Collision Detection in Parallel

Mae Eidson, Ray Man

May 2024

1 Summary

We implemented two-phase collision detection with triangle meshes, or polyhedrons with only triangular faces, with an emphasis on the broad-phase portion. Our algorithm relies on placing objects in a fixed grid subdivision of space based on their size, and improves beyond the naive O(n^2) by only checking for bounding ball intersections between objects of adjacent grid cells. In order to permit parallelism, our narrow phase checks are performed entirely after the broad-phase; these are also parallelized. We then analyzed the speedup gained from these parallel versions, and evaluate the effectiveness of different OpenMP schedulers and various re-orderings of steps of the algorithm.

2 Background

The goal of collision detection is to check whether two 3D objects in space intersect with each other or not, with the intent of allowing a collision handler to decide what to do afterwards. Collision detection has a variety of use cases, including physics and medical simulations, as well as video games. Checking whether two objects are intersecting is expensive and a cheap $O(n^2$ is often insufficient even with bounding balls, so collision detection in a system is generally split into two phases. The broad-phase only considers, in our case, bounding balls of objects and spatially organizes them in some manner to avoid ever considering two distant objects, and determines which objects could be intersecting according to the bounding ball, referring these possible intersections to the narrow-phase. The narrow-phase takes each suspected pair and exhaustively checks their meshes for collisions.

Our basic broad-phase approach, described more in depth later, maintains a fixed 3d grid partitioning of space. Each ball then marks the grid cells that it overlaps with; in practice, we use the axis-aligned bounding box for this ball, and mark every grid cell this overlaps with. Note that we don't use tighter bounding boxes for anything else because they can be expensive to calculate and update in some use cases. Then, for any non-empty grid cell, the objects overlapping it can only also have a common cell overlap. However, this suffers from large objects. Now, if we instead use multiple levels of grids, with grid cell widths scaling by powers of 2, and then only have object mark grid cells that they are smaller than, we can limit the number of grid cells that an object can occupy to the number of hierarchies, times 8 when overlapping a highest hierarchy corner, but in practice usually a much smaller multiple. We then check each object against each other object in the grid cells it covers in the smallest hierarchy the object is in, which suffices. The grid is actually efficiently implemented with a map data structure of sparse grid occupancy.

There is no real one-size-fits-all collision detection algorithm for all purposes or really anything other than specified purposes; as such, our constraints are designed both around our algorithm and a plausible use case. For our purposes, we use discrete time steps and discrete collision detection, meaning that we move objects according to their velocity at the end of each "tick", or time step, and simply check whether objects overlap in the resulting position. While this does mean objects can clip through each other slightly, the alternative is very expensive and is essentially a completely different task, as the entire narrow-phase we use would fail. As such, we restrict the maximum velocity of objects. For use cases such as video games, fast moving objects, particularly bullets, are often not treated the same way as typical slow moving objects, requiring an entire extension of the algorithm and very game-specific rules; as such, it doesn't make sense for our algorithm to accommodate this. Lastly, most collision detection algorithms of this family require the maximum object size to be decided at initialization time, as having an object be larger than the size of grid boxes causes failures in the broad-phase.

Our algorithm is robust to such failures but sees slowdowns, and no workaround exists for this. Our basic approach is to use several levels of grid box sizes, and could change the number of levels to better suit a scenario, but there is a trade-off here. Specifically, having the smallest box size be too large is slow on small objects as boxes will contain relatively far away objects, having the largest box size be too small results in large objects covering several boxes and leading to slowdowns, and having too many levels makes updating the grid occupancy slow. Thus, for the sake of fair testing, we fixed constraints on object sizes and used a fixed number of levels. In general, we represent meshes as an array of vertices, where vertex coordinates are relative to the center of the mesh. Faces of the mesh are indices of the vertex coordinates within the array. This representation allows for more efficient updates at the end of ticks, and is standard for applications like this. The algorithm takes an input consisting of an array of meshes, and, every tick of the simulation, outputs a list of meshes that collide with each other, with the idea that a collision handling system would then decide what to do; such a system is outside the scope of this project, however.

Consider the following simplified 1D example containing a few objects of different sizes (Figure 1). Mesh objects, now just 1D intervals, their sizes, and their placements are shown at the bottom of the figure. Hierarchy sizes are fixed at initialization time, and this example shows a hierarchy with four different size levels. Objects are placed in the smallest level that is still larger than its size, and then noted in every level with larger size. For example, while object A is small enough to fit in level 4, it is noted in levels 1, 2, and 3. We note that this diagram numbers hierarchies, labeled resolutions here, in the opposite direction that our own program and pseudocode does.

For the broad phase detection, we use these bounding boxes to note down potential collision areas. Sequentially, this means iterating through the hierarchy from top to bottom. For each level, we examine each mesh that belongs to that level (this means meshes for which this level is the lowest), and mark down all of its neighbors in the current or neighboring boxes as potential collision targets for the narrow phase. Looking at hierarchy level 1, middle-right cell, one could see that while objects D, E are in it, that is not their smallest hierarchy, so we only need to consider C against everything in this cell for the broad-phase. The fact that a hierarchy, specifically 3, has D and E not sharing any cells, means that we will never need to narrow-phase check them since they cannot overlap.

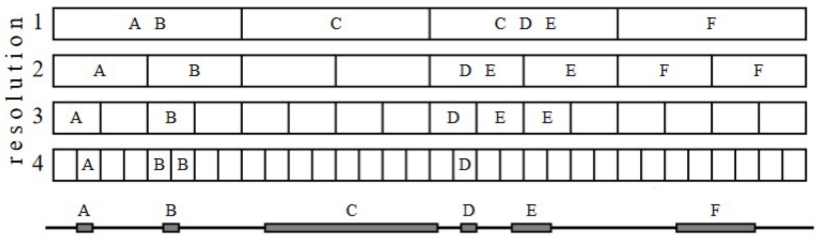

Figure 1: Hierarchical bounding boxes example [2]

Based on the general knowledge of this algorithm, we were able to create pseudocode which informed the intuition for our sequential implementation (see Algorithm 1). The broad phase portion is indicated from lines 1-23, with lines 24-27 covering the narrow phase check.

This is still a slight oversimplification for the first for loop. Instead of resetting the hashmaps, during the position updates at the end of a tick, we update the hashmaps for when objects entered or exited cells rather, resulting in fewer hashmap and hashset inserts and deletions.

Looking at the way that this algorithm navigates through each level within

Algorithm 1 One Tick of Collision Detection (Serial)

```
Let hierarchies $= 0, 1, 2, \dots k$
Initialize occupancy cell hashmap for each hierarchy, $g[0] \dots g[k]$
where $(key, value)$ is $(grid\,coord, objects\,id\,hashset)$
for $m$ in meshes do
    $s =$ the bounding ball for $m$
    for $h$ in hierarchies do
        if $s$ has diameter larger than size of hierarchy $h$ boxes, $2^h$ here then
            for each grid box $b$ of hierarchy $h$ that ball $s$ intersects with do
                insert $m$ into grid cell $g[h][b]$
            end for
        end if
    end for
end for
for $h$ in hierarchies do
    for every non-empty cells/box $b$ in hierarchy h do
        for mesh $m_1$ in $b$, looked up from the $g[h][b]$ set do
            for mesh $m_2$ in $b$ do
                check if $m_1, m_2$ bounding balls intersect
                add this to a stack of narrow-phase checks to make
            end for
        end for
    end for
end for
for every unique pair $m1, m2$ marked for narrow-phase check do
    for every face $f1$ from $m1$ and every face $f2$ from $m2$ do
        Check if $f1, f2$ intersect, if so, mark $m1, m2$ as colliding
    end for
end for
Reset the cell hashmap
```

hierarchy, one could imagine that levels can be explored in parallel. In addition, the narrow phase check processes also seems extremely parallelizable, since pairs of objects are checked independently of each other. For the purposes of OpenMP, we can think of each hierarchical level exploration as a separate task, and we can use the shared address space model to create a shared stack data structure where pairs of potential collisions between bounding boxes become noted on the shared stack whenever a thread accesses the stack. However, as we found, most importantly, parallelizing over each box b in the second group of nested loops is optimal, with very specific methodology for aggregating the results of threads. This is significant because the broad-phase takes about 80% of the single threaded runtime and suffers from extreme workload imbalance, as high density areas are far more expensive to populate cells for than low density areas, with runtime scaling squared to density in a region. There are essentially no immediate dependencies in the program, but the algorithm is liable to recognize a broad-phase pair twice and needs to aggregate the pairs from each thread in some way. We also note that this relatively cleanly parallelizable code is the result of several iterations of rearranging and modifying the algorithm that will explained later.

Furthermore, it is extremely difficult to attain any coherency in the already expensive g hashmap lookups aside from having the hierarchy loop be where it is instead of inside of the $m1$ mesh loop, which is what a previous iteration of the algorithm did. We did require ourselves to maintain the invariant that a narrow phase check can never be done twice, as this gets prohibitively expensive in some common use cases. As such, we experimented with instead of moving narrow phase checks inside of the broad phase loop and checking a growing set of previously performed narrow phase checks beforehand, and with having some threads handle the second for block while having some threads handle the third.

3 Approach

We wrote our code in C++, as we were targeting the OpenMP library for shared address space parallelism. The OpenMP library supports static, dynamic, and guided scheduling for parallel tasks, and its shared address space model also works for the shared stack structure during the broad phase detection. For experimentation, we mainly used the Gates clusters (GHC) to access multithreaded code up to 64 threads, as well as the Pittsburgh Supercomputer Cluster (PSC) in order to access more threads. Otherwise, we do not use new libraries, languages, or APIs to run our algorithm, and instead focus on the implementation and optimization instead.

In the end, we used a simple map that only permits parallel access, but not updates, as our attempts to parallelize the first for loop chunk(in the pseudocode) ended up being slower than a well written non-parallel approach. All memory is shared, as the bulky second loop body (in pseudocode) is the slowest and requires very unpredictable access. Furthermore, it would not have made sense to partition the mesh array in memory because static batching for those loops performed terribly due to poor workload balancing from varying concentrations of meshes in areas. Ultimately, we used dynamic batching for the broad phase second loop, specifically over cells, and static batching for the narrow-phase check since tetrahedron pairs each cost roughly the same amount to check.

The key implementation detail comes from how we sent the line 19 checks to make to the third for loop. For this, we had each thread build its own list and set of these, as aggregating them into a single list could lead to heavy stalling due to contention to it, especially at higher thread counts. We then implicitly concatenated these lists for the line 24 for loop, which became a loop over the total number of narrow checks. Each check index would then map to an index in a certain thread's list of checks as produced in line 19, and we had some additional data structures so each thread in the line 24 loop would know exactly which thread's list it was currently looking at with a single check. To prevent double checking narrow phases that had been spotted by two different threads, we check for duplicates against each other set. This setup is theoretically a direct improvement over most other rearrangements and approaches we worked with, and does empirically perform better.

In terms of optimizations and changes to the original serial implementation, the entire broad-phase underwent two full rewrites and is completely unrecognizable from the original serial version. Firstly, a few major algorithmic changes include switching from dynamically chosen grid-partitioning of space to a static grid; the former was slightly slower, especially on many threads. The multiple hierarchies came later due to large objects being far too expensive with a single small grid cell size. This, and struggling with maintaining a parallel list of duplicate narrow-check requests, further necessitated separating the narrow and broad phase checks into fully separate for loops.

The previously mentioned approach with dynamically chosen grid partitioning is commonly referred to as quad-trees(2D) or octo-trees(3D), where each node is a box in space that gets split into 4 or 8 by child nodes about a point chosen within the box that roughly evenly distributes the objects in the box. This was parallelizable, but actually just suffered from contention issues in dense areas, especially at the start, making it slower on any number of threads. After making the hierarchy update, we tried every single coherent permutation of the loops for the narrow-phase, or second loop chunk of the pseudocode, finding that having the hierarchy loop anywhere other than the outside led to dropoffs at high thread counts, likely because accessing every hierarchy at the same time in different threads is led to minor memory stalls. The old alternative was having it further inside, and looping over every mesh and then finding the bounding boxes of the correct hierarchy to look at for it. We do note that this was prior to some other very significant changes, but see no reason to reverse this order after this. Lastly, we tried several arrangements of the narrow-phase and feeding line 19(pseudocodes) requests to it, and this is the most frictionless method we could find with respect to parallelism. Other approaches involving explicitly coalescing the list make no sense in retrospect given what our final solution was and performed worse, but made sense at the time prior to finding this method. The final method is a bit difficult to describe, refer to the nested loops in the submission code that calls function *meshmesh*.

We briefly attempted to parallelize within single narrow-phase checks, specifically by making a task to check two mesh faces against each other. However, the granularity of this was far too low, even on relatively complicated meshes, considering how many narrow phase checks had to be performed anyways.

As a point of reference, the pre-hierarchy change runtimes on 1 and 8 threads on 200000 meshes were 28 seconds and roughly 5.6 seconds, respectively, versus 3.3 and 1.15 respectively on the final submission version. Immediately after the hierarchy update, we actually saw only about $1.5x$ speedup on 8 threads for this m. In the results below, we increased the mesh count, which increased the density or concentration of meshes in space, which vastly skews the speedup. We further note that the optimal parallelization method and even single threaded method are vastly different on lower densities, and our speedups earlier were better simply because we managed to cut out a lot of parallelizable work, especially from the difference in how the equivalent of the second loop works for dynamic boxing.

4 Results

To measure performance, we examined times for the serial version, timing the amount of seconds it took for the algorithm to compute the amount of collisions in the request, and then ran the code with parallelism enabled under static and dynamic schedulers to obtain speedup times and construct tables and graphs.

We experimented with different sizes of meshes, and would generate different numbers of differently-sized mesh tetrahedrons to see how having larger numbers of objects would scale parallel-wise. As the request gets larger and more complex, the parallelism will be more easily seen and the speedup metric larger, as the overhead of assigning threads to tasks will decrease in comparison to part of the code being parallelized. Note that the order of magnitude of the algorithm is roughly $O(n^2/m)$, where m is some large constant, increasing the number of tetrahedrons by an order of magnitude also dramatically increases the time it takes for this algorithm to run. We settled on testing collision requests with about 750,000 tetrahedron meshes, using tetrahedrons because these shorten the embarrassingly parallel narrow-phase and more accurately represent the weaker broad-phase speedup, as the broad-phase is completely independent of the number of faces in a mesh while the narrow-phase is entirely dependent on this.

Overall, we found that static scheduling was too slow to get good results, as different levels of the hierarchy would not be well load-balanced. As a result, to test the overall speedup, we conducted tests using the dynamic scheduling configuration, and obtained multi-threaded speedup based off of the baseline of single-threaded CPU code, using the very large request of 750,000 tetrahedron meshes as a problem size. The overall speedup graphs and tables are referenced in the following figures. Importantly, we see speedup improvements as we increase the number of threads, although the speedup curve is not perfectly linear.

Threads	Time (sec)	Speedup
1	51.447	1x
2	30.68	1.8x
4	17.9299	3.1x
8	11.623	4.7x

Figure 2: Table for 750,000 Tetrahedrons, Dynamic Scheduling on GHC

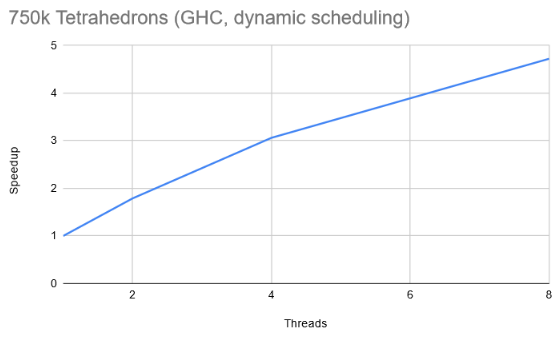

Figure 3: Speedup Graph for 750,000 Tetrahedrons, Dynamic Scheduling on GHC

Threads	Time (sec)	Speedup
1	63.7	1x
2	37.31	1.7x
4	22.53	2.8x
8	15.94	4.00x
16	11.55	5.5x
32	9.51	6.7x
64	7.74	8.2x

Figure 4: Table for 750,000 Tetrahedrons, Dynamic Scheduling on PSC

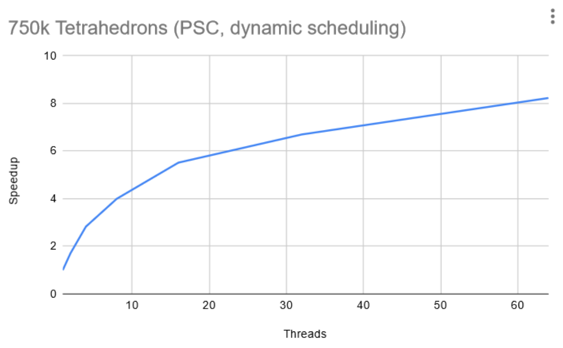

Figure 5: Speedup Graph for 750,000 Tetrahedrons, Dynamic Scheduling on PSC

Generally, we expect that as the problem size increases, the speedup will also increase, as the overhead and sequential portion starts becoming more insignificant in comparison to the larger and now more parallelizable portion. This shows up in the data as well: a problem size of 100,000 tetrahedrons yields a time of 1.37 seconds on 1 thread, 0.95 seconds on 2 threads, and 0.713 seconds on 4 threads, which is only a 1.92x speedup. When we increase the problem size to 500,000 tetrahedrons, we see a time of 28.1 seconds on 1 thread, 17.06 seconds on 2 threads, and 10.7 seconds on 4 threads, for a speedup of 2.6x on 4 threads, much greater than when we had the smaller problem size. Going deeper, this specifically has to do with how we increased the number of objects without changing the size of space they were in. As the concentration of objects increases, the narrow-phase gets progressively more expensive, reducing the granularity of its loops. This does actually worsen the workload balance, but this ended up not mattering quite as much as the granularity.

Depending on the concentration of an area, the size of a task for the broad-phase can vary by a multiple of at least a few hundred, as each grid cell is handled by a single thread in the second loop chunk. We experimented with rearrangements of the for loops that avoid this, but they each have their own drawbacks and were empirically slower on any number of threads. However, as such, we're left with potentially granular and potentially meaty tasks that also don't see any better speedup when batched at all.

On another side, the hashmap structure leads to completely random access patterns across the majority of the program's memory. With more time, we would've liked to explore alternative data structures, but this would likely require another full rewrite of the algorithm because everything is extremely finicky.

Roughly 80% of execution time, both single threaded and parallel, is spent on the second loop chunk(in pseudocode), or the broad-phase checking, as intended by our testing parameters. This ratio decreases as we increase number of threads: on larger threadcounts, we find that approximately 67% of the time is used on the broad-phase detection, while 33% was used on the narrow phase detection. As a result, we spent a lot of time optimizing the broad phase detection of the code, experimenting with different configurations of the for loops to obtain a result that was easily parallelizeable across multiple threads. Potential room for improvement would be further examining the code for the broad phase detection stage for further parallelization opportunities. Because a lot of collision detection scenarios are also heavily customized to that specific use case, creating an algorithm that best suits a specific context and takes advantage of that type of configuration would also improve the speedup. For example, we know that the hierarchy levels are initialized at runtime. However, the values for those levels might differ for different types and size classes of polygons, and knowing and fine-tuning those values will also improve speedup.

For this project, we experimented on a CPU implementation with OpenMP. However, it is true that this implementation runs on the shared address space model instead of a message passing model. The benefit to using a message passing library such as OpenMPI is that OpenMPI can run on any machine, while CUDA is reliant on GPU usage and both CUDA and OpenMP assumes use of a shared address space model.

Considering a variant of collision detection implemented using a message passing model in a library such as OpenMPI, the narrow-phase detection seems potentially sound, since each thread could be assigned a different batch of mesh pairings to compare for collisions, and sorting the mesh pairings by difficulty would perhaps allow for better load balancing as well as a smaller batch size. However, one must also consider a broad phase detection when thinking about a message passing version of this algorithm, since each thread will potentially explore its own level in the hierarchy, and communicate with all other threads after all threads explore their own levels. These levels within the hierarchy might not be load balanced well, and may also contain double-counted results which may not be updated until the threads are able to sync and communicate with each other, which may incur overhead.

During the project pitch, we originally conceived of writing a version of the collision detection program that utilizes GPU programming in CUDA. Some implementations of other collision detection algorithms in CUDA have been attempted in the past, including approaches like sort and sweep and spatial subdivision, which rely on sorting the cells, then conducting eight passes to update the states of mesh objects and check mesh collisions [1]. However, the hierarchical algorithm that we used does not match well with any implementations of the algorithms we found for CUDA online, and we believe taking advantage of GPU programming would require a significantly different algorithm.

5 List of Work

We both frequently collaborated together for different iterations and on the final paper, and as a result, have approximately contributed 50% of the project.

References

[1] Scott Legrand. Broad-Phase Collision Detection with CUDA.

[2] Brian Mirtich. Efficient Algorithms for Two-Phase Collision Detection. December 1997.