- Ed Nagy
- George Nagy
- Guilherme C. Pena
- Clark K. Ray, III
- Salles Viana Gomes de Magalhaes
- Tong Zhang
Terrain hydrography

- Given a raster terrain, perhaps 50000×50000
- Assume that each cell gets 1 unit of rain.
- Assume that all the water in a cell flows out to its lowest neighbor.
- Compute the water flowing through each cell.
- Application: compute streams and rivers.
Hydrography complication

- Basins (aka depressions, local minima).
- Perhaps nested.
- Basins trap water
- Long rivers don’t form.
- Some basins are real (Death Valley).
- Many are artifacts of the finite sampling.
- Two common solutions.
  - Burn a stream through the divide.
  - Fill the basin to overflowing.
Fill the basin to overflowing

- It’s slow, possibly quadratic time.
- One nice solution:
  - Make all the water from a cell flow to its lowest neighbor, whether or not that neighbor is lower or HIGHER.
  - Then find shortest paths from ocean to all other points.
  - Basins are no longer a special case.
  - Fewer special cases is good.
- Our faster solution:
  - Run the water flow backwards.
  - Raise the ocean slowly.
  - As the water flows over a divide, then
  - Fill everything in the basin up to that level.
  - Done carefully, this is almost linear time.
Raising the ocean

Times:

- 50000x50000 terrain: 1-2 hr.
- much better than:
  - Terraflow: 1 day or more,
  - r.wat.seg fails on big data sets.
Hydrography open questions

Terrain approximation hurts the hydrography.

- Approximating terrain as an array of elevation posts can sometimes give very wrong flows.
- E.g. water flowing on a smooth curved surface.
- This is independent of the resolution.

Possible solution:

- Don’t have the water necessarily (all?) flow to the lowest neighbor.
- This isn’t physical, but so what?

More accurate hydrography models include

- ground absorption of water
- water flow rate downhill

But they’re slow.

- Merge in our fast but idealized algorithm?
Why is speed good?

- "Quantity has a quality all its own."
- Batch becomes interactive.
- Now can
  - test hypotheticals
  - do massive Monte Carlo simulations.
Terrain Viewshed computation

- Given a raster terrain,
- What can an observer see?
- E.g. cell phone tower seeing customers’ phones.
- The size of the viewshed is not always closely related to the observer’s elevation.
- Problem 1: testing one target takes linear time in LOS length, so N-point terrain takes $N^{3/2}$ time.
- Problem 2: usually the line of sight travels between elevation posts.
Our viewshed speedups

1st
- Run lines of sight to only the peripheral targets.
- For each such peripheral target, compute visibility of every target adjacent to the LOS.
- Time is now expected constant per target.

2nd
- For the external algorithm, which pages the terrain
- We know when we’re finished with a block of terrain.
- So, we can do less I/O than the virtual memory manager.
Viewshed implementation

- Largest test: 200K by 122K points.
- 11K seconds.
- Our implementation of Fishman et al failed at 1/4 as many points.
- On data sets that it could do, we were several times faster.
The interpolation rule for elevations between adjacent posts has a large effect on computed viewsheds.

So, what’s the best interpolation?

Some interpolation errors are better than others.
  - Airline pilots have different preferences than cross-country hikers.

A proper rule seems to require a theoretical model of terrain.
Multiple observer siting

- Given a raster terrain, find locations for a large group of observers to jointly cover the most terrain.
- Optimal siting seems to require exponential time,
- So use good heuristics.
Our siting algorithm

Multistep:

- Sample to estimate visibility index of every point.
- Keep only the best 1000 or so points.
- Compute their viewsheds.
- Greedily insert points into the set of final observers.

Enforcing intervisibility.

Volume of the union of many cubes

- Given 100M overlapping congruent cubes,
- Find the volume, area, etc. their union.
- Results are exact; no sampling involved.
- This is a demo of the synergy between several of our techniques.
- Current method:
  - Pair up the cubes; find the union polyhedron of each pair.
  - Pair up the pair polyhedra and find their union polyhedra.
  - Repeat \(\lg(N)\) times.
  - Compute the union of the resulting, very complicated, polyhedron.
Our advance on the volume of the union

- Computing the volume of a polyhedron requires local information about each vertex.
- We need for each vertex,
  - its location,
  - local info about each adjacent face.
- We do not need any more global topological info:
  - not even complete edges or faces.
  - no info about loops or shells.
- Computing only the required info is much easier.
Computing the vertices of the union

- An output vertex is either
  - an input vertex
  - the intersection of three input faces, or
  - the intersection of an edge with a face.
- and the output vertex must not be contained in any input cube.
Required fast subroutines

Times are expected and assume i.i.d. input cubes.

- Find all 3-face intersections in constant time per output intersection.
- Find all edge-face ...
- Test whether a point is contained in any input cube in constant time per point.

All good.

- Theoretical analysis. Linear in output size.
- Implementation and verification. 5821 CPU secs for 100M cubes.

Also, it parallelizes well (in contrast to previous methods).
Nearest point computation.

- Preprocess a set of fixed points, so that we can query with a new test point to return the closest fixed point.
- Previous methods use something like a k-d tree.
  - Superlinear time.
  - Does not parallelize well.

Sample data: UNC complete powerplant, 5.4M points.

Our advance on nearest points.

- Simply use a uniform grid.
- Implemented in Thrust/CUDA on Nvidia GPU.
- For uniform i.i.d. points, it’s provably fast.
- For real data that is locally 2D (e.g., object surface scans),
  - we are slower.
  - but so is everyone else.
- Tested on 184M points in 3D.
- Preprocess and query times both 100x faster than FLANN.
- In higher dimensions,
  - preprocessing 100x faster than FLANN.
  - query faster up to 4D.
Nearest point query times

Query Time vs Number of Fixed Points

- Nearpt3
- NearptD
- FLANN

Time (s)

Number of Fixed Points

$10^4$ to $10^8$
Overlaying two maps (planar graphs) in 2D.

- Big Example:
  - overlay two maps (US Water Bodies, US Block Boundaries)
  - 54,000,000 vertices, 737,000 faces
  - 149 elapsed seconds (plus 116s for I/O).
- Techniques:
  - minimal representations, for simplicity,
  - uniform grid, for fast intersection detection,
  - rational numbers, to prevent roundoff errors,
  - Simulation of Simplicity, for degeneracies,
  - OpenMP, for parallel speedup.
to·pol·o·gy

tpäljē/
noun

1. ...

2. the way in which constituent parts are interrelated or arranged. "the topology of a computer network"

3. I’ll include local geometry
   - location
   - directions

4. Contrast to more global topology
   - complete edges, faces (however, will use these sometimes)
   - edge loops, face shells
   - hierarchies of inclusions
Prior art

- 9 relations in topology
- Morse complexes
- hydrography hierarchy
- winged edges, half edges
- manifold objects
- regularized set ops
How little info does a polygon need?

- Set of vertices is ambiguous.
- Set of edges is good.
  - point in polygon
  - area, center of gravity
- The computation is a map-reduce.
Point Inclusion Testing on a Set of Edges

- "Jordan curve" method
- Extend a semi-infinite ray.
- Count intersections.
- Odd $\equiv$ inside.
- *Obvious but bad alternative:* sum subtended angles.
  Implementing w/o arctan, and handling special cases wrapping around $2\pi$ is tricky and reduces to Jordan curve.
Area Computation on a Set of Edges

- Each edge, with the origin, defines a triangle.
- Sum.
- Extends to any mass property, including (using a characteristic function) point inclusion.
Advantages of Set of Edges 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 space to store.
- Easy parallelization.
  - Partition edges among processors.
  - Each processor sums areas independently, to produce one subtotal.
  - Total the subtotals.
Augmented vertices: another minimal polygon representation

- Augmented vertices: add a little to each vertex.
- My examples will use rectilinear polygons, but all this works on general polygons
- 8 types of vertices.
- Assign a sign, $s = \pm 1$ to each type.
- Now, each vertex defined as $v_i = (x_i, y_i, s_i)$
What augmented vertices can do

- Area: $A = \sum x_i y_i s_i$
But... don’t we always know the edges? (so what’s the point of this?)

- Not always.
- Compute the area of the intersection of two polygons.
- Application: how much do they interfere?
- We know the input polygons’ edges.
- However finding the output polygon’s edges is harder than merely finding the augmented vertices.
- Two types of output vertices:
  - Some input vertices,
  - Some intersections of input edges.
- All output vertices must be inside an input polygon.
- Find candidate output vertices by intersecting pairs of input edges.
- Filter.
- Apply area equation to surviving vertices.
Map overlay

- Input: two maps 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.

- UsWaterBodies: 21,652,410 vertices, 219,831 faces.
- UsBlockBoundaries: 32,762,740 vertices, 518,837 faces.
The five components of map overlay

- simple flat topologically local data structures
- parallelizable
- uniform grid
- simulation of simplicity
- rational numbers
Parallel and memory notes

Massive shared memory

- is an underappreciated resource.
- External memory algorithms are not needed for many problems.
- Virtual memory is obsolete.
- $40K buys a workstation with 80 cores and 1TB of memory.

Parallel computing

- Almost all processors, even my smart phone, are parallel.
- Algorithms that don’t parallelize are obsolete.
- One Xeon core is 20x more powerful than one CUDA core.
- Nvidia GPUs are almost ubiquitous.
Why parallel HW?

- More processing $\rightarrow$ faster clock speed.
- Faster $\rightarrow$ 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!
- $\implies\iff$
- Serial processors have hit a wall.
Parallel HW features

- IBM Blue Gene / Intel / NVidia GPU / other
- Most laptops have NVidia GPUs.
- Thousands of cores / CPUs / GPUs
- Lower clock speed 750MHz vs 3.4GHz
- Hierarchy of memory: small/fast → big/slow
- Communication cost >> computation cost
- Efficient for blocks of threads to execute SIMD.

OS, per 6/2013 http://top500.org :

- runs on 187th fastest machine
- & variants run on 1st through 186th.
Parallel computing

- We use OpenMP (w. shared memory) and CUDA/Thrust (w. Nvidia GPU).

- Our machine:
  - dual 8-core Intel Xeon: 32 hyperthreads.
  - 128GB main memory.
  - Peak Linpack speed: 358Gflops.
  - (Compare: Apple 6s iPhone: 1Gflops.)
  - Nvidia K20Xm compute processor: 2496 CUDA cores @ 706MHz, 6GB memory.
  - cost in 2012 < $15K.
OpenMP

- Shared memory, multiple CPU core model.
- Good for moderate, not massive, 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.
- Only valid cost metric is real time used.
- Programs with 2 threads can execute more slowly than with one.
CUDA and Thrust

- NVIDIA’s parallel computing platform and programming model.
- C++ small language extensions and functions
- Direct access to complicated GPU architecture.
- Nontrivial learning curve: Efficient programming is an art.
- Thrust is C++ template library for CUDA based on STL.
- Functional paradigm: can make algorithms easier to express.

- Hides many CUDA details: good and bad.
- Powerful operators all parallelize: scatter/gather, reduction, reduction by key, permutation, transform iterator, zip iterator, sort, prefix sum.
Multiprecision big rationals

- Solves problem of roundoff error when intersecting lines.
- Slivers no longer matter.
- Code runs slower, but ok.
- Efficiency concerns:
  - Number size depends on computation tree depth. Ok.
  - Millions of heap allocations are inefficient, esp. in parallel. Not ok.
    - Not mentioned in documentation; must infer from experiments.
    - Use Google’s allocator.
    - Refactor code to minimize allocations.
Simulation of simplicity

- Solves problem of geometric degeneracies.
- E.g., vertex of one map coinciding with vertex of the other map.
- Pretends to add a different order of infinitesimal to each coordinate in one map.
  \[(x_i, y_i, z_i) \rightarrow (x_i + \epsilon^{3i}, y_i + \epsilon^{3i+1}, z_i + \epsilon^{3i+2})\]
- Now, coincidences cannot happen, even in intersections.
- Implementation: analyze what effect these infinitesimals would have on every predicate in the program, and
- Recode all the predicates.
- \(if(a_1 \leq b \& b \leq a_2)\) becomes \(if(a_1 \leq b \& b < a_2)\)
Uniform grid

Summary

- Overlay a uniform 3D grid on the universe.
- 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 disadvantage:** It’s so simple that it apparently cannot work, especially for nonuniform data.
- **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 November’s ACM SIGSPATIAL GIS Cup award.

USGS Digital Line Graph; VLSI Design; Mesh
Time analysis for finding edge intersections

For i.i.d. edges (line segments), show that 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 = $\Theta (N^2 L^{-2})$.
▶ Each edge overlaps $\leq 2(G/L + 1)$ cells.
▶ $\eta \overset{\Delta}{=} # \text{ edges per cell, is Poisson; } \bar{\eta} = \Theta(N/G^2(G/L + 1))$.
▶ 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 $T = \Theta(N + N^2 L^{-2})$.
▶ $= \Theta$ (size of input + size of output).
The five components of map overlay

- simple flat topologically local data structures
- parallelizable
- uniform grid
- simulation of simplicity
- rational numbers
Map overlay data structure

Set of edges with names of adjacent polygons

(node 1, node 2)
Left: 1, Right: 2

"outside"
Edge intersection with uniform grid

4x7 uniform grid.
Blue map: 8 edges
Black map: 16 edges

sometimes subdivide once.
Locating vertices
Computing output polygons

Case 2
\((i_6, w) \rightarrow \text{outside other map}\)

- \((u, w)\) divided into 7 segments.
- 5 will be in output.
Parallel implementation

- This algorithm has little data dependency.
- Very parallelizable.
- Uniform grid creation: edges in parallel.
- Locate vertices in polygons.
- Compute intersections: cells in parallel.
- Compute output edges: process input edges in parallel.
Experimental results

| Maps: | Grid size: | BrSoil × BrCounty 200×200 | | UsAq. × UsCounty 400×400 | | UsW Bodies × UsB Bound. 2000×2000 |
|-------|-------------|---------------------------|----------------------|--------------------------|----------------------|
| Threads: | Time (sec.) | Parallel speedup | Time (sec.) | Parallel speedup | Time (sec.) | Parallel speedup |
| Read maps | 1.0 | 1.0 | 5.3 | 5.5 | 73.1 | 74.5 |
| Make grid | 2.0 | 0.6 | 14.2 | 4.4 | 185.9 | 58.0 |
| Refine 2-level grid | 6.3 | 0.4 | 8.4 | 0.5 | 161.6 | 9.9 |
| Intersect edges | 1.0 | 0.1 | 2.6 | 0.3 | 505.5 | 30.9 |
| Locate vertices | 4.8 | 0.4 | 15.3 | 1.7 | 379.0 | 38.5 |
| Comp. output faces | 0.5 | 0.1 | 0.9 | 0.2 | 110.4 | 11.8 |
| Write output | 1.0 | 0.6 | 4.5 | 4.6 | 40.4 | 41.6 |
| Total w/o I/O | 14.6 | 1.6 | 41.4 | 7.1 | 1342.4 | 149.1 |
| Total with I/O | 16.6 | 3.6 | 51.2 | 17.2 | 1455.9 | 265.2 |
Future

- overlay 3D triangulations (tetrahedralizations)
  - Salles Magalhaes PhD
  - in parallel, w/o roundoff errors, fast, etc etc
  - hard problem: intersecting the faces

- longer term: additive manufacturing algorithms.