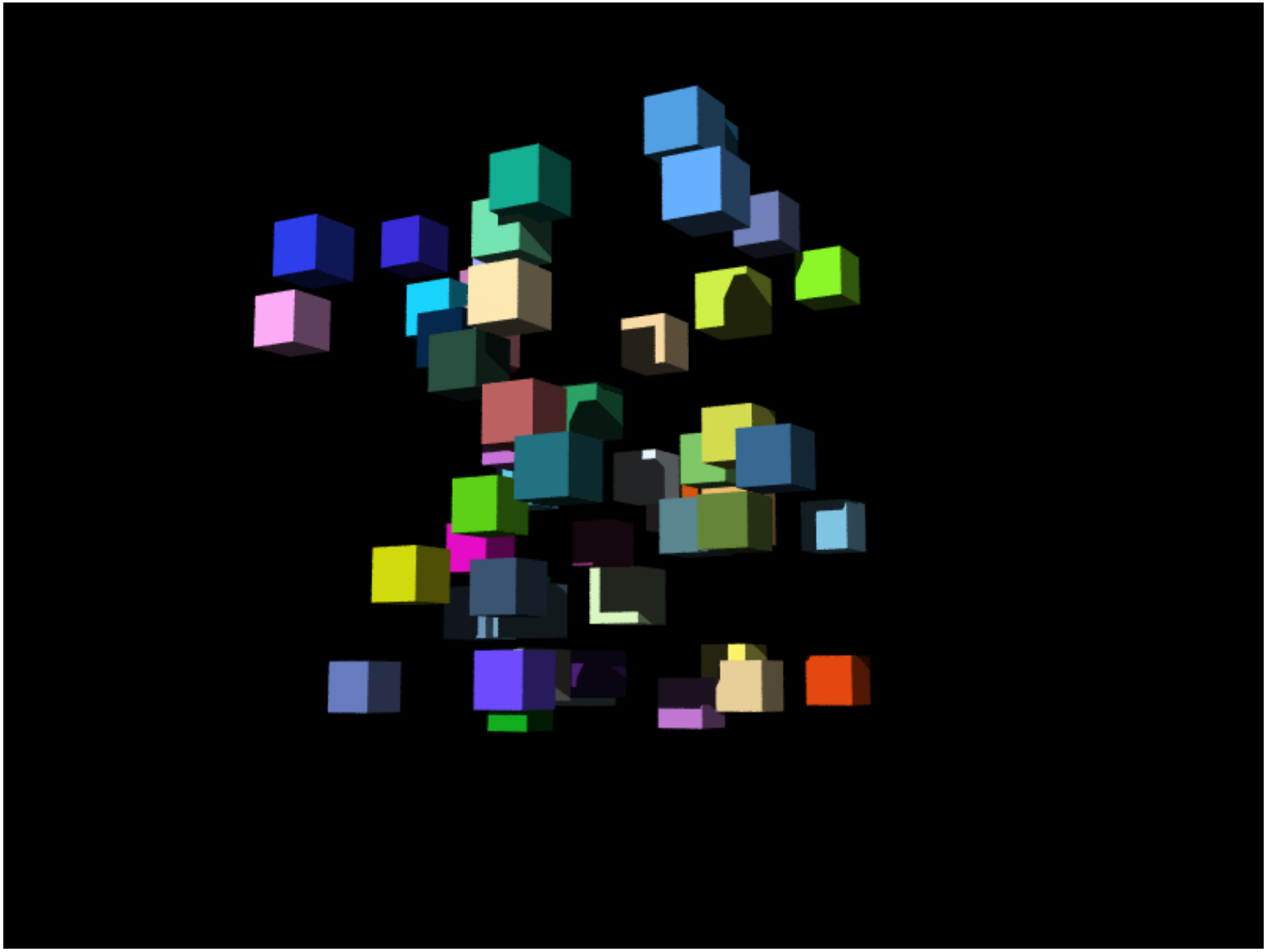
ParCube

W. Randolph Franklin and Salles V. G. de Magalhães,
Rensselaer Polytechnic Institute

2017-11-07

Which pairs intersect?



Abstract

- Parallelization of a 3d application (intersection detection).
- Good (uniform grid, radix sort) vs bad (octree, recursion) data structures.
- The good parallel algorithm is also a good sequential one.
- Functional programming via Thrust is a useful abstraction level.
- Challenge: expressing the algorithm using those primitives.
- Capability of inexpensive HW (neither MPI nor BG nor Spark nor cloud).
- Up to 130× faster than CGAL (Computational Geometry Algorithms Library).

Prior art

- Zomorodian and Edelsbrunner
  - uses segment and range trees to find 1D intersections.
  - 3 1D intersections are necessary (but not sufficient) for 3D intersection.
  - very efficient in practice, though adversarial inputs exist.
  - not parallelizable.
  - used in CGAL.
- PBIG
  - parallelizes with CUDA
  - uniform grid
  - complex CUDA-specific optimizations, compression
  - very fast, parallelizable.
- ParCube (this talk)
  - as fast or faster than PBIG.
  - simpler.
  - higher level abstraction, not restricted to CUDA.

Parallel good; massive better(?)

- Almost all processors, even my smart phone, are parallel.
- Algorithms that don't parallelize are obsolete.
- Nvidia GPUs are almost ubiquitous.
- Thousands of cores execute SIMT in warps of 32 threads.
- Hierarchy of memory: small/fast $\rightarrow$ big/slow
- Communication cost $\gg$ computation cost

Massive: IBM Blue Gene, Hadoop, Spark, cloud.

- Each processor has little memory.
- MPI, expensive communication.
- If you need it, then you need it.

However you can do a lot on one server or one GPU.

Thrust

- C++ template library for CUDA based on STL.
- Functional paradigm: 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, runlength encode/decode.
- Execution cost relative to CUDA: perhaps factor of 3.
- Many possible back ends (just recompile):
  - GPU: CUDA,
  - CPU: OpenMP, TBB, sequential.

Uniform grid

Summary

- Overlay a uniform 3D grid on the universe.
- Find cells overlapping each input primitive.
- In each cell, store set of overlapping primitives.

Properties

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

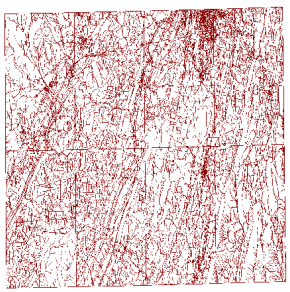
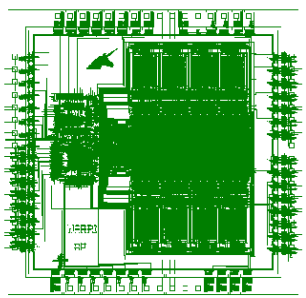
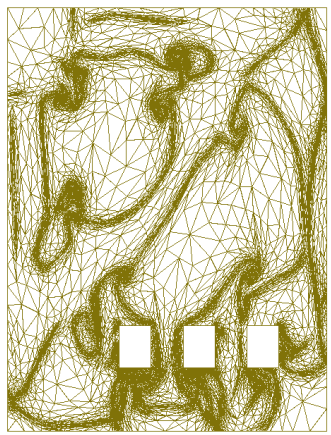
How it works to find intersections

- 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; CFD Mesh

Uniform Grid Time Analysis

For i.i.d. edges (line segments) in E^2, the time to find edge–edge intersections 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 $= \Theta\left(\frac{N^2}{L^2}\right)$.
- Each edge overlaps $\leq 2\frac{G}{L} + 1$ cells.
- $\eta \stackrel{\Delta}{=}$ # edges per cell, is Poisson; $\overline{\eta} = \Theta\left(\frac{N}{G^2}\left(2\frac{G}{L} + 1\right)\right)$.
- Expected total # xsect tests: $G^2\overline{\eta^2} = \Theta\left(\frac{N^2}{G^2}\left(2\frac{G}{L} + 1\right)^2\right)$.
- Total time: insert edges into cells + test for intersections. $T = \Theta\left(N(2\frac{G}{L} + 1) + \frac{N^2}{G^2}(2\frac{G}{L} + 1)^2\right)$.
- Minimized when $G = \Theta(L)$, giving $T = \Theta\left(N + \frac{N^2}{L^2}\right)$.

- Time $= \Theta$ (size of input + size of output). ■

ParCube: Find pairwise cube intersections

- Necessary function in
  - collision detection
  - complex boolean operations
  - near point detection
- 3D is harder than 2D. (Sweep planes?!)
- Using N=10^7 cuts out the toy algorithms,
- Output sensitive algorithm required.
- Easy extension to bipartite (red-blue) intersection detection, which would cause trouble for sweep lines.

ParCube algorithm summary

- I use specific numbers here for clarity.
- Input: 10^7 cubes, length 0.0025.
- Every step parallelizes.
- Overlay a 400x400x400 grid; cells slightly larger than cubes.
- Compute array of (cell,cube) pairs; $8 \cdot 10^7$ pairs.
- Sort to form ragged array of cubes in each cell.
- Compute array of (cube, cube) pairs from all pairs of cubes in each cell.
- Total: 10^8 potentially intersecting pairs.
- Test pairs for actual intersection; find $6 \cdot 10^6$.
- Time from when array of input cubes is in computer to when have list of intersecting pairs.
- On Nvidia GeForce Titan X GPU: 0.33 elapsed seconds.
- 131x faster than CGAL.
- Asymptotic time is output sensitive: linear in output size.

Computing (cell, cube) array

- Determine, parallelly, the cells that each cube overlaps.
- Store all those pairs in one array.
- Could use a global atomic read-increment-store counter pointing to the latest pair in the array.
- That's very slow and doesn't scale well.
- Instead: precompute where each pair will go.
- Then can store the pairs parallelly.
- Given the choice of grid size, each cube overlaps 8 cells (or, rarely, fewer).
- Precomputing each pair's location is easy.
- Pair #j from cube #i is global pair 8i+j.
- Lower-bound function on cube ids computes dope vector.
- Reduce-by-key function computes number of cubes in each cell (which varies from cell to cell).
- Can find j-th cube of i-th cell in constant time.

Computing (cube, cube) array parallelly

- This is harder because different cells have a different number of (cube,cube) pairs that might intersect.
- k cubes in a cell $\rightarrow \binom{k}{2}$ pairs in that cell.
- Order combo pairs: (1,0), (2,0), (2,1), (3,0), (3,1), ...
- Can compute the ids of the two cubes in i-th pair.
- Given a vector with the number of cubes in each cell, map to compute a vector of the number of pairs.
- Scan it to create a dope vector for each cell's list in the global (cube,cube) array.
- Now, for the i-th entry in the global (cube,cube) array:
  - Lower-bound computes cell id and pair id I in that cell.
  - from I compute the ids of the two cubes.
- Write the global (cube,cube) array in parallel.
- Filter it testing whether each pair actually intersects.
- Sort and uniquify it, since some pairs were found twice (in different cells).
- Result is an array of all the intersecting cube pairs. ■

Commentary

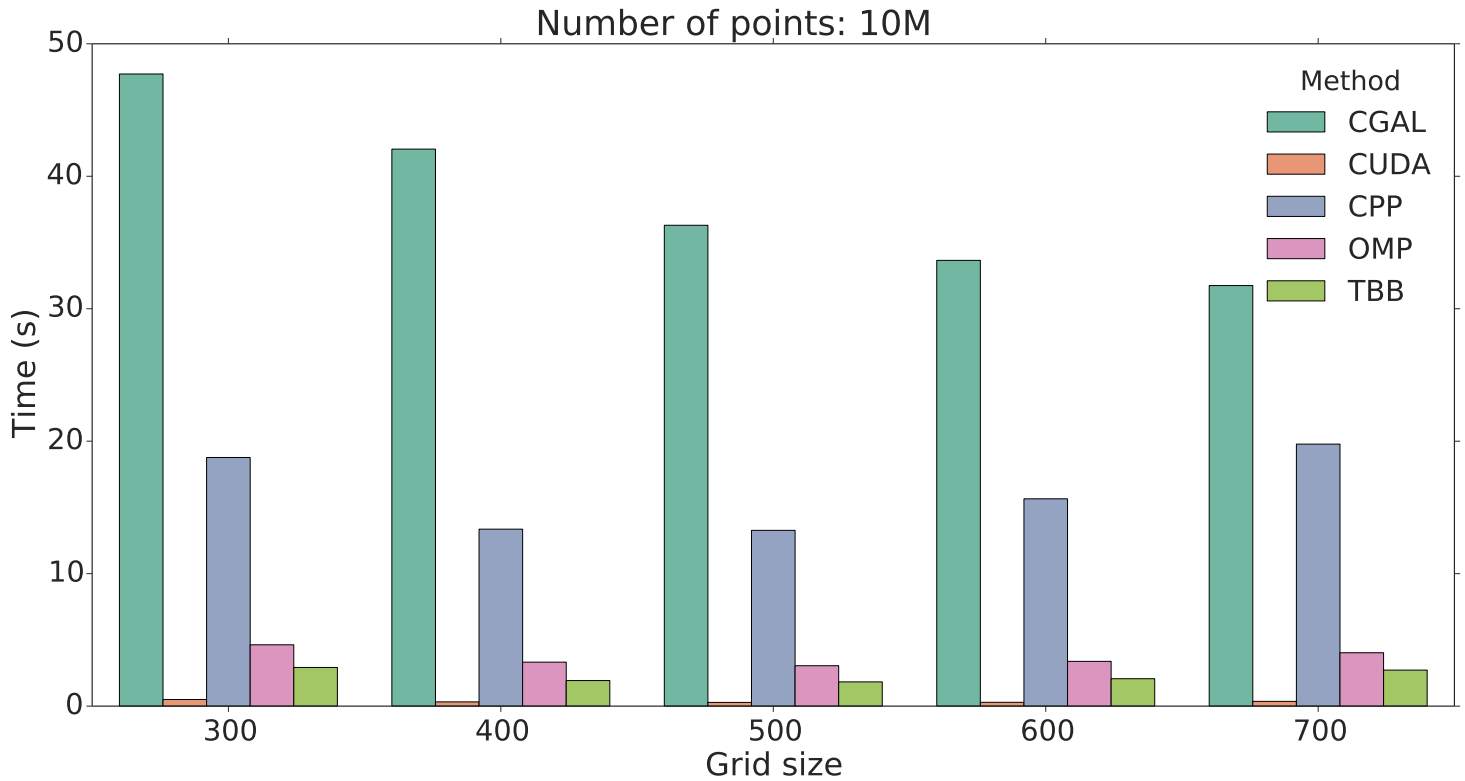
- Possible backends: sequential, OpenMP, TBB, CUDA.
- Hardest part: expressing algorithm within restrictions of Thrust, especially storing (cube, cube) pairs.
- Resulting program:
  - Straight line.
  - $<$ 200 lines of code (plus supporting files).
- Even sequential is sometimes 3x faster than CGAL.
- More sophisticated algorithms are slower.
- Sweep lines not so good in 3D.
- ParCube would extend to higher dimensions.
- ParCube not fully optimized; less abstraction might run 3x faster.

Validation

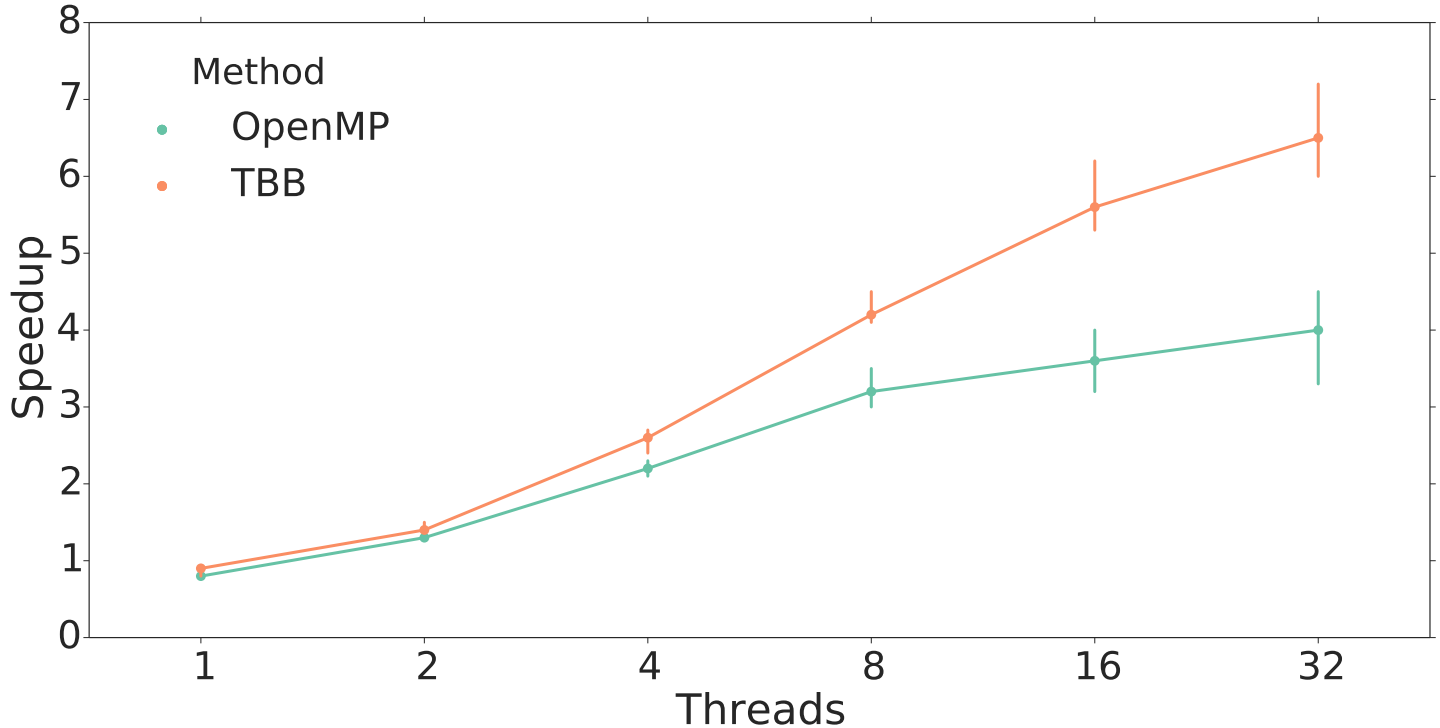
- Separate implementation by different person, using CGAL.
  - Couldn't get PBIG to work, so used its reported times.
- Hardest part was ensuring intersection test did floating roundoff compatibly with CGAL.
  - $(a + b) - b \neq a$
- Compared lists of intersecting pairs for sample parameters.
  - Perfect match.
- All our SW is freely available for nonprofit research and education.
  - It is research-quality not commercial-quality.

Experimental performance comparison

Times for 10^7 cubes with different grid (and cube) sizes, comparing CGAL and ParCube (various backends).



Parallel speedup on dual 8-core multicore Intel Xeon



Smaller datasets are faster

- 100,000 cubes: 0.01 - 0.02 sec (video frame rate)
- 1M cubes: .04 - .1 sec
- 10M cubes: .28 - .5 secs

General lessons, and Future

- You can do a lot on a GPU...
- including finding multiple-object intersections.
- Even a $700^3 = 343 \cdot 10^6$ cell uniform grid indexing 10^7 cubes works.
- Simple regular algorithms work very well and parallelize.
- Should extend to other Geometry and CAD problems.
- Would be applicable to 7D for robot configuration space collisions.
- Now intersecting 3D triangulations with millions of triangles, rational numbers, simulation of simplicity, uniform grid, OpenMP. (talk on Fri).
- Next trying to compute intersecting graded material properties in additive manufacturing.