Parallel intersection detection in massive sets of cubes

W. Randolph Franklin
Rensselaer Polytechnic Institute
Troy, NY, USA
mail@wrfranklin.org

Salles V. G. Magalhães
Universidade Fed. de Viçosa
Viçosa, MG, Brazil
salles@ufv.br

ABSTRACT

We present ParCube, which finds the pairwise intersections in a set of millions of congruent cubes. This operation is required when computing boolean combinations of meshes or polyhedra in CAD/CAM and additive manufacturing, and in determining close points in a 3D set. ParCube is very compact because it is uses a uniform grid with a functional programming API. ParCube is very fast; even single threaded it usually beats CGAL's elapsed time, sometimes by a factor of 3. Also because it is FP, ParCube parallelizes very well. On an Nvidia GPU, processing $10M$ cubes to find $6M$ intersections, it took 0.33 elapsed seconds, beating CGAL by a factor of 131. ParCube is independent of the specific parallel architecture, whether shared memory multicore Intel Xeon using either OpenMP or TBB, or Nvidia GPUs with thousands of cores. We expect the principles used in ParCube to apply to other computational geometry problems. Efficiently finding all bipartite intersections would be an easy extension.

CCS CONCEPTS

• **Theory of computation → Nearest neighbor algorithms**; *MapReduce algorithms*; *Computational geometry*; • **Computing methodologies** → *MapReduce algorithms*;

KEYWORDS

Parallel Programming, Computational Geometry, Intersection, Close Points, Near Points, Uniform Grid, Functional Programming, Thrust, Map-Reduce Algorithms

ACM Reference Format:
W. Randolph Franklin and Salles V. G. Magalhães. 2017. Parallel intersection detection in massive sets of cubes. In *Proceedings of BigSpatial'17:6th ACM SIGSPATIAL Workshop on Analytics for Big Geospatial Data , Los Angeles Area, CA, USA, November 7–10, 2017 (BigSpatial'17)*, 7 pages.
https://doi.org/10.1145/3150919.3150921

1 INTRODUCTION

3D geometry applications often require finding the intersecting pairs in a large set of small objects. For instance, in additive manufacturing (also known as 3D printing) or in computational fluid dynamics (CFD), we might wish to interpolate some property varying over space, such as density, from one mesh of an object to another mesh of the same object. This requires knowing the volumes of the intersections of all the tetrahedra of one mesh with the tetrahedra of the other mesh. That requires knowing which pairs of tetrahedra intersect.

Parallel processing is desirable, because the laws of physics do not leave an obvious path to increasing the CPU speed for a single processor. Indeed, a bit of information is effectively stored on an integrated circuit in a small capacitor charged through a resistor. This model remains valid when the bit is stored in a transistor that is in either a *on* or *off* state. Increasing the speed of switching this bit requires reducing the bit's resistance and capacitance. That has physical problems, e.g., the circuit cannot be shrunk any more, and also increases the power consumption to the point where the circuit cannot be cooled.

Several competing parallel architectures are now available. The easiest to program, often with OpenMP [27] or Intel Threading Building Blocks (TBB) [16], is a multicore Intel Xeon processor. For example, one processor board may have dual CPUs, each with 14 cores, each running two hyperthreads, for a total of 56 parallel threads. Each thread runs its own instruction stream, and each has access to the whole main memory, which could be up to 2TB. However when two threads might write the same address, expensive interlocking with atomic operations is required. A naive parallel program can easily take more elapsed time when using more threads.

Nvidia's GPUs are another widely used and very affordable platform. GPUs, which started as graphics and gaming coprocessors, are today part of about one third of all personal computers. In response to customer demand, scientific computing functionality has been added, and video output capability sometimes removed. Many supercomputers comprise thousands of GPUs.

This paper uses the Nvidia GeForce GTX Titan X, a high-end GPU costing USD1000 and introduced in 2015, with 3072 CUDA cores clocked at 1GHz, 12GB of memory, and a peak single precision floating performance of 7 TFLOPS. However, nothing in ParCube is specific to this model; it was developed on a Thinkpad W540 laptop introduced in 2014. That machine has a smaller Quadro K2100M GPU, which has only 576 cores clocked at 0.67Ghz. Although one CUDA core is about only 5% as powerful as one Intel Xeon core, still the aggregate power of a modern GPU is impressive.

The most direct way to program an Nvidia GPU is with the CUDA API, which is C++ with some language extensions and library routines. However, at this low level, programming must consider the hierarchy of execution threads into warps, blocks, and grids, and several classes of memory, such as registers, shared, local, constant, and global, with capacities and speeds ranging over factors of over 100. A *warp* is a set of 32 threads executing the same instruction (or

BigSpatial'17, November 7–10, 2017, Los Angeles Area, CA, USA

ACM ISBN 978-1-4503-5494-3/17/11…$15.00
https://doi.org/10.1145/3150919.3150921

idle); this is called Single Instruction Multiple Thread (SIMT). For best performance, consecutive threads should access consecutive words of the GPU memory. If one warp is blocked while waiting for resources, then another warp can be efficiently scheduled to run. One good introduction to GPU programming is [2].

Various APIs built on top of CUDA are available, abstracting away some of its complexities (and opportunities to optimize). This project uses *Thrust*, [24, 26]. Thrust is a set of C++ templates in header files, modeled on STL, the Standard Template Libary. Thrust has mostly a *Functional Programming* (FP) design. The only Thrust data structure is a *vector*. Vectors can be mapped and reduced. One useful vector transformation is the *gather*, which permutes the vector by computing $c_i \leftarrow b_{a_i}$. Thrust also has an efficient *radix sort*, which is not FP since it is in-place. Except for precomputing a combination table, ParCube's code is straight line, and apart from sorting, individual vector elements are not assigned, and no variable needs to be changed from its initial value, all of which follows the FP model.

Thrust's model is also very efficient. Routines execute in parallel, utilizing as many cores as are available. The execution performance penalty for using Thrust compared to a lower level abstraction layer closer to CUDA, such as Nvidia CUB [25], might be a factor of three. In return, the programming is much easier.

Another advantage of Thrust's level of abstraction is that it allows any of several backends, each exploiting different parallel hardware, to be selected at compile time. We have tested ParCube with these backends: regular single threaded C++, CUDA, OpenMP, and TBB. Therefore, ParCube can utilize whatever parallel hardware is available.

The proper use of Thrust requires that an algorithm be expressed in terms of the available functions. Explicit loops have very poor performance. Each function call has a setup cost to initialize the parallel environment, so that a few operations on large vectors are much preferred to many operations on small vectors. The global GPU memory has a latency of over 100 cycles, so that following pointers will be slow unless many parallel threads do this simultaneously, while executing the same instructions. All this implies that lists of different lengths, tree-based data structures and recursive algorithms do not easily parallelize.

Another advantage of this model is that the source code is very compact—much of the scaffolding has been stripped away. That also assists optimizing compilers and other automated tools that reason about and transform the code.

2 PRIOR ART

The pairwise intersection of boxes is a classical computational geometry problem. Solutions are typically based either on the plane-sweep technique [4, 28], or employ uniform grids [11], or hierarchical structures [3, 7, 19] to create a set of potentially intersecting pairs of boxes, which is further refined by testing each pair for intersection.

One well known sequential algorithm for detecting intersections of boxes is Zomorodian and Edelsbrunner [28]. This was designed to be very efficient in practice and is a hybrid of a scanning and a streaming algorithm. The basic idea is that a pair of intersecting boxes must intersect in all dimensions. Thus, it performs a series of 1D intersection computations (in each dimension) using either a scanning approach or a streamed approach based on segment and range trees (the scanning algorithm is also employed to prune the trees). An adversary could pessimize this with a set of boxes strung out along a diagonal so that they intersected in each separate dimension, but not in 3D. However, it is very efficient in practice and can be generalized for higher dimensions, and so was implemented in CGAL [5].

PBIG is another parallel algorithm for detecting intersections of boxes on GPUs, Lo et al. [19]. It is based on a uniform grid to cull the pairs of intersecting boxes. *PBIG* is complex, with assorted low-level optimizations tailored to CUDA, such as compression. It reports a speedup of 110× against the sequential algorithm available in CGAL.

The contributions of our algorithm are as follows:

(1) ParCube is very compact because it is uses a uniform grid with a functional programming API.
(2) ParCube is very fast; even single threaded it usually beats CGAL, sometimes by 3×.
(3) Also because it is FP, ParCube parallelizes very well.
(4) On an Nvidia GPU, processing $10M$ cubes to find $6M$ intersections, it took 0.33 elapsed seconds, beating CGAL by 131×.
(5) ParCube is independent of the specific parallel architecture, whether shared memory multicore Intel Xeon using either OpenMP or TBB, or Nvidia GPUs with thousands of cores.
(6) We expect the principles used in ParCube to apply to other computational geometry problems.

3 THE ALGORITHM

In the following, symbol names may be multiple alphanumeric characters, and multiplication of symbols is always expressed explicitly, e.g., $a \cdot b$. Vectors are boldfaced.

ParCube is a prototype designed to illustrate the power of geometric programming at a higher level of abstraction, largely functional programming, and using the uniform grid for parallel geometric processing. The input is a set $\mathcal{B} = \{b_i\}$ of n cubes, each with edge length l inside a universe of size $1 \times 1 \times 1$. Cube b_i is defined by (x_i, y_i, z_i), the coordinates of its lower left front corner. To reduce special cases, we require that

$$0 < x_i, y_i, z_i < 0.9999 - l.$$

The concept of ParCube extends to more general objects, such as tetrahedra, but the implementation would be tedious. In any case, tetrahedra could be enclosed in cubes and ParCube applied to produce pairs of tetrahedra likely to intersect, because their enclosing boxes do.

The desired output is a set $\{(b_{i1}, b_{i2})\}$ of pairs of cubes that intersect, i.e.,

$$\left(\left|x_{b_{i1}} - x_{b_{i2}}\right| \le l\right) \wedge \left(\left|y_{b_{i1}} - y_{b_{i2}}\right| \le l\right) \wedge \left(\left|z_{b_{i1}} - z_{b_{i2}}\right| \le l\right).$$

If the coordinate type is float, then roundoff errors make this indeterminate in some cases, so that different programs compute slightly different sets of intersecting pairs. This pilot project ignores that; in other programs, we address it by computing with big rational numbers that use sufficient digits to store computation results exactly [12, 13, 22].

The naive quadratic-time algorithm to find intersecting pairs of cubes would simply test all pairs. We use a *uniform grid* data structure to filter for subsets of cubes that probably intersect [1, 6, 8, 11, 15, 17, 20, 21, 23]. The uniform grid partitions the $1 \times 1 \times 1$ universe into a 3D grid of $g \times g \times g$ grid cells, for some natural number g that is a function of the data. Abstractly, each grid cell will store a set of the cubes intersecting it. The proper concrete realization of this is important; we use a *ragged array*. All the cells' contents are stored catenated in one array, $\boldsymbol{a}$. A second array, $\boldsymbol{b}$, called the *dope vector*, with $g^3 + 1$ elements, points to the start of each cell's contents in the first array. So, the i-element of the j-th cell is

$$\boldsymbol{a}_{i+\boldsymbol{b}_j}$$

and the j-th cell contains $\boldsymbol{b}_{j+1} - \boldsymbol{b}_j$ elements. While constructing the dope vector, we will temporarily use an auxiliary array with the same size as the total number of *(cell, cube)* intersections, storing, for each intersection, the id of the cell intersecting the cube.

Barbieri et al. [3] uses fast atomic operations to implement linked lists for the contents of the cells. Linked lists are dynamic, and permit elements to be inserted or deleted at any time. The extra space they require could be compressed with CDR coding [14], but that seems incompatible with SIMT. In contrast, once a ragged array's dope vector is computed, it is static, with most changes requiring $\theta(n)$ time. However, it uses less memory, and can be accessed in constant time. ParCube exploits that to assign array elements in parallel.

For the uniform grid, we will choose $g = \lfloor 0.9999/l \rfloor$. This ensures that each cube will intersect at most a block of $2 \times 2 \times 2 = 8$ grid cells. In the unusual case that a cube does not intersect the whole block, we will assume that it does. This will later cause a few extra cube-cube tests, but the intersecting pairs will still be correctly computed.

The precise choice of g is not critical, within perhaps 50%. As discussed in the cited papers, which also give experimental results, parts of the algorithm run slower and other parts faster, making the total time rather stable with varying g.

One misconception about the uniform grid needs to be addressed. It is not a voxelization of the universe. The data is not discretized. If the input is kept constant, changing g will not change the output, but only the compute time.

For independently and identically distributed (i.i.d.) input, the uniform grid reduces the expected execution time to linear in the size of the input plus output [9]. This is still true if the density of the input satisfies a Lipschitz condition, where the densest region is a constant multiple of the average density, as the number of cubes increases. Two major parts of the proof of the execution time are as follows.

(1) The expected number of possibly intersecting cube pairs that need testing is the mean of the square of the number of cubes in each cell. However when the cubes are i.i.d, the number of cubes in a cell is a Poisson random variable, so the mean of the square is the square of the mean.
(2) Because of the constant ratio between the cell size and the cube size, the probability that a pair of cubes that both intersect the same cell intersect each other in the cell is constant.

The result is that the expected time spent testing cube pairs for intersection is linear in the number of pairs that are found to intersect.

A grid cell id is a 4-byte integer formed from the x, y, and z indices of the cell in the grid as follows: $x \cdot g^2 + y \cdot g + z$. (We assume that $g \leq 1000$.) The algorithm's strategy is as follows.

3.1 Strategy

(1) Compute a ragged array of the cubes overlapping each grid cell, as follows.
   (a) Compute a vector of the *(cell, cube)* pairs, ordered by cube id. We will iterate over the cubes. Because we assume that each cube overlaps 8 cells, we know each particular pair's location in the vector, and so can do this in parallel.
   (b) Sort that vector by cell id and compact it to produce the ragged array of cubes in cells.
   (c) Note that this works quite well regardless of the different numbers of cubes in the various grid cells.
(2) Compute a vector of *(cube, cube)* pairs, listing the potential intersections.
   (a) Given the number of cubes in each cell, we can compute the number of *(cube, cube)* pairs in each cell.
   (b) Note that there is a lexicographic ordering of all combinations of 2 objects selected from a set of k objects. We can determine the 2 objects forming the i-th combination of the $\binom{k}{2}$ combinations.
   (c) Thus we can build the vector of *(cube, cube)* pairs in parallel.
(3) Finally, in parallel, test whether each *(cube, cube)* pair actually intersects.

The challenge was to realize that high-level algorithm in terms of Thrust functions. The detailed algorithm goes as follows.

3.2 Details

(1) Number the input points in order, so that the vector of point ids is $(0, 1, 2, 3, \cdots)$.

   Note. *Note that, in 1D, coordinate x will fall into grid cell $\lfloor x \cdot g \rfloor$.*

(2) Compute the vector of the points' cell ids, $\boldsymbol{bp}$, from the point coordinates: $\boldsymbol{bp_i} = \lfloor \boldsymbol{x_i} \cdot g \rfloor \cdot g^2 + \lfloor \boldsymbol{y_i} \cdot g \rfloor \cdot g + \lfloor \boldsymbol{z_i} \cdot g \rfloor$.

   Note. *Thrust* transform *maps a function onto every element of a vector, or onto every pair of corresponding elements from two vectors. One vector may be an index vector* $(0, 1, 2, 3, \cdots)$. *That would not be stored explicitly; each element would be generated on demand with a* make_counting_iterator *function.*

(3) Compute vector $\boldsymbol{r}$ to repeat each point id 8 times, i.e., $\boldsymbol{r_i} = \lfloor i/8 \rfloor$ with *transform*. Since each cube intersects 8 cells, $\boldsymbol{r}$ will be the indices of the cubes in the list of *(cell, cube)* pairs. E.g.,

$$\boldsymbol{r} = (0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, \cdots)$$

(4) Compute

$$\boldsymbol{\alpha} = (0, 1, g, g+1, g^2, g^2+1, g^2+g, g^2+g+1)$$

the relative ids of the 8 grid cells intersecting cube i, offset from cell bp_i containing the lower left front corner.

(5) Compute vector $\boldsymbol{c1}$ as $\boldsymbol{bp}$ with each element repeated 8 times with *gather*. That is, $\boldsymbol{c1}_i = \boldsymbol{bp}_{\lfloor i/8 \rfloor}$. E.g., if $\boldsymbol{bp} = (3, 8, 2, \cdots)$ then

$$\boldsymbol{c1} = (3, 3, 3, 3, 3, 3, 3, 3, 8, 8, 8, 8, 8, 8, 8, 8, 2, 2, \cdots)$$

(6) Compute vector $\boldsymbol{c2}$ as $\boldsymbol{\alpha}$ catenated together sufficiently many times; $\boldsymbol{c2}_i = \boldsymbol{\alpha}_{\ \bmod (i,8)}$ with a *gather*. E.g.,

$$\boldsymbol{c2} = (0, 1, 5, 6, 25, 26, 30, 31, 0, 1, 5, 6, 25, 26, 30, \cdots)$$

(7) Compute $\boldsymbol{cubes} = \boldsymbol{c1} + \boldsymbol{c2}$ (elementwise) with a *transform*. $\boldsymbol{cubes}$ is the list of indices of the cubes intersecting the cells in the list of *(cell, cube)* pairs. E.g.,

$$\boldsymbol{cubes} = (3, 4, 8, 9, 28, 29, 33, 34, 8, 9, 13, 14, 33, 34, \cdots)$$

(8) Bring together all the cubes in each cell thus: Sort $\boldsymbol{r}$ and $\boldsymbol{cubes}$ by $\boldsymbol{r}$ together, i.e., find and apply the permutation of $\boldsymbol{r}$ that orders it, and apply the same permutation also to $\boldsymbol{cubes}$.

NOTE. *Thrust* lower_bound($\boldsymbol{a}$,v) *binary-searches the ordered vector* $\boldsymbol{a}$ *and returns where to insert* v *to keep the result ordered.* lower_bound($\boldsymbol{a}$,$\boldsymbol{v}$) *binary-searches the ordered vector* $\boldsymbol{a}$ *to see where to insert each* $\boldsymbol{v}_i$*. This parallelizes very well.*

(9) Compute $\boldsymbol{dc}$, a dope vector showing where in $\boldsymbol{cubes}$ each new cell's cubes start with *lower_bound* on $\boldsymbol{r}$ and an index vector $(0, 1, 2, \cdots)$. The j-th cube intersecting cell i will be $\boldsymbol{cubes}_{\boldsymbol{d}_i+j}$.

(10) Free $\boldsymbol{r}$ since it won't be needed again.

(11) Compute vector $\boldsymbol{nc}$, the number of cubes in each cell, by applying a Thrust *adjacent_difference* function to $\boldsymbol{dc}$. That is,

$$\boldsymbol{nc}_i = \boldsymbol{dc}_{i+1} - \boldsymbol{dc}_i$$

Now we have a ragged array of the cubes in each cell. We have computed it in parallel with a few Thrust function calls. □

Assuming that the vectors now look like this:

$$\boldsymbol{dc} = (0, 2, 3, 3, 5, 7, 8, 10)$$

$$\boldsymbol{cubes} = \boxed{1, 2}\boxed{3}\boxed{\ }\boxed{4, 5, 6}\boxed{7}\boxed{8, 9, 10}$$

$$\boldsymbol{nc} = (0, 2, 1, 0, 3, 1, 3)$$

the harder part is forming the set of potentially intersecting pairs of cubes to produced this output:

$$\boldsymbol{pairs} = \boxed{(2, 1)}\boxed{(5, 4), (6, 4), (6, 5)}\boxed{(9, 8), (10, 8), (10, 9)}$$

(12) Compute vector $\boldsymbol{np}$, the number of pairs of cubes in each cell, by mapping a $\binom{k}{2}$ *choose* function over $\boldsymbol{nc}$ with a *transform*. E.g.,

$$\boldsymbol{np} = (0, 1, 0, 0, 3, 0, 3)$$

(13) Compute ntp, the total number of cube pairs by reducing $\boldsymbol{np}$. That is, $ntp = \sum_i \boldsymbol{np}_i$. E.g., $ntp = 7$.

(14) Allocate $\boldsymbol{pairs}$, a vector of size ntp of pairs of cube numbers, to hold the pairs of cubes.

NOTE. *Thrust* exclusive_scan *computes partial sums. If* $\boldsymbol{b}$ = exclusive_scan($\boldsymbol{a}$) *then* $\boldsymbol{b}_i = \sum_{j=0}^{i-1} \boldsymbol{a}_j$.

(15) Compute $\boldsymbol{dp}$, a dope vector for $\boldsymbol{pairs}$, by applying the *exclusive scan* function to $\boldsymbol{np}$. Cube pair j in cell i will be $\boldsymbol{pairs}_{\boldsymbol{dp}_i+j}$. E.g., $\boldsymbol{dp} = (0, 1, 1, 1, 4, 4, 7)$

(16) Compute a lexicographically ordered list $\boldsymbol{u}$ of $\binom{m}{2}$ ordered pairs, where m is the maximum number of cubes in any cell, i.e.,

$$(1, 0), (2, 0), (2, 1), (3, 0), (3, 1), (3, 2), (4, 0), (4, 1), \cdots$$

Implementation choices not taken include:

(a) Precompute this and compile it into the source, so ParCube would have contained no explicit loops. However, the execution time saved would be insignificant.

(b) Code an explicit function computing this. However such a function requires a *sqrt* and is slower to execute than a run-time table lookup.

(17) Compute $\boldsymbol{h0}$, a run length expansion of $\boldsymbol{dp}$ using $\boldsymbol{np}$ as the run lengths. That is, in $\boldsymbol{h0}$, $\boldsymbol{dp}_i$ is repeated $\boldsymbol{np}_i$ times. E.g., $\boldsymbol{h0} = (1, 4, 4, 4, 7, 7, 7)$.

NOTE. *Thrust* exclusive scan by key *extends the* exclusive_scan *with an additional argument, a key vector. It does an exclusive scan separately for each range of consecutive equal keys. With the value vector* $(1, 1, 1, 1, 1, 1, \cdots)$ *and the key vector* $(2, 2, 2, 3, 6, 6)$*, the result is* $(0, 1, 2, 0, 0, 1, \cdots)$.

(18) Compute $\boldsymbol{h1}$ with an *exclusive scan by key* with key vector $\boldsymbol{h0}$ and value vector a list of 1. Now, within each cell, $\boldsymbol{h1}$ will index the pairs, from 0 up.
Let $\boldsymbol{u0}$ be the vector containing the first element of each pair of $\boldsymbol{u}$, and $\boldsymbol{u1}$ the vector of the second elements of each pair.

(19) The following steps will compute $\boldsymbol{pairs}$, a list of pairs of possibly intersecting cubes. We will describe the process for computing the vector of the ids of the first cube in each pair; the vector of second cube ids is analogous.
Use *transform* and *gather* to combine vectors $\boldsymbol{h0}$, $\boldsymbol{h1}$, and $\boldsymbol{u0}$ into vector $\boldsymbol{pairs0}$ as follows. $\boldsymbol{pairs0}_i = \boldsymbol{cubes}_{\boldsymbol{h0}_i+\boldsymbol{u0}_{\boldsymbol{h1}_i}}$

(20) Similarly, compute $\boldsymbol{pairs1}$, the second component of each pair.

(21) Combine $\boldsymbol{pairs0}$ and $\boldsymbol{pairs1}$ into a vector of cube pairs $\boldsymbol{pairs}$.

NOTE. *Thrust* remove_if *compacts a vector by removing elements satisfying a given predicate. Like all Thrust functions, this executes efficiently in parallel.*

(22) Use *remove_if* to remove cube pairs that don't actually intersect from $\boldsymbol{pairs}$. (This could be made pure FP with a function to copy the vector and remove elements from the copy. It still parallelizes well.)

(23) Sort $\boldsymbol{pairs}$.
This and the previous step were done in that order because computation is faster than I/O on the GPU, and the previous sorting step is often the slowest step.

NOTE. *Thrust* unique *removes duplicate pairs from a sorted vector.*

(24) Use *unique* to remove duplicate pairs. They occurred when the same two intersecting cubes both intersected the same two (or more) cells.

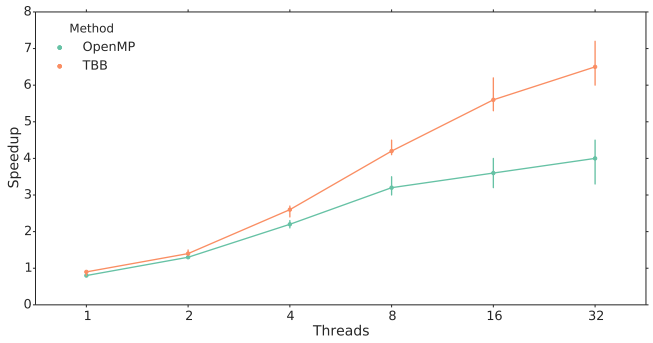

Figure 1: Parallel speedup with OpenMP and TBB

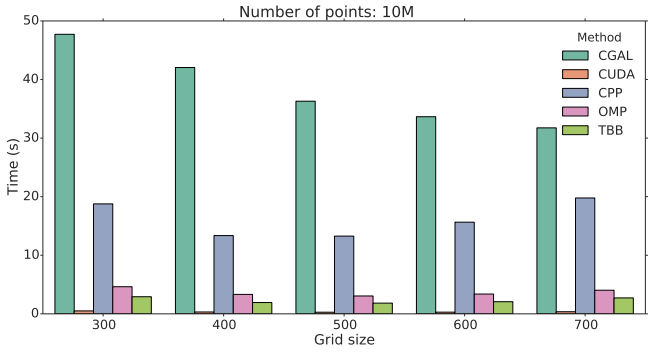

Figure 2: Times for 10M cubes of different implementations and different grid sizes

We have now computed the unique pairs of intersecting cubes. □

4 IMPLEMENTATION AND TESTING

We implemented ParCube in C++ using g++-7 and Thrust 1.8 on a dual 8-core 3.1 Ghz Intel E5-2687W Xeon CPU with 128GB of main memory running Ubuntu linux. However nothing in ParCube depends on those specifics.

We tested four different backends:

(1) regular single-threaded C++ on the Xeon,
(2) OpenMP (OMP), using 1, 2,4, 8, 16, or 32 threads on the Xeon,
(3) Intel Intel Threading Building Blocks (TBB), using 1, 2,4, 8, 16, or 32 threads on the Xeon, and
(4) CUDA 8.0 on an Nvidia GeForce GTX Titan X GPU.

We validated ParCube as follows. One of us implemented ParCube. The other implemented another program using CGAL to solve the same problem. We compared the number of intersections found by the two programs for each test data set, and, for some tests, also compared the lists of intersections. After debugging, everything matched.

The essence of ParCube is under 200 lines of code. It is freely available for nonprofit research and education. The authors welcome external questions, comments, and feedback. To simplify the programming, ParCube was written to process a list of axis-aligned cubes all of the same size, but the concepts generalize. Our test cases were random uniform i.i.d lists of cubes. We tested up the largest cases that would fit in the GPU. For each number of input cubes, we tested several different cube sizes. Our performance metric was wall clock elapsed time for the computation, starting after the data had been read in. CPU time is not widely used for benchmarking parallel programs.

Our largest tests are shown in Table 1 on the following page. Each reported time is the average of five runs. Some highlights are as follows.

(1) ParCube was up to 131× faster than CGAL.
(2) ParCube's relative performance was usually better on the harder cases: larger datasets, and data with more intersections.
(3) Even running single-threaded with no special backend, ParCube was usually faster than CGAL, and for 10M cubes, always faster, by up to a factor of three.
(4) When using the Intel-specific TBB, one thread was slower than not using TBB, but 32 threads ran about 7× faster than single-threaded C++.
(5) The more generic OpenMP did not parallelize as well, and ran about 4× faster on 32 threads.
Figure 1 shows the parallel speedup for TBB and OpenMP.
(6) One reason that multithreading exhibits less than a linear speedup is that the Xeon overclocks itself when running fewer threads. Indeed, power consumption in modern processors depends on the number of computations being performed each second. That number is totaled over all the threads running on the processor. A processor can measure its own temperature. When it gets hot, it lowers its clock speed to reduce its power consumption. In contrast, when only one thread is running, the processor's clock speed may be considerably higher than its official speed. That is, the processor can automatically overclock itself. This is nice, but does complicate timing tests.
(7) ParCube ran best on the GPU: up to 54× faster than the single-threaded CPU version, and 131× faster than CGAL.
(8) On small datasets, say 100K cubes with 41K unique intersections, it ran fast enough that the time (0.01 secs) was hard to measure.

Table 2 gives the times for various components of the test run with $N = 10M$, $G = 400$, $L = 0.0025$, which found $6M$ unique intersections in 0.325 elapsed seconds on the GPU. (The total appears slightly off because each component time was rounded for display.) The lines include references to the steps in Subsection 3.2. If a step is not listed here, its time is bundled together with the immediately following step that is listed, so that the total time is correct. There is nothing special about this particular test case. Different test cases can have different relative times, especially when they have different numbers of intersections. The relative performance CGAL and the various backends of ParCube also depends on the grid size; see Figure 2, which shows $N = 10M$. As always, the cube size is slightly less than the grid cell size.

5 CONCLUSION AND FUTURE POSSIBILITIES

ParCube admits of many possible extensions.

Table 1: ParCube vs CGAL performance

Times are wall-clock seconds starting after data is read in. Speedup compares CGAL to ParCube GPU.

#	Grid	Edge	#	#Xsect	#Uniq	CGAL	ParCube													Our
							GPU	C++	OpenMP						TBB						
Pts	Size	Len	Pairs	Pairs	Pairs	≀	≀	≀	1 th.	2 th.	4 th.	8 th.	16 t.	32 t.	1 th.	2 th.	4 th.	8 th.	16 t.	32 t.	Speed up
10K	100	0.01	3K	1K	416	0.01	0.01	0.04	0.05	0.03	0.02	0.02	0.02	0.03	0.04	0.03	0.02	0.02	0.02	0.02	1.29
100K	100	0.01	326K	139K	41K	0.2	0.01	0.11	0.13	0.09	0.06	0.04	0.04	0.07	0.12	0.07	0.05	0.04	0.04	0.04	14.41
100K	200	0.005	42K	18K	5K	0.15	0.01	0.33	0.41	0.23	0.14	0.1	0.09	0.1	0.38	0.21	0.12	0.08	0.07	0.06	10.55
100K	300	0.003 33	19K	5K	2K	0.13	0.02	0.94	1.12	0.62	0.38	0.26	0.23	0.21	1.08	0.6	0.34	0.21	0.16	0.15	5.42
1M	100	0.01	32M	14M	4M	4.46	0.1	3.14	3.72	2.43	1.43	1.06	0.98	0.89	3.41	2.18	1.25	0.74	0.59	0.52	44.48
1M	200	0.005	4M	2M	512K	3.03	0.04	1.12	1.35	0.89	0.55	0.39	0.36	0.37	1.42	0.78	0.44	0.27	0.21	0.19	84.64
1M	300	0.003 33	2M	513K	152K	2.45	0.04	1.63	1.89	1.16	0.71	0.51	0.45	0.4	1.97	1.04	0.56	0.35	0.27	0.24	62.6
1M	400	0.0025	1M	216K	64K	2.2	0.06	2.89	3.45	2.19	1.17	0.82	0.69	0.61	3.32	1.93	0.97	0.6	0.46	0.39	35.8
10M	300	0.003 33	185M	51M	15M	47.72	0.5	18.77	22.01	14.1	8.9	6.08	5.28	4.63	21.86	14.36	7.78	4.61	3.47	2.92	96.21
10M	400	0.0025	128M	22M	6M	42.05	0.33	13.36	15.78	10.54	6.31	4.34	3.76	3.32	15.62	10.02	5.62	3.28	2.39	1.93	131.83
10M	500	0.002	100M	11M	3M	36.3	0.28	13.27	15.83	10.58	6.14	4.13	3.45	3.05	15.52	9.42	5.25	3.09	2.15	1.83	130.92
10M	600	0.001 67	83M	6M	2M	33.65	0.29	15.65	19.23	11.46	6.79	4.5	3.91	3.38	18.56	11.33	5.98	3.49	2.43	2.07	114.23
10M	700	0.001 43	70M	4M	1M	31.75	0.36	19.78	23.53	14.73	8.61	5.65	4.59	4.03	23.72	14.61	7.46	4.36	3.1	2.72	88.82

Table 2: Component times for the $N = 10M$, $G = 400$, $L = 0.0025$ test

Item # refers to the list in Subsection 3.2.

Component	Item #	Time (s)
Gather	5	0.002
Sort (cell,cube) pairs	8	0.076
Lower bound	9	0.021
Adjacent difference	11	0.004
Enumerate combinations	16	0.007
Decode run length	17	0.040
Compute h1	18	0.024
Compute pairs1 and 2	19	0.014
Form pairsc	21	0.013
Sort pairs	23	0.120
Unique pairs	24	0.006
Total		**0.325**

(1) To start, extend it to handle cubes of different sizes.
(2) Finding *bipartite* intersections would also be easy. Here, cubes are painted red or blue, and we want only the red–blue intersections. When there are few red–blue intersections but many unwanted red–red and blue–blue intersections, sweep-line algorithms are inefficient. (The problem is that they must detect and discard the unwanted intersections, or, alternatively, they must preprocess either the set of red-red intersections or the set of blue-blue intersections.) However a uniform grid's expected time is still linear in the expected number of red–blue intersections. ParCube would need to be modified as follows.
   (a) In each cell, count n_r the number of red cubes and n_b the number of blue cubes. The maximum possible number of cube-cube pairs in that cell will be $n_r \cdot n_b$.
   (b) Reserve space in each cell for that many pairs.
   (c) Instead of iterating through all $\binom{n_r+n_b}{2}$ combinations of cubes in each cell, iterate through all $\binom{n_r}{2}$ combinations of red cubes and all $\binom{n_b}{2}$ combinations of blue cubes, writing pairs of a red cube and a blue cube into the cube-cube vector.

   Note. *In this context, as throughout ParCube, the abstract operation* iterating *is implemented with parallel Thrust functions.*

   (d) Test each red-blue cube pair for actual intersection.
(3) Implement more powerful algorithms, such as computing boolean combinations, and mass properties of boolean combinations, of many objects, whose individual properties are graded, i.e., vary over space. Building on [10, 18], this would be very useful in additive manufacturing.
(4) Extend ParCube to process datasets in higher dimensions, such the 7D that some robotic planning occurs in.
(5) Optimize ParCube at a lower level by fusing separate Thrust calls to reduce the quantity of data processed, using alternative libraries when indicated (e.g., on a CPU, the parallel STL sort is much faster than Thrust's sort), profiling, and finally, recoding in a lower level tool, perhaps CUB.
(6) Partition datasets too large to fit in the GPU's memory into chunks, each small enough to fit. This would effectively be constructing a 2-level grid, with each coarse cell small enough to fit in the GPU. There would need to be a final step to remove duplicate intersecting pairs, where the two copies occur in different coarse cells.

The most general lesson from this work is that simple 3D geometric algorithms expressed with uniform grids and functional programming concepts can process large datasets quite efficiently on either GPUs or multicore CPUs.

This research was partially supported by CAPES (Ciencia sem Fronteiras - grant 9085/13-0), CNPq.

REFERENCES

[1] Varol Akman, Wm Randolph Franklin, Mohan Kankanhalli, and Chandrasekhar Narayanaswami. 1989. Geometric Computing and the Uniform Grid Data Technique. *Computer Aided Design* 21, 7 (1989), 410–420.

[2] Momme Allalen, Vali Codreanu, Nevena Ilieva-Litova, Alan Gray, Anders Sjöström, Volker Weinberg, Maciej Szpindler, and Alan Gray. 2017. Best Practice Guide GPGPU, January 2017. http://www.prace-ri.eu/best-practice-guide-gpgpu-january-2017/, (retrieved 2017-08-29). (26 Jan 2017).

[3] Davide Barbieri, Valeria Cardellini, and Salvatore Filippone. 2014. Fast uniform grid construction on gpgpus using atomic operations. *Parallel Computing: Accelerating Computational Science and Engineering (CSE). ADVANCES IN PARALLEL COMPUTING* 25 (2014), 295–304.

[4] Jon Louis Bentley and Derick Wood. 1980. An Optimal Worst Case Algorithm for Reporting Intersections of Rectangles. *IEEE Trans. Computers* 29 (1980), 571–577.

[5] Computational Geometry Algorithms Library. 2003. CGAL. http://www.cgal.org/ (retrieved 27-April-2008). (2003).

[6] Salles V. G. de Magalhães, W. Randolph Franklin, Wenli Li, and Marcus V. A. Andrade. 2014. Fast map generalization heuristic with a uniform grid. In *22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (ACM SIGSPATIAL 2014)*. Dallas, Texas, USA.

[7] Herbert Edelsbrunner. 1983. A new approach to rectangle intersections part I. *International Journal of Computer Mathematics* 13, 3-4 (1983), 209–219.

[8] Wm Randolph Franklin. 1984. Adaptive Grids for geometric operations. *Cartographica* 21, 2–3 (Summer – Autumn 1984), 161–167. monograph 32–33.

[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] Wm Randolph Franklin, Narayanaswami Chandrasekhar., Mohan Kankanhalli, Manoj Seshan, and Varol Akman. 1988. Efficiency of uniform grids for intersection detection on serial and parallel machines. In *New Trends in Computer Graphics (Proc. Computer Graphics International'88)*, Nadia Magnenat-Thalmann and D. Thalmann (Eds.). Springer-Verlag, 288–297.

[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] Maurício G. Gruppi, Salles V. G. Magalhães, Marcus V. A. Andrade, W. Randolph Franklin, and Wenli Li. 2016. Using Rational Numbers and Parallel Computing to Efficiently Avoid Round-Off Errors on Map Simplification. *RBC. Revista Brasileira de Cartografia (Online)* 68 (2016), 1221–1230. online at http://www.lsie.unb.br/rbc/index.php/rbc/article/view/1857.

[14] David Gudeman. 1993. *Representing type information in dynamically typed languages*. Technical Report 93-27. Dept of Computer Science, U Arizona.

[15] David Hedin and W. Randolph Franklin. 2016. NearptD: A Parallel Implementation of Exact Nearest Neighbor Search using a Uniform Grid. In *Canadian Conference on Computational Geometry*. Vancouver Canada.

[16] Intel. 2017. Threading Building Blocks. https://www.threadingbuildingblocks.org/, (retrieved 2017-10-07). (2017).

[17] Mohan Kankanhalli. 1990. *Techniques for Parallel Geometric Computations*. Ph.D. Dissertation. ECSE Dept, Rensselaer Polytechnic Institute.

[18] Mohan Kankanhalli and Wm Randolph Franklin. 1995. Area and Perimeter Computation of the Union of a Set of Iso-Rectangles in Parallel. *J. Parallel Distrib. Comput.* 27 (1995), 107–117.

[19] Shih-Hsiang Lo, Che-Rung Lee, I-Hsin Chung, and Yeh-Ching Chung. 2013. Optimizing Pairwise Box Intersection Checking on GPUs for Large-Scale Simulations. *ACM Trans. Model. Comput. Simul.* 23, 3, Article 19 (July 2013), 19:1–19:22 pages.

[20] Salles V. G. Magalhães, Marcus V. A. Andrade, W. Randolph Franklin, and Wenli Li. 2015. Fast exact parallel map overlay using a two-level uniform grid. In *4th ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data (BigSpatial)*. Bellevue WA USA.

[21] Salles V. G. Magalhães, Marcus V. A. Andrade, W. Randolph Franklin, and Wenli Li. 2016. PinMesh – Fast and exact 3D point location queries using a uniform grid. *Computer & Graphics Journal, special issue on Shape Modeling International 2016* 58 (Aug. 2016), 1–11. Awarded a reproducibility stamp.

[22] Salles V. G. Magalhães, W. Randolph Franklin, and Marcus V. A. Andrade. 2017. Fast exact parallel 3D mesh intersection algorithm using only orientation predicates. In *25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (ACM SIGSPATIAL 2017)*. Redondo Beach, CA.

[23] Chandrasekhar Narayanaswami. 1991. *Parallel Processing for Geometric Applications*. Ph.D. Dissertation. Electrical, Computer, and Systems Engineering Dept., Rensselaer Polytechnic Institute. UMI no. 92-02201.

[24] Nvidia. 2015. CUDA Toolkit Documentation. online, http://docs.nvidia.com/cuda/thrust/, accessed 2016-02-23. (Aug 2015).

[25] Nvidia. 2017. CUB. https://nvlabs.github.io/cub/, (retrieved 2017-10-07). (2017).

[26] Nvidia. 2017. CUDA Education & Training. https://developer.nvidia.com/cuda-education-training, (retrieved 2017-08-28). (2017).

[27] OpenMP Architecture Review Board. 2016. The OpenMP API specification for parallel programming. online, http://www.openmp.org/, accessed 2016-02-23. (Feb 2016).

[28] Afra Zomorodian and Herbert Edelsbrunner. 2000. Fast Software for Box Intersections. In *Proceedings of the Sixteenth Annual Symposium on Computational Geometry (SCG '00)*. ACM, New York, NY, USA, 129–138.

October 9, 2017