PARALLELIZED BOID SIMULATION

Josiah Miggiani, Ryan Huang

SUMMARY

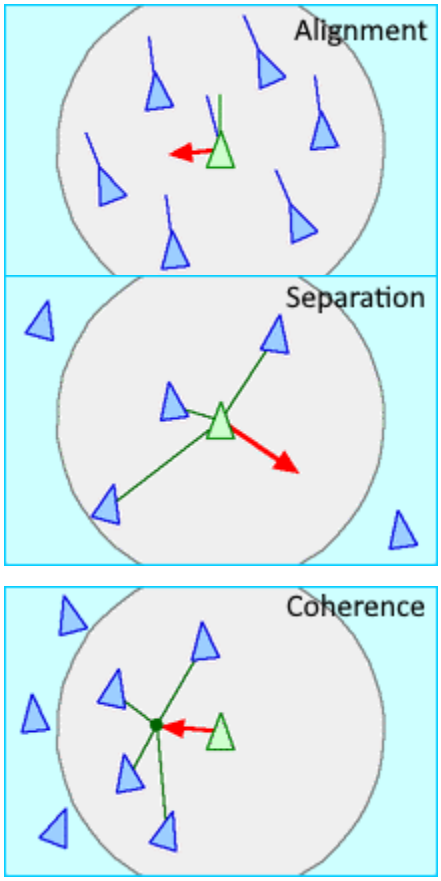
In our project, we parallelized the boids behavioral algorithm, which simulates the complex, aggregate motion of a flock of independent actors. We developed three different parallel approaches to improve upon the sequential algorithm: naive, octree, and spatial hash. We explored different methods of optimizing shared memory for finding neighbor boids between simulation ticks in order to reduce algorithmic complexity and improve performance as simulations ramp up in scope, including boid count and simulation width. On our local machine (Ryzen 7 3700X, 16 threads) we achieve a speedup of 10x, and on Bridges-2 our 128 core implementation achieves a maximum speedup of 234x. We demonstrate that the boids algorithm can be parallelized while remaining cognizant of many spatial dependencies, notably without sacrificing simulation fidelity through approximation.

BACKGROUND

Boids were first proposed by Craig W. Reynolds in his paper submission to SIGGRAPH '87, "Flocks, Herds, and Schools: A Distributed Behavioral Model." In his words, "The aggregate motion of the simulated flock is the result of the *dense interaction* of the *relatively simple behaviors* of the individual simulated birds." Though each boid is independently responsible for deciding the course of motion it takes, these decisions are informed by each boid's local perception of their dynamic environment. The interplay between behavioral rules programmed into the boids and their understanding of their local surroundings (such as nearby boids) is what produces such compelling aggregate dynamics.

In order to better understand how boid simulation can be parallelized, it's first necessary to understand the basic algorithm that dictates boid flight planning. All boid behavior is constrained by three fundamental rules (or forces):

1. **Alignment** (or velocity matching)
2. **Separation** (or collision avoidance)
3. **Coherence** (or flock centering)



Boid alignment seeks to maintain a similar velocity to nearby flockmates. Velocity is a vector quantity, representing heading (orientation) and speed.

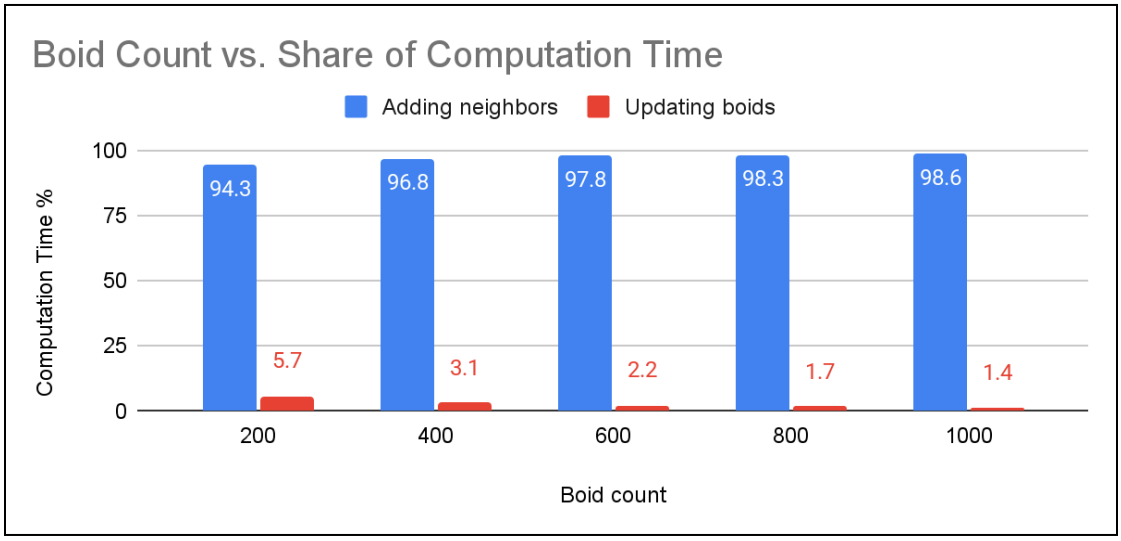
Separation requires that each boid attempts to avoid flying into other boids. As a boid nears in spatial proximity to another, it will alter its heading to avoid collision.

Coherence dictates that boids seek to fly closer towards nearby boids. More specifically, it makes a boid want to be near the center of the flock. Coherence causes the boid to take a heading that moves it closer to the centroid of nearby flockmates.

To calculate these forces, each boid has to gather information from every other boid within a certain vision radius to determine its next action. As Reynold's original paper points out, "A naive implementation of the basic flocking algorithm would grow in complexity as the order of the square of the flock's population ("$O(N^2)$")... this is because each boid must reason about each of the other boids, even if only to decide to ignore it." The algorithm inherently features a great deal of dependencies between all boids that change from one simulation tick to the next. While temporal locality is not guaranteed to be exploitable since each boid shifts towards and away from all other boids each tick, spatial locality is certainly present. Each boid is interested in other boids within a "neighbourhood sphere," and thus an optimization that reduces the search space from the entire simulation volume to relevant boids in some rough proximity would be highly useful.

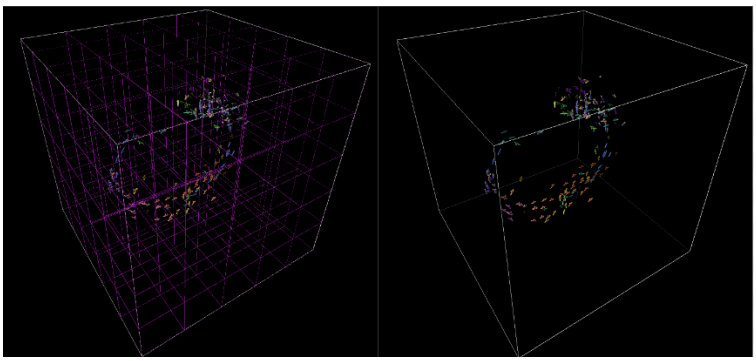
Thus, it comes naturally that a potential solution to this problem is dynamic spatial binning, in which all boids are sorted spatially so that the neighborhood queries can be expedited. With spatial bins, a boid can investigate the bins nearest to itself and only those bins, rather than examining the entirety of the simulation space. Even better, this bin structure can be placed in shared memory – threads can query independently and simultaneously since they are all readers. Simultaneous writers can be enabled with careful attention to atomicity. What the simulation is not is amenable to SIMD execution. Individual boids are wildly divergent in flight behavior, and so the simulation is not suited for such parallelization techniques.

An initial investigation into the algorithm shows that finding neighbors dominates computation time. This share only grows as boid counts increase. As such, we target spatial binning methods to aid in parallelizing adding neighbors.



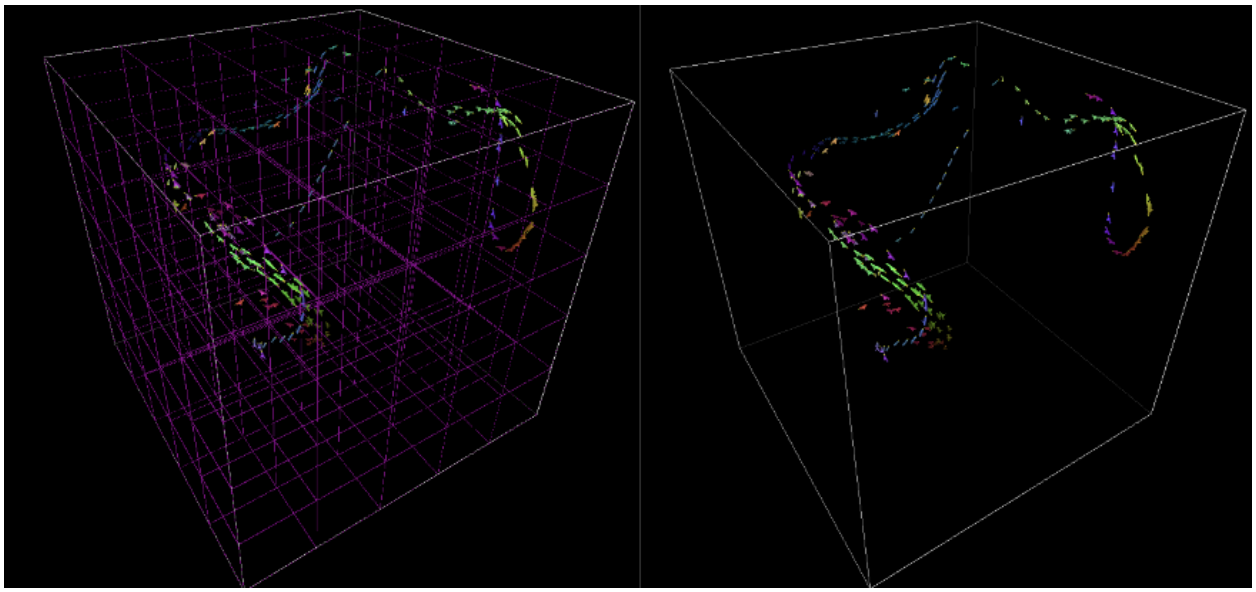
Spatial Hash:

The spatial hash partitions the simulation box into larger cells. The boids are then hashed and binned by cell location. The hash table is used to provide a shortlist for information gathering, giving each boid a sense of which objects are possible valid influences. The hash table is updated during each time step to keep an accurate track of the boids. Since we parallelized the updating process of each boid, there are parallel writes to this data structure, so we must add additional measures to ensure thread safety.



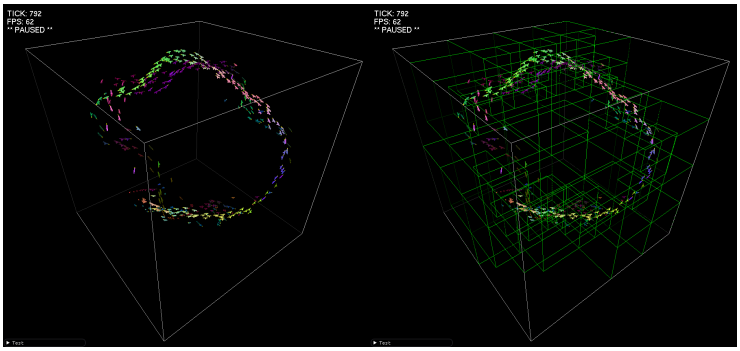
We use fine grain locking to give each cell its own lock. This allows for some parallelization as different cells of the hash table may be updated as boids are spread apart, but this greatly slows down as boids clump and contend for locks in the same cells. This issue is particularly emphasized as the coherence parameter of boids insist that boids will try to flock towards each other leading to many boids occupying similar space in the hash table.

To limit contention, we can increase the number of partitions by decreasing cell size in the hash. The smaller cell sizes limit the number of boids possibly contending for the same cell. It also shortens the list of possible neighbors by limiting the search to a smaller area, closer to the boid's vision radius due to the smaller cell size. However, as the cells increase, we add additional memory accesses to more hashed cells. Since the cells are hashed, the memory accesses have poor spatial locality, so we see a great increase in computationally expensive cache misses.



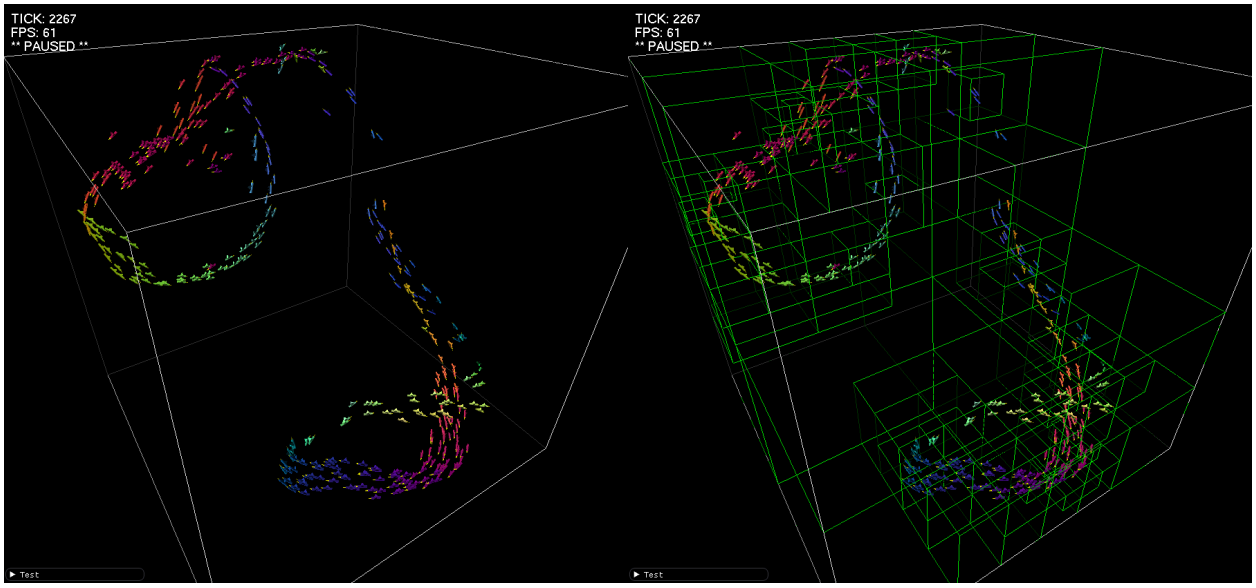
Octrees:

Octrees are a hierarchical tree data structure where each node contains up to eight children, and are the three-dimensional equivalent of quadtrees. In our simulation, since we restrict the simulation bounds to a cubic volume, our octrees are also cubic and span the entire simulation space. Octrees serve to partition the global boid vector into spatially distributed nodes for O(logN) neighbor queries (per boid). When boids are stored into the octree, each boid traverses from the root node downwards until it finds a child octree that still has available capacity. If the boid reaches a leaf octree that is full, the boid will determine the child octant it belongs to (being one of eight subpartitions) and create a new child octree node to add itself to. Thus, for each level descended in the octree, the octree's size halves across all three dimensions.



Once the octree has been fully populated, boids can query the root for its neighbors. In order to facilitate this, each octree has an associated axis-aligned bounding box (AABB) which is used in AABB-Sphere intersection tests. The neighbor query function performs a spatial query on the entire octree, hierarchically pruning children octrees (or branches) if the intersection test returns a negative result. For each valid octree, the distances between stored boids and the querying boid are determined to narrow down true neighbors based on the neighborhood radius.

In the graphics to the right, each green cube represents its corresponding octree and bounding box. For our implementation, we define each octree to have a capacity of eight boids; details on this decision in the results section.



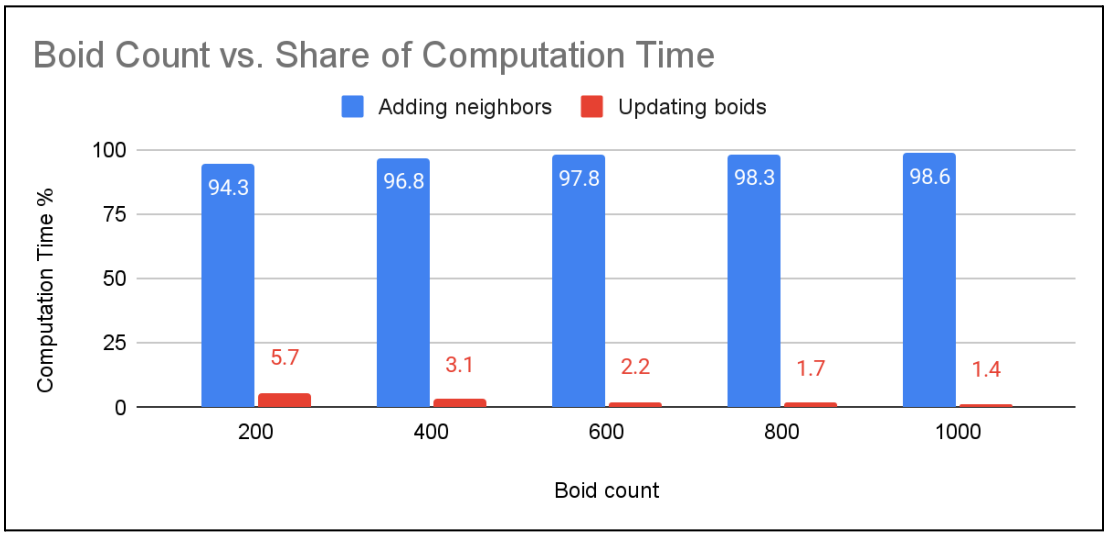
APPROACH

The naive version of our code was adapted from Thomas Rouch's implementation, explained in his Medium article located here.

The way that Rouch's implementation works is through the management of the "Boid" object. At each loop of the simulation, every boid iterates through every other boid and determines if the boid is within its vision proximity. If another boid is close enough, it adds that boid as a neighbor. All of the neighbor information within that time step is compiled into the boid. After each boid collects its neighbors they all go through an update step where they update their position based on compiled neighbor information and clears all of its neighbors in preparation for the next simulation cycle.

To consistently test and gather accurate data, we had to change certain aspects of the code. For one, we had to ensure the code was deterministic, making our tests and simulations consistent across implementations to accurately gauge speedup. To accomplish this, we had to make modifications such as assigning a unique RNG and real distribution to each boid. We also had to change the simulation to take uniform time steps, or ticks, rather than using GLUT_ELAPSED_TIME as this is a register-callback based counter that measures wall-clock time, and is not well suited for evaluating simulations. Lastly, we implemented non-visual simulation (and toggle functionality) and initialization/computation time reporting, similar to asst3. For the graphical simulation, we have a framerate detector that measures the duration of performing each system evolution update.

Once the simulation was repurposed for consistent benchmarking, we proceeded to apply parallelization techniques on the simulation. Again, an initial investigation tells us that the portion of the code that would benefit the most from parallelization is the information gathering stage.



For our parallelization we primarily utilized OpenMP. For our preliminary implementation of parallelism we deployed multiple threads to acquire a single boid and finish gathering the neighboring information. The boids are stored in a C++ vector in which querying for neighbors requires iterating through the dynamic array, comparing positions, and finally compiling neighbor information if it is within our boids vision. This implementation gave us very limited speedup. The addition of multiple threads iterating through the same vector to look for neighboring boids means that there is likely a lot of cache invalidations from other threads as they update their working boid with new information limiting our speedup. We can try to improve our speedup by adjusting the algorithm in which we find neighboring boids.

Spatial Hashing:
The goal with hashing was to isolate the boids that were close to the working boid without having to iterate through every single boid and check the distance. We accomplished this by attempting to hash the boids by position, this way we could just check the position of our working boid, and instantly get a grasp of which boids are possible neighbors. We start by partitioning the simulation area into large chunks. Boids are initialized in their respective positions. These positions are used to calculate their cell coordinates which we hash and place into a hash table. At every iteration, each working boid takes its own position before update and its position after update. If they populate a new cell, the hash table updates the boid by removing it from the old cell and placing it into the new cell.

To make the hash table thread safe at first, we placed the hash table into the critical section, ensuring that concurrent updates to the hash table would not contaminate the information, losing or overproducing boids leading to undefined and erratic boid behavior. However, this was a large bottleneck in our implementation as updates to the hash table had to be performed sequentially. Since we were able to speed up the neighbor gathering phase, the update portion grew to take up a larger portion of the computation time. This is especially true with the non optimal memory accessing patterns associated with hashing meaning that updating the hash was taking a sizable chunk of computation time.

To speedup the update times of the hash, we allowed for fine grain locking, locking individual cells of the hash instead of the whole hash table at a time. This allows for greater concurrency as multiple cells of the hash are now able to be updated at the same time. This leads to much better performance in some scenarios when the simulation width is bigger leading to more diverse and spread groups of boids. However in smaller simulations this change is not as noticeable as boids tend to group around similar clusters of cells, leading to high contention locks, limiting our capacity for parallelism.

Octrees:
The intent of using octrees is much like any other sorting data structure; incur some initial construction penalty to insert all input elements, and amortize this cost when searching through the completed tree. Octrees provide for spatial sorting across three dimensions, and are well suited for the boids neighbor detection task. When boids are inserted into the octree, it traverses the children octrees to which it is a valid member (only one octant per parent octree) until there is capacity available at a leaf node. As each of these octrees has an associated bounding box, when a boid searches for its neighbors, if an octree's bounding box does not come within the neighborhood radius, that octree and all of its children are immediately ruled out, reducing the search space the boid must explore.

As octrees are divided into 8 children every level, a boid searching for neighbors falls into the order of O(logN). In contrast, for the naive approach each boid must visit every single other boid blindly, on the order of O(N).

To first ensure that octrees would be compatible with our project, we found a lightweight C++ implementation by Stefan Annell which "[functions with any] vector class and any generic data blob stored alongside its position." After adapting the implementation to fit the boids simulation, we received promising results. An introductory sweep over increasing boid counts with all other parameters set to default illustrated increasing speedup over the naive implementation. In hindsight, perhaps a better test would be to sweep over thread counts as well to see scaling behavior.

Boids	200	400	800
Ref Speedup	122.99%	152.12%	160.57%

With these results, we set out to write our own implementation of octrees, using Annell's implementation as a reference. As a result of his code being entirely generic, it has rather poor readability. Our implementation is designed to fit our boids project exactly. When querying for neighbors, we only specify the AABB-Sphere intersection test used for the neighborhood radius.

Thus, on each simulation tick, we construct an octree and populate it with the updated boid positions. After this initial cost, we enter the neighbor query phase and parallelize over boids, where each thread is assigned a boid and searches the octree for that boid's neighbors. Being entirely read operations, we ensure thread-safety. Lastly, we parallelize over boids once more and perform a fast update using the boid's neighbors. Notably, we do not parallelize the octree construction phase. This is a very clear bottleneck, but unpredictably destructive harmful performance at higher thread counts led us to choose project integrity throughout over gains at reduced cores. This is investigated in much more detail in the results section, where we examine sources of speedup limiting.

RESULTS

In our project, our primary performance metric of interest is speedup. While not critical to our results, we can also demonstrate frame rate improvements when rendering the simulation graphically in real-time. We only present total speedup, as initialization time is constant and negligible (0.006ms) across all runs.

The original application, developed by Thomas Rouch and available at his public repository, is a graphical simulation implemented in C++ that comes with knobs for specifying a number of parameters, such as boid count, separation, cohesion, and alignment. To limit our experimental space, we keep the default separation/cohesion/alignment coefficients, and discard obstacles and targets that his repository also provides. We find that results with these knobs succeed in generating compelling flock behavior. However, these knobs can absolutely affect parallelization performance, and could be explored in future works. For example, increasing cohesion and separation hypothetically leads to spread-out boids clustered in the same general spatial region, which could worsen octree performance by incurring more false queries.

In addition to the many knobs the base application comes with, we introduce the following input parameters that are specified at runtime:

- -i implementation (naive|octree|hash)
- -n number of threads
- -b number of boids
- -t (simulation) time in ticks
- -w (simulation) width
- -s seed

Besides the other parameters whose use is obvious, specifying a seed ensures a simulation is guaranteed fully deterministic from one run to the next (assuming stable boid count throughout the simulation), which is critical for meaningful results gathering. Each boid is assigned its own seed (based on the input seed), random number generator, and uniform_real_distribution for controllably-random behavior, while ensuring thread-safety.

We use these parameters to shape our experiments in order to explore parallelization effects across our experimental space. Once the simulation completes, parameters and results are logged to simulation_results.csv, with some results such as initialization and computation time also printed to the terminal.

```
Python
# Runs a simulation using spatial hashing and 128 threads on 1600 boids for 2000 ticks in a boundary space of 100x100x100 with seed 999
./main -i hash -n 128 -b 1600 -t 2000 -w 100 -s 999
```

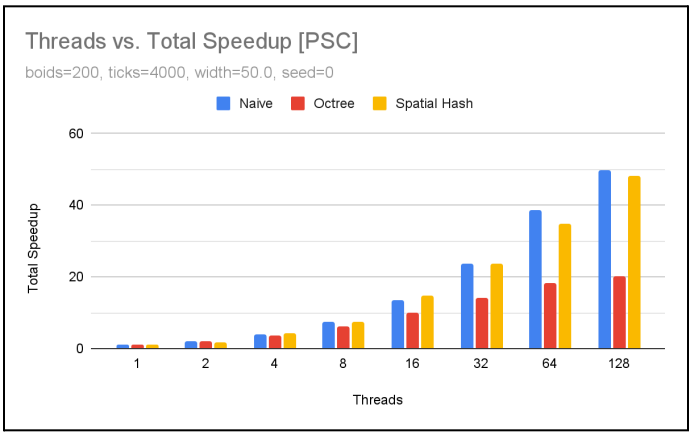
Additionally, we created a git branch that strips all visualization functionality (and heavy graphics packages) from the project so that it can be run on PSC's Bridges-2. This pure simulation branch is what we will submit.

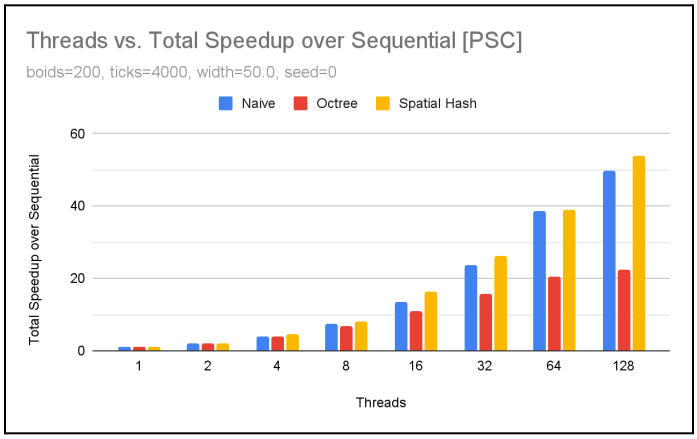
PSC RESULTS:

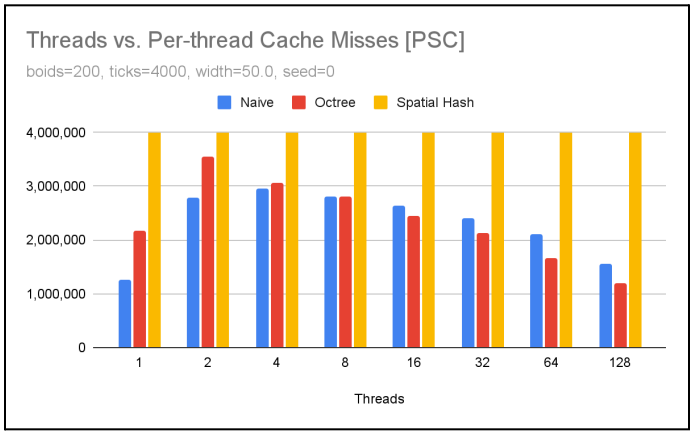
The bulk of our experimentation is conducted on the Bridges-2 cluster. Interested in ramping simulation complexity, we conduct three sweeps over increasing boid counts.

200 BOIDS

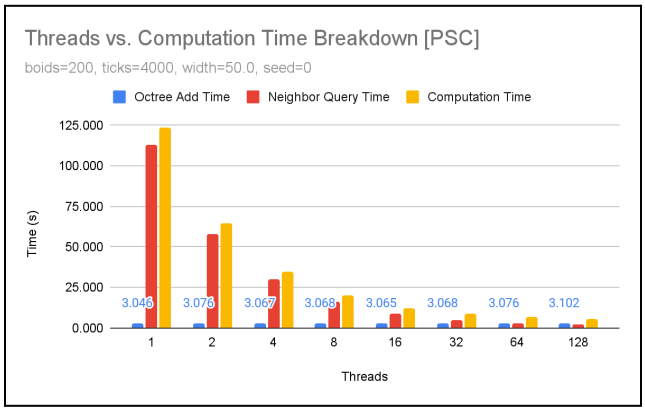
In this first experiment we see that the naive implementation demonstrates the best scaling relative to single-core performance, but spatial hashing scaling keeps up, while actually performing better relative to the sequential simulation.

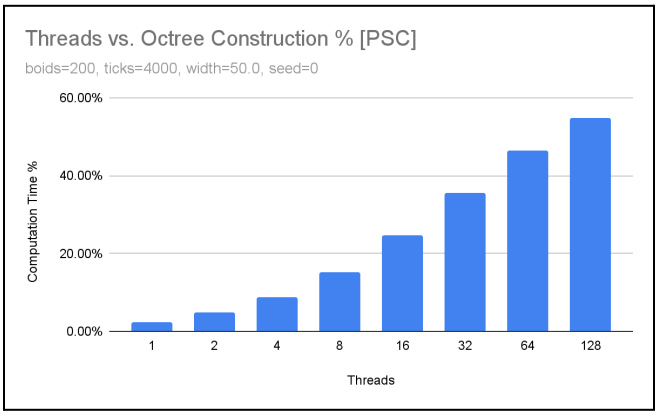






Remarkably, octree performs quite poorly in comparison to the other two implementations, leading us to lead further inquiry and collect more data.





When rerunning the 200 boids test for the octree implementation, we discovered that the construction of the octree structure was constant time despite computation time scaling quite well with increasing thread count. This proves a very large bottleneck and severely limits the overall speedup available to us via Amdahl's law. Of course, this was logical as we did not parallelize the construction of the octree. We first considered hand-over-hand locking, but realized fine-grained locking would reduce the critical section further; instead of locking an entire node (or two at a time), we can lock just the data field when adding boids, and lock the children field when creating a new leaf node.

However, once we reran with our lightweight parallelism locally, we anecdotally noted worse overall performance, particularly for octree construction time. While low threads resulted in similar octree construction times, as threads approached 16, construction worsened by several factors. Having not recorded results, we tried sweeping over the PSC 200 boids test mentioned above to provide quantitative backing, only to discover that construction time generally improves with fine-grained locking and parallelism, but seemed to explode at the maximum thread count.

octree (fine-grained locking)								
Threads	1	2	4	8	16	32	64	128
Octree Construction Time	3.109	1.892	1.381	1.004	0.797	0.715	0.869	3.235

Hoping to confirm this was not a fluke, we ran on 128 threads numerous more times, only to yield wildly divergent results, ranging from ~1.7s to ~100s.

Run #	1	2	3	4	5	6	7
Octree Construction Time	7.11	3.93	1.73	2.51	25.35	25.49	25.83
Neighbor Query Time	8.18	5.19	2.82	3.71	78.74	79.47	78.11
Update Time	6.39	3.24	1.07	1.87	47.07	47.27	47.38
Computation Time	21.68	12.36	5.62	8.08	151.15	152.24	151.32

Even worse, it appeared that whatever was causing this terrible behavior would leak over and affect both neighbor query and boid update times, the latter being especially absurd considering it was previously only 5% of the computation time at worst and entirely sequential. Computation times were in the range of ~5.6s to over 150s.

It is possible that lock contention and cache thrashing became suddenly significant at 128 threads, leading to performance degradation. To address this, we attempt to limit the maximum number of threads allowed to construct the octree in parallel. When running with `#pragma omp parallel for num_threads(64)` over the construction loop, 128-threaded performance would exhibit the same divergent behavior, and now 64 thread performance showcased similar issues.

Unable to pin why this occurred, we ultimately chose to leave the octree construction section sequential, as the sudden, unpredictable, and massive performance hits meant keeping the error-prone parallelization was untenable. Future work would benefit from a much closer look into this strange behavior.

Using perf stat, we also collected cache miss statistics for the entire sweep. Octree actually demonstrates the best caching at scale, whereas the spatial hash is substantially worse when it comes to caching. The hash implementation's cache misses in the previous figures were truncated so as to not dwarf the visibility of naive and octree implementations.

Here are some tables that illustrate it better:

Total cache misses (4000 ticks, 200 boids)								
Threads	1	2	4	8	16	32	64	128
Naive	1,251,314	5,574,325	11,829,695	22,408,695	42,371,858	76,685,704	134,967,450	199,238,333
Octree	2,180,504	7,108,352	12,281,346	22,401,078	39,311,858	67,959,445	106,660,847	152,210,972
Hash	3,080,390,070	2,877,761,072	2,765,864,260	3,073,198,844	2,746,998,359	2,885,990,265	2,948,778,134	2,795,844,352

Per-thread cache misses (4000 ticks, 200 boids)								
Threads	1	2	4	8	16	32	64	128
Naive	1,251,314	2,787,163	2,957,424	2,801,087	2,648,241	2,396,428	2,108,866	1,556,549
Octree	2,180,504	3,554,176	3,070,337	2,800,135	2,456,991	2,123,733	1,666,576	1,189,148
Hash	3,080,390,070	1,438,880,536	691,466,065	384,149,856	171,687,397	90,187,196	46,074,658	21,842,534

Despite exhibiting nearly 14x more total cache misses than the other implementations at its most favorable comparison (128 threads, hash vs. naive), a fascinating result is that total cache misses remain fairly constant across different thread counts. The vast increase in cache missing is likely due to the erratic memory access patterns associated with hashing. Since the cache misses are constantly occurring due to the poor spatial locality of hash access, there is little increase in cache misses due to increasing thread counts. Even if cache lines are missed due to coherence misses, they are unlikely to play a substantial part in the large number of misses due to hashing.

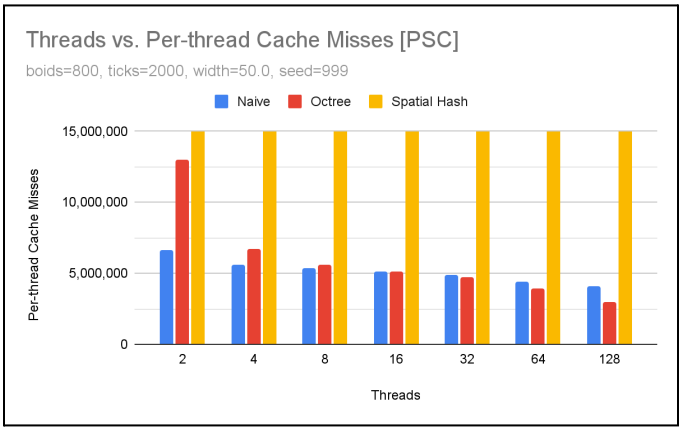
800 BOIDS

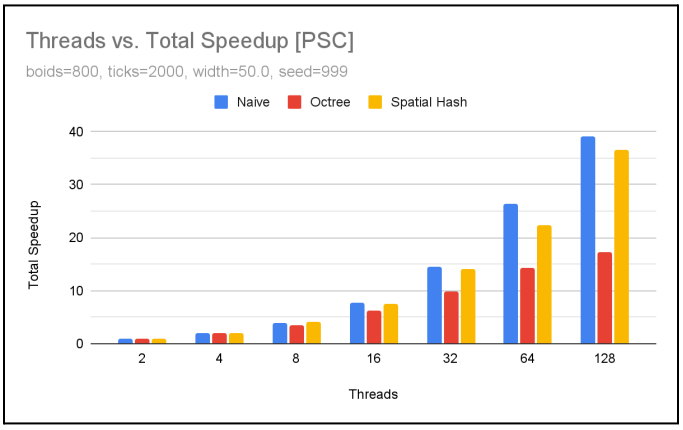
To cut down on evaluation time, simulation duration is cut down from 4000 ticks to 2000 ticks, and single-core simulations are not evaluated (besides the sequential implementation).

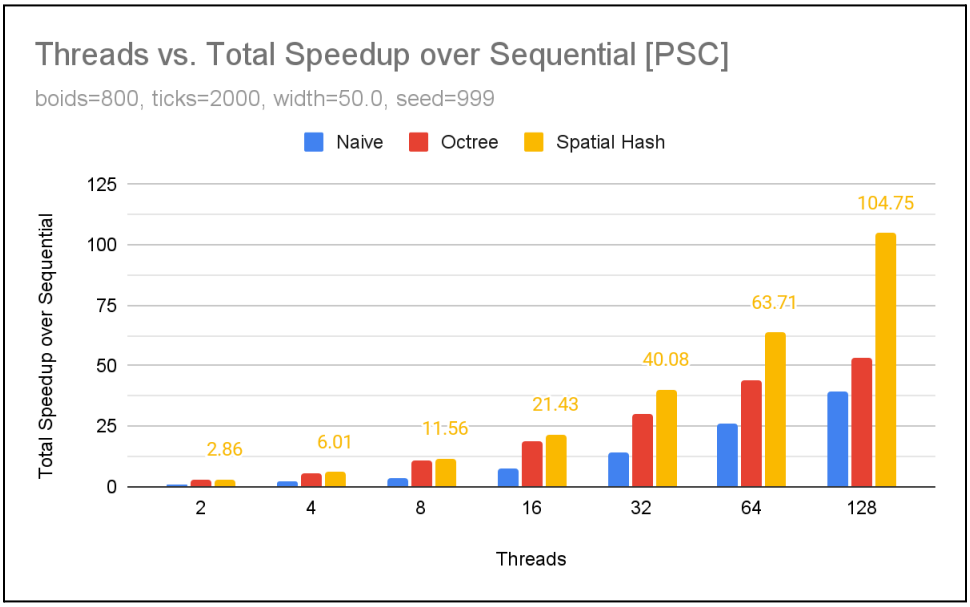
With increased simulation complexity coming in the form of 4x more boids (and thus increased dependencies), we see that the octree and hashing implementations begin to leverage their algorithmic advantage over the naive implementation. The octree demonstrates the best performance at reduced core counts, but does not show the same continued scaling capability as the spatial hashing implementation does. Cache miss behavior is largely the same, with octrees illustrating improved cache utilization as cores increase, while total misses remain constant across threads for spatial hashing.

Total cache misses (2000 ticks, 800 boids)							
Threads	2	4	8	16	32	64	128
Naive	13,261,593	22,504,727	42,618,480	81,636,151	155,463,346	283,892,482	520,525,252
Octree	25,977,207	26,819,100	44,656,258	81,514,100	150,892,628	249,239,577	382,251,814
Hash	18,393,235,527	22,640,998,683	22,959,956,709	24,258,473,898	23,013,180,671	23,942,747,965	21,577,427,903

Per-thread cache misses (2000 ticks, 800 boids)							
Threads	2	4	8	16	32	64	128
Naive	6,630,797	5,626,182	5,327,310	5,102,259	4,858,230	4,435,820	4,066,604
Octree	25,977,207	26,819,100	44,656,258	81,514,100	150,892,628	249,239,577	382,251,814
Hash	9,196,617,764	5,660,249,671	2,869,994,589	1,516,154,619	719,161,896	374,105,437	168,573,655

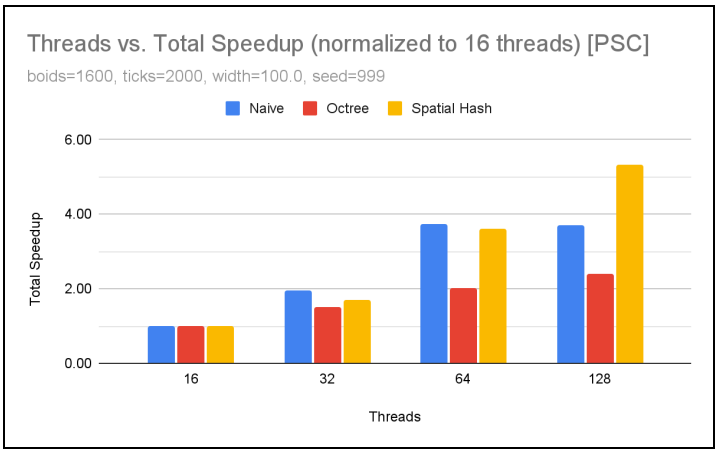




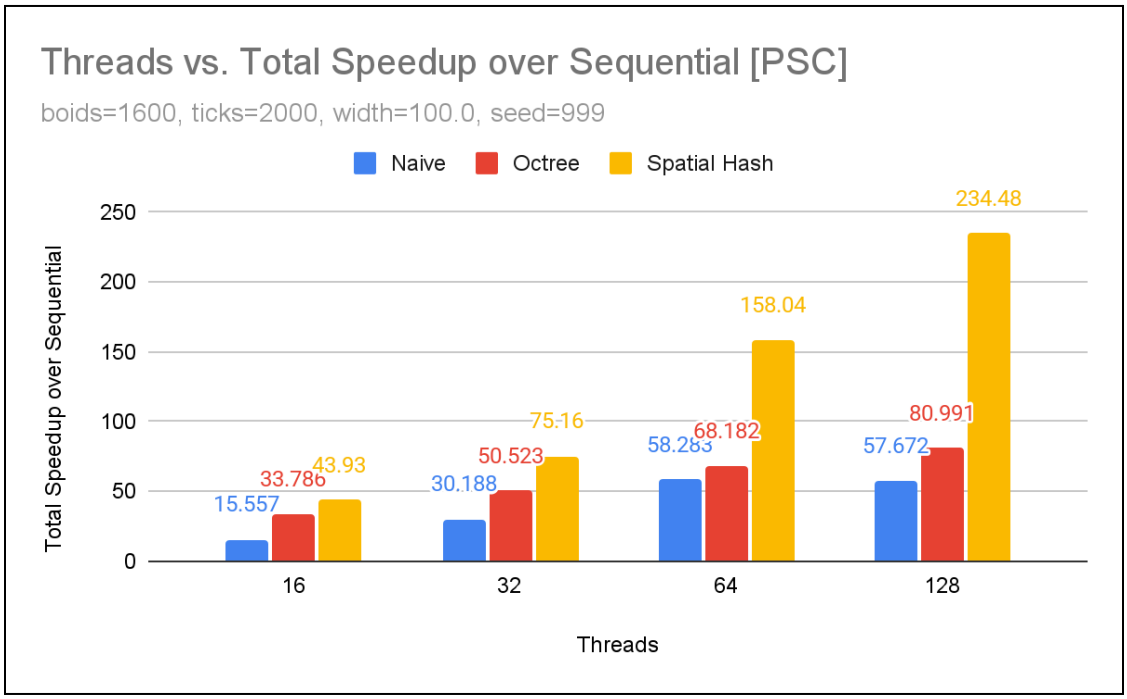


1600 BOIDS

In addition to doubling the boids from the last experiment, we also increase the size of the bounds to cumulatively represent a highly complex simulation scenario. We do not run the experiment on thread counts lower than 16 to reduce overall evaluation time, with the exception of the sequential implementation, which took nearly 40 minutes to complete.



In this final experiment on Bridges-2, the spatial hash implementation shines, scaling much better than the other two implementations given a more complex scenario. While intra-implementation speedup trends largely mirror previous results, we see that the naive implementation plateaus at 128 cores. Compared to the sequential implementation, we see that both binning implementations outperform the naive implementation, but the spatial hashing implementation demonstrates superscaling across all thread count configurations. We can likely attribute this superscaling behavior to the improved algorithmic complexity of our binning methods. The added caches are not likely to help hashing as the misses are occurring due to non optimal memory access rather than capacity. The octree implementation also demonstrates superscaling to a lesser degree, and falls off quicker. It likely falls off because of a bottleneck in sequential octree construction, limiting the scalability of the implementation for high thread counts. While the spatial hashing implementation's results indicate that it would not maintain the superscaling into even greater thread counts, it stands heads above the competition as simulation complexity multiplies.

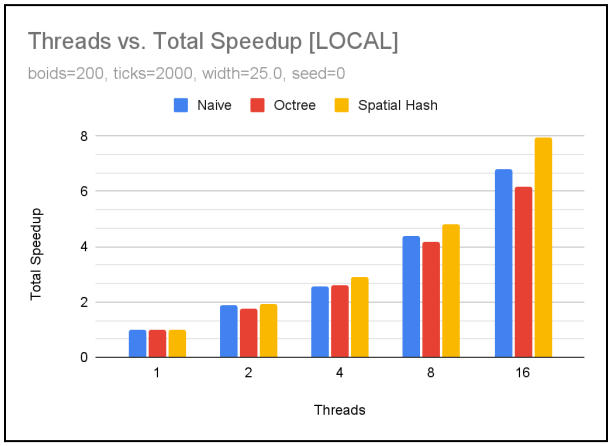


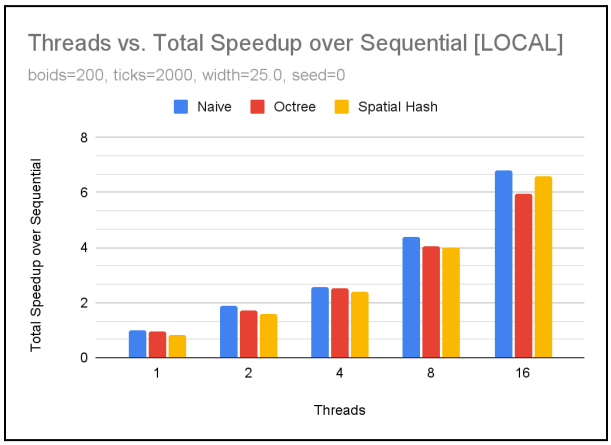
LOCAL RESULTS:

The machine we used to evaluate locally is equipped with a Ryzen 7 3700X CPU, which has 8 cores and 16 total threads. As we were running our simulations in WSL, we did not have access to hardware performance counters and thus did not report cache statistics for these tests.

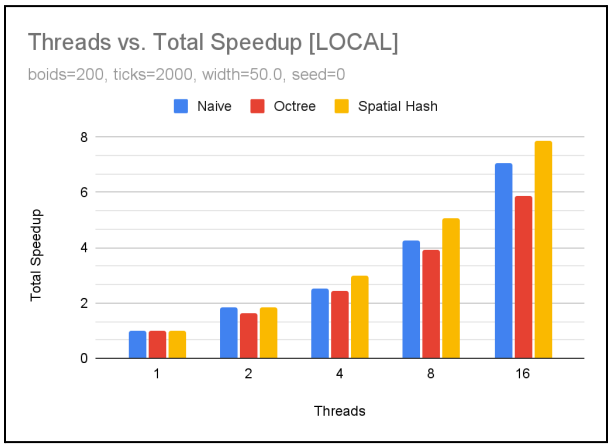
One parameter we have yet to isolate and explore is simulation width. This parameter directly contributes to the sparsity of the data, which we hypothesized that spatial binning would improve, by reducing the number of neighbor candidates considered per each boid.

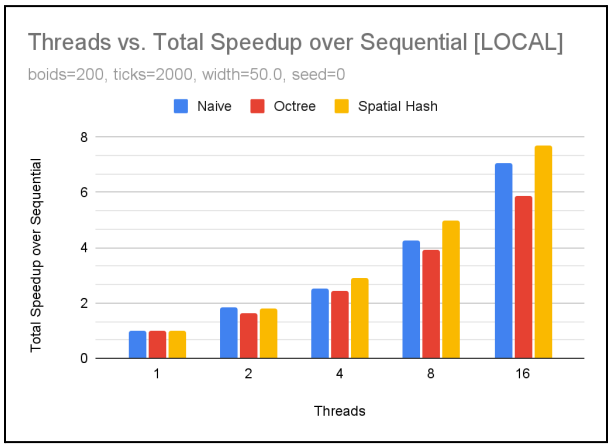
WIDTH = 25.0



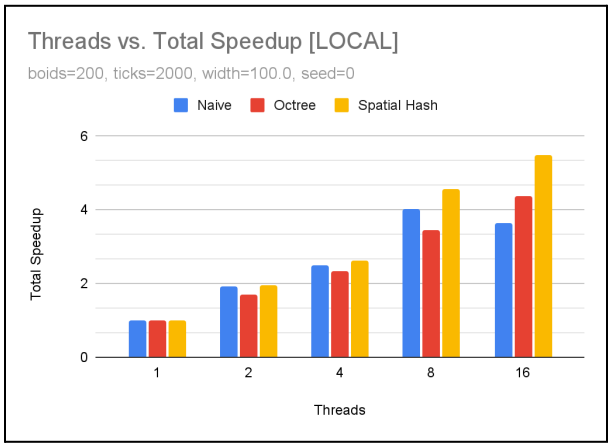


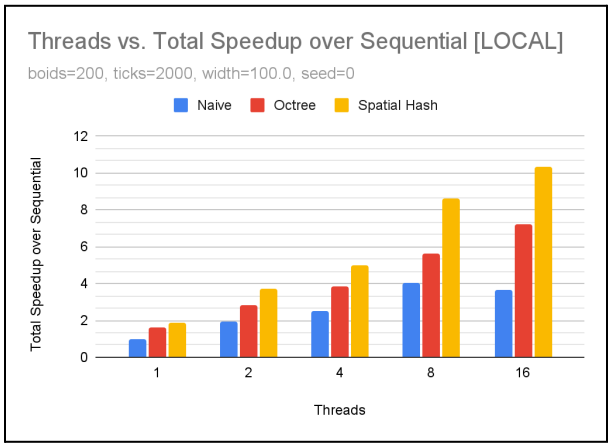
WIDTH = 50.0



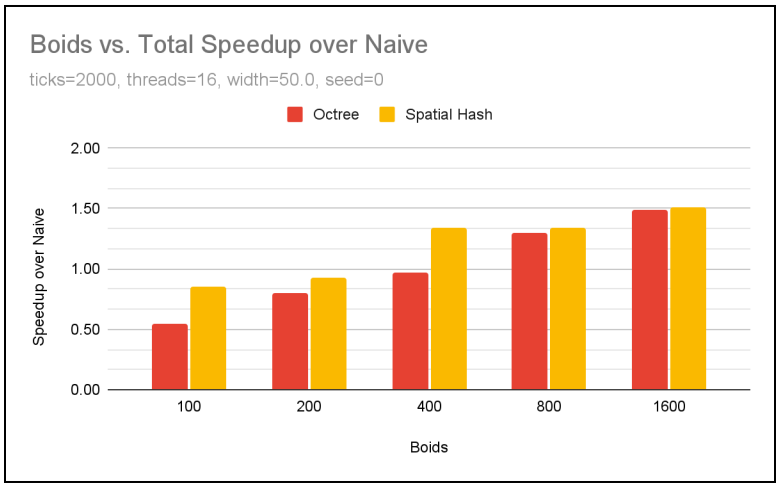


WIDTH = 100.0



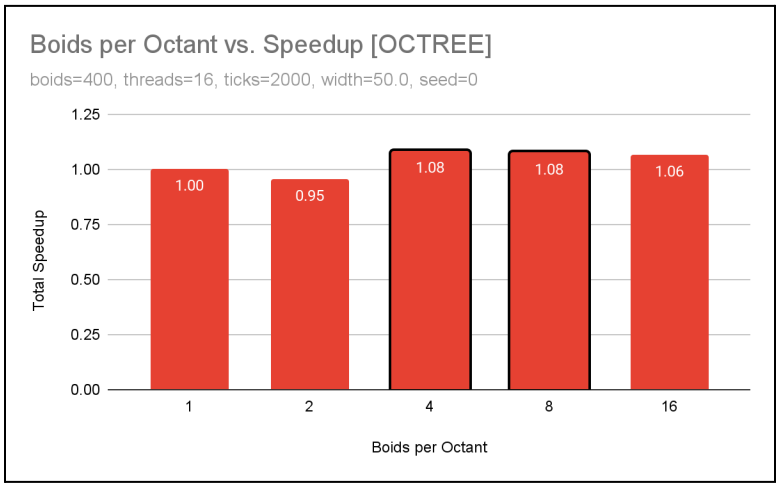


As the results show, as the simulation boundaries expand, spatial binning further demonstrates its advantage in sparse boid environments. When the simulation is tightly constrained, boids are extremely likely to be neighbors with each other (especially for the default neighborhood radius = 10.0), so the advantage of shortlisting candidate boids becomes less pronounced. For the octree and hash implementations, this just means extra overhead just to manually double-check the near-same quantity of candidate boids. As the space expands and boids fly more freely, narrowing down the number of initial candidate neighbors becomes much more crucial.



Following this thread, we sweep over boid count as well, this time keeping simulation width fixed at 50. We see a similar trend, where the binning algorithms show improved performance for simulations involving higher boid counts. The naive algorithm faces $O(N^2)$ candidates for each simulation tick, while the octree and spatial hash consider candidates on the order of O(NlogN) every simulation tick.

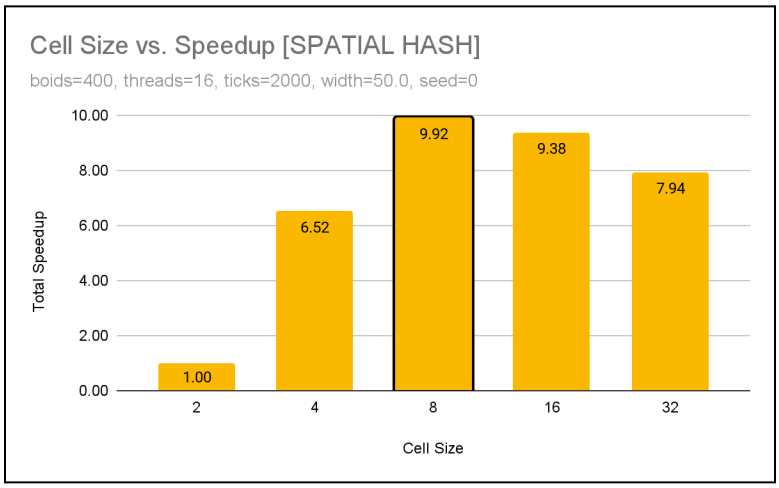
We can conclude that our spatial binning approaches are effective at handling increasing simulation complexity, particularly boid counts and simulation boundary size.



Two last parameters to sweep over are specific to the binning implementations. As alluded to prior, octrees are constrained to have capacity for exactly 8 elements. In this experiment, we sweep over boids allotted per octree node.

Overall, we see minimal difference between the allotments, but we choose capacity to be eight for all of our other experiments.

Secondly, we sweep over spatial hash cell size. Cell size allows us to have some more precision when developing a shortened list of neighborhood candidates. The smaller we make the cell size, the more certain we can be that the nearby hashed cells contain boids within the working boid's vision. However, as we decrease cell size, we are also increasing the number of hash lookups needed to find possible neighbors. Hashing has poor locality, so decreasing the cell size can lead to an extreme increase in expensive memory access. We try to find an appropriate middle ground to conduct our results gathering on. With these parameters, we determine that 8 is the best cell size.



RESULTS CONCLUSION:

In our project proposal, we set out to accomplish the following set of objectives:

1. Optimize the boid simulation using binning techniques to reduce neighbor querying
2. Achieve an amortized algorithmic complexity of O(NlogN) for N boids
3. Ultimately achieve a total speedup of 10x over the sequential implementation on 8 cores, surpassing the ideal 8x via intelligent shared memory usage

On all accounts, we have achieved these initial goals. We explored two different approaches, being octrees and spatial hashing for improved neighbor querying, and presented results that demonstrate impressive scaling for increasingly complex simulation schemes, such as increased boid counts and simulation boundary sizes. For the 800 boids experiment on Bridges-2, we saw both the octree and hash approaches attain speedups over 10x compared to the sequential implementation (naive only achieved ~4x), and in our most complex experiment of 1600 boids with simulation width = 100, octree and spatial hashing improved significantly over the naive approach. Spatial hashing has shown itself to be very well suited for this task, achieving a maximum 234x speedup on 128 threads in that experiment. For scaling the simulation to larger widths and boid counts, hashing provides the best improvement with low overhead and great pruning benefits. Our octrees approach could use some further optimization in octree construction. Currently the construction is sequential, greatly hindering our capacity for speedup especially as the simulation scales to more involved and complex scenarios. Further investigation will be taken.

REFERENCES

[1] – Reynolds, C. W. (1987). Flocks, herds, and schools: A distributed behavioral model. Computer Graphics, 21(4), 25-34.
[2] – Craig Reynold's Personal Website: Boids
[3] – Mastering Flock Simulation - Thomas Rouch
[4] – Thomas Rouch's Boids Simulator
[5] – Stefan Annell's Generic Oct/Quadtree

LIST OF WORK, CREDIT DISTRIBUTION

Josiah: Deterministic runs, benchmarking infrastructure (e.g. command-line arguments, timing, CSV logs), octree approach, and all results gathering and graph creation.

Ryan: Naive parallelism, frame-rate and time-constrained benchmarking, spatial hashing approach

Total project credit should be distributed equally, 50% - 50%.