Parallel Volume Computation of Massive Polyhedron Union

W. Randolph Franklin[*]

Abstract

We present a parallel implementation of UNION3, an algorithm for computing the volume, surface area, center of mass, or similar properties of a boolean combination of many polyhedra. UNION3 has been implemented on the special case of the union of congruent isothetic cubes, and tested on $100,000,000$ cubes. The algorithm uses a volume formula that does not require first computing the explicit union. All that is needed is the set of output vertices, including each vertex's position and neighborhood. UNION3's computation does not use randomization or sampling, and its output is exact. The implementation is parallel, using OpenMP. When using 32 threads, UNION3's elapsed time is only a few minutes, and the speedup, compared to using one thread, is a factor of ten. When executed on uniform random i.i.d input, UNION3's execution time is linear in the number of output vertices. UNION3 is intended as an example of a parallel geometry algorithm whose techniques should be broadly applicable.

1. INTRODUCTION

Parallel computation is necessary for compute-intensive applications for several reasons. CPU clock speeds have hardly increased in several years. Likewise, superscalar and pipeline techniques for executing several instructions in one cycle have plateaued. Conversely, a machine with a several CPUs each with several cores each capable of running two threads, all using shared memory, is relatively economical. Better, NVIDIA GPU-based parallel processors are now available in most laptops for a few hundred dollars, and massively powerful versions are available as addon desktop cards for a few $hundred (as gaming cards) or $thousand (the professional versions). Theoretically, their double precision floating computation speed is in the teraflops.

Complementing the processors, massive amounts of memory are now affordable, allowing fast random access to large datasets, and rendering virtual memory obsolete. Complementing this hardware are software techniques such as OpenMP for the multicore CPUs and CUDA for the NVIDIA compute cards, although efficiently using them is an arcane art. For example, in 2013, a machine with dual 8-core 16-thread 3.4GHz Intel Xeon CPUs, two state-of-the-art NVIDIA cards (K20x and K5000), 128GB of memory, and the usual extras costs under $15K.

This poses a problem for computational geometry. Most algorithms are not amenable to parallelization. Asymptotic times of $\Omega(N \log N)$ are too slow.

2. PARALLELIZATION SUMMARY

OpenMP is a common tool for parallelizing C++ code on multi-threaded shared-memory systems. Its effective use requires that the algorithm be expressed as a sequence of parallelizable loops, which are indentified to OpenMP with directives. Such loops' iterations must be independent of each other. Many threads' reading a common variable in parallel is free. However, any parallel writes to common memory must be conceptually guarded with semaphores. Using a `#pragma for reduction` to have each parallel iteration sum an addend into a common total variable is free. Using a `#pragma atomic` to atomically increment and capture the value of a common variable is cheap. (That is useful, for instance, for each thread to obtain a unique array index to write into.) Using the required `#pragma critical` for any more sophisticated write to a common variable is prohibitively expensive, perhaps hundreds of thousands of cycles per operation. Finally, rules such these are not often written down, but must be intuited from experience. Those are the constraints for designing a good parallel OpenMP algorithm.

OpenMP is not suitable for massive parallelism because the shared-memory model does not scale up to many processors. For the most economical massive parallel platform, NVIDIA GPUs, the usual API is CUDA, perhaps hidden with abstractions like Thrust. However that is beyond this extended abstract.

3. ALGORITHMIC TECHNIQUES

We prefer to spatially organize geometric data with a *uniform grid*. That imposes a 1-level grid on the universe, with each cell containing the set of elements that intersect it. One must resist the temptation to adaptively subdivide some cells, which simultaneously increases programming time, number of lines of code, execution time and space, while decreasing parallelizability.

In the current example, an element is a cube, a face, or an edge, and the grid's size is up to $1000 \times 1000 \times 1000$ when the number of cubes, $N = 100,000,000$. Since the number of elements in each grid cell varies from cell to cell, the obvious concrete realization of the set is a linked list. Our implementation halves the storage by using a ragged array. The input is processed twice, the first time only to count the number of elements in each grid cell. Then, the required storage is allocated and the dope vector for the ragged array is computed. Finally, the input is processed again and the uniform grid ragged array is populated. Processing the input twice is especially efficient on a parallel machine because, the first time, nothing is written except for incrementing an array of counters. Using memory more efficiently also permits larger examples to be run. When the algorithm is transitioned to CUDA, this will be especially important because, there, memory accesses are more expensive than computation.

Our polyhedron volume computation formula uses only the polyhedron's set of vertices' positions and neighborhoods. For a polyhedron with axis-aligned faces, there are eight octants around each vertex, and from one to seven are interior to the polyhedron. We need that local information. We do not need any more global information, including complete edges or faces or any topological information such as nested edge loops or face shells. The formulae do not use any form of sampling, and are exact apart from roundoff error during the computation.

For any boolean combination of polyhedra, each output vertex is one of three types: an input vertex, the intersection of an input edge with an input face, or the intersection of three input faces. Call the set of all such possibilities *candidates*. When the boolean operator is the *union*, the output vertices are precisely those candidates that are outside all the input polyhedra. The uniform grid permits testing whether a candidate is outside all the input polyhedra in expected constant time per candidate.

[*]mail@wrfranklin.org, Rensselaer Polytechnic Institute, Troy NY USA

Notation: Let the universe be a $1 \times 1 \times 1$ region. Let N be the number of input cubes. Let L be the inverse of the cubes' edge length. Let the uniform grid have $G \times G \times G$ cells. In our hardest example, $N = 100{,}000{,}000$, $L = 400$ and $G = 1000$.

All expected times in this extended abstract assume that the input is uniform and independently identically distributed, and that G is chosen optimally as N increases. To make the examples interesting, we choose $L = \theta(N^{1/3})$, which keeps the total volume of all the input polyhedra, $\theta\left(NL^{-3}\right)$, constant. G may be set to keep the expected number of elements per cell, $\theta\left(N^{2/3}G^{-2}\right)$ for $G > L$, constant as $N \to \infty$. That makes $G = \theta(L) = \theta(N^{1/3})$. Testing whether a candidate vertex c is outside all the input cubes requires identifying which grid cell contains c and then testing c against all the cubes overlapping that cell.

In addition, for the candidates formed from the intersection of input elements, only elements that overlap a common cell are tested for intersection. This permits a further optimization when $G >> L$, when a cell will often be completely inside some cube. Such a cell can contain no output vertices. If such cells are marked at an early stage, then no edges or faces need to be stored into such cells, nor tested for intersection. That saves time and space. The resulting expected time is linear in the size of the input plus output. Details, including formulae with derivations, are given in [1, 2].

4. TEST IMPLEMENTATION

Although the theory applies to general polyhedra, our test implementation handles only congruent axis-aligned cubes. They store more compactly, the code is briefer and implemented more quickly, and their edges and faces intersect with no roundoff error. Our hardware was described above; it runs Ubuntu linux. Simulation of simplicity was used to handle any coincidental coordinate values. The complete program is only 824 lines of C++ code.

UNION3 processes small cases very quickly, taking only 0.02 elapsed seconds for $N = 1000$ and $L = 10$, to compute a union volume of 0.573 and area of 19. At this speed, it could be used to efficiently test for collisions among moving rigid objects, each approximated as a union of cubes, as follows. Compute the volume of the union of the set of possibly colliding objects. If it changes, the objects have started to overlap.

The larger case of $N = 1{,}000{,}000$, $L = 100$, $G = 200$, took 3.3 seconds to find that the volume was 0.602.

Our hardest example processed $N = 100{,}000{,}000$ and $L = 400$ with $G = 1000$ in 473 elapsed seconds. That is over 10 times faster than the sequential elapsed time of about 5000 seconds. Some statistics of the computation are as follows. UNION3 used 57GB of main memory. The volume of the union polyhedron was 0.765. If the cubes had not intersected at all, the output volume would have been 1.56, meaning that a point was on average contained in 1.56 cubes. The output surface area was 831, compared to a total area of 3750 for the input cubes. Of the 800,000,000 input vertices, 186,366,729 were not inside any cube, and so became output vertices. There were another 395,497,686 output vertices that were the intersection of 3 of the 600,000,000 input faces, and 811,328,383 output vertices that were the intersection of one of the 600,000,000 input faces with one of the 1,200,000,000 input edges. 258,461,149 of the 1,000,000,000 grid cells were completely covered by some cube.

These numbers show that the sample dataset had many overlaps, and so was nontrivial and would exercise any boolean combination program. These numbers did not change when UNION3 was run in serial vs in parallel. That lends confidence to the correctness of the parallel implementation. The output volume and area did not change when G changed, although the numbers of intersections changed, because of the covered cells. That lends confidence to the correctness of the grid data structure.

UNION3's time is output-sensitive. When run with $N = 100{,}000{,}000$ but somewhat smaller cubes with $L = 500$, the parallel elapsed clock time was reduced to 396 seconds, with $G = 1000$. Because of fewer intersections, only $50GB$ of memory was needed.

5. EXTENSION TO GENERAL POLYHEDRA AND BOOLEAN OPERATIONS

Although UNION3 does not require that the global topology of the input be explicit, it does require that it be correct. UNION3 is also completely intolerant of topological errors during the computation caused by roundoff errors. An example would be to incorrectly determine that an edge intersected a face, when it did not. Therefore uniting general polyhedra would require computing with big rational numbers, perhaps using GMPXX.

To compute the volume of a general boolean function f on a large number of input polyhedra, we would write f in disjunctive normal form. The top level union needs the output vertices, with neighborhoods, of each low level intersection. Implementing this would be a challenge, but looks quite doable.

6. CONCLUSIONS

The significance of this result is as follows. • This implementation is merely a proof of principle to demonstrate a nontrivial geometric operation on a large database that is accelerated by a factor of ten when run in parallel. • The concepts extend to the union of any set of polyhedra, with general topology (multiple nested shells of faces containing multiple nested loops of edges). • Since a general Boolean operation can be expressed in terms of unions and intersections, and since intersections are easier than unions, this could be extended, with some effort, to compute any mass property of any Boolean operation of a set of polyhedra. • The algorithm is quite parallelizable. • UNION3 would appear to compare quite favorably with existing methods.

The next step is to reimplement UNION3 in CUDA on our NVIDIA K20x GPU Accelerator.

Our source code, albeit only prototype quality, is freely available for nonprofit research and education; we welcome stress tests and error reports.

This research was partially supported by NSF grant IIS-1117277.

7. REFERENCES

[1] 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.

[2] 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.