A Voxel-Based Parallel Collision Detection Algorithm

Orion Sky Lawlor
Dept. of Computer Science
University of Illinois at Urbana-Champaign
olawlor@acm.org

Laxmikant V. Kalé
Dept. of Computer Science
University of Illinois at Urbana-Champaign
kale@cs.uiuc.edu

ABSTRACT

Two physical objects cannot occupy the same space at the same time. Simulated physical objects do not naturally obey this constraint. Instead, we must detect when two objects have collided– we must perform collision detection. This work presents a simple voxel-based collision detection algorithm, an efficient parallel implementation of the algorithm, and performance results.

Categories and Subject Descriptors

I.6.m [**Simulation and Modeling**]: Miscellaneous—*Collision Detection*

General Terms

Algorithms

Keywords

parallel geometry, collision detection, contact

1. INTRODUCTION

When using CAD/CAM to design a machine, we must ensure the simulated machine parts never pass though one another. If the simulated parts collide, we must correct the design.

In computer graphics and animation, we often want to ensure that objects behave in a physically plausible way. This means checking if the simulated objects penetrate one another– if they do, the modeling tool or animator will want to know.

In motion planning for robotics, we must check if a robot arm will collide with anything as it executes a proposed command. The command will have to be modified if a collision is possible.

In simulating a car crash or tearing metal, at each timestep we must check if any objects intersect. If they do, we must deform or displace the objects.

Collision detection, also known as *contact* or interference detection, is the problem of determining whether a given set of objects overlap. If the objects overlap, in some situations we need to find

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICS'02, June 22-26, 2002, New York, New York, USA.
Copyright 2002 ACM 1-58113-483-5/02/0006 ...$5.00.

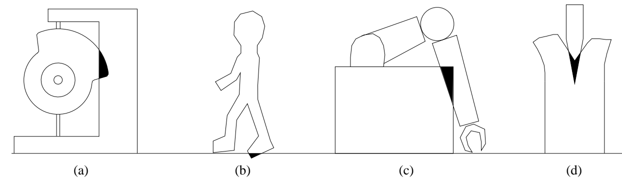

Figure 1: Collision detection for: (a) CAD/CAM (b) Computer animation (c) Robotics (d) Structural simulation

exactly which parts of the objects overlap and the depth of the overlap region. The problem can be formulated in 2D, 3D, or higher dimensions; can vary with time; and, as shown above and in Figure 1, is relevant to many domains.

1.1 Summary

In this work, we present a scalable high-level parallel solution to a large subclass of collision detection problems. Our approach is to divide space into a sparse grid of regular axis-aligned voxels distributed across the parallel machine. Objects are then sent to all the voxels they intersect. Once all the objects have arrived, each voxel becomes a self-contained subproblem, which is then solved using standard serial collision detection approaches. This voxel-based approach efficiently and naturally separates many objects that cannot ever collide, by placing them in separate voxels. Simultaneously, voxels bring together adjacent objects that may intersect.

In theory, the basic voxel algorithm works for any object type or serial collision detection method. In practice, the method works best when there is not too much scale disparity– objects much larger than a voxel will be sent to several voxels, wasting space and time; while structures much smaller than a voxel must be dealt with entirely via the serial approach. Further, the serial collision detection method used must be able to work with only a subset of the problem's objects– the subset that lies within the voxel.

For problems which conform to these limitations, however, the serial and parallel performance of the voxel scheme is excellent.

1.2 Prior Work

Many researchers, from many domains, have addressed the problem of collision detection. Following Lin and Gottschalk in [**?**], we can separate out two main areas of work: n-body collision detection, the "broad phase" which determines which pairs among n objects may intersect; and pairwise collision detection, the "narrow phase" which decides whether a single pair of objects collide. Any n-body algorithm depends on a pairwise algorithm to perform the final tests.

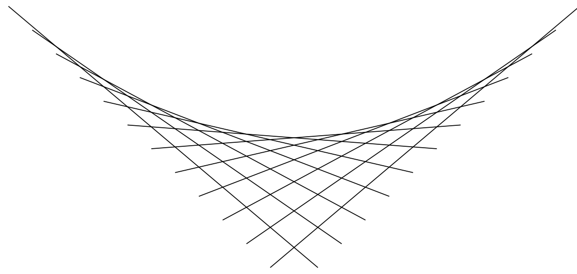

Figure 2: n **line segments arranged to have** $\binom{n}{2}$ **collisions**

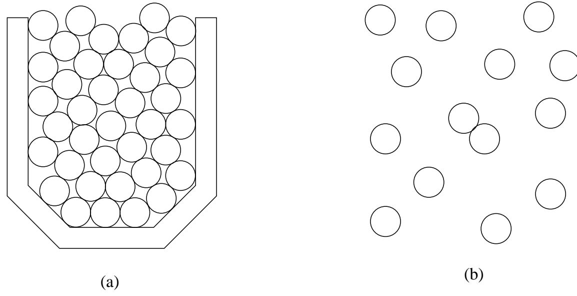

Figure 3: n **circles; (a) with** $O(n)$ **collisions; (b) with** $O(1)$ **collisions**

Finally, although the two problems are distinct, the common and literature usage of the term "collision detection" can refer to work in either n-body or pairwise detection. The voxel method, whose presentation forms the bulk of this work, addresses the n-body problem.

The basic problem of pairwise collision detection is to determine whether two given objects collide. Farouki et al [**?**] show how to do pairwise intersections for quadrics. Cameron [**?**] defines "S-Bounds" to perform pairwise intersections for constructive solid geometry. The famous papers by Lin and Canny [**?**], and Gilbert, Johnson, and Keerthi [**?**], use convex optimization techniques to intersect a pair of convex polytopes in time logarithmic in the number of vertices. This last result is sometimes misinterpreted to mean the n-body collision detection problem can be solved in logarithmic time, which is not the case.

1.3 n-body collision detection

Given an algorithm to determine if a given pair of objects collide, the obvious way to determine which of n objects collide is simply to try all $\binom{n}{2}$ possible pairs. Because $\binom{n}{2} = O(n^2)$, this approach requires $O(n^2)$ pair tests.

In fact, *any* correct n-body collision detection algorithm has a worst-case time in $O(n^2)$. This is because it is possible to arrange n objects (see Figure 2) so that every object collides with every other object, for $O(n^2)$ separate collisions.

However, because physical objects cannot pass though one another, nearly any plausible physical situation actually has at most $O(n)$ collisions, as in Figure 3(a). Many situations found in practice have as few as $O(1)$ collisions, as shown in Figure 3(b).

Since most problems have only a few collisions, it is quite often possible to do collision detection in faster than $O(n^2)$ time. There are several ways to do this, each with their own advantages. Hubbard [**?**] points out that most collision detection research ignores the time dependence of collision problems. Although several methods for fully four dimensional "spacetime" collision detection are known, and described in Jiménez's survey [**?**], a standard approach is to sweep out or ignore time dependence and treat each timestep or frame of the simulation as a conceptually separate problem. For rigid objects with bounded velocities and accelerations, the collision scheduling method of [**?**] can be effective.

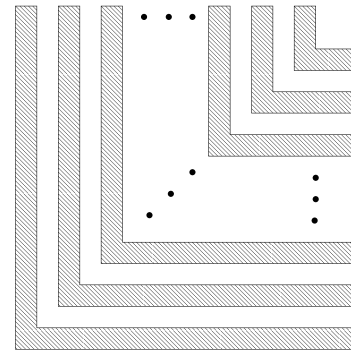

Figure 4: n **nested 'L' figures do not collide, but every bounding box intersects every other bounding box.**

Most other n-body approaches use some sort of bounding volume. Bounding volumes, although shown quite effective in practice, are subject to rather unlikely worst cases, such as the nested 'L' shapes shown in Figure 4. Suri, Hubbard, and Hughes [**?**] show that if the object aspect ratio and "scale factor"[1] are bounded, then bounding boxes introduce no more than a linear amount of overhead. Zhou and Suri [**?**] extend this result to include bounded *average* aspect ratio and scale factor. These theoretical results confirm the practical effectiveness of bounding boxes.

There are two major approaches to n-body collision detection. In object subdivision, objects are divided into parts that cannot intersect. Hubbard [**?**] divided objects using bounding spheres; Gottschalk et al. [**?**] examine oriented bounding boxes; Klosowski et al. [**?**] use discrete-orientation polytopes (k-DOPs); and Krishnan et al. [**?**] use a higher-order bounding volume. By contrast, with spatial subdivision, space itself is divided up to separate objects. The voxel method is the simplest spatial subdivision scheme, although there are others.

Brown, Attaway, Plimpton, and Hendrickson [**?**] use a parallel recursive coordinate bisection (RCB) spatial subdivision scheme to perform collision detection in a transient dynamics application. The parallel implementation of RCB requires at least $O(\lg p)$ synchronized communication steps, however, so their approach has a rather high synchronization overhead. However, theirs is scalable, production-level work, and should be considered a proven alternative to the voxel method.

Several other spatial subdivision schemes have been described, such as the sweep-line method, binary space partition [**?**], octrees [**?**] and methods derived from ray tracing. By reduction to sorting, however, any subdivision scheme based on comparing object locations will require $\Omega(n \lg n)$ time; with parallel implementations that require multiple synchronization steps. The voxel method, described next, in some situations achieves $O(n)$ performance with excellent parallel efficiency.

[1] Ratio of the volume of the bounding boxes of the smallest and largest object.

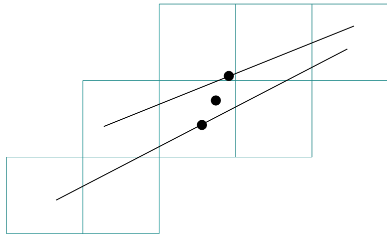

Figure 5: Large objects may span several voxels. The dots show the centers of the two lines and the midpoint between centers.

2. THE VOXEL METHOD

The n-body, spatial subdivision approach we examine in the remainder of this work is the *voxel method*. A regular, axis-aligned grid of voxels is imposed over the problem domain. Each object is added to all the voxels it touches, then each voxel's list of objects is collided separately. Any collision detection subscheme may be used to detect collisions within a voxel, from the simple all-pairs test to a recursive application of the voxel method.

We are deliberately vague about exactly what an "object to be collided" means in this context, because the exact type of object is completely irrelevant. The voxel method is exactly the same and works almost equally well whether colliding boxes, triangles, tetrahedra, hexahedral finite elements, polyhedra, spheres, NURBS surface patches, or even complicated composite objects. Of course, any implementation has to pick some representation; two fairly popular, general choices are bounding boxes and triangles.

The voxel method was first described by Turk [**?**] for molecules, but his approach hashed the grid locations into buckets, ignoring hash collisions, and used the naive all-pairs algorithm on each bucket. Zyda et al. [**?**] describe the NPSNET system, which uses a parallel 2D dense grid structure quite similar to the voxel method, but Zyda's implementation was limited to simple vehicle/vehicle and vehicle/terrain interactions, and also used the naive all-pairs approach within each grid cell.

When the objects to be collided are of nearly uniform size, the voxel method is ideal and gives excellent results. For objects much smaller than a voxel, the division into voxels does not help much, and the voxel method degenerates to the serial scheme used within a voxel.

Objects much larger than a voxel will be duplicated across many voxels, as shown in Figure 5. In this case, a careful implementation can avoid duplicating collision tests across voxels by using a convention to decide which voxel is responsible for testing these shared objects. For example, only the voxel that contains the midpoint between the object centers need test those objects for collision—see the dots in Figure 5. Even with this optimization, however, objects much larger than a voxel still waste both space and time.

Voxels should be implemented as a sparse grid, via a hash table in most cases. A sparse implementation is not required, but can be a good deal more space and time efficient if the problem domain is not naturally bounded or if objects are distributed nonuniformly. A significant advantage of a sparse grid is the fact that empty voxels never get created, and hence cost nothing.

Although normally implemented in 3D, gridding works well in any number of dimensions, although high-dimensional problems may consume large quantities of memory. The common timestep "sweeping" approach for spacetime collision detection, illustrated in Figure 6, can be seen as a kind of gridding along the time axis of spacetime.

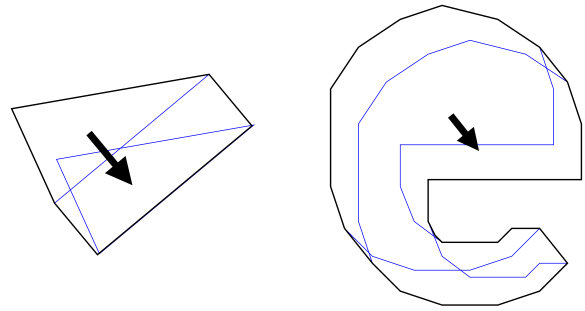

Figure 6: Sweeping objects between timesteps corresponds to a coarse timestep grid in 4D

The voxel approach is easiest to implement via the sloppy strategy of adding each object to all the voxels touched by the object *bounding box*, which may be many more voxels than are touched by the actual object. This sloppiness can be tolerated because any irrelevant objects in a voxel will be ignored by the collision detection subscheme. For elongated objects, whose bounding box does not fit well, this can be quite wasteful; for long and skinny or flat objects, the 3D digital differential analyzer approach of [**?**] could improve performance.

However, since the voxel size is normally chosen so that most objects fit in a single voxel, gridding only the bounding box normally works well. The fast gridding method described in Appendix A is ideal for this case.

3. PARALLEL IMPLEMENTATION

Unlike many other object- or space-division collision detection schemes, the voxel method is naturally[2] parallel. The description of a parallel implementation of the voxel algorithm is given below.

3.1 CHARM++

The sparse voxel grid is implemented on top of the portable parallel runtime system CHARM++, presented in [**?**]. CHARM++ targets both shared- and distributed-memory parallel machines, and runs on machines from workstations to clusters to supercomputers. CHARM++ is a parallel library for C++, but includes bindings for C and FORTRAN90.

A parallel application in CHARM++ consists of a number of relatively small, self-contained *parallel objects*. These objects communicate via remote method invocation, an RPC protocol similar to CORBA or Java RMI, but asynchronous. Since each processor houses several objects, while one object is waiting for data, other local objects can use the CPU. This data-driven approach thus automatically and transparently overlaps communication and computation, leading to better performance and easier application development. For unstructured, dynamic problems like collision detection, this approach is much more useful than the lower-level primitives provided by MPI.

Parallel objects are implemented as ordinary C++ classes. They communicate with other parallel objects using a local *communication proxy* object, another regular C++ class. Proxies are automatically generated from an interface file that lists each classes' remote methods.

In this sense, CHARM++ is quite similar to CORBA. However, CHARM++ targets tightly coupled parallel machines and aims for

[2]The more common but, to us, less descriptive term is "embarrassingly parallel."

extremely efficient communication between thousands of in-process objects. The overhead imposed by CHARM++ is just a few microseconds, rather than the millisecond-scale cost of CORBA.

The CHARM++ array framework, presented by the author in [**?**], supports parallel objects called *array elements* that can be dynamically created and destroyed, participate in reductions and broadcasts, and migrate from one processor to another. Array elements are identified across the parallel machine by an *array index*, which need not be a contiguous range of integers– it can be a sparse user-defined data structure such as a multidimensional location, bitvector, or string.

Since array elements can migrate, CHARM++ can improve load balance by occasionally migrating some objects from heavily loaded processors to less loaded processors. Migration-based load balancing can automatically compensate for application-induced load imbalance, minimize communication volume, handle variation among machines, or accommodate background load. The load balancing system is described in [**?**].

CHARM++ is an excellent foundation for parallel program design, research and implementation. Several major, highly scalable parallel applications have been developed using CHARM++, including [**?**].

3.2 Interface

There are currently two implementations of the voxel algorithm. One implementation uses an axis-aligned bounding box to represent an object; in the other, a triangle is an object.

Triangles are an extremely popular lowest common denominator surface representation, and can be efficiently represented and processed. Details on the specific input format and some triangle-specific optimizations are presented by the author in another work [**?**].

Bounding boxes are more general than triangles in the sense that even a complex object can be collided using only its bounding box, at least at the broad phase. Bounding boxes can also be constructed to contain multiple objects, providing an additional degree of freedom for the user. Of course, bounding boxes may intersect when the actual objects do not, so the bounding box implementation is only the first step. A subsequent "narrow phase" is still needed to check the few possible colliding pairs of objects for any actual intersections.

The input to either implementation of the voxel algorithm is a set of objects—bounding boxes or triangles—presented on each processor. The output is a list of objects that collide; different pieces of this list are returned on every processor.

3.3 The Algorithm

The voxel method's parallel implementation is perhaps most succinctly described by its CHARM++ interface file:

```
module parCollide {
  message objList;

  array [3D] Voxel {
    entry Voxel();
    entry [createhere] void add(objList *);
    entry void startCollision(void);
  };

  group Manager : syncReductionMgr {
    entry Manager(CkArrayID voxels);
    entry void voxelMessageRecvd(void);
  };
};
```

`Voxel` is a grid of voxels– a sparse three-dimensional CHARM++ parallel array of objects. This allows voxels to be created on any processor, receive messages from any processor, and migrate from processor to processor.

`Voxel::add` is called to add a list of objects to the voxel. Because `add` is declared `createhere`, if an `add` message is sent to a non-existent grid location, CHARM++ will create a *new* `Voxel` object on the sending processor to handle the request. `objList`, the "message" parameter to `add`, contains a list of objects to be collided by the voxel.

`Voxel::startCollision` runs the serial collision detection subscheme on all the voxel's accumulated objects. We use a spatial recursive coordinate bisection scheme; although any method can be used. An interesting approach might be to recursively apply the voxel method, especially if many objects lie in a single voxel.

`Manager` is a CHARM++ *group*, a special collection of parallel objects with exactly one object on every processor. The `Manager` coordinates and synchronizes the collision detection computation.

3.4 Steps

The steps in one collision detection computation are as follows.

1. Clients give the local `Manager` their objects to be collided.
2. The local `Manager` determines which voxels each object touches, and calls `Voxel::add` for the appropriate voxels.
3. The `Managers` synchronize, to ensure that all voxels have received all their objects.
4. `startCollision` is broadcast to all voxels, which then run their serial collision detection algorithm.
5. The resulting lists of collisions are returned to the clients.

As usual in CHARM++, because this library's clients are almost always parallel objects themselves, there can be several clients per processor, or perhaps none at all on a particular processor.

In step 2, the local `Manager` actually accumulates the objects to be sent off, then finally sends all the objects for a voxel in a single message. Step 2 is an all-to-all communication step, so this simple "accumulate and send" communication optimization is just one of a variety of optimizations that can be applied. Such communication optimizations are the subject of ongoing research.

The message sends in step 2 actually *create* the voxels, if they haven't already been created. By default, the creation happens on the sending processor– this makes all future communication between this `Manager` and the voxel efficient. If the objects given on a single processor are clustered well, most will never be sent over the network.

The synchronization in step 3 is needed because voxels may receive objects from several processors. Since any processor can send an object to any voxel, "Have all objects been delivered?" is a global property, obtainable only via a global operation. Recall that in CHARM++, however, other local objects will automatically use the CPU during this single synchronization.

After the synchronization, each voxel runs its separate serial collision detection algorithm and reports the results.

3.5 Parallel Details

Unlike in MPI, although messages can be sent to `Voxels`, a `Voxel` is not a processor; nor is a processor responsible for some fixed set of `Voxels`. CHARM++ allows many `Voxels` to share a single processor efficiently—the only limitations are memory usage and bookkeeping overhead. Most programs work best with

between 10 and 10,000 `Voxels` per processor, but `Voxels` can be dynamically created, and often migrate between processors for load balance.

Consider, then, step 2 of our parallel algorithm. A list of objects destined for some `Voxel` with index $(17, 9, -4)$ can be given to CHARM++ on some processor. Somehow, the processor must determine if that `Voxel` already exists somewhere on the parallel machine. If so, the objects need to be sent to the `Voxel`. If not, a new `Voxel` must be created, ideally on the same processor.

Further, objects destined for the same, nonexistent `Voxel` may be given to two different processors at the same time. CHARM++ must resolve this race condition—exactly one of the processors must create a new `Voxel`; and the others' objects must be sent there.

It's easy to imagine using a centralized `Voxel` registry to map grid locations to processors, and to organize `Voxel` creation. However, a centralized registry is is not scalable and would quickly become a serial bottleneck. The natural solution, then, is to use a distributed registry.

This is the approach used by CHARM++. To deliver a message to an unknown `Voxel` location, it computes a "home" processor[3] for that location via a simple hash function. This home processor will either inform the sender of the `Voxel`'s location, or authorize the sender to create a new local `Voxel`. Home processors thus act as a distributed registry to keep track of which `Voxels` exist, and where. Processors cache the location of recently referenced `Voxels`, which means the home processor need not be consulted before every message send.

Home processors also provide an efficient, distributed means to support migrating array elements. The array framework also properly supports broadcasts to all existing `Voxels`, used in step 4 above; as well as reductions across all `Voxels`, used in step 5. The details of how to support these operations even with ongoing migrations are presented by the author in another work [**?**]. Of course, CHARM++'s migratable parallel objects are not a voxel method-specific construct—they are useful, and are used, in many other contexts.

3.6 Load Balancing

The CHARM++ automatic load balancer [**?**] measures the load and communication patterns of each parallel object. It then uses a load balancing "strategy"[4] to determine where each object should be migrated for optimal performance. Performing load balancing automatically at run-time enables an application to react quickly to irregularity in the problem structure.

In fact, since collision detection is often only a small part of a much larger parallel program, perfect load balance for the collision library is actually not necessary. As long as the program is structured to allow the execution of different libraries to interleave (as is always the case in CHARM++), different processors may have different amounts of collision work yet all have the same amount of work overall.

Consider the work done by a parallel machine during an unsynchronized period– that is, between two global synchronizations. In symbols, let A_i and B_i represent the amount of work needed by two libraries A and B on processor i. Assume further that A and B are independent, so they do not depend on one another—a situation that CHARM++supports quite well. Then for load balance we require that each processor have the same amount of work overall:

$$\forall_{i,j}\ A_i + B_i = A_j + B_j \tag{1}$$

This is often ensured by requiring that every library impose the same amount of work on every processor, or:

$$\forall_{i,j}\ (A_i = A_j) \wedge (B_i = B_j) \tag{2}$$

It is clear that 2 implies 1, but 1 does not imply 2. That is, library load balance is sufficient but not necessary for global load balance. Thus the usual parallel programming approach of load balancing every library is not actually necessary. A load balancer can use the extra degrees of freedom obtained by replacing 2 with 1 to, for example, minimize communication overhead.

For the voxel method, the CHARM++ automatic load balancer is free to migrate both the clients, which do work before the synchronization; and the voxels, which do work afterwards. Hence the load balancer can balance the collision algorithm in the context of the larger program.

4. PERFORMANCE

The voxel algorithm presented in this work has excellent theoretical as well as practical performance.

4.1 Theoretical Performance

Let n denote the number of objects. If the average number of voxels spanned by each object is bounded by a constant, the voxel method sends each object a constant number of times, and collides $O(n)$ objects, and hence imposes no asymptotic overhead on the intra-voxel serial scheme.

If, further, the intra-voxel serial scheme runs in linear time, then the voxel method is also linear time overall. If the serial scheme is not linear; but the number of objects in each voxel is bounded, the serial scheme's input size is constant and hence runs in constant time, so the voxel method is again linear time.

In the parallel implementation, after a global synchronization step each voxel forms an independent serial subproblem. Thus assuming enough voxels that good load balance is possible[5] the time required is $O(n/p + \lg p)$. For large enough n, the first term dominates and the time required is $O(n/p)$.

To reiterate, for a variety of common cases, the voxel method runs in $O(n)$ serial time, and $O(n/p)$ parallel time.

4.2 Serial Performance

To test the performance of our implementation of the voxel method independently from that of the serial collision detection scheme, we used an optimization taken from computer graphics– never test the geometric models for self-intersection. This optimization is especially important for triangular models, which may consist of millions of adjacent triangles. The optimization is implemented by adding a "model number" to each object, and skipping the serial collision detection test if all the objects in a voxel came from the same model. Of course, this optimization still allows you to detect collisions between models, which are likely the interesting collisions. If self-intersections are desired, this check could be disabled or each object can be assigned a different model number. Finally, in our tests we always used individual triangles as objects; this is actually a worst case for our method, since more aggregation generally improves our performance.

These performance results always use only one model, and hence the implementation is always able to skip the serial collision detection step. This nicely factors out the contribution of the serial colli-

[3] The same concept is used in most distributed shared memory cache coherence schemes.

[4] Several built-in strategies exist; or a custom strategy can be written.

[5] The degree of parallelism is limited by the number of voxels.

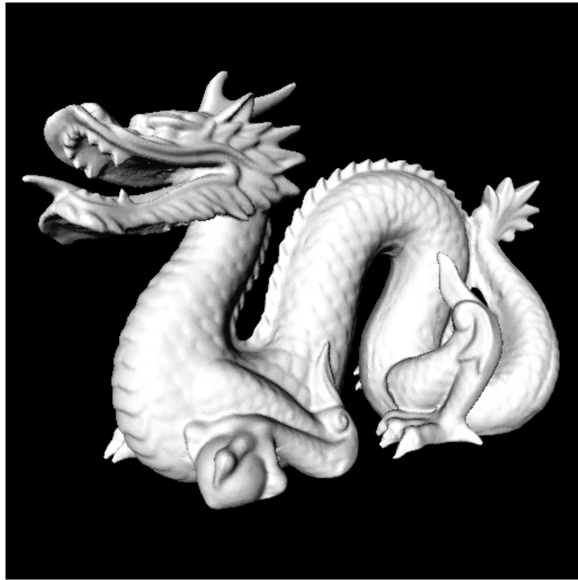

Figure 7: Dragon model.

sion detection scheme from the parallel scheme. The implementation, and the remainder of the computation, from object distribution to the final collection of results, is otherwise unmodified.

The 871,414 triangle[6] dragon model of Figure 7 presented in [**?**] took 3.132 seconds to collision detect on a single processor of an Origin 2000.[7]

The remaining tests use a much simpler model—a tessellated plane perturbed by small harmonic waves. This simple model is easy to scale to any desired number of triangles, while retaining much of the character of an actual surface.

Figure 8 shows the wall-clock time per complete collision detection for a range of different numbers of triangles on a slow Linux PC.[8] The smallest figure shown is 1,024 triangles, which take 1.4 milliseconds or 1.4 microseconds per triangle. The largest figure is 1,048,576 triangles, which take 2.1 seconds or 2.1 microseconds per triangle. This much larger dataset cannot fit entirely in cache and is hence slightly slower.

This serial result is competitive with the figure of 2 microseconds per triangle[9] given for the serial algorithm of Gottschalk et al. in [**?**]. Further, the observed performance of the voxel algorithm is indeed linear in the number of triangles.

4.3 Parallel Performance

We ran a simple scaling benchmark with 65,536 polygons per processor on ASCI RED. The wall-clock time per timestep for various numbers of processors is shown in Figure 9. A program with perfect speedup would have a constant time per step. Instead, we see a slow, logarithmic rise in the time per step due to the $O(\lg p)$ synchronization overhead.

The smallest run shown, 65,536 triangles on a single processor, takes 0.44 seconds per step. The largest run shown, 65,536 triangles on each of 1,500 processors or 98.3 million triangles, takes

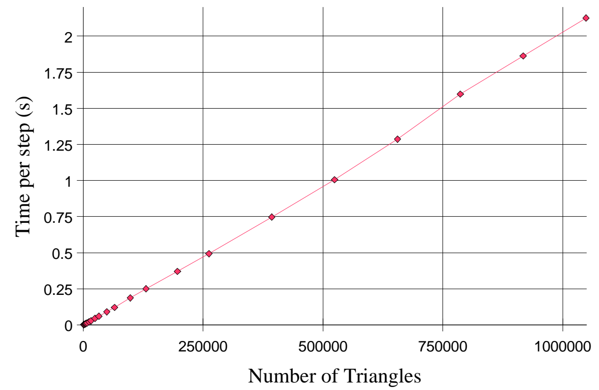

Figure 8: Time per step for serial benchmark

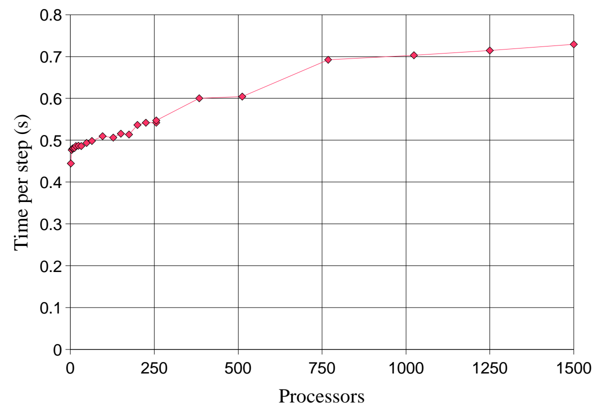

Figure 9: Time per step for a scaling parallel benchmark, with a fixed 65,536 polygons per processor.

0.73 seconds per step. This is a scaled speedup of 915, or a parallel efficiency of 60 percent.

The observed parallel performance is indeed excellent. It is also comparable with the result of 40 milliseconds per timestep (approximately 20-fold faster) for 1875 objects per processor (approximately 35-fold fewer) for the parallel RCB scheme described in [**?**]. However, that work was part of a production code and included extra computation and communication associated with mechanics.

The parallel implementation also scales down for smaller models and fast response time, such as for interactive applications. 32 processors of a 195 MHz Origin2000 system can handle 300,000 triangles at the good interactive rate of 30 milliseconds per step.

5. CONCLUSIONS AND FUTURE WORK

We have demonstrated a parallel collision detection algorithm based on regular space subdivision. We have implemented the algorithm in CHARM++ and extensively analyzed its performance. The results are theoretically and practically quite competitive with published work.

5.1 Future Work

The most fundamental limitation of the voxel method is that all the voxels are the same size and orientation, which limits performance for extremely small or large objects. A promising area of future research is to use a nonuniform grid, ideally determined au-

[6] Hence this is a 871,414-object model.
[7] 195MHz MIPS R10000, IRIX 6.5
[8] 400MHz AMD K6-3, Linux 2.4.2
[9] On a 195MHz MIPS processor.

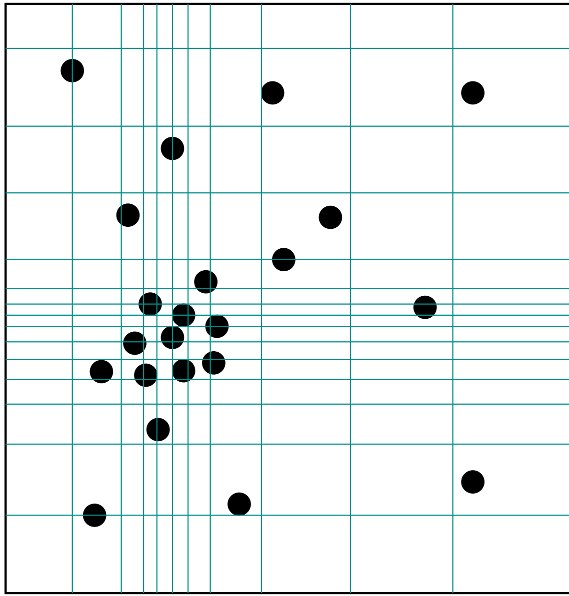

Figure 10: Objects could warp the grid in a relativistic fashion.

tomatically from the object density. A multi-level grid such as an octree would be an easy way to achieve this end, although it may be difficult to maintain $O(n)$ performance. A more effective method might be to use the local object size or density, possibly from the previous step, to efficiently adjust the grid resolution in a spatially dependent manner. A per-axis application of this method is shown in Figure 10; and Gaede and Günter [**?**] provide a comprehensive survey of possible techniques.

A much simpler goal is automatic determination of the ideal voxel size. Voxels that are too small have too much overhead; voxels that are too large do not create enough parallelism and lump together too many objects. The tradeoff should be better analyzed and, if possible, automatic.

Voxels need not form a regular grid at all. There are several other regular and irregular ways to tile 3D space. A less angular grid cell, with a higher volume to surface area than a cube, would be less likely to bring together objects that will never collide. The implementation could also make much better use of temporal and spatial coherence, as suggested in [**?**] and [**?**].

The implementation currently requires one barrier operation per collision, which could become a bottleneck for small problems on large machines. Some sort of global operation is needed to ensure all objects have been delivered to the voxels; but a systolic approach, continually summing the current send and receive counts, might be more proactive.

We currently only determine which objects intersect. This is sufficient for some applications, such as motion planning, clearance checking, and many graphics applications. However, a more complete solution would go on to determine the entire set of penetrating points, the penetration distance, and the penetration direction for each point. Kawachi and Suzuki [**?**] use a “discrete closest feature list” to efficiently determine these quantities via an algorithm similar to the voxel method.

Finally, we are implementing a real mechanics application that uses this collision detection implementation. We hope to quantitatively demonstrate the benefits of our dynamic load balancing capabilities using this real application.

6. REFERENCES

[1] S. Bandi and D. Thalmann. An adaptive spatial subdivision of the object space for fast collision of animated rigid bodies. In *Proceedings of Eurographics '95*, pages 259–270, August 1995. http://ligwww.epfl.ch/ thalmann/.

[2] S. Brown, S. Attaway, S. Plimpton, and B. Hendrickson. Parallel strategies for crash and impact simulations. *Computer Methods in Applied Mechanics and Engineering*, 184:375–390, 2000.

[3] R. Brunner and L. Kalé. Adapting to load on workstation clusters. In *The Seventh Symposium on the Frontiers of Massively Parallel Computation*, pages 106–112, February 1999.

[4] S. Cameron. Approximation hierarchies and s-bounds. In *Proceedings of Symposium on Solid Modeling Foundations and CAD/CAM Applications*, pages 129–137, 1991.

[5] B. Curless and M. Levoy. A volumetric method for building complex models from range images. In *Proceedings of ACM Siggraph '96*, pages 303–312, 1996.

[6] R. Farouki, C Neff, and M. O'Connor. Automatic parsing of degnerate quadric-surface intersections. *ACM Transactions on Graphics*, 8:174–203, 1989.

[7] H. Fuchs, Z. Kedem, and B. Naylor. On visible surface generation by a priori tree structures. *Proceedings of ACM Siggraph*, pages 124–133, 1980.

[8] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, June 1998.

[9] E. Gilbert, D. Johnson, and S. Keerthi. A fast procedure for computing the distance between objects in three-dimensional space. *IEEE Journal of Robotics and Automation*, RA-4:193–203, 1988.

[10] S. Gottschalk, M. Lin, and D. Manocha. Obb-tree: A heirarchical structure for rapid interference detection. In *Proceedings of ACM Siggraph '96*, pages 171–180, 1996.

[11] Chris Hecker. Let's get to the (floating) point. *Game Developer Magazine*, Feb/Mar, 1996.

[12] P. Hubbard. *Efficient Collision Detection for Animation and Robotics*. PhD thesis, Brown University, 1994.

[13] P. Jiménez, F. Thomas, and C. Torras. 3d collision detection: A survey. *Computers and Graphics*, 25(2):269–285, 2001.

[14] L. Kale and S. Krishnan. Charm++: Parallel Programming with Message-Driven Objects. In *Parallel Programming using C++*, pages 175–213. 1996. http://charm.cs.uiuc.edu/.

[15] L. Kalé, R. Skeel, M. Bhandarkar, R. Brunner, A. Gursoy, N. Krawetz, J. Phillips, A. Shinozaki, K. Varadarajan, and K. Schulten. NAMD2: Greater scalability for parallel molecular dynamics. *Journal of Computational Physics*, 151:283–312, 1999.

[16] K. Kawachi and H. Suzuki. Distance computation between non-convex polyhedra at short range based on discrete voronoi regions. In *Proceedings of Geometric Modeling and Procesing*, pages 123–128, Hong Kong, 2000. IEEE.

[17] J. Klosowski, M. Held, J. Mitchell, H. Sowizral, and K. Zikan. Efficient collision detection using bounding volumes of k-dops. In *Siggraph '96 Visual Proceedings*, page 151, 1996.

[18] S. Krishnan, A. Pattekar, M. Lin, and D. Manocha. A higher order bounding volume for fast proximity queries. In *Proceedings of Third International Workshop on Algorithmic Foundations of Robotics*, 1998.

[19] O. Lawlor. A grid-based parallel collision detection

algorithm. Master's thesis, University of Illinois at Urbana-Champaign, March 2001. http://charm.cs.uiuc.edu/papers/.

[20] O. Lawlor and L. Kalé. Supporting dynamic parallel object arrays. In *Proceedings of International Symposium on Computing in Object-oriented Parallel Environments*, Stanford, CA, Jun 2001. http://charm.cs.uiuc.edu/papers/.

[21] M. Lin. *Efficient Collision Detection for Animation and Robotics*. PhD thesis, University of California, Berkeley, 1993.

[22] M. Lin and J. Canny. Efficient algorithms for incremental distance computation. *IEEE Conference on Robotics and Automation*, pages 1008–1014, 1991.

[23] M. Lin and S. Gottschalk. Collision detection between geometric models: A survey. In *Proceedings of IMA Conference on Mathematics of Surfaces*, 1998. http://www.cs.unc.edu/ dm/collision.html.

[24] S. Suri, P. Hubbard, and J. Hughes. Analyzing bounding boxes for object intersections. *ACM Transactions on Graphics*, 18 no. 3:257–277, July 1999.

[25] G. Turk. Interactive collision detection for molecular graphics. Master's thesis, University of North Carolina at Chapel Hill, 1989.

[26] Y. Zhou and S. Suri. Analysis of a bounding box heuristic for object intersection. *Journal of the ACM*, 46 no. 6:833–857, November 1999.

[27] M. Zyda, W. Osborne, J. Monahan, and D. Pratt. Npsnet: Real-time collision detection and response. *Journal of Visualization and Computer Animation*, 4, number 1:13–24, 1993.

APPENDIX

A. EFFICIENT GRIDDING

The inner loop of the voxel method outlined in this work computes the grid locations spanned by an object, and then inserts the object into the object lists at each grid location. For simplicity, we compute the grid locations based on the object bounding box. Hence one extremely common operation is to convert the coordinate of each face of a bounding box, a floating-point number, into a grid location, an integer. However, on modern architectures, converting floating-point numbers to integers is often slow.

In certain situations, a novel but simple technique [**?**] can be used to dramatically speed up floating-point to integer conversions. In particular, given a double-precision source and integer destination array, we may use either the first or the second two C statements below:

```
/*Normal way:*/
dest[i] = (int)floor(src[i]);
/*Bizarre but fast way:*/
float f = (float)(src[i] + convert);
dest[i] = *(int32 *)&f;
```

The reason is that an IEEE single precision floating-point number's mantissa field overlaps the low bits of an integer, as shown in Figure 11. If the grid size is a power of two, we can convert a coordinate into a grid location by shifting and extracting out the appropriate bits of the coordinate.

Floating-point numbers are always stored in the "normalized" form $2^m 1.xxx_2$, so we can *shift* the mantissa of a floating point number by simply adding a constant. For example, given a floating-point coordinate with binary representation $xxx.yyy_2$, after adding $2^{23} 1.1000_2$, we get

$$2^{23} 1.\underbrace{10000000000000000000xxx}_{\text{23bits}}$$

By adding a constant, we have shifted the grid location xxx into the low bits, and by rounding to single precision we have removed the excess bits yyy that represent the sub-gridcell location.

This floating-point number can be compared, incremented, and decremented as if an integer. The IEEE exponent field corrupts the high bits of this integer, but these can easily be subtracted off. Since the origin of the grid coordinates is arbitrary anyway, we leave the exponent. The mantissa is only 23 bits, but this is still enough to represent 8 million grid cells along each axis, or 10^{20} cells total. Converting a double-precision floating point number to a 64-bit integer in the same way would yield a 52-bit grid index.

By adding a "borrow protection" bit to the constant, this method can accept both positive and negative input coordinates. By subtracting the coordinate origin, we can accomplish a shifting as well as scaling. We can also scale the constant to extract out a different set of bits, simulating any power-of-two grid size; this is less flexible but much more efficient than scaling the input values to change the grid size. Finally, we must compensate for IEEE rounding, which rounds to the nearest bitpattern rather than truncating. The ideal constant for this method thus has the form:

$$\underbrace{1.5}_{\text{borrow protect}} * \overbrace{2^{23}}^{\text{shift}} - \underbrace{0.5}_{\text{round}}$$

Thus given a power-of-two grid size and origin, the optimized C code to compute the conversion constant and then convert a floating point value `src` to an integer grid cell index `dest` is as follows.

```
/* gridsize must be a power of two*/
double convert = (1.5*(1<<23)-0.5)
                        *gridsize-origin;
float f = (float)(src + convert);
int dest = *(int32 *)&f;
```

Of course, the same method can be applied in any language that allows us to quickly interpret the bits of one data type as bits of another data type.

Table 1 shows the time per floating point to integer conversion for both the conventional and optimized approaches. The performance difference between the two is dramatic– a factor of 2 to 40, depending on the architecture. Because mapping one bounding box onto a grid requires six such conversions, the total savings by using the optimized method is a microsecond or two per object– a significant speedup.

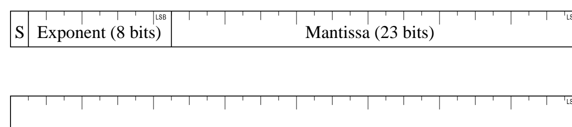

Figure 11: An IEEE single precision floating-point number and a 32-bit integer

Machine[10]	*Normal*	*Fast*
1.5 GHz AMD Athlon XP[11]	54 ns	5 ns
500 MHz Pentium 3 Xenon[12]	236 ns	14 ns
195 MHz MIPS R10000[13]	109 ns	21 ns
300 MHz SPARC Ultra 10[14]	245 ns	30 ns
332 MHz PowerPC 604e[15]	281 ns	127 ns
240 MHz PA-RISC 8200[16]	648 ns	15 ns

Table 1: Comparing conventional and optimized conversion times.

[10] Computed as wall clock time for 100 million conversions divided by 100 million

[11] Linux 2.4.7, gcc 2.96 -O3

[12] Linux 2.4.2, gcc 2.96 -O3

[13] IRIX64 6.5, gcc 2.95.2 -O3

[14] Solaris 2.6, gcc 2.95.2 -O3

[15] AIX for IBM SP, gcc 2.95.2 -O3/VisualAge AIX 5

[16] HP-UX 10.20, gcc 2.95.2 -O3