Computing intersection areas of overlaid 2D meshes

W. Randolph Franklin [1] and Salles Viana Gomes de Magalhães [2]

[1] *ECSE Dept, Rensselaer Polytechnic Institute, Troy NY USA, https://wrf.ecse.rpi.edu/*
[2] *Departamento de Informática, Universidade Federal de Viçosa, MG, Brasil*

Abstract

Parover2 is a parallel algorithm and preliminary implementation to compute the area of every nonempty intersection of any face of one 2D mesh with any face from another mesh over an overlapping domain. This is the hard part of cross-interpolating data from one mesh to another, for when the faces one mesh have an attribute that would be useful for the faces of the other mesh. Parover2, implemented using a map-reduce paradigm in Thrust, can quickly process millions of faces. The expected execution time is linear in the number of intersections, which is usually linear in the number of input faces. A uniform grid quickly determines the pairs of input edges that might intersect. Local topological formulae compute the areas of the output faces from the sets of their (vertex, edge) adjacencies w/o needing to compute the faces' global topologies.

1 Introduction

Suppose that a polygon $\mathcal{P}$ has been meshed, or partitioned, into smaller faces, in two different ways, M_0 and M_1 for two different applications. Each mesh is optimal for some application, and would be suboptimal for the other application. Assume that the faces of M_0 have some property, such as mass, that would be useful for the faces of M_1. (If the density varies, then the mass is not simply the area.) One quick approximation for the mass of f', a face of M_1. is a weighted sum of the masses of the overlapping faces of M_0, with the weights being the areas of the intersections of f' with the overlapping faces of M_0. The compute-bound component is to identify all the nonempty intersections of any face of M_0 with any face of M_1, and compute their areas. This paper presents Parover2, an algorithm and preliminary parallel implementation for that.

Parover2's execution time is linear in the number of intersections, which is generally linear in the number of faces. An initial cull to locate pairs of faces likely to intersect is achieved with a uniform grid. Parover2 uses local topological formulae to compute the areas of the output faces (aka outfaces) from their vertices and half-edges. The algorithm is largely a series of map-reduce steps, implemented with Nvidia's Thrust, with an OpenMP backend.

3D-EPUG-Overlay, Magalhães et al [5, 11, 12, 13, 17] is a somewhat similar idea. Working in 3D, that finds the complete intersection polyhedra, where we find only areas. However it directly uses OpenMP, while we use Thrust, so our possible implementation platform will also include an Nvidia GPU. 3D-EPUG-Overlay also uses rational numbers to completely prevent roundoff errors. Because of the specialized nature of its output, Parover2 is much faster. However 3D-EPUG-Overlay handles geometric degeneracies with Simulation of Simplicity.

Common algorithms for computing intersections include plane sweep lines (in 2D) and quad or octrees [2, 20]. According to Audet [1], "the plane sweep strategy does not parallelize efficiently, rendering it incapable of benefiting from recent trends of multicore CPUs and general-purpose GPUs", and "While effort has been expended to parallelize the plane sweep on CPU, most recently by ([18, 19]), none of the proposed candidates result in an algorithm amendable to fine-grained SIMD parallelism such as with GPUs".

QuickCSG [6] is an efficient parallel intersection algorithm using a kd-tree index. However, it assumes vertices are in general position, and does not handle floating point numerical non-robustness [21], although the user may randomly perturb the input to help.

CGAL [3] operates on Nef polyhedra, which are boolean combinations of half-spaces. LibiGL [15] is an exact and parallel algorithm, Zhou et al.[21], for performing booleans on meshes. LibiGL is also much faster than CGAL for Nef Polyhedra, but is still slower than fast inexact algorithms such as QuickCSG.

The simpler problem of computing the area of the intersection of two polygons is useful in interference detection in robotics. However, the usual solution is either more complicated; it first computes the intersection polygon, and then its area, or it's simpler and tests only whether the intersection is nonempty.

While there appears little archival literature on directly computing intersection areas, some web pages exist. That is, while every CAD package can perform boolean operations on polygons and polyhedra, and then can compute mass properties, optimizing the composition of those two functions is rather rare. Nevertheless, Hardy [14] gives an algorithm with code for the area of the intersection. Polylib [16] operates on objects that are unions of d-D polytopes. Edwards [7] describes how to find the area of the intersection of two polygons with a graphing calculator.

2 Data structures

Parover2 reads the input meshes in one format, and computes the output mesh in a different, simpler, format. Each format is sufficient for the necessary computation. The input mesh format is designed to compute the outface vertices. The outface format is designed compute the outface areas.

The input mesh format is the set of its edges, $\{(x_0, y_0, x_1, y_1, f_l, f_r)\}$. $(x_0, y_0), (x_1, y_1)$ are the coordinates of one edge's end vertices. f_l and f_r are the identifiers (ids) of its adjacent faces to the left and the right. There are no lists, and no explicit vertices, explicit faces, or higher level topology such as the edges around a vertex, the edges around a face, nested faces, etc. That is, although several edges may contain the same vertex, we do not record this. We do not record in one place all the vertices contained in one face, etc. This info could be derived if we needed it, which we don't. This is simpler than quad edges and doubly connected edge lists, because it permits fewer operations (but it permits all the operations that we need).

The outface format is even simpler; it does not even store complete edges. It is a set of edge-vertex incidences, as de-

scribed later. However it permits the local topological formulae described next to compute the outface areas.

3 Local topological formulae

This section will motivate the power of local topological formulae, by presenting an algorithm for computing the area of a face from either the set of its vertex positions and their neighborhoods, or the set of its vertex-edge incidences. The neighborhood of a vertex is defined as the direction vectors of its adjacent edges, and which one of the two sectors that they define is inside the face. This algorithm does not use the lengths of the adjacent edges, nor the vertices at their other ends. It resembles Green's theorem, except that, while Green's theorem computes a polygon's area by integrating along its boundary edges, we use only the boundary vertices.

The general formula is given in [8]. For a simple example, consider a diagonal rectilinear face f, as shown in Figure 1. All edges have slope $+1$ or -1. The face may have multiple components and nested islands and holes. Each vertex will be assigned a sign bit, s, determined by its neighborhood. Then the area of this isothetic diagonal f is $\sum_i s_i x_i^2$. For Figure 1, the area would be $0 - 1 + 4 - 9 + 16 - 4 = 6$. As this formula is a map-reduction, it may be efficiently computed in parallel.

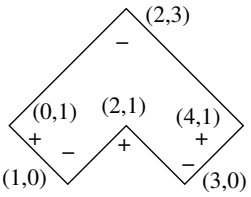

Figure 1: Diagonal rectilinear polygon

The proof is by induction. It is clearly true if f is a rectangle. If two rectangles with a common edge are united and those common edges (and four vertices) removed, then the area formula remains valid. In this way, any face may be built up. Any mass property, such as a higher order moment, may be likewise computed.

There are many formulae using different input formats. Another face area formula map-reduces the set of oriented edges, summing the signed areas subtended by the edges and the coordinate origin.

We compute the outface areas with a formula using the set of *(vertex, edge)* adjacencies. Each vertex neighborhood comprises two adjacencies. For vertex v with adjacent edges e_1, e_2, they would be $(v, \widehat{e_1})$ and $(v, \widehat{e_2})$. The normalization is because we know the direction of the edge but not its length. One adjacency is represented as the triple $(v, \widehat{t}, \widehat{n})$. v is the position of the vertex. $\widehat{t}$ is a unit direction vector along the edge. $\widehat{n}$ is a unit direction vector perpendicular to $\widehat{t}$ pointing to the inside side of the edge. $\widehat{n}$ adds only one bit of information.

The area of the general f is $\left(\sum \left(v \cdot \widehat{t}\right)\left(v \cdot \widehat{n}\right)\right)/2$. This is proved by dropping a perpendicular from the origin to each edge of f, and partitioning f into $2n$ right triangles. The vertices of one triangle will be the origin, a perpendicular foot, and one of the two end vertices of that foot's edge. Then, the signed area of one such triangle is $\left((v \cdot \widehat{t})(v \cdot \widehat{n})\right)/2$.

The above data structure is simpler, smaller, and faster than the well known half-edge data structure of the doubly connected edge list, since we do not use both ends of an edge together. It is much simpler to compute outfaces in this format than to also compute their edges.

4 Outface area computation strategy

Each outface is the intersection of two input faces (aka infaces), one from each input mesh. An outface is identified by the ordered pair of those two infaces. PAROVER2 computes the outface areas by a map-reduction over the vertex-edge adjacencies. As it processes the input, it does not compute each outface all at once. Rather, it will compute an output vertex with all its vertex-edge adjacencies, then another output vertex with all its vertex-edge adjacencies, and so on. So, it accumulates the outface areas incrementally. It never computes the output edges. (If necessary, the adjacencies could be paired up to produce the output edges.)

There are two types of output vertex-edge adjacencies: an adjacency of one of the input meshes M_i, and an intersection of an edge of M_0 with an edge of M_1.

4.1 Output adjacencies that are input adjacencies

1. Call the input adjacency, h.
2. Without loss of generality, assume that h is in mesh M_0.
3. h is adjacent to two infaces, call them f_l and f_r.
4. Let the vertex of h be v. v is contained in some face, say f', of the other mesh, M_1.
5. h is part of the two outfaces (f_l, f') and (f_r, f').
6. The normal vector component of h may need to be negated depending on which outface we are considering.
7. PAROVER2 will compute an area component for the two outfaces. Each component is of the form *(outface-id, area-component)*.
8. The total area of each outface is obtained by summing its components, as described later.

There are three nontrivial parts here: the tedious process of getting the several different special cases correct, storing the area components, and point location. Storing the area components is complicated. We are computing and storing the components in parallel. We do not know in advance the ids of the nonempty outfaces. We do not know in advance how many components each outface will have (aka how many vertices it has). PAROVER2 uses a vector of the *(outface-id, area-component)* pairs as follows.

1. The size of the vector is four times the number of input edges in the two meshes combined.
2. The i-th input edge will create output pairs numbered $4i$ to $4i + 3$. So, the output pairs can be written in parallel w/o needing semaphores or locks. Specifically, we define a function that maps from index i to output pair i, and then map that function over the index vector $0, 1, 2, ...$
3. Finally we sort the vector by outface-id and perform a parallel reduce-by-key, which sums the area components with the same outface-id.
4. The slowest step is the sort. However, every alternative, such a hash table keyed by the outface-id, or linked lists, appears worse.

For locating which face of M_1 contains v, we use a uniform grid, aka bucket sort. The expected query time is constant per point location, independent of the map size. The preprocessing algorithm goes as follows:

1. Determine the maximum linear size of any face in either direction (x or y).
2. Superimpose a grid of $g \times g$ cells over the meshes. g is chosen so that a cell is slightly taller and wider than the largest face.

3. For each input edge e, compute a superset of size 4 of the grid cells that intersect e. Because of the choice of g, the actual number of cells intersecting e ranges from 1 to 3. For programming ease, we circumscribe e with a box and use all the cells intersecting that box. This avoids exactly computing—in parallel—the intersecting cells, which would require identifying, in constant time, the i-th intersecting cell.
4. Working in parallel over the edges, form a vector of these pairs.
5. Sort it by cell id.
6. Use a parallel scan function to find the start of each cell's edges in the sorted vector, and the number of edges in each cell.

The total preprocessing execution time is linear in the input size because (a) for sorting by cell ids, a linear-time radix sort is applicable, and (b) the other steps are linear in the number of pairs (and fast).

The query algorithm to locate which face mesh M_1 contains point q from mesh M_0 is this:

1. Compute which grid cell contains q. (This is simply two modulo operations on q's coordinates.)
2. Process the edges of M_1 that are in the same cell as q as follows, in parallel over the edges, as follows. (a) Run a vertical line up and down from q through the cell. (b) Skip edges that are not intersected. (c) For each edge that is intersected, compute the vertical distance from q to the edge. (d) Return the absolutely closest edge.
3. If no edge was vertically above or below q, then the containing face is the external face. This is because the grid size is large enough that it is impossible to a face to extend beyond a cell in both directions.
4. Otherwise the containing face is one of the two faces adjacent to the closest edge. Which one is determined by whether the edge is above or below q and whether its first vertex is to the left or right of its second vertex.

The execution time to query q is linear in the number of edges in its grid cell. Since the cell size is slightly larger than the largest face, that depends on the ratio between the average face size and the largest face size, which is a constant < 4 for many common distributions of face sizes, including uniform and Gaussian. So, the expected query time per point is constant.

4.2 Output adjacencies that are the intersections of two input edges

This case differs from the previous case in two ways. First, we must compute the intersections of edges from M_0 with edges from M_1. Each intersection is a vertex of four outfaces and generates two adjacencies per face, for a total of eight outface adjacencies per intersection. Second, there is no need for point location because here we know the outface ids.

Note that two edges that intersect each other must both intersect the same grid cell, but not all pairs of edges in the same cell will intersect. The edge pairs in a cell can be indexed so that the i-th pair can be determined in constant time. If the edges are independently and identically distributed, then the probability of a pair of edges in the same cell intersecting each other is constant, independent of the total number of edges.

The algorithm goes as follows.

1. Compute the maximum possible number of edge intersections per cell and allocate space for a vector of intersections from all cells.
2. Populate that vector with the pairs of possibly intersecting edges. Note that this parallelizes because of the above observations. I.e., we can determine in constant time the i-th element of that vector.
3. Filter that vector by whether or not the edges do intersect.
4. Map the resulting vector into a vector of octuples of outface adjacencies.
5. Sort, reduce by key, and sum.

This parallelizes well. The expected execution time is linear in the number of input edges. An adversary could generate bad input cases, but other data structures such as quadtrees can also be made to perform poorly, and they don't parallelize well. We do not believe that real data would have this problem.

5 Implications of the target platform

The design choices in Parover2 are motivated by the goal of eventual execution on an Nvidia GPU. (Nvidia was chosen because it is currently by far the most common and most cost-effective GPU.) A GPU executes thousands of threads in parallel, with the threads grouped by 32 into warps. All 32 threads in one warp simultaneously execute the same instruction, ideally on data from successive words in memory. The only exception is that some threads in the warp may be idle.

Efficient algorithms for a GPU prefer data structures that are arrays of plain old data types, not even arrays of structures. Conditionals hurt performance. Pointers are strongly to be deprecated. Even randomly accessing elements is not ideal.

Therefore complex and adaptive data structures such as sweep lines and trees are very difficult to parallelize. This is especially true when using hundreds or thousands of threads. All this gets much worse in 3D. Hence our preference for simple flat data structures like uniform grids.

Simple flat data structures have another advantage; they are more compact; they take less space. In many parallel programs, the I/O time to read and write the data dominates the computation time. Although hosts and devices have a lot of memory as described above, it is slow, but they have small fast caches and register banks.

6 Implementation

Parover2 is implemented in C++ on a dual 14-core 2.0GHz Intel Xeon with 256GB of memory, running Ubuntu Linux. The parallel environment is Nvidia's Thrust, which is a set of parallel C++ classes and routines that map and reduce vectors. It is a parallel extension of parts of the Standard Template Library and Boost[4]. Thrust is reasonably mature, and seems to represent a good medium being a high level abstraction and being efficient. Thrust has several possible backends, including OpenMP and CUDA. Parover2 currently uses OpenMP, with CUDA support being debugged.

Our initial test cases are pairs of overlapping square meshes. Times are shown in Table 1. The test times start after the data has been read. The additional time to read the data, from ASCII files stored in real memory, is about 80% of the processing time. If the identical job is rerun, the time may vary by 10%. The parallel speedup on this system with 28 threads and 56 hyperthreads was a factor of 6.3. One limiting factor is that the processors automatically overclock from 2.0GHz to 3.2GHz when lightly loaded, but run more slowly when executing many parallel threads. This is typical of multicore CPUs, and serves

No. input edges	No. output faces	Elapsed time (sec)
220	400	.023
3,720	3,600	.032
40,400	40,000	.082
361,200	360,000	.47
4,004,000	4,000,000	6.2

Table 1: Elapsed time to compute intersection areas of two meshes

to limit the heat. The CPU time increased massively with the parallelism; for 56 threads it was 198 CPU seconds. This is irrelevant because the usual metric for parallel programs is elapsed wall-clock time.

7 Extension to 3D

The goal of this project is to compute the volumes of all the intersecting regions from two overlaid meshes in 3D. The mesh polyhedra may be tetrahedra, hexahedra, or other polyhedra. The conceptual extension is small; the difficulties mostly practical. Indeed, we expect that the tools that make Parover2 work, i.e., local topogical formulae and uniform grid, will be even more valuable in 3D, in Parover3.

Each input mesh for Parover3 is a set of faces, each tagged with the ids of its adjacent polyhedra. The output is the set of the pairs of input polyhedra that have a nonempty intersection, together with that intersection's volume.

Our volume formula for a polyhedron is a map-reduce over the set of 3D adjacencies. A 3D adjacency is a 4-tuple $(v, \widehat{t}, \widehat{n}, \widehat{b})$, where v is the position of a vertex, $\widehat{t}$ is a unit vector tangent to an adjacent edge, $\widehat{n}$ is a unit vector normal to $\widehat{t}$ and in the plane of a face adjacent to that vertex and edge, and $\widehat{b}$ is a unit binormal vector, normal to both $\widehat{t}$ and $\widehat{n}$ and pointing towards the interior of that face.

If v has k adjacent edges, then it will have $2k$ adjacencies. The volume of the polyhedron is $\left(\sum((v\cdot\widehat{t})(v\cdot\widehat{n})(v\cdot\widehat{b})\right)/6$. The unnecessity of any global topological info, even complete edges or faces, makes this formula easier to apply to the intersection of two 3D meshes.

Union3 [9, 10] is another parallel algorithm and implementation that demonstrates the validity and efficiency of these ideas. Union3 computes the volume of the union of millions of congruent isothetic cubes. When the cubes' positions are i.i.d., the expected execution time is linear in the number of cubes, even if the number of face-edge and face-face-face intersections grows superlinearly. The reason is that only those intersections that are not inside any input cube become output vertices. That number grows only linearly. When a cell is completely inside a cube, the possibly superlinear number of intersections inside it are never computed. A more detailed theoretical analysis, with implemention details and test results is given in the references.

References

[1] Samuel Audet, Cecilia Albertsson, Masana Murase, and Akihiro Asahara. 2013. Robust and Efficient Polygon Overlay on Parallel Stream Processors. In *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (SIGSPATIAL'13)*. ACM, New York, NY, USA, 304–313. https://doi.org/10.1145/2525314.2525352

[2] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. 2008. *Computational Geometry: Algorithms and Applications* (3rd ed.). Springer-Verlag TELOS, Santa Clara, CA, USA.

[3] CGAL. 2018. Computational Geometry Algorithms Library. (2018). Retrieved 2018-09-09 from https://www.cgal.org

[4] Beman Dawes, David Abrahams, and Rene Rivera. 2010. Boost C++ libraries. (2010). Retrieved 2018-09-09 from http://www.boost.org/

[5] Salles Viana Gomes de Magalhães. 2017. *Exact and parallel intersection of 3D triangular meshes*. Ph.D. Dissertation. Rensselaer Polytechnic Institute, Troy, NY, USA.

[6] Matthijs Douze, Jean-Sébastien Franco, and Bruno Raffin. 2015. *QuickCSG: Arbitrary and faster boolean combinations of n solids*. Ph.D. Dissertation. Inria-Research Centre, Grenoble–Rhône-Alpes, France.

[7] C. C. Edwards. 2018. How to Find the Area of the Intersection of Two Polygons. (2018). Retrieved 2019-03-18 from https://www.dummies.com/education/graphing-calculators/how-to-find-the-area-of-the-intersection-of-two-polygons/

[8] Wm Randolph Franklin. 1987. Polygon properties calculated from the vertex neighborhoods. In *Proc. 3rd Annu. ACM Sympos. Comput. Geom.* 110–118.

[9] W. Randolph Franklin. 2004. Analysis of Mass Properties of the Union of Millions of Polyhedra. In *Geometric Modeling and Computing: Seattle 2003*, M. L. Lucian and M. Neamtu (Eds.). Nashboro Press, Brentwood TN, 189–202.

[10] Wm. Randolph Franklin. 2005. Mass Properties of the Union of Millions of Identical Cubes. In *Geometric and Algorithmic Aspects of Computer Aided Design and Manufacturing, DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, Ravi Janardan, Debashish Dutta, and Michiel Smid (Eds.). Vol. 67. American Mathematical Society, 329–345.

[11] W. Randolph Franklin, Salles V. G. Magalhães, and Marcus V. A. Andrade. 2017. 3D-EPUG-Overlay: Intersecting very large 3D triangulations in parallel. In *2017 SIAM conference on industrial and applied geometry*. Pittsburgh PA USA. (talk).

[12] W. Randolph Franklin, Salles V. G. Magalhães, and Marcus V. A. Andrade. 2017. An exact and efficient 3D mesh intersection algorithm using only orientation predicates. In *S3PM-2017: International Convention on Shape, Solid, Structure, & Physical Modeling, Shape Modeling International (SMI-2017) Symposium*. Berkeley, California, USA. (poster).

[13] W. Randolph Franklin, Salles V. G. Magalhães, and Marcus V. A. Andrade. 2018. Exact fast parallel intersection of large 3-D triangular meshes. In *27th International Meshing Roundtable*. Alberqueque, New Mexico.

[14] Normal Hardy. 2018. Area of Intersection of Polygons. (2018). Retrieved 2019-03-18 from http://www.cap-lore.com/MathPhys/IP/

[15] Alec Jacobson, Daniele Panozzo, et al. 2016. *libigl: A Simple C++ Geometry Processing Library*. Retrieved 2017-10-18 from http://libigl.github.io/libigl/

[16] Vincent Loechner. 2010. PolyLib - A library of polyhedral functions. (2010). Retrieved 2019-03-18 from http://icps.u-strasbg.fr/PolyLib/

[17] Salles V. G. Magalhães, Marcus V. A. Andrade, W. Randolph Franklin, Wenli Li, and Maurício Gouvêa Gruppi. 2016. Exact intersection of 3D geometric models. In *Geoinfo 2016, XVII Brazilian Symposium on GeoInformatics*. Campos do Jordão, SP, Brazil.

[18] Mark McKenney, Roger Frye, Mathew Dellamano, Kevin Anderson, and Jeremy Harris. 2016. Multi-core parallelism for plane sweep algorithms as a foundation for GIS operations. *GeoInformatica* (2016), 1–24. https://doi.org/10.1007/s10707-016-0277-7

[19] Mark McKenney and Tynan McGuire. 2009. A Parallel Plane Sweep Algorithm for Multi-core Systems. In *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (GIS '09)*. ACM, New York, NY, USA, 392–395. https://doi.org/10.1145/1653771.1653827

[20] Robert Sedgewick. 1988. *Algorithms (2nd Ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[21] Qingnan Zhou, Eitan Grinspun, Denis Zorin, and Alec Jacobson. 2016. Mesh Arrangements for Solid Geometry. *ACM Trans. Graph.* 35, 4, Article 39 (July 2016), 15 pages.