Parallel Volume Computation of the Union of Many Cubes

W. Randolph Franklin
ECSE Dept., 6026 JEC
Rensselaer Polytechnic Institute
110 8th St.
Troy, NY, USA
wrf@ecse.rpi.edu

ABSTRACT

We present an algorithm and implementation for computing volumes and areas of the union of many congruent axis-aligned cubes. Its expected execution time is linear. It has been tested to 100M cubes. The ideas extend to any mass property of the union of any polyhedra, and to online computations as more inputs are added. The algorithm is mostly a series of map-reduce operations and so parallelizes quite well. Inserting a new cube and recomputing takes constant expected time.

The algorithm combines local topological formulae with a uniform grid. It does not build a computation tree of height log(N), but rather computes all the possibly relevant intersections in one step. It is an exact computation, not a sampling or cellular decomposition technique. Most of the operations are a map-reduce, and so they parallelize quite well, better than more complicated data structures.

Categories and Subject Descriptors

I.3.5 [**Computational Geometry and Object Modeling**]: Geometric algorithms, languages, and systems

General Terms

Algorithms, Experimentation, Performance

Keywords

polyhedron, volume, boolean operation, mass property, parallel

1. INTRODUCTION

Sometimes a polyhedron $\mathcal{P}$ is created by boolean operations on a large set of primitive objects for the sole purpose of computing some mass property, such as its volume. For example, we may be interested in the volume of material removed by a cutting tool. Each segment of the tool's path, intersected with the object being cut, induces its swept polyhedron. The total polyhedron swept out is the union of those separate polyhedra; we are interested in its volume. Interpreted slightly differently, we might be interested in whether the volume is nonzero, since that is an indicator of whether the tool intersected the object at all.

A third application of mass properties of boolean combinations of polyhedra arises from interpolating some quantity from one partition of E^3 to another, as follows. Consider two overlapping triangulations T and U of the same region of E^3. Each tetrahedron t of T has some property such as mass. (Assume that the underlying medium is compressible, so that the mass is not simply the volume.) We need to know the masses of the tetrahedra u of U. A reasonable interpolation strategy to find the mass of u proceeds as follows.

1. Find t_i the tetrahedra of T with nonempty intersections with u.
2. Let m_i be the mass of t_i.
3. Find v_i the volume of each such intersection.
4. Note that $\sum v_i = vol(u)$.
5. Approximate mass of u as follows

$$\frac{\sum_i v_i m_i}{\sum_i v_i} \tag{1}$$

This is called the *cross area* problem in E^2 and is discussed in more detail later. One application is computing estimated populations of watershed polygons from the known populations of census districts. While superficially this may appear to be quite a different problem, at its core is quite similar. Indeed we have used the same techniques to implement this algorithm in E^2.

2. PRIOR ART

There appear to be few results for object combination in E^3 except for the following special cases: efficiently intersecting convex polyhedra [8, 7, 30, 33], intersecting a convex polyhedron with a general one [11, 27], fast detection of polyhedral intersection [10], and uniting convex polyhedra [2]. Constructive Solid Geometry (CSG) is well documented, [5, 12, 23], though its efficient methods tend to use bounding boxes. Its algorithms for boolean combinations are described in [24, 25]. As with the current paper, in CSG, objects are formed by boolean operations on overlapping primitives.

[37] presents a fast algorithm for intersecting boxes in 3D. Various heuristics are combined to optimize the implementation, for which tests on 500,000 boxes were reported. Although the general optimization techniques are very useful, extending this particular algorithm to compute properties of a union is not at all obvious since the characteristics of intersections and unions of boxes are quite different. E.g., intersections are convex, while unions are often not even connected. [13] computes any Boolean combo of two *local* polyhedra in $O(N \log N)$ time. However since the faces of a union are often not simplices, our polyhedra are not local. [36] gives a new algorithm to report intersecting pairs among n objects in a fixed dimension d. Again, this is considerably easier than computing unions.

Another topological representation for polyhedra is presented in [4]. However, it appears more concerned with global topology and ordering information. That strongly contrasts to our approach, which eschews that level of complexity in favor of sets. A nice collection of mass formulae for polyhedra is presented in [28], which derives them with surface and volume integrals. Again, this uses global properties of the polyhedra. Detecting whether a boolean combination is null, which is a special case of volume computation, is presented in [34].

In 2-D, the well-known plane sweep methods for line segment intersection take from $T = \theta(k + n \lg n)$ for a complex topological sweep up to $T = \theta((n + k) \lg n)$ for a simple plane sweep. n is the number of input line segments, and k the number of output intersections. For the large values of n in this paper, the $(\lg n)$ is significant. Of course, these methods' worst-case time is much better than ours, since an adversary can hurt the performance of our "input-sensitive" method. Also our computation model is more powerful and more realistic than that used by the plane sweep methods; we can compute *modulo* in constant time.

The prior art on mass properties of boolean combinations of polyhedra is similarly sparse, [3, 29]. Some of these local topological formulae could well date to the discovery of analytic geometry, [9], but actual citations earlier than [15, 31, 19] are elusive. Large geometric and cartographic data sets are now being processed, perhaps with external algorithms, [35, 6].

This paper presents UNION3, an algorithm for computing the volume and other mass properties of the union of many polyhedra, without combining the inputs pair by pair in a tree of height $\lg n$, in expected linear time.

UNION3 does not have the following properties. It is not a Monte Carlo or other statistical sampling technique. The output is exact to the arithmetic precision of the computations. UNION3 is also not a voxelization or octree method, where the universe is partitioned into voxels, and each object is represented as the union of a set of voxels, and everything is only as accurate as the voxel size. Although superficially similar, our uniform grid is quite different. Different grid sizes affect only the execution time and space, but have no effect on the result. Input cubes generally overlap grid cells only partially.

This paper updates and extends results that were earlier presented in [17, 16], which should be consulted for more details of the basic algorithm, implementation, and time analysis. Those papers also present more evidence for the general applicability of the uniform grid used later in this paper.

3. CURRENT ALGORITHM

The usual algorithm for computing the volume of the boolean combination of many polyhedra proceeds by computing $\mathcal{P}$ by a series of boolean operations on pairs of smaller polyhedra, then computing the volume of $\mathcal{P}$. If the number of input polyhedra is n, then $n - 1$ boolean operations are required, in a computation tree of depth $\geq \lg n$. There may be considerable intermediate swell; the intermediate polyhedra may have many more vertices than $\mathcal{P}$. Even though the resulting volume is merely a scalar, the complete topology of $\mathcal{P}$ and all the intermediate polyhedra is determined. Finally the complexity of the data structures makes this difficult to parallelize. This paper presents optimizations that reduce those problems.

4. MINIMAL INFORMATION TO COMPUTE VOLUME

Computing $m(\mathcal{P})$, the volume M of polyhedron $\mathcal{P}$, does not require complete explicit knowledge of $\mathcal{P}$. For example, $F = \{f\}$, the set of faces, is sufficient. Indeed each f, together with $\mathcal{O}$, the coordinate origin, induces a tetrahedron t, and

$$m(\mathcal{P}) = \sum_i m(t_i) \tag{2}$$

Note the considerable advantages of Equation 2.

1. No global topology is required. There is no need to correctly process shells of faces. Therefore the code is much simpler, with less possibility of error.
2. Non-manifold conditions, such as two shells of faces coincident at a vertex are not a problem.
3. The simpler data structure requires less memory, which requires less I/O, and which causes fewer virtual memory and cache faults, causing a compounding increase in execution speed.
4. If the algorithm computing $\mathcal{P}$ could be simplified by computing only the faces rather than the complete topology, then we could utilize that. Later in this paper, we will.
5. When executed in a parallel environment Equation 2 is a *reduction* operation; a scalar computed independently in each process or thread is summed to give the result. Widely used parallel tools such as *OpenMP* and *MPI* implement reduction particularly efficiently; see Section 11 below.

Other compact representations are possible. The set of vertices does not work, because that does not uniquely define the polyhedron. However, the set of vertices, augmented with some local information, does work. For instance, for a vertex v of a polygon, let the angles of the two edges be α and β. Then the polygon's area is $\mathcal{A}$ where

$$A_v = \begin{cases} \frac{1}{2}x^2(\tan\alpha - \tan\beta) & \text{if } (0,\infty) \text{ is on the inside side of v} \\ -\frac{1}{2}x^2(\tan\alpha - \tan\beta) & \text{otherwise} \end{cases}$$

$$\mathcal{A} = \sum A_v$$

The characteristic function of a general polyhedron, which is 1 for a point p iff p is contained in the polyhedron, may be determined as follows. For trihedral v in E^3, $\chi_v(p)$ is defined using the vertex angles of the adjacent faces, f_1, f_2, and f_3, with angle at v α, β, and γ.

$$\chi_v(p) = \begin{cases} \frac{2\pi-\alpha-\beta-\gamma}{4\pi} & \text{if } p \text{ is inside all 3 face planes.} \\ \frac{\alpha-\beta-\gamma}{4\pi} & \text{if } p \text{ is outside } f_1 \text{ but inside } f_2 \text{ and } f_3. \\ & \text{etc.} \\ \frac{\alpha+\beta-\gamma}{4\pi} & \text{if } p \text{ is outside } f_1 \text{ and } f_2 \text{ but inside } f_3. \\ & \text{etc.} \\ \frac{\alpha+\beta+\gamma-2\pi}{4\pi} & \text{if } p \text{ is outside all 3 face planes.} \end{cases}$$

$$\chi = \sum \chi_v(p)$$

Finally, χ may be integrated to give the polyhedron's volume.

Here are some formulae for axis-aligned polyhedra. For instance, the volume of a cube with vertices (x, y, z) may be expressed as

$$\mathcal{M} = \sum_i s_i x_i y_i z_i, \text{ where } s_i = \pm 1 \tag{3}$$

For any particular vertex, $s = +1$ iff an odd number of the three faces adjacent to that vertex are on the high side of the cube. The volume of any axis-aligned polyhedron $\mathcal{P}$ is still $\mathcal{M} = \sum_i s_i x_i y_i z_i$, if the definition of s is extended to every local vertex geometry. $\mathcal{P}$ may contain non-manifold vertices, and multiple nested components with holes. Our implementation uses this formula. The above formula easily extends to other mass properties, such as mass moments of any order and to point inclusion testing in $\mathcal{P}$.

Likewise, lower dimensional mass properties, such as face area and edge length may be computed by reduction operations over a function of each vertex and its local geometry. Throughout this paper "volume" includes any mass property.

Axis-aligned polygons rotated by $\pi/4$ so that their edges's slopes are ± 1 have a 4-parameter family of area formulae:

$\forall\alpha\forall\beta\forall\gamma\forall\delta$

$$\mathcal{A} = \sum_i s_i(\alpha {x_i}^2 - (1-\alpha){y_i}^2 + \beta x_i + \gamma y_i + \delta)$$

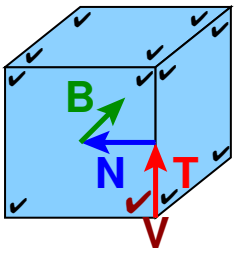

Figure 1: Neighborhood of a polyhedron vertex

An extension to general polyhedra that does not even use each vertex's complete neighborhood is described in [32, 15]. Suppose that we have only the set of incidences of output vertices, edges lines, and face planes, together with which side is inside. For instance, for a cube, each vertex would induce six such incidences; see Figure 1. Then if V is the vertex position, $\widehat{T}$ is a unit vector along the edge incident on it, $\widehat{N}$ is a unit vector normal to $\widehat{T}$ in the plane of the face, and $\widehat{B}$ is a unit vector normal to both $\widehat{T}$ and $\widehat{N}$ pointing into the polyhedron, then the volume is

$$\mathcal{M} = -\frac{1}{6}\sum (V\cdot\widehat{T})\,(V\cdot\widehat{N})\,(V\cdot\widehat{B}) \tag{4}$$

Similar formulae obtain for other mass properties, such as the surface area and edge length:

$$\mathcal{A} = \frac{1}{2}\sum (V\cdot\widehat{T})\,(V\cdot\widehat{N}) \tag{5}$$

$$\mathcal{L} = -\frac{1}{2}\sum (V\cdot\widehat{T}) \tag{6}$$

[29] also describes efficient polyhedron formulae.

5. FINDING THE VERTICES AND NEIGHBORHOODS

Computing $M(\mathcal{P})$ requires determining the set of vertices for $\mathcal{P}$, and the neighborhood of each vertex, viz., the directions of each face and edge adjacent on that vertex. The direction of an edge is its semi-infinite ray. We do not explicitly know its length or the other endpoint, although later we will discuss linking the rays up in pairs to find the edges explicitly. The direction of a face is the face plane, the two edge rays, and which of the two possible sectors that they form is the face interior.

The vertices of $\mathcal{P}$ fall into three classes:

1. Some of the input vertices of the input polyhedra,
2. Some of the intersections of edges and faces from the input polyhedra, and
3. Some of the intersections of three faces from the input polyhedra.

Call the possible output vertices *candidate vertices*. Each candidate vertex must pass a test to become an actual output vertex. For a candidate vertex to be an actual output vertex, it must be adjacent to both the interior and to the

exterior of $\mathcal{P}$. (The boundary of $\mathcal{P}$ at the vertex must also be nonplanar, but that is not a problem.) The actual test depends on the boolean operations and on its one to three input polyhedra.

When $\mathcal{P}$ is the union of the input polyhedra, the test is particularly easy; simply that the candidate vertex is not in the interior of any other input polyhedron. *Other* is included to avoid having to define (and then compute correctly) the meaning of inclusion of one of the vertices of a polyhedron inside itself. This implies that we are testing the candidate vertices for inclusion against different polyhedra, not against $\mathcal{P}$.

Once a candidate vertex is certified as an actual output vertex, the directions of its adjacent edges and faces in $\mathcal{P}$ are derived from the adjacent edges and faces of the one to three input polyhedra that formed it. Implementing this is merely a messy case analysis.

6. CULLING THE POTENTIAL INTERSECTIONS

The success of this procedure relies on efficiently finding the candidate vertices. To simplify the explanation, we will assume we are computing the volume of the union of many congruent, axis-aligned (aka rectilinear or isothetic), cubes, in a universe of size $1 \times 1 \times 1$. Let n be the number of input cubes. Let l be the edge length. Identifying the 3-face vertices by testing all triples of faces, in time $\theta(n^3)$, is infeasible. However, as n increases, l typically falls. Since this paper's algorithm is intended for n in the millions, the asymptotic relation of l to n is of interest.

Since each face has area l^2, the expected number of 3-face intersections is $x_{fff} = \theta(n^3 l^6)$. What is a reasonable relation between n and l; i.e., what is a reasonable model for inputs of ever increasing size?

We might assume that the total volume of all the input cubes is constant as n increases. That is, nl^3 is a constant, so that $l = \theta(n^{-1/3})$. Then,

$$x_{fff} = \theta(n) \tag{7}$$

which is good.

The expected number of face-edge intersections is

$$n_{fe} = \theta(n^2 l^6) = \theta(1) \tag{8}$$

We need an efficient culling technique to find the candidate vertices in expected constant time per vertex. For the order of magnitude of n under consideration, even a factor of $\lg n$ is undesirable. Also, complex data structures are undesirable. There is comparatively little prior art on efficient operations on E^3.

We will use a *uniform grid*[14, 1, 20] to cull the faces and edges for sets that are likely to intersect. A uniform grid has one parameter, an integer g typically in the range 5 to 1000. It partitions the $1 \times 1 \times 1$ universe into $g \times g \times g$ cells. Each cell knows the set of input vertices, edges, and faces that are incident on it. One edge or face may be incident on several cells. The candidate vertex identification process proceeds as follows.

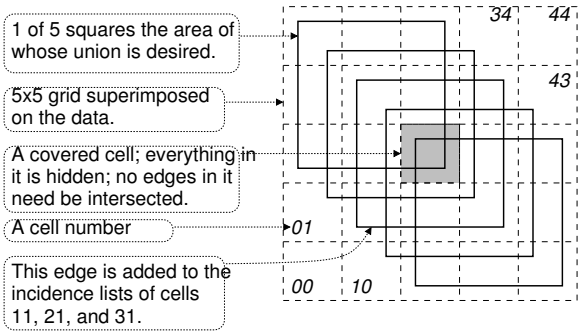

Figure 2: Covered cells

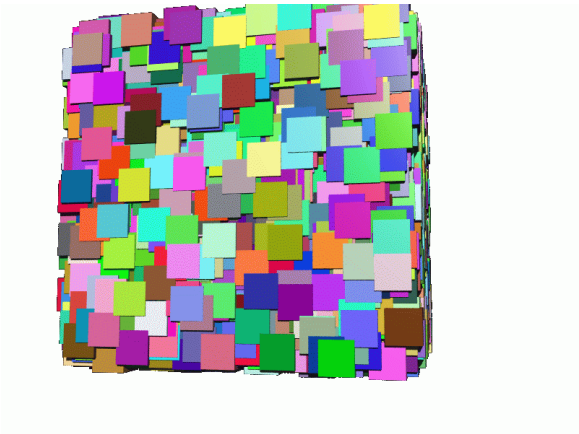

Figure 3: Seriously overlapping cubes

1. Iterate through the input cubes. For each cube, find the grid cells that it overlaps, and insert the cube's vertices, edges, and faces into the appropriate cells' incidence sets.

   Also, if a cube should completely cover a cell, then mark that cell as *covered*. Since all output vertices are adjacent to the exterior of $\mathcal{P}$, a covered cell cannot contain any output vertices. The covered cell concept is illustrated, in E^2, in Figure 2.

   The covered cell concept does not affect the asymptotic complexity in this case, but greatly reduces the execution time when there are many large input cubes as illustrated in Figure 3, which would otherwise be bad because of the large number of intersections. In another context, hidden surfaces, the covered cell is necessary to reduce the asymptotic time to linear. Indeed, then the number of intersections of pairs of edges can be superlinear, but the covered cell reduces the number of edge intersections that potentially visible, and that are computed, to linear.

2. Iterate through the grid cells. In each cell that is not covered, intersect all triples of faces, and intersect all edges against all faces.

Analyzing the uniform grid's performance requires a rule for g. Setting $g = c/l$ for some c is reasonable; $c = 4$ has proven to work well in practice. The optimal c depends on the relation of l to n, but the optimum is broad. For covered cells to occur, we must have $c > 1$.

The theoretical worst case time of the uniform grid is very bad since an adversary may pile all the input objects into a few cells. However, in practice a form of Lipshitz condition obtains so that the worst case cell does not have more than a few orders of magnitude more objects than the average. This has been observed in various diverse domains in E^2 and E^3, from edges in an integrated circuit design to roads in a USFS digital line graph to faces in triangulations in a finite element model.

With the value for g chosen above, the expected number of objects per grid cell, and so the time to process each cell, is constant. The total time to find the candidate vertices is the number of cells, $g^3 = c^3/l^3 = \theta(n)$, that is, linear in the input.

7. CULLING THE CANDIDATE VERTICES

The previous section identified a superset of the output vertices required for the volume computation. Now we must test each vertex to see whether it is in the interior of any input cube (other than one of the one to three cubes that defined it). The goal is constant time per vertex; again the method is the uniform grid, as follows.

1. To test a candidate vertex, determine which grid cell contains it.
2. If that is a covered cell, then delete this vertex.
3. Otherwise, test this vertex against all the input cubes incident on that cell; if any contain the vertex, then delete this vertex.

8. VOLUME COMPUTATION

From the set of output vertices determined above, the volume of $\mathcal{P}$ is the sum of a simple function of each vertex and its neighborhood. Any other mass property follows similarly. We also compute surface area and edge length.

9. HANDLING DEGENERACIES

In the real world of computer aided design, degenerate positions of objects are common. Consider one cube placed to rest on top of another. Even with random input, as n grows, the risk of an accidental degeneracy rises. The nature of the formulae used in this paper, involving adding large positive and negative numbers that almost cancel out, is that one wrong vertex will cause a completely erroneous volume. This is good for debugging, but not otherwise desirable. The solution is *simulation of simplicity* (SoS). SoS works by conceptually perturbing each coordinate by an infinitesimal quantity, so that no two coordinates are identical. Different orders of infinitesimals are added to each coordinate if necessary, so that there will also be no concidences among intersections. No actual infinitesimals are required in the implementation, rather their effect on any predicates is determined and the predicate recoded to effect that. E.g., suppose that x_i is perturbed to $x_i + \epsilon^i$. Then the test

$$x_i < x_j \tag{9}$$

is changed to (in C language notation) to

$$(x_i < x_j)||((x_i == x_j)\&\&i < j) \tag{10}$$

This has a logical but perhaps unexpected effect on the total face area of $\mathcal{P}$ when two cubes are just touching on a common face. SoS might cause the two cubes to be united into one connected polyhedron, so that that common face does not contribute to the output area. Otherwise, it might cause the two cubes to be infinitesimally separated so that that face contributes twice.

10. IMPLEMENTATION

Computing the volume, area, and edge length of the union of many random aligned congruent cubes has been implemented as a C++ program. Although the uniform grid is conceptually simple, an efficient implementation does take care. Since we wanted to test on the largest possible n, and run in parallel, minimal storage and simplicity were the goals. The main design decision is how to store the contents of each grid cell, since the number of elements incident on a cell varies widely. Although a linked list is obvious, it is almost always wrong. Indeed, each pointer requires as much storage as the element (depending on the element's size). It is impossible to randomly access the k-th element in constant time. We used an STL `vector` class, but that was also suboptimal in this context. One problem is that reallocating the vector as it grows fragments the heap so that each separate allocation takes more time. That is, the total memory processing time is superlinear in the number of allocations. Also, the memory fragmentation causes the heap size to be unnecessarily large, reducing the maximum problem size that will fit in core.

In other applications, we have computed the contents of the grid cells twice as follows.

1. Allocate an integer array c of size $g \times g \times g$, initializing the elements to 0. This will keep a count of the number of objects (vertices, edges, or faces) incident on each cell.
2. Process the input polyhedra, to update those counts. Do not, at this time, store the incidence information.
3. Let h be the total number of incidence relations, $h = \sum_{0 \le i,j,k < g} c_{ijk}$.
4. Allocate a an array of h integers, to serve as a ragged array for the contents of the uniform grid
5. Repurpose c to be a dope vector into a, c_{ijk} pointing to the start of that grid cell's incident objects.
6. Reprocess the input polyhedra, this time to store the incident objects to each grid cell.

The cost of this method is that the input is processed twice. The advantage is that minimal storage is used and storage is

not reallocated in the heap. Indeed, given an upper bound on its size, the big array can be allocated staticly at compile time.

An alternative possible technique would be to store the contents of the cells, together with each element's cell, in one large array, then sort the array. This would be more attractive for parallel execution since efficient parallel sorting routines are widely available.

The summary of the previous paragraphs is that, although the uniform grid is conceptually simple, efficient implementation takes care.

The big problem with a uniform grid is that almost everyone feels the need to complexify it by adapting it to the varying densities of the different cells, to turn it into something approaching an octree. Both theoretical analysis and our practical experience show this to be *wrong*. Increasing its complexity makes the code harder to debug, slower to execute, and harder to parallelize.

We tested this implementation on sets of random cubes, whose coordinates were generated with a Tausworth random number generator. Particularly to be avoided are linear congruential generators since they generate points that fall on a comparatively small number of parallel planes[26].

Various factors give us confidence in the algorithm's and implementation's correctness.

1. We hardwired in small examples, both nondegenerate and degenerate, whose union volumes were known.
2. The computed volume is the sum of many large quantities of both signs, and also must be in the range $[0, 1]$. Most errors will produce an out-of-range number.
3. For larger examples, we tested the same sets of cubes with different values of g, to check the consistency of the grid processing.
4. Since the cubes are random i.i.d uniformly distributed, $E[V]$, the expected volume of the union of a large number of small cubes is an easy computation. Indeed, unless the cubes are so dense that effects near the edge of the universe predominate,
$$E[V] = 1 - (1 - l^3)^n \tag{11}$$
5. Testing the parallel implementation is accomplished by seeing whether the result, not just the volume but counts of vertices and intersections of different types, is independent of the degree of parallelism.

Our code is also freely available for non-profit research and education for anyone else to test.

11. PARALLEL PROGRAMMING ENVIRONMENTS

This is a quick review of parts of OpenMP, an API for multithreading on shared memory parallel systems, and MPI, or Message Passing Interface, an API for more distributed environments.

The OpenMP computation model assumes that a process can have multiple threads that all access the same shared memory. This allows one process to more fully utilize a multicore processor. The user adapts a C++ program for OpenMP by adding *pragma*s or directives before `for` loops whose iterations may be executed in parallel. Inside the loop, other pragmas identify critical regions that must be executed serially. For instance if data is being appended to a common array, reading and incrementing the array index must be executed in only one thread at a time. *Reduction* variables, which are computed as the sum of a subtotal from each loop iteration, are implemented particularly efficiently.

Implementing actual OpenMP programs shows that falling into a critical region is extremely slow. Indeed the execution time of the example given above, with multiple threads appending into the same array, is prohibitive. We have observed such code to take more clock time in parallel than when executed in one thread without using OpenMP at all. Even writing large amounts of data in parallel to different memory locations seems to be inefficient. The only efficient way to get data out of an OpenMP computation seems to be via a reduction, i.e., by writing very little global data. That aligns very well with the point of this paper.

The hardware limit of OpenMP is that too many threads is impossible to implement efficiently. The solution is MPI, a completely different API intended for multiple, perhaps tens of thousands, separate processes, each with its own address space, typically on separate processors, even with different architectures. The MPI API allows the processes to send and receive messages to and from each other. The limitation of MPI is that communication is much slower than computation. An efficient MPI program often uses a master process to distribute parts of a large computation and collect results, called *scatter – gather*. There is often no other interprocess communication.

12. PARALLEL IMPLEMENTATION

The algorithm described in this paper is eminently suitable for parallel execution. The first stage, computing the cube-cell incidences, is a scatter-gather operation. Each cube may be processed independently of the others. In practice, because each cube takes so little time and starting a parallel process or thread is comparatively so slow, there should be many fewer processes or threads than cubes. The major limitation is congestion when the information is gathered at each cell, since something has to be serialized here. For instance, our experience with OpenMP is that critical regions are very slow. However, this is still much better than parallelizing a sweep line.

The second stage, processing the grid cells to compute each one's contribution to the volume is totally parallelizable. The grid cells do not affect each other. The only output datum is a scalar. The numbers for the various cells are combined with a very efficient reduction operation.

13. TEST RESULTS

We implemented the algorithm in C++ on an Ubuntu linux 2.4GHz quad-Opteron with 64GB of main memory. As n grew, we chose l to cause many overlapping cubes, because we can handle that difficult case so well. g was set at $4/l$ except when that would use too much memory. Table 1 shows some sample statistics. For n from 100K to 100M, the execution time is only slightly worse than linear.

14. COMPUTING THE EXPLICIT UNION

The technique described above, for computing the set of vertices with their neighborhoods, can be used to compute the explicit union, should that be required. By *explicit union* we mean the set of edges and faces. Determining the edges goes as follows.

1. Each vertex defines one end and the direction of every edjacent edge; call these *half-edges*.
2. For each half-edge, determine the infinite line that contains it, and associate that half-edge with that line.
3. For each infinite line, sort its half-edges by vertex location, and combine adjacent pairs of vertices into individual edges, while performing consistency checks.

The faces are more complicated:

1. Each vertex of $\mathcal{P}$ defines, for each adjacent face of $\mathcal{P}$, one vertex and the half-edges of its two adjacent edges.
2. For each adjacent face of each vertex of $\mathcal{P}$, determine the infinite plane containing it, and associate the face that we have (i.e., vertex and two half-edges) with that plane.
3. For each infinite plane, we have a 2-D problem of finding the edges and faces[18].
   (a) Associate the half-edges with infinite lines in the plane, and link them into edges.
   (b) Perform planar graph traversals to identify the faces, while checking consistency. Note that a face may have genus greater than zero.

The above process of finding edges and faces from vertices is worse, both in lines of code and in execution resources, than finding only the vertices and their neighborhoods. That is what motivated the research into local topological formulae.

Nevertheless, this is an algorithm for computing the explicit boolean combination of many primitive polyhedra in time linear in the size of the input plus output, if the expected number of vertices along each infinite edge is fixed (for random input, it's 2).

It may be that the sets of edges and faces are insufficient, and that the complete topology is required. In that case, shells of faces would need to be computed, together with their nesting relationships. However, that would seem to be usually unnecessary.

15. ONLINE COMPUTATION

The above algorithm for computing the volume of the union of many polyhedra has been expressed in the form of an offline, batch, process, where all the input is known before the computation. Extending this to an online algorithm where the volume is updated as new input polyhedra are read is mostly easy:

1. Insert the new polyhedron's vertices, edges, and face into the uniform grid.
2. Test each previous output vertex to determine whether it is still an output vertex. It would fail if it is contained in the new polyhedron.
3. Test each new input vertex to see whether it is an output vertex.
4. If the new polyhedron covers one or more grid cells, then delete those cells' contents from the cell and tag the cell accordingly.
5. Test the new edges and faces against existing edges and faces in the same cells to find more new output vertices.
6. Update the volume accordingly.

Notes:

1. Optimal performance assumes that the input polyhedra are generated by a stationary process. If their statistical distribution changes over time, then g, the uniform grid size, may become suboptimal.
2. The concrete data structure for the members of each grid cell must allow efficient online insertions. That excludes both the ragged array and forming an array of all the *(cell, contents)* pairs as they are detemined and then sorting it and then creating the grid data structure.
3. Because the expected size of each cell's contents is constant, processing one online polyhedron should take constant expected time. However the constant factor in the time will be larger than for offline processing.

16. EXTENSION TO GENERAL POLYHEDRA

The volume of the union algorithm has been expressed and implemented in terms of congruent axis-aligned cubes. The theory generalizes to any input polyhedra but the implementation would become much harder in the following ways.

1. More special cases would need consideration, for inside vs outside etc, when the faces are not axis-aligned. This is just detail work.
2. The general volume formula in Equation 4 would be needed.

Table 1: Test results

n	$1/l$	g	n_{vout}	n_{fff}	n_{fe}	n_{cov}	n_{cp}	n_{cf}	n_{ce}	V	A	L	Time
100K	20	80	9K	17K	26K	467K	283K	441K	231K	0.977	7.460	1002.	3.05
800K	40	160	38K	76K	112K	3899K	1387K	2147K	1083K	0.988	7.712	2102.	26.
2700K	60	240	87K	180K	259K	13M	3678K	5681K	2780K	0.992	7.812	3215.	95.
6400K	80	320	159K	355K	484K	31M	8347K	12M	6107K	0.994	7.963	4426.	255.
12500K	100	400	249K	578K	762K	62M	14M	23M	10M	0.995	7.985	5572.	502.
21600K	120	480	351K	803K	1071K	108M	21M	32M	15M	0.996	7.933	6582.	947.
34300K	140	560	485K	1171K	1496K	172M	34M	52M	24M	0.996	8.002	7820.	1381.
51200K	160	800	643K	1669K	2012K	505M	41M	54M	23M	0.997	8.073	9128.	2691.
72900K	180	900	794K	1967K	2459K	721M	49M	65M	27M	0.997	7.984	10026.	3776.
100M	200	1000	1007K	2805K	3197K	989M	73M	97M	40M	0.997	8.109	11556.	5821.

Legend:

1K	10^3
1M	10^6
n	number of input cubes
l	edge length
g	grid size
n_{vout}	number of output vertices
n_{fff}	number of face-face-face intersections
n_{fe}	number of face-face intersections
n_{cov}	number of covered cells
n_{cp}	number of cell-polyhedron incidences
n_{cf}	number of cell-face incidences
n_{ce}	number of cell-edge incidences
V	volume of $\mathcal{P}$
A	area of $\mathcal{P}$
L	edge length of $\mathcal{P}$
Time	CPU time in seconds

3. Numerical roundoff errors would start to cause topological errors. With axis-aligned faces, there is no roundoff. This must be handled correctly because this algorithm is completely intolerant of topological errors. It does not store the topology explicitly, but it does require that the implicit topology determined by the explicit set of vertices and neighborhoods be consistent.

   The answer is probably to compute with big rationals in CGAL. Our computation tree has maximum depth three regardless of the number of input polyhedra, which is a big advantage, so the rationals would not grow too much.

4. Removing degeneracies with Simulation of Simplicity would become more complicated. SoS is intolerant of roundoff error, but with big rationals that would not be a problem.

17. EXTENSION TO OTHER BOOLEAN OPERATIONS

Can this algorithm be extended to computing a more general tree of boolean operations as a flat, one-level, process? The change would be in the rule for culling the candidate output vertices. The candidate vertices of the different polyhedra would need to satisfy different rules to be selected. The question is whether each vertex is adjacent to both the interior and exterior of the output polyhedron. For instance, consider

$$\mathcal{A} \cup (\mathcal{B} - \mathcal{C}) \tag{12}$$

The following are the output vertices.

1. Vertices of $\mathcal{A}$ that are outside $\mathcal{B}$ or inside $\mathcal{C}$.
2. Vertices of $\mathcal{B}$ that are outside $\mathcal{A}$ and outside $\mathcal{C}$.
3. Vertices of $\mathcal{C}$ that are outside $\mathcal{A}$ and inside $\mathcal{B}$.
4. All intersections of a face of $\mathcal{A}$ with a face of $\mathcal{B}$ and a face of $\mathcal{C}$.
5. Intersections of an edge of $\mathcal{A}$ with a face of $\mathcal{B}$, which are outside $\mathcal{C}$.
6. Intersections of an edge of $\mathcal{B}$ with a face of $\mathcal{A}$, which are outside $\mathcal{C}$.
7. Intersections of an edge of $\mathcal{A}$ with a face of $\mathcal{C}$, which are inside $\mathcal{B}$.
8. Intersections of an edge of $\mathcal{C}$ with a face of $\mathcal{A}$, which are inside $\mathcal{B}$.
9. Intersections of an edge of $\mathcal{B}$ with a face of $\mathcal{C}$, which are outside $\mathcal{A}$.
10. Intersections of an edge of $\mathcal{C}$ with a face of $\mathcal{B}$, which are outside $\mathcal{A}$.

For larger boolean expressions, these rules can be mechanically generated, as follows. Let $f(a_1, a_2, ...)$ be a boolean function corresponding to the boolean operation on the input polyhedra. A vertex of input polyhedron a_1 is an output vertex iff $f(0, a_2, a_3, ...) \neq f(1, a_2, a_3, ...)$.

There is a problem of what information needs to be stored in the uniform grid? Another view is, how can these expressions be optimized? When the only boolean operation is *union*, the only vertex test is, "does any other polyhedron contain this vertex?" That is efficiently implemented with the covering cell idea. The goal would be for constant expected evaluation time per vertex; whether this is possible is unknown.

18. OVERLAYING TWO TRIANGULATIONS

Overlaying two triangulations in E^2 or E^3 to identify all the nonempty intersections, and compute their volumes, between a triangle or tetrahedron of one triangulation with

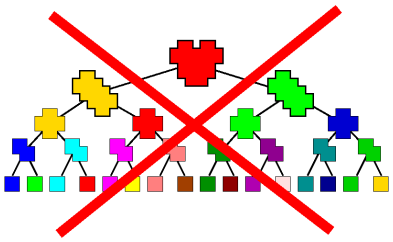

Figure 4: Don't need this computation tree

one of the other is another application of these ideas. However, instead of finding one volume of the union of all the primitives, we are finding many volumes, each the intersection of two primitives. Ignoring efficiency considerations, the process goes as follows.

1. Find all the intersections between a face or edge of one triangulation and an edge or face, respectively, of the other triangulation.
2. Each such intersection is one vertex of six (in general) different output intersection polyhedra.
3. Compute the intersection's contributions to the six output polyhedra and sum them into six entries of a hash table, keyed by the sorted pair of the two intersecting tetrahedra that would make each output polyhedron.
4. At the end, each nonempty entry in the hash table is the volume of a nonempty intersection.
5. Use those volumes for cross-interpolation of the desired property of the first triangulation onto the second.

With a uniform grid, the expected execution time is linear in the number of intersections, [21]. We implemented this in E^2 and tested on an example of the coterminous US counties intersected with hydrography polygons, totalling 3000 polygons and about 130,000 edges. The biggest component of the sub 1 CPU second execution time was reading and writing the data[22].

19. CONCLUSIONS AND FUTURE WORK

Computing the union of many primitive objects does not require combining them pair by pair in a tree of height $\lg n$, see Figure 4, but rather may be done in one step. Optimizing the composition of operators, such as union and volume can be very efficient. The expected execution time is linear, and large sets of inputs may be processed efficiently.

Future work may include more optimizations, extension to other boolean operations, and porting the code from OpenMP to MPI to enable greater parallelism.

20. ACKNOWLEDGEMENT

This research was partially supported by NSF grant CMMI-0835762.

21. REFERENCES

[1] V. Akman, W. R. Franklin, M. Kankanhalli, and C. Narayanaswami. Geometric computing and the uniform grid data technique. *Computer Aided Design*, 21(7):410–420, 1989.

[2] B. Aronov, M. Sharir, and B. Tagansky. The union of convex polyhedra in three dimensions. *SIAM J. Comput.*, 26:1670–1688, 1997.

[3] H. Bieri and W. Nef. A sweep-plane algorithm for computing the volume of polyhedra represented in boolean form. *Linear Algebra and its Applications*, 52–53:69–97, 1983.

[4] E. Brisson. Representing geometric structures in d dimensions: topology and order. In *SCG '89: Proceedings of the fifth annual symposium on Computational geometry*, pages 218–227, New York, NY, USA, 1989. ACM.

[5] S. Cameron. Efficient bounds in constructive solid geometry. *IEEE Comput. Graph. Appl.*, 11(3):68–74, 1991.

[6] Center for Geometric and Biological Computing. External memory algorithms and data structures. `http://www.cs.duke.edu/CGC/subjects/external.html`, 2001.

[7] B. Chazelle. An optimal algorithm for intersecting three-dimensional convex polyhedra. In *Proc. 30th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 586–591, 1989.

[8] B. Chazelle and D. P. Dobkin. Intersection of convex objects in two and three dimensions. *J. ACM*, 34(1):1–27, Jan. 1987.

[9] R. Descartes. *Principia Philosophiae*. Ludovicus Elzevirius, Amsterdam, 1644.

[10] D. P. Dobkin and D. G. Kirkpatrick. Fast detection of polyhedral intersection. *Theoret. Comput. Sci.*, 27(3):241–253, Dec. 1983.

[11] K. Dobrindt, K. Mehlhorn, and M. Yvinec. A complete and efficient algorithm for the intersection of a general and a convex polyhedron. In *Proc. 3rd Workshop Algorithms Data Struct.*, volume 709 of *Lecture Notes Comput. Sci.*, pages 314–324, 1993.

[12] D. Eppstein. Speed-ups in constructive solid geometry. Report 92-87, Dept. Inform. Comput. Sci., Univ. California, Irvine, CA, 1992.

[13] J. Erickson. Local polyhedra and geometric graphs, 2003.

[14] W. R. Franklin. *Combinatorics of hidden surface algorithms*. Technical Report, Center for Research in Computing Technology, Harvard Univ., Cambridge, MA, June 1978.

[15] W. R. Franklin. Polygon properties calculated from the vertex neighborhoods. In *Proc. 3rd Annu. ACM Sympos. Comput. Geom.*, pages 110–118, 1987.

[16] W. R. Franklin. Analysis of mass properties of the union of millions of polygedra. In M. L. Lucian and M. Neamtu, editors, *Geometric Modeling and Computing: Seattle 2003*, pages 189–202. Nashboro Press, Brentwood TN, 2004.

[17] W. R. Franklin. Mass properties of the union of millions of identical cubes. In R. Janardan, D. Dutta,

and M. Smid, editors, *Geometric and Algorithmic Aspects of Computer Aided Design and Manufacturing, DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, volume 67, pages 329–345. American Mathematical Society, 2005.

[18] W. R. Franklin and V. Akman. Reconstructing visible regions from visible segments. *BIT*, 26:430–441, 1986.

[19] W. R. Franklin, N. Chandrasekhar, M. Kankanhalli, V. Akman, and P. Y. Wu. Efficient geometric operations for CAD. In M. J. Wozny, J. U. Turner, and K. Preiss, editors, *Geometric Modeling for Product Engineering*, pages 485–498. Elsevier Science Publishers B.V. (North-Holland), 1990.

[20] W. R. Franklin, N. Chandrasekhar., M. Kankanhalli, M. Seshan, and V. Akman. Efficiency of uniform grids for intersection detection on serial and parallel machines. In N. Magnenat-Thalmann and D. Thalmann, editors, *New Trends in Computer Graphics (Proc. Computer Graphics International'88)*, pages 288–297. Springer-Verlag, 1988.

[21] W. R. Franklin and M. S. Kankanhalli. Volumes from overlaying 3-D triangulations in parallel. In D. Abel and B. Ooi, editors, *Advances in Spatial Databases: Third Intl. Symp., SSD'93*, volume 692 of *Lecture Notes in Computer Science*, pages 477–489. Springer-Verlag, June 1993.

[22] W. R. Franklin, V. Sivaswami, D. Sun, M. Kankanhalli, and C. Narayanaswami. Calculating the area of overlaid polygons without constructing the overlay. *Cartography and Geographic Information Systems*, pages 81–89, Apr. 1994.

[23] D. H. Laidlaw, W. B. Trumbore, and J. F. Hughes. Constructive solid geometry for polyhedral objects. *Comput. Graph.*, 20(4):161–170, 1986. Proc. SIGGRAPH '86.

[24] Y. T. Lee and A. A. G. Requicha. Algorithms for computing the volume and other integral properties of solids. I. Known methods and open issues. *Commun. ACM*, 25:635–641, 1982.

[25] Y. T. Lee and A. A. G. Requicha. Algorithms for computing the volume and other integral properties of solids. II. A family of algorithms based on representation conversion and cellular approximation. *Commun. ACM*, 25:642–650, 1982.

[26] G. Marsaglia. A current view of random number generators. In *Proceedings, Computer Science and Statistics: 16th Symposium on the Interface*, Atlanta GA USA, 1984. Elsevier. (keynote address).

[27] K. Mehlhorn and K. Simon. Intersecting two polyhedra one of which is convex. In L. Budach, editor, *Proc. Found. Comput. Theory*, volume 199 of *Lecture Notes Comput. Sci.*, pages 534–542. Springer-Verlag, 1985.

[28] B. Mirtich. Fast and accurate computation of polyhedral mass properties. *Journal of Graphics Tools*, 1:31–50, 1996.

[29] B. Mirtich. Fast and accurate computation of polyhedral mass properties. *J. Graphics Tools*, 1(2):31–50, 1996.

[30] D. E. Muller and F. P. Preparata. Finding the intersection of two convex polyhedra. *Theoret. Comput. Sci.*, 7:217–236, 1978.

[31] C. Narayanaswami and W. R. Franklin. Determination of mass properties of polygonal CSG objects in parallel. In J. Turner, editor, *Proc. Symposium on Solid Modeling Foundations and CAD/CAM Applications*, pages 279–? ACM/SIGGRAPH, June 1991.

[32] C. Narayanaswami and W. R. Franklin. Determination of mass properties of polygonal CSG objects in parallel. *Internat. J. Comput. Geom. Appl.*, 1(4):381–403, 1991.

[33] K. Sugihara. A robust and consistent algorithm for intersecting convex polyhedra. *Comput. Graph. Forum*, 13(3):45–54, 1994. Proc. EUROGRAPHICS '94.

[34] R. B. Tilove. A null-object detection algorithm for constructive solid geometry. *Commun. ACM*, 27:684–694, 1984.

[35] L. Toma, R. Wickremesinghe, L. Arge, J. S. Chase, J. S. Vitter, P. N. Halpin, and D. Urban. Flow computation on massive grids. In *Proc. ACM Symposium on Advances in Geographic Information Systems*, 2001. see also `http://www.cs.duke.edu/geo*/terraflow/papers/bib.html`.

[36] A. Vigneron. Reporting intersections among thick objects. In *The nature and future of the catalog, Phoenix, AZ : Oryx Press:127-36*, 2002.

[37] A. Zomorodian and H. Edelsbrunner. Fast software for box intersections. *International Journal of Computational Geometry and Applications*, 12(1-2):143–172, 2002.