Local topology and parallel overlaying large planar graphs

W. Randolph Franklin and Salles V. G. de Magalhães, RPI

2016-02
Goal of this talk

- minimal geometry representations for polygons etc.
- applied to overlaying two plane graphs (GIS maps), combining
  - minimal reps, for simplicity,
  - uniform grid, for fast intersection detection,
  - rational numbers, to prevent roundoff errors,
  - Simulation of Simplicity, for degeneracies,
  - OpenMP, for parallel speedup.
- 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).
- next step: overlay 3D meshes.
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.
Aim

- new ways to look at relations between objects in space
- to facilitate spatial operations
  - area
  - overlay
- what is minimal explicit type of info need?
  - fewer special cases
  - less code
  - less debugging
- goal: to do something
  - better,
  - faster,
  - in parallel,
  - on bigger datasets
- All this is intended to be used.
topology

tpälj/
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
More prior art

Winged edge

Figure 4 in Guibas and Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi Diagrams
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: \( A = \sum x_i y_i s_i \)
Vertex incidences: YAMPR

- Another minimal data structure.
- like half edges.
- Only data type is incidence of an edge and a vertex, and its neighborhood. For each such:
  - $V = \text{coord of vertex}$
  - $T = \text{unit tan vector along edge}$
  - $N = \text{unit vector normal to } T \text{ pointing into the polygon.}$
- Polygon: $\{(V, T, N)\}$ (2 tuples per vertex)
- Perimeter $= - \sum (V \cdot T)$.
- Area $= \frac{1}{2} \sum (V \cdot T)(V \cdot N)$
- Multiple nested components ok.
- Parallelizable.
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.
- Salles Viana Gomes de Magalhães presented this at BIGSPATIAL in Nov.
- However, first some foundations:
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!
- $\rightarrow\leftarrow$.
- 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.
Massive Shared Memory

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- Virtual memory is obsolete.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

```cpp
const long long int n(5'000'000'000);
static long long int a[n];
int main() {
    double s(0);
    for (auto &e in a) e = i;
    for (auto e in a) s += e;
    std::cout << "n=\" << n << ", s="
                << s << std::endl; }
```

Runtime: 60 secs w/o opt to loop and r/w 40GB. (6 nsec / iteration)
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.
const int n(500000000);
int a[n], b[n];
int k(0);
int main () {
    #pragma omp parallel for
    for(int i = 0; i < n; i++) a[i]=i;
    #pragma omp parallel for
    for(int i = 0; i < n; i++) {
        #pragma omp atomic capture (or critical)
        j = k++;
        b[j] = j;
    }
    double s(0.);
    #pragma omp parallel for reduction(+:s)
    for (int i=0;i<n;i++) s+=a[i];
    cout << "sum: " << s << endl; }

CUDA

- NVIDIA's parallel computing platform and programming model.
- C++ small language extensions and functions
- CUDA compiler nvcc picks this apart.
- Direct access to complicated GPU architecture.
- Nontrivial learning curve: Efficient programming is an art.
- Assists like Unified Virtual Addressing trade execution vs programming speed.
- My advice: don’t over optimize; next generation will be different.
Thrust

- 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.
- Surprisingly efficient algorithms like bucket sort.
- Possible back ends: CUDA, OpenMP, sequential on host.
Thrust Example

```cpp
struct dofor {
    __device__ void operator()(int &i) { i *= 2; }
};

int main(void) {
    thrust::device_vector<int> X(10);
    thrust::sequence(X.begin(), X.end()); // init to 0,1,
    thrust::fill(Z.begin(), Z.end(), 2); // fill with 2
    // compute Y = X mod 2
    thrust::transform(X.begin(), X.end(), Z.begin(),
        Y.begin(), thrust::modulus<int>());
    thrust::for_each(X.begin(), X.end(), dofor());
    thrust::copy(Y.begin(), Y.end(), // print Y
        std::ostream_iterator<int>(std::cout, "\n"));
}
```
Other techniques used in big example

- rational numbers
- simulation of simplicity
- uniform grid
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
Uniform Grid Time Analysis

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 $= \Theta(N^2L^{-2})$.
- Each edge overlaps $\leq 2(G/L + 1)$ cells.
- $\eta \overset{\Delta}{=} \#$ 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 $T = \Theta(N + N^2L^{-2})$.
- $= \Theta$ (size of input + size of output).
Five components of big example

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

Next: Salles’s ACM BIGSPATIAL talk
Future Modeling of Valid Terrain

My big long-term unsolved problem is to devise a mathematics of terrain.

Goals: Math that
  ▶ allows the representation of only legal terrain (= height of land above geoid),
  ▶ minimizes what needs to be stated explicitly, and
  ▶ enforces global consistencies.

Why? To put compression and other ops on a logical foundation.
Terrain properties

- Messy, not theoretically nice.
- Often discontinuous ($C^{-1}$).
- Many sharp local maxima.
- But very few local minima.
- Lateral symmetry breaking — major river systems.
- Different formation processes in different regions.
- Features do not superimpose linearly; two canyons cannot cross and add their elevations.
- $\mathcal{C}^\infty$ linear systems, e.g., Fourier series, are wrong.
- Multiple related layers (elevation, slope, hydrology).
Current representations

- Array of elevation posts.
- Triangular splines, linear or higher.
- Fourier series.
- Wavelets

Theory vs practice:
- Slope is derivative of elevation, but
- that amplifies errors, and
- lossy compression has errors, so
- maybe we want to store it explicitly.

Also, shoreline is a level set, but see next slide.
Inconsistencies between layers

Elevation contours crossing shoreline
Math should match physics

- Fourier series appropriate for small vibrations, not terrain.
- Truncating a series produces really bad terrain.
- Anything, like Morse complexes, assuming continuity is irrelevant.
- Fractal terrain is not terrain.
- Wavelets: how to enforce long-range consistency?
- Topology, by itself, is too weak.
- Terrain is not linear, not a sum of multiples of basis function.
Terrain formation by scooping

- Problem: Determine the appropriate operators, somewhere inside the range from conceptually shallow (ignoring all the geology) to deep (simulating every molecule).
- One solution: **Scooping.** Carve terrain from a block using a scoop that starts at some point, and following some trajectory, digs ever deeper until falling off the edge of the earth.
- Properties: Creates natural river systems w/ cliffs w/o local minima.
- Every sequence of scoops forms a legal terrain.
- Progressive transmission is easy.

(Chris Stuetzle, *Representation and generation of terrain using mathematical modeling*, PhD, 2012.)
Terrain formation by features

- Represent terrain as a sequence of features — hills, rivers, etc..
- plus a combining rule.
- This matches how people describe terrain.
- Progressive transmission.
- The intelligence is in the combining rule.

How compact is this rep? How to evaluate it?
Implications of a better rep

- Put earlier empirical work on a proper foundation.
- Formal analysis and design of compression.
- Maximum likelihood interpolation, w/o artifacts.
- Treat more sophisticated metrics, like suitability for operations like path planning, or recognizability.
- Close the loop to pre-computer descriptive geometry.
PhD research: An efficient algorithm for computing the exact overlay of triangulations

Salles Viana Gomes de Magalhães, PhD. Student
Prof. Dr. W Randolph Franklin, RPI/Supervisor
Prof. Dr. Marcus V. A. Andrade, UFV
Wenli Li, PhD. Student
Myself

- Universidade Federal de Vicosa, Brazil – 2005-2010.
  - GIS since 2007
  - Areas: HPC, GIS, algorithms ...
  - Dr. Andrade

  - Dr. Franklin
  - Dr. Andrade
  - Wenli Li
Map overlay

- Two vectorial maps are superimposed.
- The intersection between polygons from the two maps is computed.
- Several applications. Ex: counties and watersheds.

- This problem extends to 3D objects (triangulations).
- Example: layers of soil x polyhedron representing excavation section.
Challenge

- Finite precision of floating point → roundoff errors.
  - Common techniques: no guarantee.

- Big amount of data & 3D → increase problem.

- Proposed solution: EPUG-OVERLAY and 3D-EPUG-OVERLAY
EPUG-OVERLAY and 3D-EPUG-OVERLAY

- **EPUG-OVERLAY**
  - **Exact**: uses rational numbers.
  - **Parallel**.
  - **Uniform Grid** for indexing.

- Next steps: 3D-EPUG-OVERLAY
  - Will use the same techniques, but for 3D triangulations
EPUG-OVERLAY

• Simple map representation.
  • No explicit global topology → easy to maintain and avoid topological errors.
  • Easy to process in parallel.

• Simple data structures.
  • Easy to parallelize
  • Efficient
Map representation

- Topological representation.
- Each region has one id.
- Edges represent boundaries.

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

“outside”
Overlay algorithm

• Find all intersections.
• Locate vertices in the other map.
• Compute output polygons.
Computing intersections

• “Brute force”: $O(|A| \times |B|)$
• Other possible technique:
  • Chazelle-Edelsbrunner $O(n \log n + k)$
  • Complicate and doesn't parallelize
• In this work: uniform grid
  • Tests: very efficient
Computing intersections

In this work: uniform grid.

- Insert edges in grid cells (edge may be in several cells).
- For each grid cell $c$, compute intersections in $c$. 

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

- Uniform Grids work well for uneven data.
- For very uneven data: 2-level uniform grid.
Locating vertices in other map

- Also implemented using a uniform grid.
- Given $p$, find the lowest edge above $p$. 
Locating vertices in other map

- Also implemented using a uniform grid.
- Given $p$, find the lowest edge above $p$. 

![Diagram of locating vertices in other map]
Locating vertices in other map

- Also implemented using a uniform grid.
- Given $p$, find the lowest edge above $p$. 
Locating vertices in other map

- Also implemented using a uniform grid.
- Given $p$, find the lowest edge above $p$. 
Locating vertices in other map

- Also implemented using a uniform grid.
- Given $p$, find the lowest edge above $p$. 
Computing output polygons

• Edges of the output polygons → computed based on input edges.
• For each input edge → three scenarios.
Computing output polygons

No intersection.

1 - edge completely inside a polygon (ex: e).
   • Create output edge.

2 - edge completely outside a polygon (ex: f).
   • No output.
Computing output polygons

3 – edge $e=(u,w)$ with intersections.

- $e$ is divided into segments.
- Segments classification → similar to the cases 1 and 2.

- $(u,w)$ divided into 7 segments.
- 5 will be in output.
Computing output polygons

3 – edge \( e=(u,w) \) with intersections.

* \( e \) is divided into segments.
* Segments classification → similar to the cases 1 and 2.

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

Case 1: inside polygon 5
Computing output polygons

3 – edge $e=(u, w)$ with intersections.

- $e$ is divided into segments.
- Segments classification → similar to the cases 1 and 2.

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

Case 2
$(i_6', w) \rightarrow$ outside other map
Parallel implementation

- This algorithm → few 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.

- Most of computers: multicore → OpenMP.

source: wikipedia
Implementation details

- Computation is performed using rational numbers → no roundoff errors.

- EPUG-OVERLAY implemented using GMPXX.

- Special cases: simulation of simplicity.
Experimental results

- EPUUG-OVERLAY implemented in C++.
- Tests:
  - Xeon E5-2687 → 16 cores / 32 threads.
  - 128 GiB of RAM.
  - Linux Mint 17
Experimental results

- 2 Brazilian and 4 North American datasets.
- Shapefiles converted to our format.

- BrCounty: 342,738 vertices, 2,959 faces
- BrSoil: 258,961 vertices, 5,567 faces.
Experimental results

- 2 Brazilian and 2 North American datasets.
- Shapefiles converted to our format.

- UsAquifers: 358,551 vertices, 3,235 faces.
- UsCounty: 3,648,726 vertices, 3,552 faces.
- UsWaterBodies: 21,652,410 vertices, 219,831 faces.
- UsBlockBoundaries: 32,762,740 vertices, 518,837 faces.
Experimental results

- Processing time.
- First level grid: created s.t. the expected number of edges-edges tests per cell = 50.
- Second level grid: 40 x 40 cells, refined when #tests > 50

<table>
<thead>
<tr>
<th>Maps: Grid size:</th>
<th>BrSoil × BrCounty 200×200</th>
<th>UsAq. × UsCounty 400×400</th>
<th>UsWBodies × UsBBound. 2000×2000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads:</td>
<td>Time (sec.)</td>
<td>Parallel speedup</td>
<td>Time (sec.)</td>
</tr>
<tr>
<td>Read maps</td>
<td>1.0</td>
<td>1.0</td>
<td>5.3</td>
</tr>
<tr>
<td>Make grid</td>
<td>2.0</td>
<td>0.6</td>
<td>14.2</td>
</tr>
<tr>
<td>Refine 2-level grid</td>
<td>6.3</td>
<td>0.4</td>
<td>8.4</td>
</tr>
<tr>
<td>Intersect edges</td>
<td>1.0</td>
<td>0.1</td>
<td>2.6</td>
</tr>
<tr>
<td>Locate vertices</td>
<td>4.8</td>
<td>0.4</td>
<td>15.3</td>
</tr>
<tr>
<td>Comp. output faces</td>
<td>0.5</td>
<td>0.1</td>
<td>0.9</td>
</tr>
<tr>
<td>Write output</td>
<td>1.0</td>
<td>0.6</td>
<td>4.5</td>
</tr>
<tr>
<td>Total w/o I/O</td>
<td>14.6</td>
<td>1.6</td>
<td>41.4</td>
</tr>
<tr>
<td>Total with I/O</td>
<td>16.6</td>
<td>3.6</td>
<td>51.2</td>
</tr>
</tbody>
</table>
Experimental results

- Processing time.
- First level grid: created s.t. the expected number of edges-edges tests per cell = 50.
- Second level grid: created once the number of tests per cell exceeds 50.
- Good speedup: ~200-300 thousand edges/vertices, up to ~3 million edges/vertices, ~20-30 million edges/vertices.

<table>
<thead>
<tr>
<th>Maps:</th>
<th>Grid size:</th>
<th>BrSoil × BrCounty</th>
<th>UsAq. × UsCounty</th>
<th>UsWBodies × UsBBound.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>200×200</td>
<td>400×400</td>
<td>2000×2000</td>
<td></td>
</tr>
<tr>
<td>Threads:</td>
<td>Time (sec.)</td>
<td>Parallel speedup</td>
<td>Time (sec.)</td>
<td>Parallel speedup</td>
</tr>
<tr>
<td>Read maps</td>
<td>1.0</td>
<td>1.0</td>
<td>5.3</td>
<td>5.5</td>
</tr>
<tr>
<td>Make grid</td>
<td>2.0</td>
<td>0.6</td>
<td>14.2</td>
<td>4.4</td>
</tr>
<tr>
<td>Refine 2-level grid</td>
<td>6.3</td>
<td>0.4</td>
<td>8.4</td>
<td>0.5</td>
</tr>
<tr>
<td>Intersect edges</td>
<td>1.0</td>
<td>0.1</td>
<td>2.6</td>
<td>0.3</td>
</tr>
<tr>
<td>Locate vertices</td>
<td>4.8</td>
<td>0.4</td>
<td>15.3</td>
<td>1.7</td>
</tr>
<tr>
<td>Comp. output faces</td>
<td>0.5</td>
<td>0.1</td>
<td>0.9</td>
<td>0.2</td>
</tr>
<tr>
<td>Write output</td>
<td>1.0</td>
<td>0.6</td>
<td>4.5</td>
<td>4.6</td>
</tr>
<tr>
<td>Total w/o I/O</td>
<td>14.6</td>
<td>1.6</td>
<td>41.4</td>
<td>7.1</td>
</tr>
<tr>
<td>Total with I/O</td>
<td>16.6</td>
<td>3.6</td>
<td>51.2</td>
<td>17.2</td>
</tr>
</tbody>
</table>
Experimental results

- Processing time.
- First level grid: created s.t. the expected number of edges-edges tests per cell = 50.
- Second level grid: 40 x 40 cells, refined when #tests > 50.
- Good speedup ~ 200-300 thousand edges/vertices

<table>
<thead>
<tr>
<th>Maps:</th>
<th>Grid size:</th>
<th>BrSoil × BrCounty 200×200</th>
<th>Time (sec.)</th>
<th>Parallel speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads:</td>
<td></td>
<td></td>
<td>1</td>
<td>32</td>
</tr>
<tr>
<td>Read maps</td>
<td></td>
<td>1.0</td>
<td>1.0</td>
<td>1</td>
</tr>
<tr>
<td>Make grid</td>
<td></td>
<td>2.0</td>
<td>0.6</td>
<td>3</td>
</tr>
<tr>
<td>Refine 2-level grid</td>
<td></td>
<td>6.3</td>
<td>0.4</td>
<td>15</td>
</tr>
<tr>
<td>Intersect edges</td>
<td></td>
<td>1.0</td>
<td>0.1</td>
<td>8</td>
</tr>
<tr>
<td>Locate vertices</td>
<td></td>
<td>4.8</td>
<td>0.4</td>
<td>12</td>
</tr>
<tr>
<td>Comp. output faces</td>
<td></td>
<td>0.5</td>
<td>0.1</td>
<td>4</td>
</tr>
<tr>
<td>Write output</td>
<td></td>
<td>1.0</td>
<td>0.6</td>
<td>2</td>
</tr>
<tr>
<td>Total w/o I/O</td>
<td></td>
<td>14.6</td>
<td>1.6</td>
<td>9</td>
</tr>
<tr>
<td>Total with I/O</td>
<td></td>
<td>16.6</td>
<td>3.6</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Maps:</th>
<th>Grid size:</th>
<th>UsAq. × UsCounty 400×400</th>
<th>Time (sec.)</th>
<th>Parallel speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads:</td>
<td></td>
<td></td>
<td>1</td>
<td>32</td>
</tr>
<tr>
<td>Read maps</td>
<td></td>
<td>5.3</td>
<td>5.5</td>
<td>1</td>
</tr>
<tr>
<td>Make grid</td>
<td></td>
<td>14.2</td>
<td>4.4</td>
<td>3</td>
</tr>
<tr>
<td>Refine 2-level grid</td>
<td></td>
<td>8.4</td>
<td>0.5</td>
<td>16</td>
</tr>
<tr>
<td>Intersect edges</td>
<td></td>
<td>2.6</td>
<td>0.3</td>
<td>8</td>
</tr>
<tr>
<td>Locate vertices</td>
<td></td>
<td>15.3</td>
<td>1.7</td>
<td>9</td>
</tr>
<tr>
<td>Comp. output faces</td>
<td></td>
<td>0.9</td>
<td>0.2</td>
<td>5</td>
</tr>
<tr>
<td>Write output</td>
<td></td>
<td>4.5</td>
<td>4.6</td>
<td>1</td>
</tr>
<tr>
<td>Total w/o I/O</td>
<td></td>
<td>41.4</td>
<td>7.1</td>
<td>6</td>
</tr>
<tr>
<td>Total with I/O</td>
<td></td>
<td>51.2</td>
<td>17.2</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Maps:</th>
<th>Grid size:</th>
<th>UsWBodies × UsBBound. 2000×2000</th>
<th>Time (sec.)</th>
<th>Parallel speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads:</td>
<td></td>
<td></td>
<td>1</td>
<td>32</td>
</tr>
<tr>
<td>Read maps</td>
<td></td>
<td></td>
<td>73.1</td>
<td>74.5</td>
</tr>
<tr>
<td>Make grid</td>
<td></td>
<td></td>
<td>185.9</td>
<td>58.0</td>
</tr>
<tr>
<td>Refine 2-level grid</td>
<td></td>
<td>161.6</td>
<td>9.9</td>
<td>16</td>
</tr>
<tr>
<td>Intersect edges</td>
<td></td>
<td>505.5</td>
<td>30.9</td>
<td>16</td>
</tr>
<tr>
<td>Locate vertices</td>
<td></td>
<td>379.0</td>
<td>38.5</td>
<td>10</td>
</tr>
<tr>
<td>Comp. output faces</td>
<td></td>
<td>110.4</td>
<td>11.8</td>
<td>9</td>
</tr>
<tr>
<td>Write output</td>
<td></td>
<td>40.4</td>
<td>41.6</td>
<td>1</td>
</tr>
<tr>
<td>Total w/o I/O</td>
<td></td>
<td>1342.4</td>
<td>149.1</td>
<td>9</td>
</tr>
<tr>
<td>Total with I/O</td>
<td></td>
<td>1455.9</td>
<td>265.2</td>
<td>6</td>
</tr>
</tbody>
</table>

Good speedup ~ 20-30 million edges/vertices
Experimental results

- Processing time.
- First level grid: created s.t. the expected number of edges-edges tests per cell = 50.
- Second level grid: 40 x 40 cells, refined when #tests > 50.
- Good speedup ~ 200-300 thousand edges/vertices.
- Up to ~3 million edges/vertices.
- ~ 20-30 million edges/vertices.

<table>
<thead>
<tr>
<th>Maps: Grid size:</th>
<th>BrSoil × BrCounty 200×200</th>
<th>UsAq. × UsCounty 400×400</th>
<th>UsWBodies × UsBBound 2000×2000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads:</td>
<td>BrSoil × BrCounty 200×200</td>
<td>UsAq. × UsCounty 400×400</td>
<td>UsWBodies × UsBBound 2000×2000</td>
</tr>
<tr>
<td></td>
<td>Time (sec.)</td>
<td>Parallel speedup</td>
<td>Time (sec.)</td>
</tr>
<tr>
<td>Read maps</td>
<td>1.0</td>
<td>1.0</td>
<td>5.3</td>
</tr>
<tr>
<td>Make grid</td>
<td>2.0</td>
<td>0.6</td>
<td>14.2</td>
</tr>
<tr>
<td>Refine 2-level grid</td>
<td>6.3</td>
<td>0.4</td>
<td>8.4</td>
</tr>
<tr>
<td>Intersect edges</td>
<td>1.0</td>
<td>0.1</td>
<td>2.6</td>
</tr>
<tr>
<td>Locate vertices</td>
<td>4.8</td>
<td>0.4</td>
<td>15.3</td>
</tr>
<tr>
<td>Comp. output faces</td>
<td>0.5</td>
<td>0.1</td>
<td>0.9</td>
</tr>
<tr>
<td>Write output</td>
<td>1.0</td>
<td>0.6</td>
<td>4.5</td>
</tr>
<tr>
<td>Total w/o I/O</td>
<td>14.6</td>
<td>1.6</td>
<td>41.4</td>
</tr>
<tr>
<td>Total with I/O</td>
<td>16.6</td>
<td>3.6</td>
<td>51.2</td>
</tr>
</tbody>
</table>
**Experimental results**

- Processing time.
- First level grid: created s.t. the expected number of edges-edges tests per cell = 50.
- Second level grid: 40 x 40 cells, refined when #tests > 50.
- Good speedup: ~200-300 thousand edges/vertices.
- Up to ~3 million edges/vertices.
- ~20-30 million edges/vertices.

<table>
<thead>
<tr>
<th>Maps: BrSoil × BrCounty 200x200</th>
<th>Time (sec.)</th>
<th>Parallel speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads:</td>
<td>1</td>
<td>32</td>
</tr>
<tr>
<td>Read maps</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>Make grid</td>
<td>2.0</td>
<td>0.6</td>
</tr>
<tr>
<td>Refine 2-level grid</td>
<td>6.3</td>
<td>0.4</td>
</tr>
<tr>
<td>Intersect edges</td>
<td>1.0</td>
<td>0.1</td>
</tr>
<tr>
<td>Locate vertices</td>
<td>4.8</td>
<td>0.4</td>
</tr>
<tr>
<td>Comp. output faces</td>
<td>0.5</td>
<td>0.1</td>
</tr>
<tr>
<td>Write output</td>
<td>1.0</td>
<td>0.6</td>
</tr>
<tr>
<td>Total w/o I/O</td>
<td>14.6</td>
<td>1.6</td>
</tr>
<tr>
<td>Total with I/O</td>
<td>16.6</td>
<td>3.6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Maps: UsAquifers × UsCounties 400x400</th>
<th>Time (sec.)</th>
<th>Parallel speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads:</td>
<td>1</td>
<td>32</td>
</tr>
<tr>
<td>Read maps</td>
<td>5.3</td>
<td>5.5</td>
</tr>
<tr>
<td>Make grid</td>
<td>14.2</td>
<td>4.4</td>
</tr>
<tr>
<td>Refine 2-level grid</td>
<td>8.4</td>
<td>0.5</td>
</tr>
<tr>
<td>Intersect edges</td>
<td>2.6</td>
<td>0.3</td>
</tr>
<tr>
<td>Locate vertices</td>
<td>15.3</td>
<td>1.7</td>
</tr>
<tr>
<td>Comp. output faces</td>
<td>0.9</td>
<td>0.2</td>
</tr>
<tr>
<td>Write output</td>
<td>4.5</td>
<td>4.6</td>
</tr>
<tr>
<td>Total w/o I/O</td>
<td>41.4</td>
<td>7.1</td>
</tr>
<tr>
<td>Total with I/O</td>
<td>51.2</td>
<td>17.2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Maps: UsWbodies × UsBBoundaries 2000x2000</th>
<th>Time (sec.)</th>
<th>Parallel speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads:</td>
<td>1</td>
<td>32</td>
</tr>
<tr>
<td>Read maps</td>
<td>73.1</td>
<td>74.5</td>
</tr>
<tr>
<td>Make grid</td>
<td>185.9</td>
<td>58.0</td>
</tr>
<tr>
<td>Refine 2-level grid</td>
<td>161.6</td>
<td>9.9</td>
</tr>
<tr>
<td>Intersect edges</td>
<td>505.5</td>
<td>30.9</td>
</tr>
<tr>
<td>Locate vertices</td>
<td>379.0</td>
<td>38.5</td>
</tr>
<tr>
<td>Comp. output faces</td>
<td>110.4</td>
<td>11.8</td>
</tr>
<tr>
<td>Write output</td>
<td>40.4</td>
<td>41.6</td>
</tr>
<tr>
<td>Total w/o I/O</td>
<td>1342.4</td>
<td>149.1</td>
</tr>
<tr>
<td>Total with I/O</td>
<td>1455.9</td>
<td>265.2</td>
</tr>
</tbody>
</table>

Mem. alloc.
Experimental results

- Processing time.
- First level grid: created s.t. the expected number of edges-edges tests per cell = 50.
- Second level grid: created s.t. the expected number of edges-edges tests per cell = 40 x 40.
- Good speedup: ~200-300 thousand edges/vertices, up to ~3 million edges/vertices, ~20-30 million edges/vertices.

Grass (serial/not exact): 5321s

<table>
<thead>
<tr>
<th>Maps: Grid size:</th>
<th>BrSoil × BrCounty 200×200</th>
<th>UsAq. × UsCounty 400×400</th>
<th>UsWBodies × UsBBound. 2000×2000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads: Time (sec.)</td>
<td>Parallel speedup</td>
<td>Time (sec.)</td>
<td>Parallel speedup</td>
</tr>
<tr>
<td>Read maps</td>
<td>1.0</td>
<td>1.0</td>
<td>5.3</td>
</tr>
<tr>
<td>Make grid</td>
<td>2.0</td>
<td>0.6</td>
<td>14.2</td>
</tr>
<tr>
<td>Refine 2-level grid</td>
<td>6.3</td>
<td>0.4</td>
<td>8.4</td>
</tr>
<tr>
<td>Intersect edges</td>
<td>1.0</td>
<td>0.1</td>
<td>2.6</td>
</tr>
<tr>
<td>Locate vertices</td>
<td>4.8</td>
<td>0.4</td>
<td>15.3</td>
</tr>
<tr>
<td>Comp. output faces</td>
<td>0.5</td>
<td>5.4</td>
<td>110.4</td>
</tr>
<tr>
<td>Write output</td>
<td>1.0</td>
<td>1.0</td>
<td>40.4</td>
</tr>
<tr>
<td>Total w/o I/O</td>
<td>14.6</td>
<td>1.6</td>
<td>41.4</td>
</tr>
<tr>
<td>Total with I/O</td>
<td>16.6</td>
<td>3.6</td>
<td>51.2</td>
</tr>
</tbody>
</table>
## Experimental results

- Why not have 3, 4, 5 levels, … , quadtree?
- Uniform grid: simple and easily parallelizable.
- More levels: +memory and +time to create.

<table>
<thead>
<tr>
<th>Maps overlaid</th>
<th>3-level grid</th>
<th>Quadtree</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1&lt;sup&gt;st&lt;/sup&gt;</td>
<td>2&lt;sup&gt;nd&lt;/sup&gt; &amp; 3&lt;sup&gt;rd&lt;/sup&gt;</td>
</tr>
<tr>
<td>BrSoil × BrCounty</td>
<td>200&lt;sup&gt;2&lt;/sup&gt;</td>
<td>40&lt;sup&gt;2&lt;/sup&gt;</td>
</tr>
<tr>
<td>UsAquifers × UsCounty</td>
<td>400&lt;sup&gt;2&lt;/sup&gt;</td>
<td>40&lt;sup&gt;2&lt;/sup&gt;</td>
</tr>
<tr>
<td>UsW Bodies × UsBBound.</td>
<td>2000&lt;sup&gt;2&lt;/sup&gt;</td>
<td>40&lt;sup&gt;2&lt;/sup&gt;</td>
</tr>
</tbody>
</table>
Experimental results

- Why not have 3, 4, 5 levels, … , quadtree?
- Uniform grid: simple and easily parallelizable.
- More levels: +memory and +time to create.

<table>
<thead>
<tr>
<th>Maps overlaid</th>
<th>3-level grid</th>
<th>Quadtree</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1\textsuperscript{st}</td>
<td>2\textsuperscript{nd} &amp; 3\textsuperscript{rd}</td>
</tr>
<tr>
<td>BrSoil $\times$ BrCounty</td>
<td>200\textsuperscript{2}</td>
<td>40\textsuperscript{2}</td>
</tr>
<tr>
<td>UsAquifers $\times$ UsCounty</td>
<td>400\textsuperscript{2}</td>
<td>40\textsuperscript{2}</td>
</tr>
<tr>
<td>UsW Bodies $\times$ UsB Bound.</td>
<td>2000\textsuperscript{2}</td>
<td>40\textsuperscript{2}</td>
</tr>
</tbody>
</table>

More time than our entire algorithm!
Next steps: 3D-EPUG-OVERLAY

- Work in progress.

- Will use similar techniques:
  - Rational numbers
  - “3D maps” represented by a set of triangles
  - Triangles: left/right objects
  - 3D uniform grid for intersection and point in polygon
  - Simulation of simplicity
  - Algorithm designed to be parallel

- EPUG-OVERLAY is efficient → 3D-EPUG0-OVERLAY will be.
Conclusions

- EPUG-OVERLAY is an efficient method.
- Use precise arithmetic, but the performance is comparable with GRASS.
- Parallelizable algorithm → use computing power of modern computers.

- Work in progress: 3D-EPUG-OVERLAY.

- Future work:
  - Compare the quality of the output.
  - Perform more theoretical analysis.
Thank you!

Acknowledgement:

Contact:
Salles V. G. de Magalhaes: vianas2@rpi.edu
W. Randolph Franklin: mail@wrfranklin.org

An efficient algorithm for computing the exact overlay of triangulations
Experimental results

- The importance of the two-level uniform grid.
- UsW Bodies x UsBBound.
- 1 level: 20,000 cells w/ 10,000+ pairs of edges
- 2 levels: 100 cells w/10,000+ pairs of edges!