Algorithms, libraries, and development environments to process huge geoinformatic databases on modern hardware.

W. Randolph Franklin

Rensselaer Polytechnic Institute
Troy, NY, USA

2015-04-15
Outline

1. Large DB vs New HW
   - Why parallel HW?
   - Massive Shared Memory
   - Geometric Databases

2. HW Tools
   - OpenMP
   - CUDA
   - Thrust

3. SW Tools
   - Multiprecision big rationals
   - CGAL
   - MATLAB
   - Arrayfire
   - C++ Matrix Packages

4. Algorithm Tactics
   - 3D Uniform Grid

5. Examples
   - Volume of Union of Many Cubes
   - Improved Parallel ODETLAP
   - Parallel Multiple Observer Siting

6. Future Modeling of Valid Terrain
   - Terrain properties
   - Current representations
   - Inconsistencies between layers
   - Math should match physics
   - Terrain formation by scooping
   - Terrain formation by features
   - Implications of a better rep

7. Summary
Large Geometric/GIS Datasets vs New HW Capabilities

- Large geometric/GIS datasets $\gg 10^6$ objects.
- New parallel HW, but with restricted capabilities.
- $\therefore$ Need new algorithms, data structures.

(Stanford U, Digital Michelangelo Project)
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 $\rightarrow$ big/slow
- Communication cost $\gg$ computation cost
- Efficient for blocks of threads to execute SIMD.
- OS:
  - 1st through 186th fastest run variants of Linux.
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- Virtual memory is obsolete.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.

Code

```cpp
const long long int n(5000000000);
static long long int a[n];
int main() {
    double s(0);
    for (long long int i=0; i<n; i++)
        a[i] = i;
    for (long long int i=0; i<n; i++)
        s += a[i];
    std::cout << "n=" << n << " s=" << s << std::endl; }```

Runtime: 60 secs w/o opt to loop and r/w 40GB. (6 nsec / iteration)
Geometric Databases

- Larger and larger geometric databases now available, with tens of millions of primitive components.

- Some needed operations:
  - nearest point
  - boolean intersection and union
  - planar graph overlay
  - mass property computation of the results of some boolean operation

- Apps:
  - Volume of an object defined as the union of many overlapping primitives. Two objects interfere iff the volume of intersection is positive.
  - Interpolate population data from census tracts to flood zones.
Outline

1. Large DB vs New HW
2. HW Tools
   - OpenMP
   - CUDA
   - Thrust
3. SW Tools
4. Algorithm Tactics
5. Examples
6. Future Modeling of Valid Terrain
7. Summary
Tools to Access the HW

- OpenMP
- CUDA
- Thrust

(Highlights of the 42nd Top500 List, SC13)
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.
- Just compiling with OpenMP slows the program.
- I’ve seen programs where 2 threads executed more slowly than with one.
- OTOH, union3 runs slightly faster with 32 threads than with 16, and 10x faster than with 1.
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; }
- NVIDIA's parallel computing platform and programming model.
- Direct access to complicated GPU architecture.
- Learning curve: Efficient programming an art.
- Assists like Unified Virtual Addressing trade execution vs programming speed.
- Following slide from http://www2.engr.arizona.edu/~yangsong/gpu2.png.

**Code Fragment**

```c
__global__ void device_greetings(void)
{
    cuPrintf("Hello, from the device!\n");
}

cudaMalloc((void**)&device_array,
    num_bytes)

cudaMemcpy(host_array, device_array,
    num_bytes, cudaMemcpyDeviceToHost);

device_greetings<<<2,3>>>()
```
GPU Architecture
Thrust

- C++ template library for CUDA based on STL.
- Functional paradigm: can make algorithms easier to express.
- Hides many CUDA details: good and bad.
- Tries to be efficient, e.g., uses radix sort.
- Powerful operators all parallelize: scatter/gather, reduction, reduction by key, permutation, transform iterator, zip iterator, sort, prefix sum.
- Surprisingly efficient algorithms like bucket sort.
Thrust Example

Code

```c
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 0,1,2, ...
    thrust::fill(Z.begin(), Z.end(), 2); // fill with 2s
    // 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(),
                 std::ostream_iterator<int>(std::cout, "\n");
}
```
Outline

1. Large DB vs New HW
2. HW Tools
3. SW Tools
   - Multiprecision big rationals
   - CGAL
   - MATLAB
   - Arrayfire
4. Algorithm Tactics
5. Examples
6. Future Modeling of Valid Terrain
7. Summary
SW Tools

- gmp++ – big rationals.
- Computational Geometry Algorithms Library – CGAL.
- Matlab.
- Arrayfire.
- Mathematics.
Multiprecision big rationals

- Solves problem of roundoff error when intersecting lines.
- Example uses gmp++ via boost.

Code

```cpp
#include <boost/multiprecision/gmp.hpp>
using namespace boost::multiprecision;
int main() {
    mpq_rational v;
    for(mpq_rational i = 1; i <= 8; ++i) {
        v += (2*i)/(2*i+1);
        std::cout << i << ": " << v << std::endl;
    }
}
```

Output

1: 2/3
2: 22/15
3: 244/105
4: 1012/315
5: 14282/3465
6: 227246/45045
7: 269288/45045
8: 5298616/765765
Gmp++ Impressions

- For 100,000 iterations, time= 0.08s.
- However, this is a particularly stressful example for rationals.
- When overlaying 2 planar maps, computation depth is only 2; CPU time tolerable.
- This is the proper solution to the problem of roundoff error with straight lines.
- Good for algorithms where computation tree is shallow. E.g., not for area of large polygon (which I don’t need accurately anyway).
Curves and Roundoff

What about this: \( x(t) = \sum_{i=0}^{3} a_i t^3 \), \( y(t) = \sum_{i=0}^{3} b_i t^3 \)

- Can use resultants to eliminate \( t \) to give \( f(x, y) = 0 \) (degree 9), but
  - curve’s topology is difficult to determine.
  - \( 0 \leq t \leq 1 \) is only a part of that curve.
- Can intersect \( f(x, y) = 0, \ g(x, y) = 0 \) to give \( h(x) = 0 \), but
  - much higher degree; extraneous roots.
- For tests, can bracket the root away from 0 with adaptive precision.
  - often fast; always correct.
- CGAL can guarantee correctness for tests w/o exact computation.
  - point vs line; circular direction of 3 points.
CGAL

- Computational Geometry Algorithms Library
- C++ classes; efficient and reliable geometric algorithms.

**Code Fragment**

```cpp
#include <iostream>
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/convex_hull_2.h>
typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef K::Point_2 Point_2;
int main() {
    Point_2 points[5] = { Point_2(0,0), Point_2(10,0) ... };
    Point_2 result[5];
    Point_2 *ptr = CGAL::convex_hull_2( points, points+5, result );
    std::cout << ptr - result << " points on the convex hull"
               << std::endl;
    return 0; }
```
CGAL Impressions

- Massive effort with algorithms and kernels.
- Doesn’t look likely to parallelize.
- Theoretically excellent base for sequential projects.
- Dozens of programmer-years.
- Many demos and examples.
- Excellent documentation.
- Dependencies on other packages (not bundled for legal reasons).
- Just installing it is nontrivial.
- Serious learning curve.
- Living project — still growing.
Matlab

- Excellent prototyping and development environment for engineers.
- Handles vectors and matrices well; otherwise not so easy.
- Efficient programming requires obscure idioms.
- Explicit loops are very expensive.
- Has many engineering tools, e.g., for signal processing.
- State-of-the-art numerical algorithms, e.g., for solving overdetermined sparse systems of linear equations.
- Expensive (and increasingly so) commercial product.
- Can utilize multiple CPU cores.
- Can call out to C.
- Exploits CPU and GPU multi-cores for certain numerical ops.
- I’ve used it for years, although considering moving to C++.
• Fast parallel matrix manipulation library for C++.

• Now free w source on github.

• Possible backends: CUDA, CPU, OpenCL.

• Compile with c++ or nvcc.

• I’m still learning it.
2D ODE TLAP – Overdetermined Laplacian Method

Application of MATLAB

• Small set of posts ⇒ complete matrix of posts

• Overdetermined linear system:
  • \( z_{ij} = h_{ij} \) for known points,
  • \( 4z_{ij} = z_{i-1,j} + z_{i+1,j} + z_{i,j-1} + z_{i,j+1} \) for all nonborder points.
  • Emphasize accuracy or smoothness by weighting the two types of equations differently.

• Original goal: fill contours to a grid w/o showing terraces; competing methods have these problems:
  • Information does not flow across contours ⇒ slopes discontinuous
  • If rays are fired from the test point to the first known point, then method is not conformal etc.
ODETLAP Advantages

Handles
- missing-data holes.
- incomplete contours,
- complete contours,
- kidney-bean contours,
- isolated points,
- inconsistent data.
• input: contours with sharp corners
• output: smooth silhouette edges, inferred top
ODETLAP computational requirements

- Depends on number of known points, which
- Affects sparsity of system.
- $400 \times 400$ data grid with 1000 points $\Rightarrow$
- Solving $160000 \times 170000$ overdetermined sparse system is fast.
- Solving a $104976 \times 104976$ data grid: 58 GB main memory, 1.8 hours on workstation with four 2.4GHz processors and 60 GB of main memory running Ubuntu 10.04.2 and 64-bit Matlab R2009a.
- Approximate iterative solution sufficient.
Code Fragment

- Interactive scripting, matrix data types, very efficient algorithms

```matlab
function [W] = odetlap8(Z, R)
leftbordind=[1:N];
cornerind=[1,N,N2-N+1,N2];
maindiag=z(ones(N2,1));
maindiag([leftbordind,rightbordind,topbordind,bottombordind])=3*R;
maindiag(cornerind)=2*R;
z2=-R;
plusonediag=z2(ones(N2,1));
plusonediag(bottombordind)=0;
B=sparse(N2,N2);
B=spdiags([maindiag,plusonediag,minusonediag,plusNdiag,minusNdiag],[0,-1,1,N,-N],B);
X=[1:np]';
sechalf=spconvert([X,zeroind,ones(np,1);(np+1) N2 0]);
B=[B;sechalf];
C=[zeros(N2,1);Z(zeroind);0];
NE=size(B,1);
WD=zeros(N2,1);
W1 = B \ C; % Solve the overdetermined linear system.
```
Parallel implementation

- Test data: $234256 \times 234256$ linear system from 5D-ODETLAP.
- Matlab CF direct solver: 49237 secs.
- Matlab iterative Conjugate Gradient (CG) solver: 5495 secs.
- CUSP: open source sparse linear system library using CUDA.
- CUSP CG solver: 179 secs.
- All but 9 secs was data transfer.

Optimization

- Use a Matlab executable (MEX).
- Write CUSP code in MEX style.
- Compile into C++. (CUDA can be compiled into C++).
- Then compile into MEX.
- Then call from Matlab.
- On $104976^2$ system, Matlab direct time: 6488 secs, Matlab CG: 40 secs, CUSP CG: 5 secs.
- Also uses less memory (only 13GB in Matlab, 512MB in GPU).
Matlab Calling CUDA

- Input data: 18x18x18x18
- CG solver: based on CUSP
- CUDA NVCC compiler
- Matlab Mex compiler
- Linear system: 4096x4096
- 5D-ODETLAP construction
- Integrated CG solver in Matlab
- Linear solution: 4096x1
- in .CU format
- in .cpp format
C++ Matrix Packages

- C++ packages that operate on matrices as does Matlab.
- Aggressive use of templates – fragile, obscure errors.
- C++ execution efficiency (excellent).
- E.g., boost, Eigen.
- Not as nicely packaged as Matlab – must search for components.
- Free, but some projects become abandoned.
Mathematica

- Large inclusive development environment, originally algebra, now general.
- Everything fits together beautifully — you can have an array of plots.
- Excellent presentation and plots.
- Some things missing beneath the surface, e.g., cannot handle large sparse systems (unlike Matlab).
- Expensive commercial product.
- Living, growing, system.
- For years, I’ve been intending to make more use of it.
Outline

1. Large DB vs New HW
2. HW Tools
3. SW Tools
4. Algorithm Tactics
   3D Uniform Grid
5. Examples
6. Future Modeling of Valid Terrain
7. Summary
• I/O more limiting than computation $\rightarrow$ minimize storage.
• For $N \gg 1000000$, $\lg N$ is nontrivial $\rightarrow$ deprecate binary trees.
• Minimize explicit topology, especially 3D.
• Plan for 3D; many 2D data structures not easily extensible to 3D, e.g., line sweep.
• E.g., Voronoi diagram: 2D is $\Theta(N \lg N)$. 3D is $\Theta(N^2)$
• Optimize function composition, e.g. $\text{Volume(Union(S))}$. 
3D Uniform Grid

Summary

• Overlay a uniform 3D grid on the universe.
• For each input primitive — cube, face, edge — 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.
• Ditto any hierarchical method like octree.

Advantage

• 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.
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 = $\Theta(N^2 L^{-2})$.
- Each edge overlaps $\leq 2(G/L + 1)$ cells.
- $\eta \triangleq$ # edges per cell, is Poisson distributed. $\bar{\eta} = \Theta(N/G^2(G/L + 1))$.
- Expected total # intersection tests: $N^2/G^2(G/L + 1)^2$.
- 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 $T = \Theta(N + N^2 L^{-2})$.

Q.E.D.
1. Large DB vs New HW
2. HW Tools
3. SW Tools
4. Algorithm Tactics
5. Examples
   - Volume of Union of Many Cubes
   - Improved Parallel ODETLAP
   - Parallel Multiple Observer Siting
6. Future Modeling of Valid Terrain
7. Summary
Examples

- Volume of Union of Many Cubes, unifying example that includes:
  - testing point inclusion in expected constant time per point independent of polyhedron size,
  - finding all 3-face intersections in expected time linear in size of output,
  - finding mass properties by map-reduce on set of polyhedron vertices.
- Improved ODETLAP with CUDA
- Parallel Multiple Observer Siting
Volume of Union of Many Cubes

- Nice unifying illustration of several ideas.
- Do a prototype on an easy subcase (congruent axis-aligned cubes).
- Idea extends to general polyhedra.
- **Not** statistical sampling — 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-Polygon Union

• Construct pairwise unions of primitives.
• Iterate.

Time depends on intermediate swell, and elementary intersection time.
• Let \( P = \text{size of union of an M-gon and an N-gon} \). 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

- \( \lg N \) levels in computation tree cause \( \lg N \) 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 right.
- However explicit output not needed for computing mass properties.
- Set of vertices with neighborhoods suffices.
Volume Determination

Box: \[ V = \sum_i s_i x_i y_i z_i \]
\[ s_i : +1 \text{or} -1 \]

General rectilinear polygons:
- 8 types of vertices, based on neighborhood
- 4 are type +, 4 –
- Area = \[ \sum_i s_i x_i y_i \]

Rectilinear polyhedra: \[ V = \sum_i s_i x_i y_i z_i \]
∃ formulae for general polyhedra.
Properties

Represent output union polyhedron as set of vertices with neighborhoods.

- no explicit edges; no edge loops.
- no explicit faces; no face shells.
- no component containment info.
- general polygons ok: multiple nested or separate comps.
- any mass property determinable in one pass thru the set.
- parallelizable.

- This is very useful because computing only the output vertices and neighborhoods is much faster than also computing the edges, faces, and global topology.
Volume Computation Overview

- Find all vertices of output object.
- For each vertex, find location and local geometry.
- Map-reduce sum over vertices, applying formula.

*Challenge:* to find those vertices in expected constant time per vertex.
Finding the Vertices

3 types of output vertex:

- Input vertex,
- Edge–face intersection,
- Face–face–face intersection.

- Find possible output vertices, and filter.
- An output vertex must not be contained in any input cube.
- Isn’t intersecting all triples of faces, then testing each candidate output vertex against every input cube too slow?
- No, if we do it right.
Parallel Implementation

- 32 thread OpenMP.
- Dual 3.4GHz Xeon, 128GB memory.
- Don’t slice up the input spatially.
- Inserting objects into cells uses atomic increment and capture.
- Very compact data structures.
- Compute element-cell incidences twice: 1st time just to count size of each cell.
- Then allocate ragged array.
- Finally compute again and populate array.
- Intersection testing and volume computation writes only with sum reduce.
- 824 executable lines of C++.
Results

- Fast small datasets: $N=1000$, $L=10$: $V=0.573$, $A=19$, $T=0.02s$.
- Medium: $N=1,000,000$, $L=100$, $G=200$: $V=0.6$, $T=3.3s$.
- Feasible large datasets: $N=100,000,000$, $L=400$, $G=1000$: $V=0.765$, $A=831$, $57GB$, $T=473s$.
- 10x faster with 32 threads than with one thread.
- 258,461,149 of the 1,000,000,000 grid cells were completely covered by some cube.
- Smaller cubes are faster: $N=100,000,000$, $L=500$, $G=1000$: $T=396s$, 50GB.

Output vertex types for big case:
- 186,366,729 of the 800,000,000 input vertices outside all cubes.
- 395,497,686 of the intersections of 3 of the 600,000,000 input faces.
- 811,328,383 of the intersections of one of the 600,000,000 input faces with one of the 1,200,000,000 input edges.
Implementation Validation

It compiles and runs w/o crashing; why look for trouble?

- Terms summed for volume are large and mostly cancel.
- Errors unlikely to total to a number in \([0,1]\).
- Expected volume: \(1 - (1 - L^3)^N\).
- Compare to computed volume.
- Assume that coincidental equivalence is unlikely.
- Construct specific, maybe degenerate, examples with known volume.
Extensions

To general boolean ops:
• Intersection of many convex polyhedra quite easy.
• Any boolean op expressible in CNF as union of intersections (common technique in logic design for computer HW).

To general polyhedra:
• Formulae are messier.
• Roundoff error would be biggest problem.
• Fatal to miss an intersection.
• Compute using rationals.

Testing polyhedron validity: Illegal volume or volume change after rigid transformation → invalid.
Improved Parallel ODETLAP

Work by Dan Benedetti

- Break problem into smaller tasks
- Run on multiple threads concurrently
- Many calculations can be carried out simultaneously
- A parallel implementation should arrive at an answer faster than a sequential implementation
  - This assumes that the problem can be broken down to a sufficiently large number of tasks
  - Overhead associated with splitting and merging threads must be accounted for
CUDA - Parallel Programming and Computing Platform

Use CUDA for parallel programming

- Allows for general purpose parallel computation on the GPU using high-level languages

- Libraries useful to ODETLAP available
  - Thrust - STL compliant template library for vector operations, including sorting, transformations, and reductions
  - CUSP - Sparse linear algebra library, containing tools for the manipulation of sparse matrices and the solving of sparse systems

- Can be used on any system with a CUDA-enabled GPU
CUDA Through MATLAB Implementation

Original CUDA implementation technique

- Construct ODETLAP sparse linear system matrix using MATLAB
- Compile CUSP C++ code containing linear system solver
- Compile using MATLAB executable (MEX) to call from MATLAB
- Requires a large amount of time for data transfer
Direct CUDA Implementation

Minimize data transfer time
- Construct linear system directly on the GPU
- Use Thrust device vectors for vectors and dense matrices
- Use CUSP coordinate matrices for sparse matrices

Solve using CUSP
- Use Generalized Minimum Residual (GMRES) method
- Provides approximate iterative solution
2D ODETLAP performance using MATLAB versus using CUDA

- MATLAB runs on 16 CPU threads at 3.1 GHz (Intel Xeon E5-2687)
- CUDA runs on 2688 GPU cores at 732 MHz (NVIDIA Tesla K20X)

<table>
<thead>
<tr>
<th>dataset</th>
<th>MATLAB</th>
<th>CUDA</th>
</tr>
</thead>
<tbody>
<tr>
<td>400x400</td>
<td>16.4 s</td>
<td>2.3 s</td>
</tr>
<tr>
<td>1024x1024</td>
<td>425.6 s</td>
<td>16.7 s</td>
</tr>
<tr>
<td>2048x2048</td>
<td>1556.8 s</td>
<td>49.8 s</td>
</tr>
</tbody>
</table>

- Greater than 30 times faster for the 2048x2048 dataset
- Smaller datasets benefit less from parallelization
  - Larger impact of overhead
  - Less parallelism due to smaller problem size
Parallel Multiple Observer Siting

Work by Wenli Li.

- Four functions: VIX, FINDMAX, VIEWSHED, and SITE.
- The OpenMP program adds a few compiler directives to the sequential program
  - In VIX, a `#pragma omp parallel for` before the `for` loop that computes the approximate visibility indexes
  - In FINDMAX, a `#pragma omp parallel for` before the `for` loop that finds the most visible points
  - In VIEWSHED, a `#pragma omp parallel for` before the `for` loop that computes the viewsheds
  - In SITE, a `#pragma omp parallel for` before the `for` loop that computes the extra areas
- CUDA uses threads and thread blocks.
- Test data: Puget Sound, 1025x1025 to 16385x16385.
Experimental Results

Speedups of the OpenMP program

Speedups of the CUDA program
Outline

6 Future Modeling of Valid Terrain
   Terrain properties
   Current representations
   Inconsistencies between layers
   Math should match physics
   Terrain formation by scooping
   Terrain formation by features
   Implications of a better rep

7 Summary
Much of the preceding is a list of tools; what’s their purpose?

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.
- $C^\infty$ linear systems, e.g., Fourier series, are wrong.
- Structure that people can recognize even though hard to formalize; see Figure.
- Multiple related layers (elevation, slope, hydrology).

Peninsulas or fjords?
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...
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.

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.
Outline Review

1 Large DB vs New HW
   Why parallel HW?
   Massive Shared Memory
   Geometric Databases

2 HW Tools
   OpenMP
   CUDA
   Thrust

3 SW Tools
   Multiprecision big rationals
   CGAL
   MATLAB
   Arrayfire
   C++ Matrix Packages

4 Algorithm Tactics
   3D Uniform Grid

5 Examples
   Volume of Union of Many Cubes
   Improved Parallel ODETLAP
   Parallel Multiple Observer Siting

6 Future Modeling of Valid Terrain
   Terrain properties
   Current representations
   Inconsistencies between layers
   Math should match physics
   Terrain formation by scooping
   Terrain formation by features
   Implications of a better rep

7 Summary
Summary

Guiding principles — to process big geometric and GIS datasets on parallel machines

• GPUs, memory are affordable.
• Build on powerful existing tools.
• Use minimal possible topology, and compact data structures.
• Use lots of memory; run BIG examples to show the linear time.

Source code (prototype quality) freely available for nonprofit research and education; I welcome stress tests and error reports.