N-CS159: Parallel Collision Detection

Lok Man Chu
Computer Science Department
San Jose State University
San Jose, CA 95192
408-924-1000
lokman.chu@sjsu.edu

ABSTRACT

Collision detection is an important computational problem that detects the intersection of two or more objects within a group of objects. While it was initially only used for video games, it now is used in computer physical simulations and robotics.

One of the biggest problems with collision detection is the fact that pairwise comparisons are required to check for intersections and thus very computationally expensive. This paper will explore how spatial partitioning can be used for broad phase pruning to increase the efficiency of pairwise comparisons and also how to effectively tackle the problem with parallel programming.

1. Introduction

1.1 Detecting Collisions

In order to perform collision detection, one has to check for intersections between objects. For objects such as circles or rectangles, it is very simple to check for intersections. With circles, there is an intersection when the distance between the center of the two circles is less than the sum of their radii. For rectangles on the same axis of rotation, intersection can be detected with a combination of four conditional statements. For more complicated polygons numerous techniques to check for collisions with their own pros and cons exist. The specifics of these techniques are out of the scope of this paper, but it is sufficient to understand that they are complicated algorithms that require more than a few of instructions unlike the circle or the rectangle.

1.1.1 The N Squared Problem

Those are the steps required to check for one intersection between two objects. However, if one is performing collision detection, presumably there is more than two objects to check against. In that case, every pair of objects need to be checked for intersection, resulting in the following checks for intersections:

$$\binom{n}{2} = \frac{n!}{2!(n-2)!} = \frac{n(n-1)}{2} \approx \frac{1}{2}n^2$$

While $O(n^2)$ algorithms are sufficient for small values of n, quadratic growth eventually balloons out of control. For this reason, steps must be taken in order to reduce the number of pairwise comparisons being made.

1.1.2 N Squared Solutions

One of the best ways to prune the number of pairwise comparisons that need to be done is through spatial partitioning. The hyperplane separation theorem states that, for two disjoint non-empty convex sets, there exists a hyperplane that separates the two sets [1]. This is used to derive the separating axis theorem which states that, two convex objects do not overlap if there exists an axis onto which the two object's projections do not overlap. More practically speaking, if you can draw a line or a plane between two objects, they are not intersecting. Following this rule, if space U is partitioned into space A and B, and objects in space U are not in both space A and B, then objects in space A cannot intersect with objects in space B. This means that pairwise checking only needs to be done within space A and Space B. If one can perfectly divide n objects into m partitions, then the number of checks for intersections is reduced by a factor of m.

$$m\binom{n/m}{2} \approx m\left(\frac{1}{2}\left(\frac{n}{m}\right)^2\right) = \frac{1}{2}\left(\frac{n^2}{m}\right)$$

1.2 Quadtrees

One of the best ways to partition objects in a space is through the use of tree based decomposition. One such form is the quadtree, which recursively divides a space into four quadrants until some condition is met. Objects are then placed in implicit data structures within these quadrants. Assuming an object does not overlap into another quadrant, pairwise collision detection only needs to occur among objects within the same leaf node. There are two main implementations of quadtrees, dynamic and static, each with their own benefits and demerits. The dynamic implementation, which is usually just called a quadtree, subdivides the space into quadrants only if the number of objects within that space is greater than some set amount. The static implementation, which is also called a tree-pyramid is a complete tree that subdivides to a given number of levels [4].

1.2.1 Choosing the Right Tree

The dynamic quadtree is good at ensuring that the number of pairwise comparisons is equal in each quadrant, but requires for the tree structure to be recreated every time the objects move. The static quadtree handles the movement of objects by clearing and inserting them back into the tree every time the object moves. While the static quadtree does not have to recreate the tree structure every time an object moves, it does have a much higher memory footprint due to subdividing quadrants that may not be occupied.

2. Implementation

2.1 Tools Used

The implementation of quadtree spatial partitioning was done in C++ using the openFrameworks toolkit for easier graphics rendering. Parallelization of code was done with OpenMP. Lastly the project was done using Microsoft Visual Studio 2017. The most updated version of the source code for this implementation can be found at github.com/slivermasterz/ParallelCollision.

2.2 Design Decisions

As the goal for this project was to determine the efficiency of parallel processing on collision detection with spatial partitioning, a working test environment was required in order to compare different implementations of the algorithm. Using circles as the objects of collision as opposed to more complicated polygons allowed for the focus of the implementation to be on the spatial partitioning and less on the intersection algorithm. The project was done in two dimensions with quadtrees as opposed to three dimensions with octrees in order to reduce the complexity of vector math required. This also makes for an easier visual representation of what is occurring as the program window is two dimensional by nature.

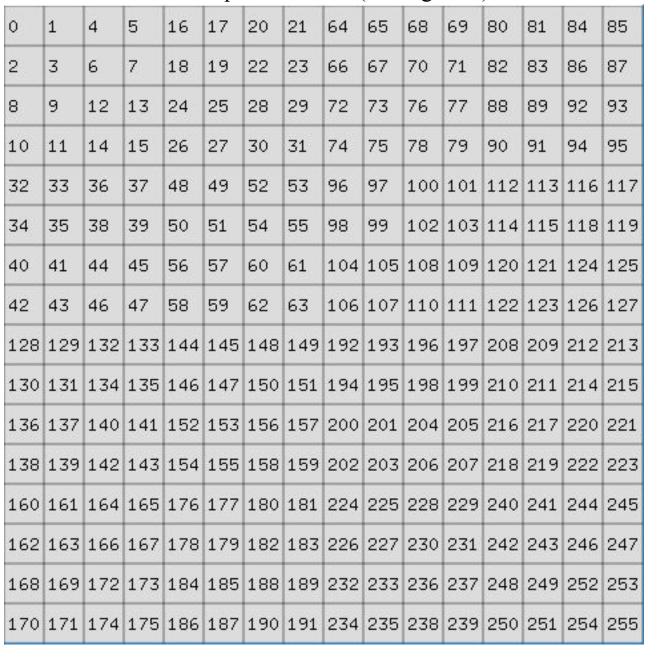
2.2.1 The Quadtree

2.2.1.1 The Box

The quadtree used was a static quadtree as the dynamic quadtree has a much higher runtime requirement when it comes to tracking moving objects. Each quadrant of the quadtree is represented by boxes, which contain a vector for all child boxes and also a vector for all objects within the box. As the quadtree is a complete tree, only the leaf boxes contained the objects. A lock was used to prevent race conditions from occurring when adding to the object vector.

2.2.1.1 The Nearest Neighbor

The quadtree structure is incredibly efficient for insertions, requiring only as many level deep of the quadtree is worth of iterations. Quadtree traversal on the other hand, is a lot more complicated and time consuming than it should be. Thus a new vector was added that contained every leaf node in the quadtree, added during quadtree creation, which allows for quick tree traversal at only a small cost to memory. The only problem with adding to the vector during quadtree creation is that the index of the boxes are not in sequential order (See figure 1). While this has



0	1	4	5	16	17	20	21	64	65	68	69	80	81	84	85
2	3	6	7	18	19	22	23	66	67	70	71	82	83	86	87
8	9	12	13	24	25	28	29	72	73	76	77	88	89	92	93
10	11	14	15	26	27	30	31	74	75	78	79	90	91	94	95
32	33	36	37	48	49	52	53	96	97	100	101	112	113	116	117
34	35	38	39	50	51	54	55	98	99	102	103	114	115	118	119
40	41	44	45	56	57	60	61	104	105	108	109	120	121	124	125
42	43	46	47	58	59	62	63	106	107	110	111	122	123	126	127
128	129	132	133	144	145	148	149	192	193	196	197	208	209	212	213
130	131	134	135	146	147	150	151	194	195	198	199	210	211	214	215
136	137	140	141	152	153	156	157	200	201	204	205	216	217	220	221
138	139	142	143	154	155	158	159	202	203	206	207	218	219	222	223
160	161	164	165	176	177	180	181	224	225	228	229	240	241	244	245
162	163	166	167	178	179	182	183	226	227	230	231	242	243	246	247
168	169	172	173	184	185	188	189	232	233	236	237	248	249	252	253
170	171	174	175	186	187	190	191	234	235	238	239	250	251	254	255

Figure 1. Index of boxes when assigned on quadtree creation

little effect on traversal, it makes finding the neighbor quadrant an extreme headache. Finding a neighbor quadrant is required by the algorithm as there is possibility that an object overlaps with another quadrant. To solve this problem, there are two potential solutions, one where the neighbor is found by traversing the quadtree recursively, or two where the index of the neighbor quadrant is calculated to be accesses through the vector of leaf boxes. The recursive option is not only overly complicated, but also relatively inefficient [2]. Calculating the index however, has a *O(1)* time complexity. While calculating the index of the neighbor is possible with the current indexing, it was deemed to be easier to reindex the boxes to be ordered left-to-right then top-to-bottom, normal English reading order [3]. This way, the cost of calculating indexes is pushed to quadtree initialization and taken away from runtime. This is done by adding boxes to a new vector in the left to right then top to bottom order through calculating the index of the box occupying that space (See figure 2). The value of the index occupying that space is calculated with a modified binary to decimal conversion, where the powers of two are replaced with powers of four. For example, the value of the index at row 3 column 14 is:

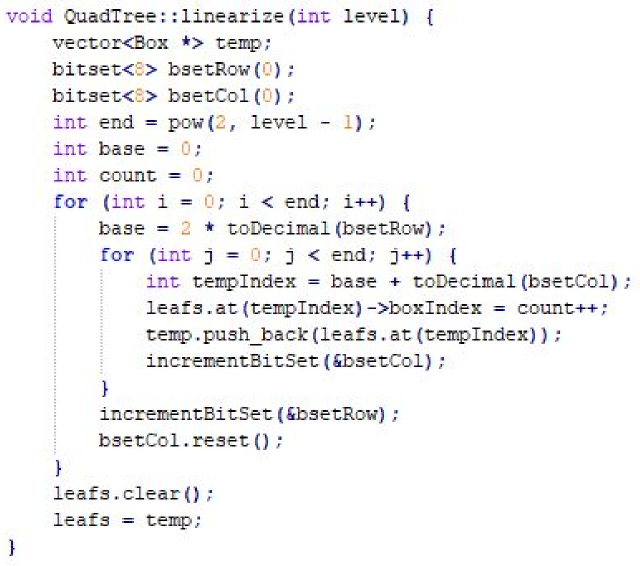
$$2_{10} = 0010_2 \;\; \& \;\; 13_{10} = 1101_2$$

$$2\,(0(4)^3 + 0(4)^2 + 1(4)^1 + 0(4)^0) + (1(4)^3 + 1(4)^2 + 1(4)^1 + 1(4)^0)$$

$$= 2(4) + 64 + 16 + 1$$

$$= 89$$

Performing this operation for all leaf nodes will result in all boxes sorted in left to right then top to bottom order. This results in incrementing or decrementing by one in order to find the left or right neighbor and incrementing or decrementing by 16 to find the top or bottom neighbor.



```
void QuadTree::linearize(int level) {
    vector<Box *> temp;
    bitset<8> bsetRow(0);
    bitset<8> bsetCol(0);
    int end = pow(2, level - 1);
    int base = 0;
    int count = 0;
    for (int i = 0; i < end; i++) {
        base = 2 * toDecimal(bsetRow);
        for (int j = 0; j < end; j++) {
            int tempIndex = base + toDecimal(bsetCol);
            leafs.at(tempIndex)->boxIndex = count++;
            temp.push_back(leafs.at(tempIndex));
            incrementBitSet(&bsetCol);
        }
        incrementBitSet(&bsetRow);
        bsetCol.reset();
    }
    leafs.clear();
    leafs = temp;
}
```

Figure 2. Code for linearizing quadrant indexes

2.2.2 The Collision Detection

Collision detection was done by adding all objects into the quadtree structure and iterating through all the boxes through the use of the leafs vector. For each box, pairwise intersection calculations must be performed for all objects in the box/ Whether or not the object overlaps into a neighbor box is also checked, and if true will be checked against every object in said neighbor box. The tree is then cleared to allow for object insertion on the next loop.

2.3 Parallelizing Challenges

2.3.1 OpenMP

Through the use of OpenMP, parallelizing loops is not too difficult. The programmer no longer needs to manually spawn threads and partition the loop, but can now use compiler instructions to parallelize the loop. As the algorithm for collision detection is just comprised of looping through leaf boxes, OpenMP's loop parallelization of loops was sufficient to parallelize collision detection.

Generally, multithreaded programs try to avoid inter process communication by returning their results in their own data structures to be merged after all other threads are completed. However, after some initial testing, merging the result took longer than adding the objects into the shared vector, thus the latter method was used. Due to the shared data structure, locks were used to prevent race conditions.

2.3.2 Loop Scheduling

OpenMP comes with three scheduling options which change the way loop iterations are divided. The default schedule is static, which equally divides all iterations between the number of threads. This scheduling is great when the tasks of each iteration generally take the same amount of time to complete. The next scheduling option is dynamic scheduling, which assigns one iteration to each thread. When a thread completes its iteration, it is assigned with the next iteration not yet executed. Dynamic scheduling can also be assigned chunks, where instead of only one iteration being assigned to the thread, some *n* number of iterations are assigned instead. This scheduling method has a much higher runtime overhead, as iterations are distributed at runtime rather than compile time with static scheduling. However, it has the advantage over static scheduling when iterations take drastically different amounts of time. There is also a third scheduling option of guided scheduling, which is like dynamic scheduling, except that the chunk size changes over run time. It starts with larger chunks and adjusts to smaller chunks if the workload is imbalanced. This is the middle ground schedule which attempts to garner the benefits of both static and dynamic scheduling.

2.3.2.1 Schedule Testing

Based on the strengths of each scheduling option, it seemed like dynamic scheduling would be the best for this program, as there are no guarantees that each leaf box contain the same amount of objects, nor is there any guarantee for the number of objects that overlap into a neighboring box. All factors which affect the runtime duration of each iteration. This sounds reasonable in theory, but does not hold up in reality.

2.3.2.1.1 Test Performed

The test for performance of each algorithm consisted of checking the number of objects required to increase the runtime duration above twenty milliseconds. Each algorithm gets to run on the same amount of objects three times, with the median time being recorded. The number of objects continue to increment until the runtime of all three attempts for an algorithm goes above twenty milliseconds. This method was taken in order to eliminate the occasional spike in runtime, which would have prematurely ended the test. It also results in smoother data.

The algorithms tested are the original pairwise n^2 algorithm, the sequential quadtree, and the static, dynamic, and guided quadtree with each quadtree tested at different depths. One test was run on a i7-7500U CPU which comes with two physical cores and four logical cores. Another test was performed on a Ryzen 7 1700 CPU with eight physical cores and sixteen logical cores.

2.3.2.1.2 Test Results

Tests on both the Intel and AMD CPUs returned interesting results. On the Intel CPU, the most efficient algorithm was the nine level quadtree with static scheduling. The AMD CPU on the other hand performed the best with the eight level sequential quadtree. The pairwise algorithm for Intel returned 1,360 and for AMD returned 2,630. Ironically, the CPU with more threads performed worse at parallelization and better sequentially (See Table 1 & 2).

2.3.2.1.3 Result Analysis

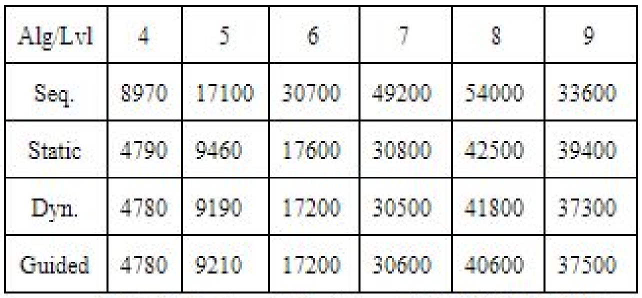
Considering the program was written and initially tested on an i5-4670k 3.9GHz CPU, it is not surprising that the best result occurred on a similar four threaded machine. The AMD processor failing to result in parallel performance exceeding sequential performance can probably be accounted for by the inefficient use of threads. There are two possible reasons. The first being that sixteen threads is overkill for these algorithms, thus causing the overhead of thread generation to exceed the speedup gained from parallelization. The second potential reason is that the algorithm is not optimized enough to perform well on sixteen threads. Perhaps an alternative assignment of threads could result in a runtime speedup. The first reason seems more likely, as the efficiency ratio between the parallel and sequential algorithms increase as the depth of the quadtree increases.Which means that the effectiveness of parallelization increases when then number of iterations increases.

Earlier it was stated that dynamic scheduling should outperform static scheduling due to the fact that the number of objects within each box are probably not equal nor close to equal. That assumption is clearly false, as the data seems to show that across the board, static scheduling is better than dynamic scheduling with the only minor exception being at the level four depth quadtree for the Intel CPU. The flaw in the initial theory is that the premise of there being an unequal amount of objects in each box is false. After a certain level of subdivisions, The size of the boxes become so small that less than five objects can exist within that space at once. Thus causing all the iterations to have near equal runtimes.

Number of Objects Before Test Failure

Alg/Lvl	4	5	6	7	8	9
Seq.	7970	15200	26300	41600	39500	28600
Static	5430	10390	19600	33900	51600	58900
Dyn.	5460	10340	19300	33800	51000	46700
Guided	5420	10280	19300	33700	51600	47800

Table 1. Test results for i7-7500U 3.5GHz CPU



Alg/Lvl	4	5	6	7	8	9
Seq.	8970	17100	30700	49200	54000	33600
Static	4790	9460	17600	30800	42500	39400
Dyn.	4780	9190	17200	30500	41800	37300
Guided	4780	9210	17200	30600	40600	37500

Table 2. Test results for Ryzen 7 1700 3.8GHz CPU

3. CONCLUSION

While the parallelization of collision detection is embarrassingly easy with OpenMP, the optimization of efficiency of said parallelization is annoyingly difficult. One needs to consider the number of tasks per iteration, the number of iterations, and the number of threads that will be used. It seems like whenever any of those factors change, the most efficient variant of the algorithm also changes. This means that a programmer working at parallelizing code, should always test on multiple environments in order to gauge the effectiveness of their parallelized code on each one.

4. ACKNOWLEDGEMENTS

Thanks to Hooey and Saeldur for lending their hardware for the tests performed. This project could not have been completed without their assistance in providing the test data and trust that the executable did not contain malignant programs.

5. REFERENCES

[1] Boyd, Stephen P. and Vandenberghe, Lieven. Convex Optimization. Cambridge University Press, Cambridge UK, 2004, 46-51.

[2] Geier, David. Advanced Octrees 4: finding neighbor nodes. The Infinite Loop. geidav.wordpress.com/2017/12/02/advanced-octrees-4-finding-neighbor-nodes/

[3] Kunio Aizawa, Koyo Motomura, Shintaro Kimura, Ryosuke Kadowaki, and Jia Fan. Constant Time Neighbor Finding in Quadtrees: An Experimental Result. ISCCSP 2008, Malta, 12-14 March 2008

[4] Samet, Hanan. Spatial Data Structures. Modern Database Systems: The Object Model, Interoperability, and Beyond, Addison Wesley/ACM Press, Reading, MA, 1995, 361-385.