High-Performance-Simulation-of-Boids-A-Comparative-Study-using-CUDA-and-Message-Passing

https://github.com/AmaadMartin/High-Performance-Simulation-of-Boids-A-Comparative-Study-using-CUDA-and-Message-Passing?tab=readme-ov-file

Summary

For our project we wrote several parallel implementations for a Boid Simulator. This included a CUDA implementation with advanced memory management, and an OpenMPI implementation that involved grid based load balancing. We tested our implementations on several benchmarks and considered multiple design choices.

Background

The concept of a "boid" stems from simulating the aggregate motion of flocks, herds, or schools seen in birds, land animals, or fish. This form of motion, intricate yet fluid and coordinated, has rarely been captured accurately in computer animation. Our project involves simulating this complex behavior using boids, an approach that treats each member of the flock as an individual but interdependent entity.

Boids operate under a distributed behavioral model. Each boid acts as an independent actor, navigating based on its perception of the environment, the laws of simulated physics, and a set of pre-defined behaviors. The interaction of these simple behaviors across multiple boids results in the complex, aggregate motion characteristic of natural flocks.

Tech Stack

For the base code and the sequential simulator we used C++. We used the CUDA and OpenMPI libraries for the parallel implementations.

CUDA

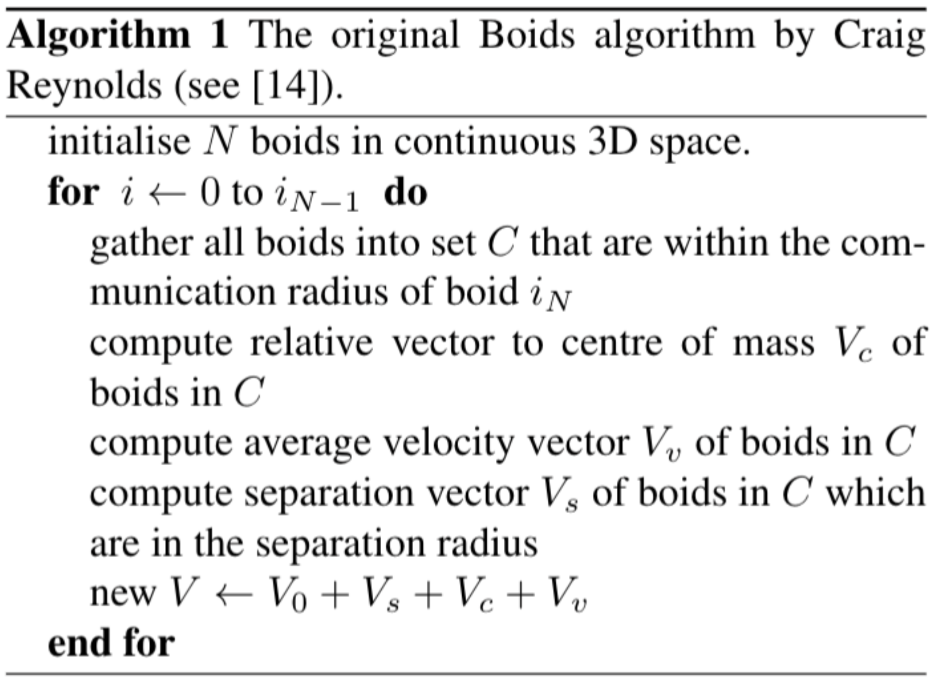
For the CUDA implementation we used ghc machine 63 which has access to an NVIDIA GeForce RTX 2080 B GPU

OpenMPI

For the OpenMPI implementation we used the PSC Bridges-2 RM Machines.

Sequential Algorithm

The sequential algorithm for simulating Boids is quite simple and similar to n-body simulations.



Algorithm 1 The original Boids algorithm by Craig Reynolds (see [14]).

```
initialise $N$ boids in continuous 3D space.
for $i \leftarrow 0$ to $i_{N-1}$ do
    gather all boids into set $C$ that are within the communication radius of boid $i_N$
    compute relative vector to centre of mass $V_c$ of boids in $C$
    compute average velocity vector $V_v$ of boids in $C$
    compute separation vector $V_s$ of boids in $C$ which are in the separation radius
    new $V \leftarrow V_0 + V_s + V_c + V_v$
end for
```

The three velocities are the separation (Move or steer to avoid crowding local flock-mates), cohesion (Move or steer toward the center of mass of local flock-mates) , and alignment (Steer in such a way to align self with the heading of local flock-mates) velocities. Although it is quite simple there is a nested for loop which causes the complexity of the algorithm to be O(n^2) with respect to the number of boids which doesn't allow for efficient simulation with high numbers of boids. However there is a lot of room for parallelization which leads to our solutions.

Parallel Solutions

The Boid simulation is heavily parallelizable in multiple ways. Each boid is updated independently which can allow for a boid based load balancing scheme or a grid based balancing scheme

CUDA

For the boid based load balancing scheme we attempted a CUDA implementation of the simulation algorithm.

Using GPUs for n-body like simulations is not a new concept and has been researched a decent amount. For our implementation we took inspiration from Simon Greens "Particle Simulation using CUDA" from Nvidia themselves.

This implementation leverages per particle parallelization which in our case translates to per boid parallelization allocating a thread to each boid in the simulation.

We allocated threads to boids in the following way:

```
int idx = blockIdx.x * blockDim.x + threadIdx.x;
if (idx < flockSize)
```

And found the best overall performance with 256 threads per block initialized as follows

```
int threads = numThreads;
int numBlocks = (size + threads - 1) / threads;
```

Simply leveraging load balancing gave speedups of up to 728x! However we were able to improve this with smarter memory management.

In Green's paper he described two algorithms for memory management. Building a grid with atomic operations and bins and building a grid using sorting. He highlighted the fact that sorting was generally better due to memory coherency when updating vectors from neighboring particles and reduced warp divergence as the closer as the particles close in space (i.e. more likely to be neighbors) were put in the same warp. In addition, he said that distribution of the particles significantly affects the atomic implementation.

The algorithm for sorting the grid is as follows:

1. Calculate Hashes for each Boid
2. Create (Hash, Boid) array
3. Sort array by hashes
4. Calculate starts and ends of each hashed value in sorted array

What this algorithm does is find each boids grid cell then hashes that cell into a 1D line to allow for sorting. Therefore, the hashes that are close in value will be close in memory. This allows for another design choice which is choosing the hash. Green discusses the trivial linear hash, however, he also talks about Z-order curve hashing and through further research I found the Hilbert Curve to also be promising.

The cells of the grids are initialized to be two times the radius of each boid so that the only cells that could interact are the neighboring eight cells around. During the update step only these cells are iterated through to update a boid.

OpenMPI

We also attempted an OpenMPI implementation of the Boids algorithm, where each process had its own memory and would only communicate necessary information. We then compared it to the CUDA implementation.

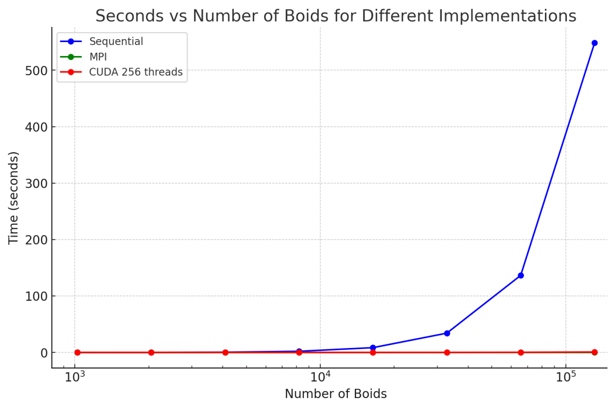
To maximize parallelism, we divided the space into the grids, where each process was assigned one of these divided spaces. Each process from then out was responsible for the Boids that resided within this space of the grid.

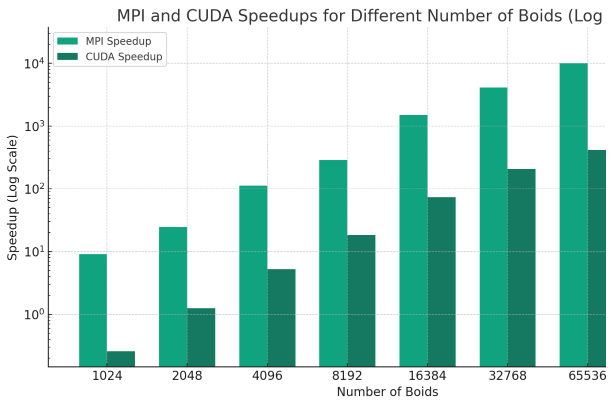
However, there still remains the possibility that there exists a boid in one grid space that is dependent on a boid in another. Thus, processes must communicate their current grid bounds to determine whether or not it is possible for another boid to be within the communication radius. If this is the case, they then send the boids that are to each other. Now that every process has the necessary boids to perform computations, we build a quad tree so we can quickly determine, which boids are within a communication radius for a specific boid. Finally, each process performs a step in our simulation according to the rules of the Boid algorithm.

Results

To benchmark our results we used 8 different numbers of boids initially in the box ((0,0), 10000, 10000) (origin, width, height) against the sequential implementation, MPI with 121 processors, and CUDA with 256 threads per block

- 1024
- 2048
- 4096
- 8192
- 16384
- 32768
- 65536
- 131072



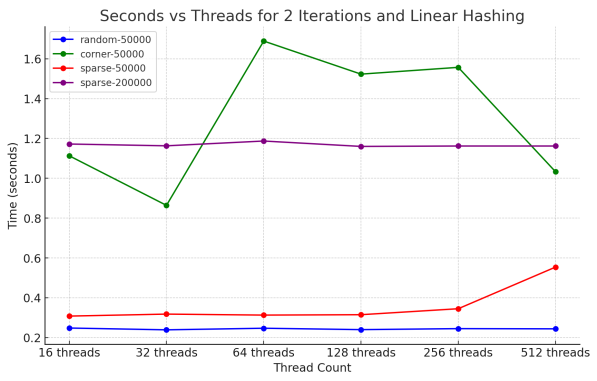


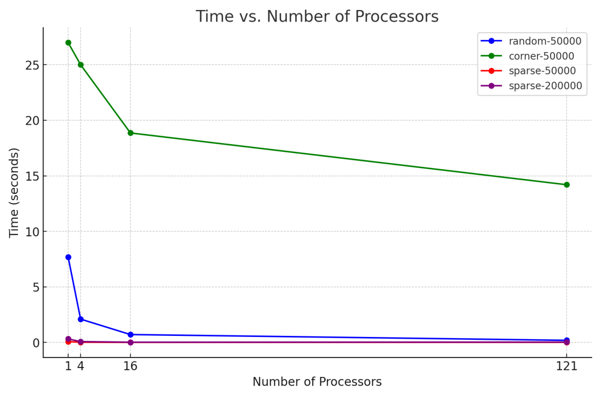
The speedups we experienced were definitely expected due to the n^2 nature of the sequential algorithm. To further test the MPI against the CUDA we could have tried even higher counts of boids but we would not be able to use the sequential algorithm as it would probably end up nearing hours for simulation.

We also tested 4 different scenarios with varying boid positions and numbers on different numbers of threads and processors.

- Random-50000: A scene of 50000 boids initially in the box ((0,0), 5000, 5000)
- Corner-50000: A scene of 50000 boids initially in the box ((0,0), 5000,5000)
- Sparse-50000: A scene of 50000 boids initially in the box ((0,0), 100000)
- Sparse-200000: A scene of 200000 boids initially in the box ((0,0), 1000000)

We collected the following graphs:





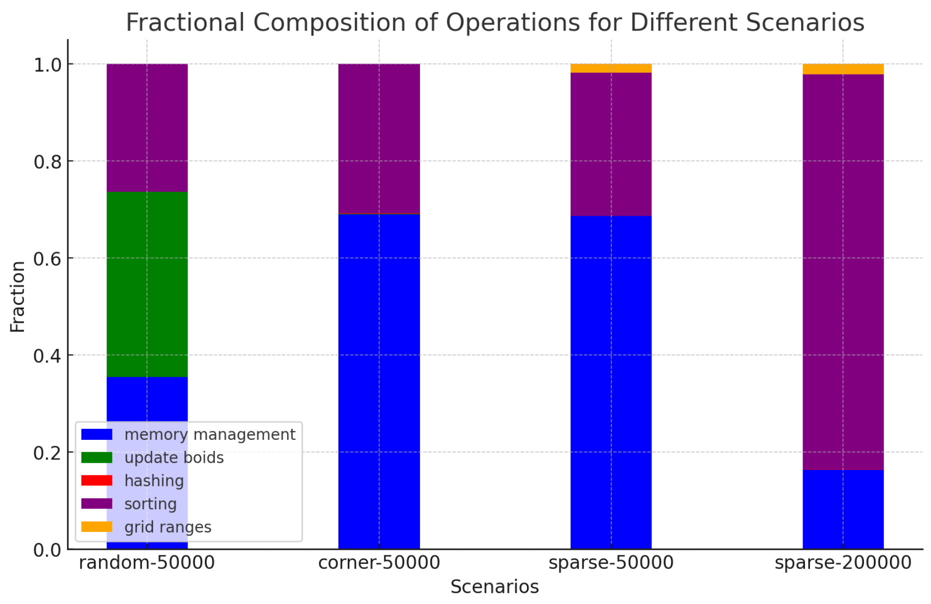
We observed that changing the threads per block for boid Based parallelization had marginal effects. This is actually not surprising, however, due to the fact that the threads in separate blocks don’t actually have any correlation with the algorithm nor share any dependencies. With regards to changing the number of processors we did expect an increase in performance as each processor corresponded to a cell and more processors should have correlated to more balanced loads. However, the corner case for MPI didn’t perform as well which was also expected as every boid has to interact with every other in this case, and the work is being done on one processor.

Further Analysis

For both implementations we analyzed a single step and divided up the time into multiple sections

CUDA

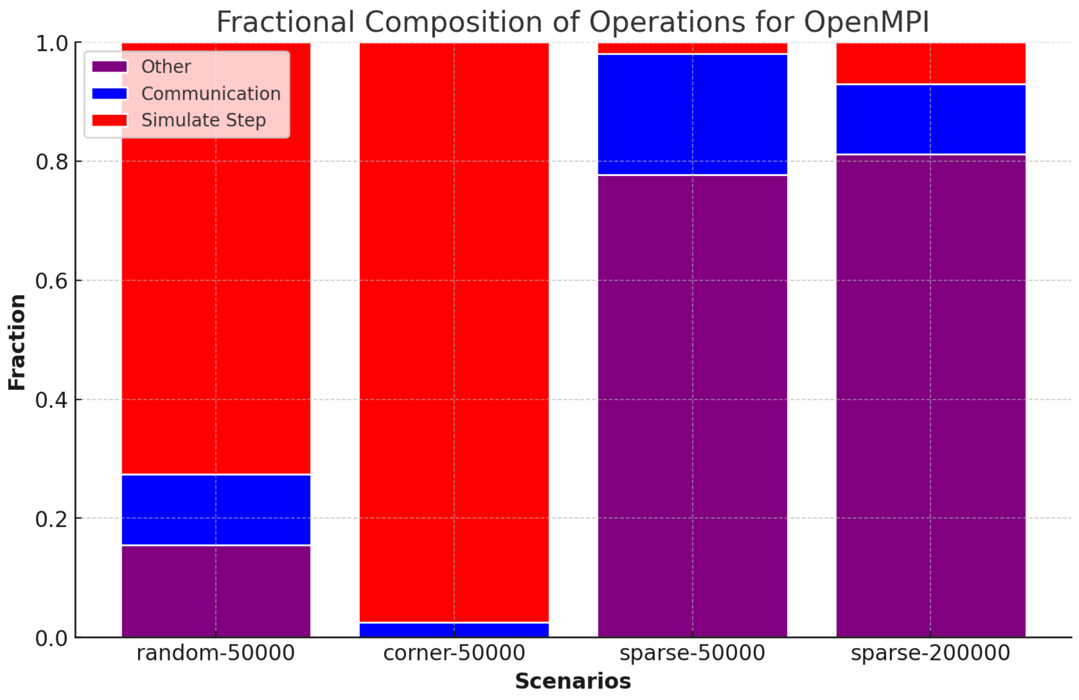
- Memory Management (copy from host to device, allocating memory)
- Update Boids (updating all boid positions and velocities
- Hashing (hashing boids)
- Sorting (Sorting (hash, boid) tuples
- Grid Ranges (Calculating starts and ends for cells)



Upon further analysis of the CUDA implementation it was evident that managing the memory took most of the time. We attempted to cut down this time as much as possible but it is known to be expensive to move memory from host to device and therefore it wasn't completely avoidable. Sorting also took a considerable amount of time. To try and mitigate this I switched from using std::sort to thrust::sort which is provided by Nvidia to support sorting utilizing GPU resources and noticed a considerable speedup. Other operations were negligible besides the update operation in the random case, however, most of our efforts already attempted to reduce this.

OpenMPI

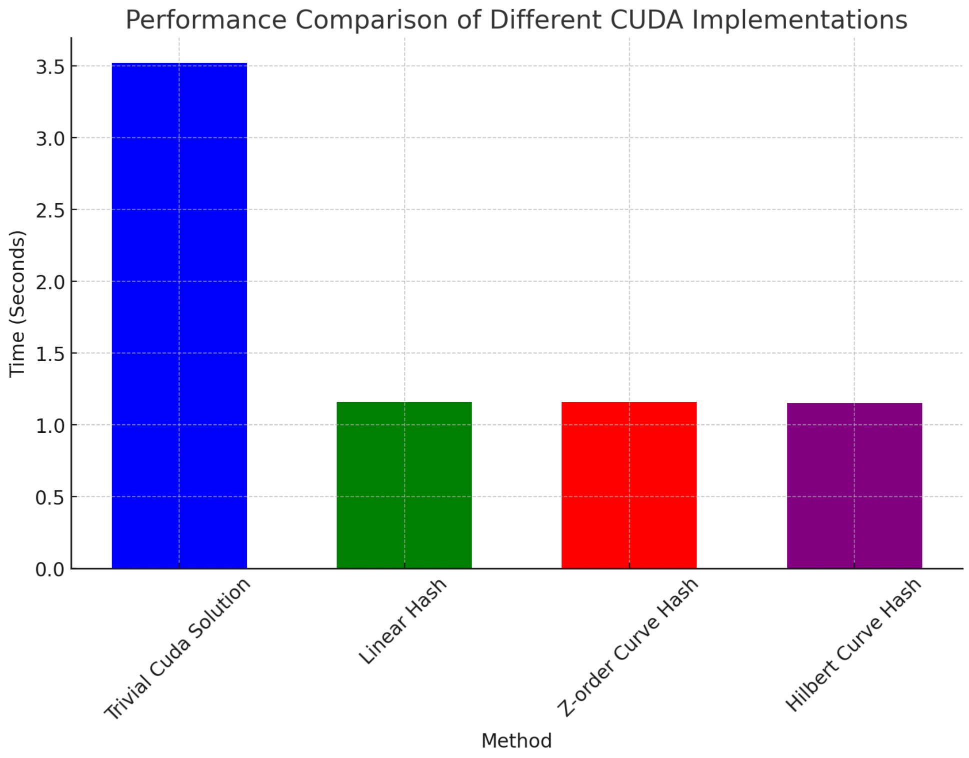
- Communication
- Simulate Step
- Other



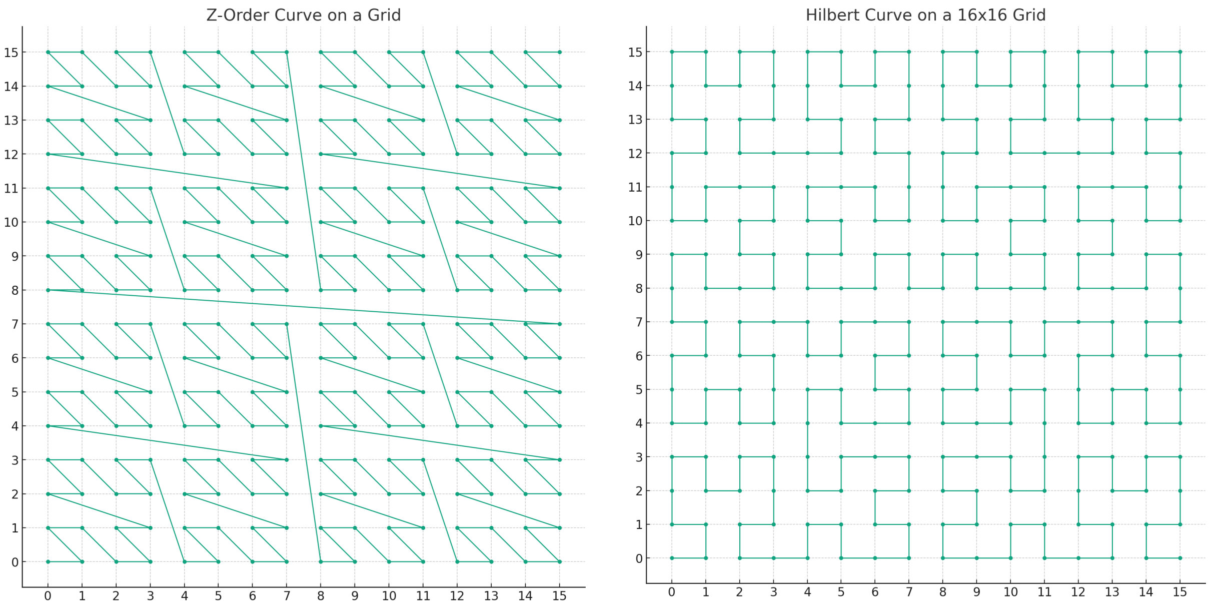
Upon analyzing this plot, we came to the conclusion that OpenMPI implementation was bottlenecked by the most expensive process. This became most apparent with the edge case corner-50000. The distribution of time was dominated by the simulated step. This could be explained by the nature of the load balancing effort of the OpenMPI implementation, where we split the grid into subgrids. Initially, a large proportion of all the boids are in the corner, and thus assigned to one subgrid. Thus, this one process was responsible for performing the expensive simulate step function, for many boids, and all the other processes had to wait. This could be improved through better load balancing efforts, such as considering the density of boids in the grid, before creating our bounds for our subgrid, instead of naively splitting the grid into equal subgrids. However, for the more common case, where the boids are more sparse, we see fair performance. We see the normally expensive operations (communication and simulate step) represent a much smaller portion of time of each step, indicating we have achieved significant speedup and improvement over a naive implementation.

Differing CUDA Implementations

We also compared the different possible Cuda implementations, including the trivial solution of just iterating over all boids for each boid, and three different hashing techniques.



It was expected that the memory managing implementations involving hashes would outperform the trivial solution, however, I did actually think different hashing schemes would provide considerable speedups. Here is what each hashing scheme looks like on the grid.



While both the Z-order Curve and Hilbert curve seem to preserve really good spatial locality I think that reason that they are both marginally better than the linear hash is that the linear hash will be able to cache the rows of the neighboring cells all at once making it comparable.

Division of Work

Task	Amaad	Wadih
Sequential implementation	X	X
CUDA Implementation	X	
MPI Implementation		X
Infrastructure and Scripting	X	X
Poster		X
Report	X	

References

Green, Simon. "Particle simulation using cuda." *NVIDIA whitepaper* 6 (2010): 121-128.

Husselmann, Alwyn, and Ken Hawick. ‘Simulating Species Interactions and Complex Emergence in Multiple Flocks of Boids with GPUS’. *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems*, 09 2011, https://doi.org10.2316/P.2011.757-012.

Reynolds, Craig W.. “Flocks, herds, and schools: a distributed behavioral model.” *Seminal graphics: pioneering efforts that shaped the field* (1987): n. Pag.