15-418 Parallel Computer Arch., Spring 2021

Parallel Boids Project Final Report

Due Date: Thursday May, 2021 22:59 EST

Group: Elan Biswas (elanb), Gustavo Silvera (gsilvera)

1 Summary

We experiment with different memory layout implementations across varying parallelization axes in order to highlight the best configuration for performance and parallel speedup of a simulation with multiple interacting and interdependent actors. We will experiment with and compare different memory layout schemes and parallelization axes with `OpenMP`, then further experiment with a GPU implementation in `CUDA`. We found that the parallel flocks alongside local boids implementation achieved the best overall performance with reasonable speedup.

2 Background

The *Boids algorithm*, developed by computer scientist Craig Reynolds in 1986, is an algorithm that simulates flocking behaviour of groups of actors. This can be generalized to a flock of birds, school of fish, herd of buffalo, and swarm robotics. In writing our algorithm, we'll follow the general approach described in this Boids Pseudocode.

2.1 Data Structures

We must maintain a collection of boids, each with its own velocity, position, and acceleration vectors. For simplicity, our implementation focused on parallelizing over a 2-dimensional space, but the boids algorithm can be extended to any number of dimensions.

Additionally, we maintain a collection of "flocks," which are subsets of the boids which are close together in space. These are not a part of the original algorithm, but they allow us to represent the simulation as multiple instances of individual Boids algorithms based on nearby flocks. Flocks also allow us to take advantage of spatial locality as well as eliminate many comparisons and distance computations in the **sense** and **plan** stages (see next sec. **Program Overview**).

In order to associate the boids to the flocks in the simulator in an efficient manner, we decided on using an `std::unordered_map` as the associative container that maps boids to flocks.

2.2 Key Operations

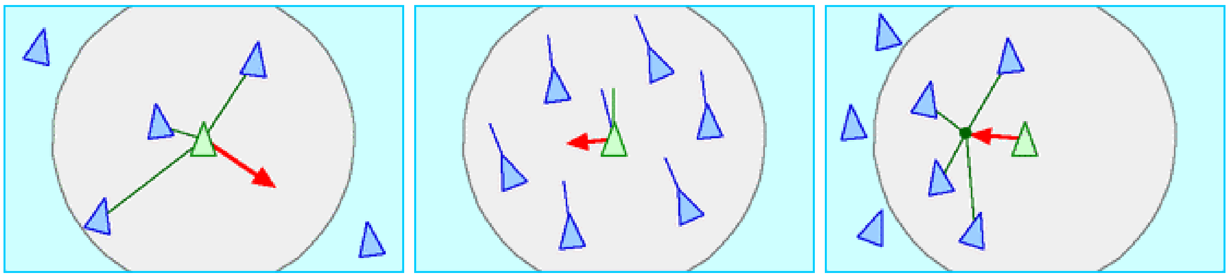


Figure 1: (Left to Right) Separation, Alignment, & Cohesion rules illustrated
Image credits: `http://www.red3d.com/cwr/boids/`

On every simulation `Tick()`, each boid's `velocity` is updated by a constant factor of its `acceleration` vector. The direction and magnitude of the vector are determined by a set of 3 rules:

1. **Cohesion**. Boids will tend to move toward the mean position of its neighboring boids. Neighboring boids are defined as falling within a distance given by a parameter `neighborhood_radius` and do not include the current boid.
2. **Separation**. Boids will repel boids that are too close to avoid collisions. We define "too close" by the parameter `collision_radius`.
3. **Alignment**. Boids will also tend to match the velocities of its neighboring boids, this aligns them towards the headings of the local flockmates.

The extent to which these rules affects the acceleration vector for a given tick is given by the parameter weights `cohesion`, `alignment`, and `separation`, respectively. Higher weights correspond to more aggressive responses to each of the three movement stimuli.

2.3 Inputs and Outputs

Inputs to the algorithm include specifications for boid interactions. As discussed, they are the `collision_radius`, `neighborhood_radius`, `cohesion`, `alignment`, and `separation` parameters. Additionally, we allow the user to specify the **number of boids**, the **number of tick iterations**, and **maximum boid velocity**.

Each tick **outputs** the new velocity and position vectors for each boid after appropriately applying the 3 rules.

2.4 Benefits of Parallelism

Each tick results in $O(n^2)$ distance computations and comparisons as each boid determines who to interact with. This results in $O(n^2)$ floating point operations to compute the acceleration vector. After the comparisons are made and the acceleration vector for each boid is computed, each boid must then compute its new velocity vector, resulting in an additional $O(n)$ floating point operations. Each of these sections provides opportunity for multiple avenues of parallelism, which will be discussed in section 3.

2.5 Dependencies

Each boid must consider the positions and velocities of its neighboring boids. This requires checking all the boids for their positions to determine proximity. Once each boid knows its neighboring boid, it must then compute its acceleration vector based upon the positions and velocities of its neighboring boids. This means that to compute its acceleration vector for a given tick, each boid depends upon the data of all other boids. Additionally, computing the position vector for a given tick, relies upon the velocity vector for that tick, which in turn relies upon the acceleration vector. This creates an inherently sequential block for the vector computations.

This means we can gather the necessary data for each boid in parallel as well as perform the vector computation blocks in parallel for each boid. However, the vector computation must occur sequentially after the data gathering.

The execution does not lend itself easily to SIMD execution since the computation of the acceleration vector requires serveral conditional checks to determine the proximity of other boids.

3 Program Overview

The program follows a generic simulator approach where all actors in the scene are updated simultaneously following a "sense $\rightarrow$ plan $\rightarrow$ act" loop. This presents the following algorithm for our parallel boids:

The **sense** phase occurs when Boids read other Boids and determine who is important to keep track of and who can be ignored.

The **plan** phase occurs when Boids determine how to change their velocity (computing accelerations) based off neighbours.

The **act** phase occurs after Boids have decided how to apply their acceleration, then they apply it to their velocity, which in turn increments their positions.

With our actors being Boids, the "sense" and "plan" components can be combined together since all Boids operations will be read-only (determining nearby boids and reading their current positions/velocities), and the "act" phase will be done all at once afterwards so all Boids advance in time at once and every Boid has the same information available to them (no Boid gets an advantage in time by being computed first).

The overall program (simulator + algorithm) essentially works as follows

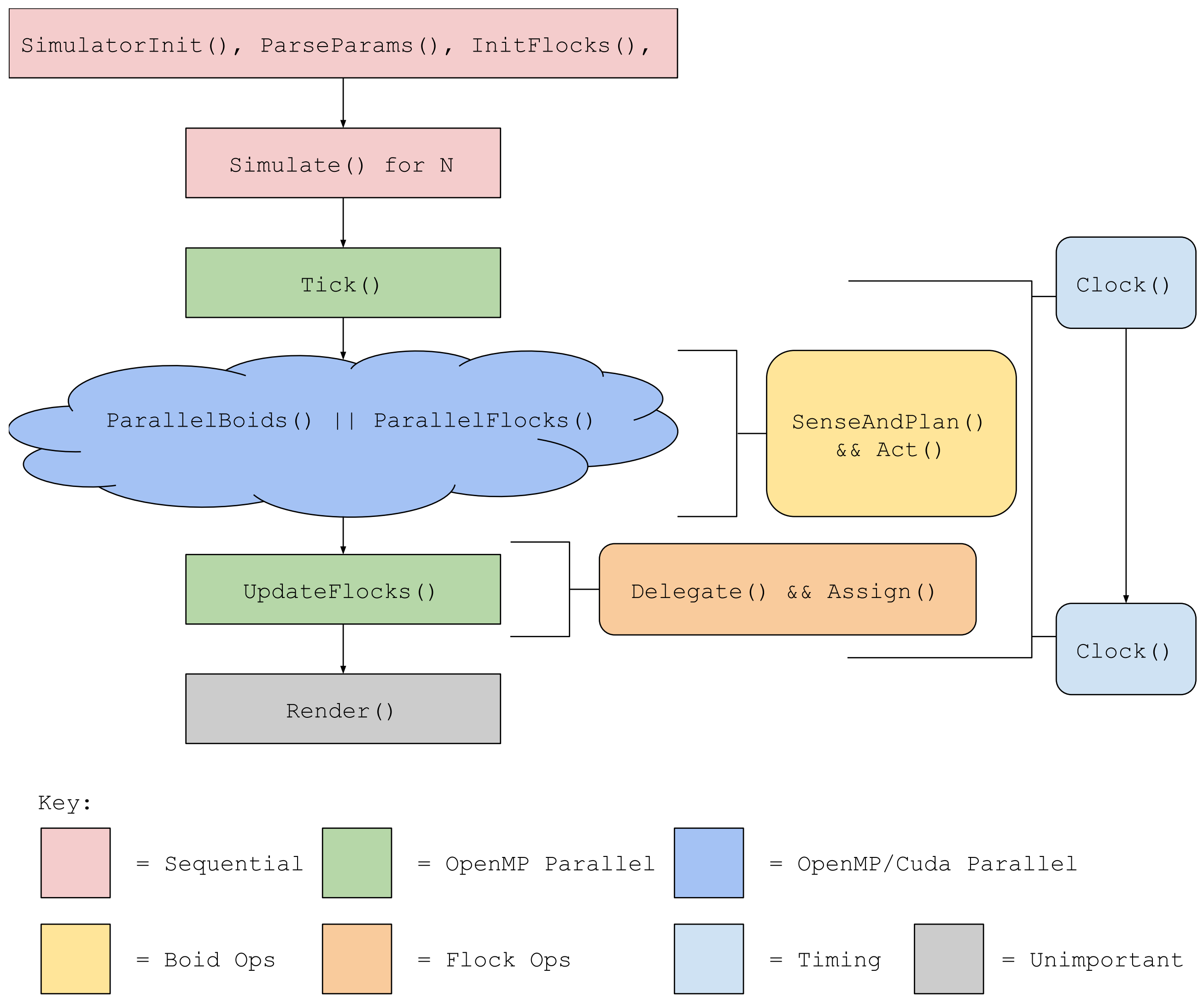


Figure 2: Diagram of entire program execution. Colour coded as described below.

Notice that the dark blue block is explicitly differentiated to highlight the different potential modes of parallelism that we will be investigating in this project.

3.1 Configurations

We realized that grouping boids together into flocks presents another data structure holding the same data that would be required to compute a valid Boids algorithm. This presents two possible axes of parallelism that we can traverse over with any parallelization techniques we learned so far. The two modes are illustrated below:

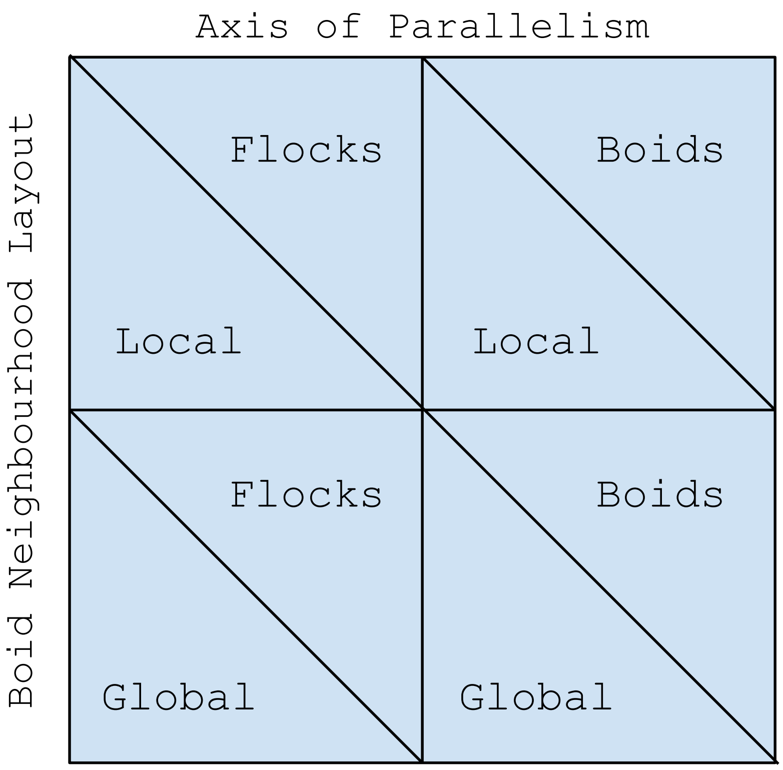


Figure 3: Different **modes of parallelism** and **memory layouts** that we're interested in

It is reasonable to consider common pairs in this matrix that might work well with each other: "local neighbourhood + parallel flocks", and "global neighbourhood + parallel boids".

3.2 Local Neighbourhood & Parallel Flocks

In the "**local neighbourhood**" case, we have every flock in the world carry ownership (memory initialization and management) of the boids that are a part of this flock. Here each flock contains their own `std::vector<Boid> LocalBoids` vector which contains all the boids in their flock which get moved around as boids switch between flocks throughout their simulation. This allows the flocks to be iterated over in a parallel fashion, and then each boid in the flock is iterated over sequentially. This presents a natural notion of the flock-boid relationship where boids are contained wholly in their flocks, yet their instances are moved around to new flocks during a flock assignment if they emigrate to another flock. This means the locations in memory of the boids are spacially contiguous in memory since they are handled by the flocks. Intuitively this might present better cache locality since the bulk of a boid's operations occurs with nearby boids, so having nearby boids in the simulation also physically nearby in memory may present a boon for performance.

The explanation above can be illustrated by Figure 4 below.

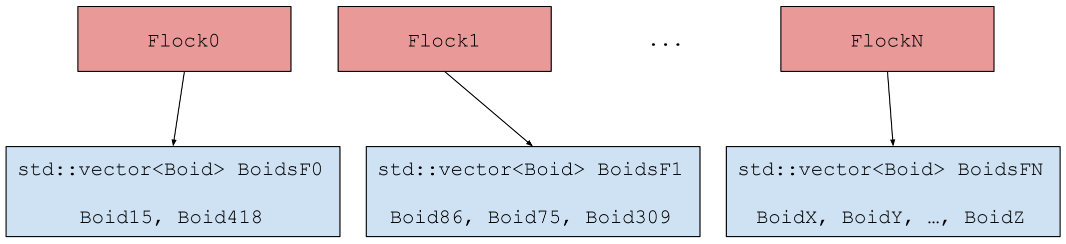


Figure 4: Diagram of `AllFlocks` memory layout

Here you can see how each individual flock contains their own `std::vector<Boid> LocalBoids` vector with arbitrary boids depending on their flock ownership in the simulator.

By intuition we'd expect that this memory layout should perform decently well and scale with parallelism since cache locality will be improved.

3.3 Example Usage

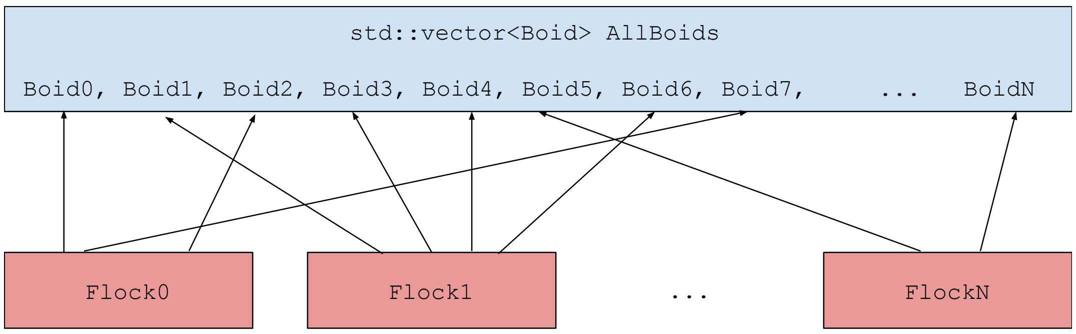
Finally, using this `LocalBoids` layout permits the following code snippet in our `ParallelFlocks()` function which emphasizes parallelization across the `AllFlocks` vector.

```
#pragma omp parallel num_threads(Params.NumThreads)
{
    /// NOTE: Here we are parallelizing across flocks
    #pragma omp for schedule(static)
    for (size_t i = 0; i < AllFlocksVec.size(); i++)
    {
        /// NOTE: SenseAndPlan is run per flock
        AllFlocksVec[i]->SenseAndPlan(omp_get_thread_num(), AllFlocks);
    }
    #pragma omp barrier
    #pragma omp for schedule(static)
    for (size_t i = 0; i < AllFlocksVec.size(); i++)
    {
        /// NOTE: Act is run per flock
        AllFlocksVec[i]->Act(Params.DeltaTime);
    }
}
```

3.4 Global Neighbourhood & Parallel Boids

In the "**global neighbourhood**" case, we have a single large `std::vector<Boid> AllBoids` vector allocation that stores all the Boids and never moves them. To respect our `Flock` API, we must still use flocks as usual so all modes of parallelism execute on the same pipeline. One major difference between `Flocks` in this case is that the individual `Flocks` no longer hold ownership of any particular Boids (in their neighbourhood). Since this is handled by the large (`static` ie. singleton) vector, all `Flocks` share the same vector. Therefore for flocks and boids to still work together as designed, the flocks must manage a set indicating which Boids in the global `AllBoids` vector belong to their "neighbourhood" by their index in the vector. This works because in the global `AllBoids` vector, the Boids remain in their initial positions throughout the entire program, they are not moved around or copied, only edited in place.

The explanation above can be illustrated by Figure 4 below.


std::vector<Boid> AllBoids
Boid0, Boid1, Boid2, Boid3, Boid4, Boid5, Boid6, Boid7, ... BoidN
Flock0
Flock1
...
FlockN
Figure 4: Diagram of `AllBoids` memory layout

Here you can see the large singular (contiguous) `std::vector<Boid> AllBoids` vector with all individual boids initialized in a continuous manner. Recall that since all Boids are initialized randomly, their position in this vector has no correlation with their associated `Flock`. Therefore each flock needs to maintain a local vector of indices indicating where in `AllBoids` are the boids from their neighbourhood.

By intuition we'd claim that this method of memory layout should perform worse than the Local neighbourhoods since each flock would operate on a single global vector and randomly access elements that could be very sparsely distributed throughout the vector, hurting spacial cache locality.

Additionally, we will have to deal with additional synchronization because all the threads (in the flock-case) will be dealing with the same memory addresses.

3.5 Example Usage

Finally, using this `AllBoids` vector permits the following code snippet in our `ParallelBoids()` function which emphasizes parallelization across the `AllBoids` vector.

```
    /// NOTE: Here we are parallelizing across Boids
#pragma omp parallel num_threads(Params.NumThreads)
{
    /// NOTE: Here we are parallelizing across boids
    #pragma omp for schedule(static)
    for (size_t i = 0; i < AllBoids.size(); i++)
    {
        /// NOTE: SenseAndPlan is run per boid
        AllBoids[i].SenseAndPlan(omp_get_thread_num(), AllFlocks);
    }
    /// NOTE: Need a barrier before all boids are allowed to act
    #pragma omp barrier
    #pragma omp for schedule(static)
    for (size_t i = 0; i < AllBoids.size(); i++)
    {
        /// NOTE: Act is run per boid
        AllBoids[i].Act(Params.DeltaTime);
    }
}
```

3.6 Other cases

The previously described pairs were emphasized as going “hand-in-hand” because of their conceptual usage and ease of implementation. However, we are still interested in implementing and measuring the effects of a “local neighbourhood + parallel boid” case and “global neighbourhood + parallel flock” case. These are both implemented in the simulator and perform reasonably well as we’ll discuss later. The usage is somewhat more awkward however and this effect can be seen in the example below:

For the “parallel boid + local neighbourhood” case we have

```
#pragma omp parallel num_threads(Params.NumThreads)
{
    /// NOTE: parallel boids w/ local neighbourhood
    // first we'll need to aggregate all the Boids from all the flocks
    std::vector<Boid *> AllBoids;
    for (const Flock *F : AllFlocks)
    {
        #pragma omp critical
        {
            std::vector<Boid *> LocalBoids = F->Neighbourhood.GetBoids();
            AllBoids.insert(AllBoids.end(),
                            LocalBoids.begin(), LocalBoids.end());
        }
    }
    // then we can proceed as usual (similar to parallel boids)
    #pragma omp barrier
    #pragma omp for schedule(static)
    for (size_t i = 0; i < AllBoids.size(); i++)
    {
        AllBoids[i]->SenseAndPlan(omp_get_thread_num(), AllFlocks);
    }
    #pragma omp barrier
    #pragma omp for schedule(static)
    for (size_t i = 0; i < AllBoids.size(); i++)
    {
        AllBoids[i]->Act(Params.DeltaTime);
    }
}
```

Note that in our implementation, the “parallel flocks + global neighbourhood” high level parallel for loop is written exactly the same because we have a hidden API function as a part of the flocks’ `Neighbourhood` called `GetBoids()` which is layout-agnostic and will return an `std::vector<Boid *> LocalBoidPtr` vector by accumulating from the global `AllBoids` vector.

4 Approach

4.1 Technologies and APIs

We primarily used OpenMP to explore the 4 axes of parallelism shown in Figure 3. We experimented with both static and dynamic scheduling and primarily targeted the Gates Cluster machines.

Toward the end of the project, we briefly experimented with a CUDA implementation, targeting the Gates Machine Nvidia RTX 2080 GPU.

4.2 Mapping to Machines

We implemented 2 means of mapping work to cores: individual boid assignment and individual flock assignment. The use can specify the desired work assignment scheme as a parameter.

OpenMP Parallel Flocks: The "flocks" data structure we use to track the spatial proximity of boids lends itself as a convenient unit of work. In this way, each core is responsible for computing the output position and velocity vectors of all of the boids in its assigned flocks.

Since flocks function as units of work, we found that capping the flock size can be used as a means of granularity control. We experimented dynamically scheduling flocks capped at different sizes to find the optimal granularity. Note also that as the algorithm progresses, the boids will converge into larger and larger flocks, so capping the flock size keeps this convergence from becoming too extreme.

To see the trade-off between large flocks and small flocks, consider the two extremes. In the case where all the boids are in a single flock, we effectively have a sequential algorithm where all the work is assigned to a single processor. In the case where all the boids have their own flock, we essentially parallelize over boids, resulting in maximally fine-grained scheduling.

OpenMP Parallel Boids: In this axis, each boid is individually mapped to a processor. We hypothesized that this would have worse performance when compared with the other assignment as it does not take advantage of the spatial proximity and locality gains provided by the flocks.

CUDA: In the CUDA implemnation, we parallelize over boids in a manner similar to the OpenMP parallel boids implementation. To port our original parallel boids implementation to CUDA, the boid data structures had to be split into a struct of arrays. In this way, we map indexes of boids to individual threads that can be used to access the necessary boid data.

4.3 Changes to the Serial Algorithm

In the original pseudocode of the Boids Algorithm, there is no notion of a "flock" in the sense of a local cluster of boids. We added this data structure to provide programmatic access to the clusters of boids that naturally form throughout the course of the algorithm.

Furthermore, we hypothesized that ignoring flocks that are far away would significantly reduce communication costs because we eliminate multiple boids with each ignored flock. Since this should also improve sequential performance as fewer boids comparisons are computed, we do not expect flock ignoring to specifically lead to significant gains in speedup.

4.4 Past Iterations of Implementation

In our first iteration of parallelizing the `Boid`'s updates, we thought of using a single function for the `Sense`, `Plan`, and `Act` functions. However, we discovered that this led to a race condition since Boid's would not be acting at the same time and thus would be reading to boids that could be changing their position, invalidating the algorithm. In our current implementation we have all boids sensing & planning simultaneously, then after a barrier all boids act together.

In previous iterations of the program we had data structures that would be wholly designed for particular configurations, such as a "parallel-boid" `Boid` and "parallel flock" `Boid` as well as different `Flock`'s for the different memory layouts. This quickly proved to be very tedious to implement and maintain across the growing codebase and therefore we decided to focus our efforts on creating a single high level `Flock` and `Boid` that would abstract away the underlying memory layout and parallelization axis.

This is mostly interesting for the `Flock`s, which now either hold a `std::vector<Boid> LocalBoids` which they fill with `Boid` instances, or have access to a global `static std::vector<Boid> AllBoids` which is shared among all the boids. This is abstracted away by a `NeighbourhoodLayout` data structure which each `Flock` contains and presents the high level functions for accessing the flock's neighbourhood (as a list of boids (local) or boid pointers (global)) in an efficient manner.

Additionally, some of our prior implementations used `double`s for all the computation including the highly used `Vec2D` class which we made to use 2-d vector operations on boids. Since we grew more concerned with the memory growth as the problem size scaled, we noticed that by changing these all to `float`'s then we'd save on a large amount of space for essentially equivalent accuracy. This also allows for more boids to fit in the caches of the processors running these (AMD 3900x) since the amount of memory per boid was decreased signficantly.

5 Results

5.1 Metrics

The primary way we measured performance is by measuring the wall clock time that is elapsed for the important components of our algorithm. In particular this ignores the sequential overhead of initializing the simulator and rendering the image, since those are additional components we use but our focus is on the computation for the Boids interactions themselves.

5.2 Experimental Setup

Our experimental setup is based on a list of parameters that can be tuned at runtime giving us easy access to fast reliable experiments. For the following tests, our params config file looked like this:

```
[Simulator]
num_boids=20000
num_iters=200
num_threads=X # tunable
render=false # don't render the scene
par_flocks=true # parallelize across flocks
[Boids]
boid_radius=2.0
cohesion=0.5
```

```
alignment=0.5
separation=0.5
[Flocks]
max_size=500
max_flock_delegation=3
is_local_neighbourhood=true
```

Our benchmark data was generated by using the same program but removing all instances of `OpenMP` yielding a completely sequential algorithm with no `OpenMP` overhead. We could then compare the output (rendered) images together and verify that the corresponding outputs per frame were equal for all boids.

Flock Granularity When experimenting with flock parallelization, we also measure how the size of the flock can affect performance in a dynamic scheduling setting.

6 Performance Graphs

6.1 Dynamic vs Static Scheduling

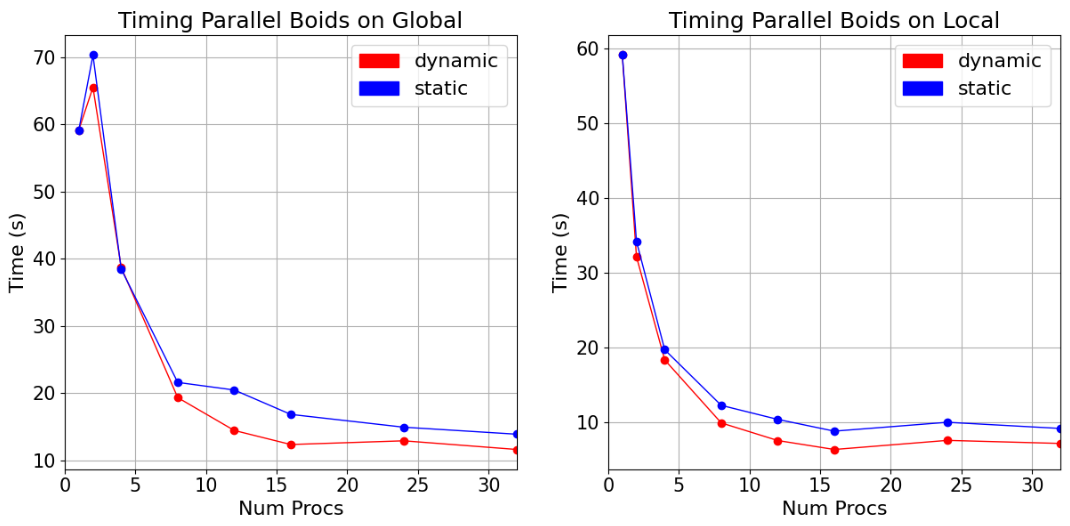


Figure 5: Time graph for Parallel Boids on Global/Local layout

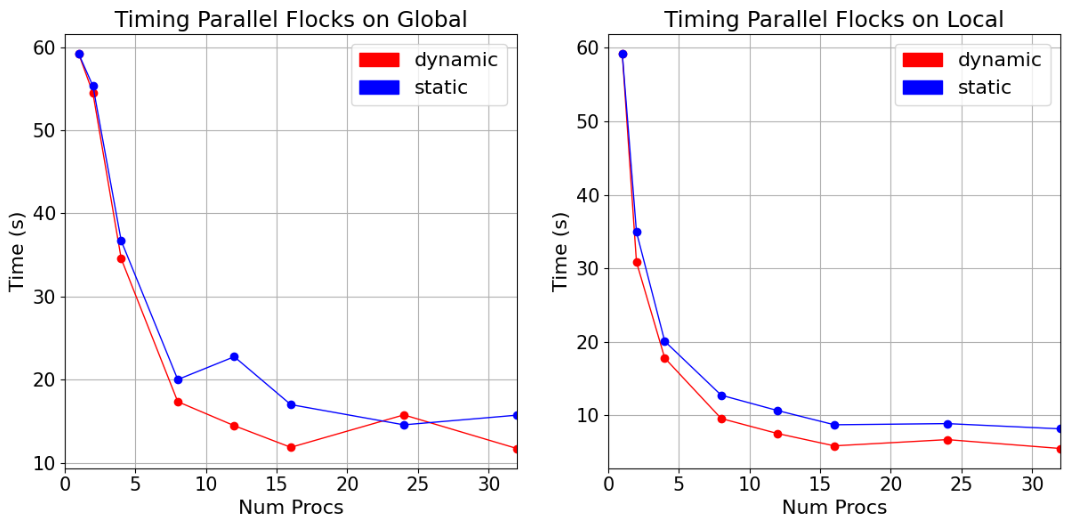


Figure 6: Time graph for Parallel Flocks on Global/Local layout

These performance graphs are compared against our fastest sequential baseline implementation (removed all `OpenMP` pragmas). The performance is fairly reasonable throughout the increase in processors but generally plateaus after 12 processors, likely due to some sequential bottleneck and Amdahl's Law. For the most part it is clear that the local memory layout performs consistently better than the global memory layout, likely due to cache locality. The problem sizes are very important because they determine whether or not our boid simulations can fit in the caches for our processors.

6.2 Speedup Graphs

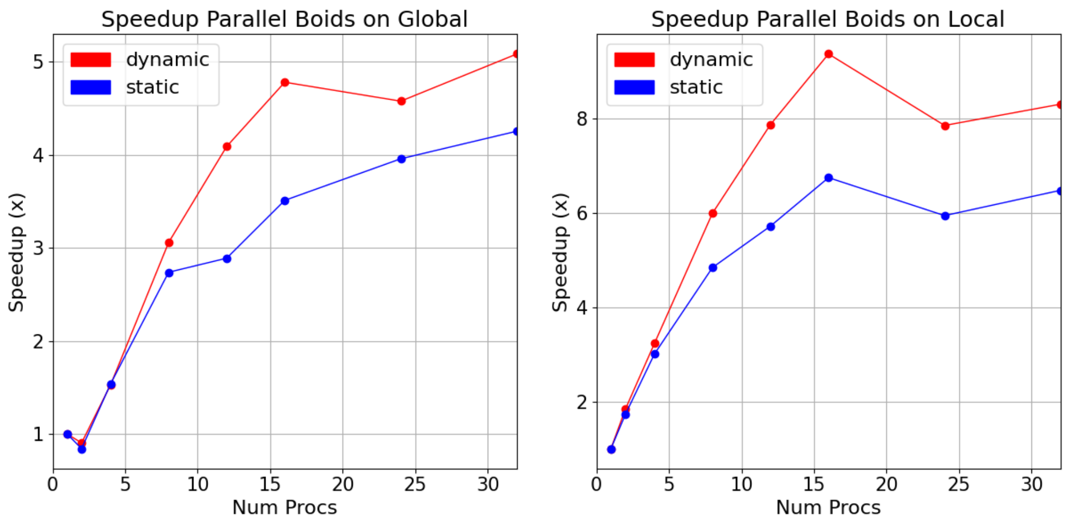


Figure 7: Speedup graph for Parallel Boids on Global/Local layout

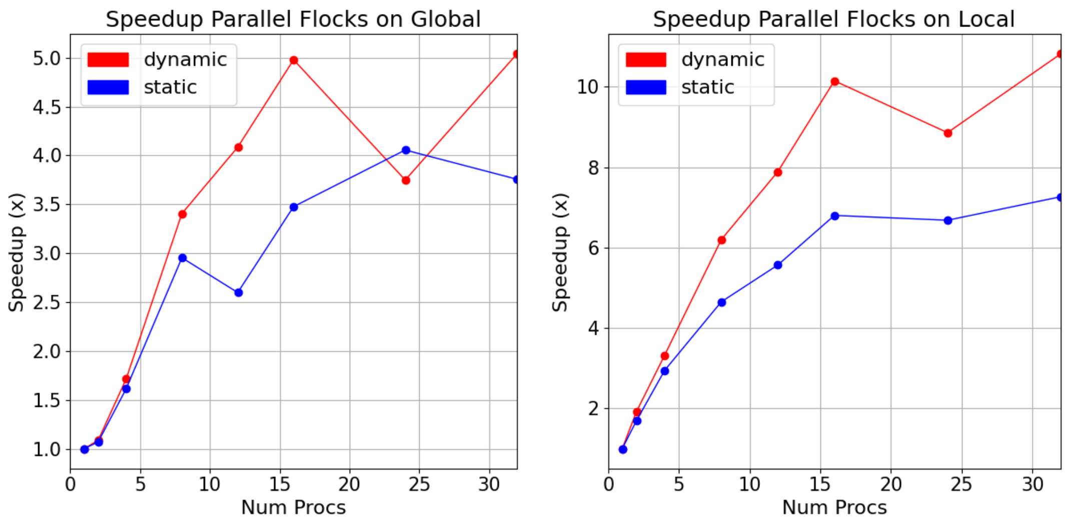


Figure 8: Speedup graph for Parallel Flocks on Global/Local layout

As seen in both the performance and speedup graphs, the **dynamic** `OpenMP` work scheduling was almost always better than the static counterpart. Although the overhead of dynamic scheduling is present, it likely improves the load balance of the processors assigned to the flocks and Boids, hence improving overall performance.

From these results we can conclude that the **Parallel Flocks + Local Neighbourhood** configuration is optimal for our Boids implementation and therefore we'll focus on this configuration for the remainder of our experiments.

6.3 Flock Granularity

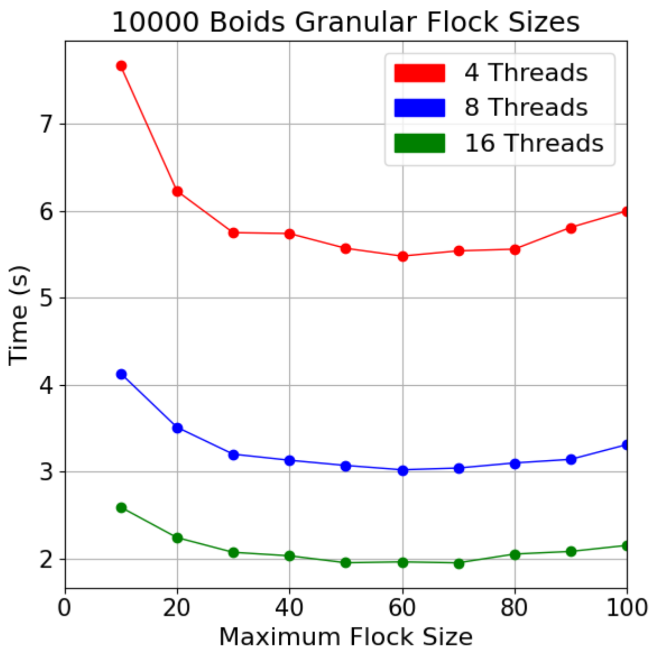


Figure 9: Run times of varying limits to flock size for granularity control.

As we predicted, we found that there is a balance between large flocks and small flocks when dynamically scheduling. In Figure 9, we record the run time of programs with varying maximum flock sizes and 10,000 boids. We see that program performance generally improves across the board as flock size increases and then worsens beyond a around a cap of 60. This means that capping the flock size at approximately 60 boids yields close to optimal dynamic work assignment. What is also interesting to note is that this optimal flock size is constant for larger processor counts.

6.4 CUDA

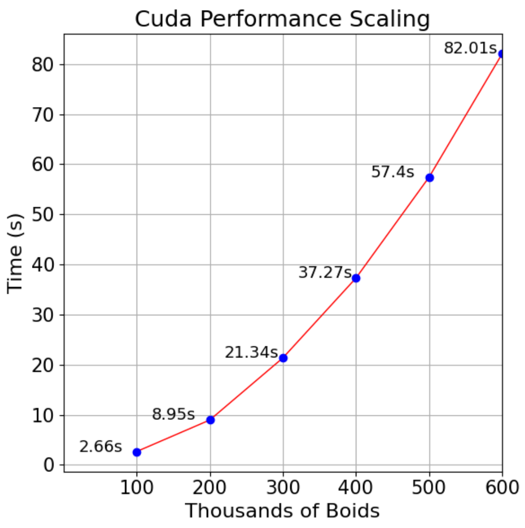


Figure 9: Run time on RTX 2080 on varying numbers of boids

We also briefly created an analyzed a CUDA implementation that parallelizes over boids. In Figure 9 we see that the performance on the GPU scales as we should expect from the $O(n^2)$ done by the algorithm. Additionally we note the following datapoint on the speedup for computing 20,000 boids for 200 iterations:

Implementation	Run time for 20,000 Boids and 200 Iterations
Sequential	59.15s
CUDA	0.3724s

For 20,000 boids, this results in a **speedup of 155.80x**.

6.5 Effect of Problem Size

The problem size is an important characteristic of the Boids algorithm because it directly correlates with the number of Boids and Flocks in the scene. Also recall that the boids must look at all their neighbours to determine what to do next, and in the worse case this accounts for an $O(n)$ iteration per boid. Across all the boids this leads to an $O(n^2)$ loop that we are traversing in parallel. Therefore, we know that the increase in problem size also increases the runtime by a quadratic factor, which severely limits our scalability on a CPU with limited parallelism.

We can see the non-linear runtime scaling occur in the following graph. This was taken on our best configuration with 8 processors.

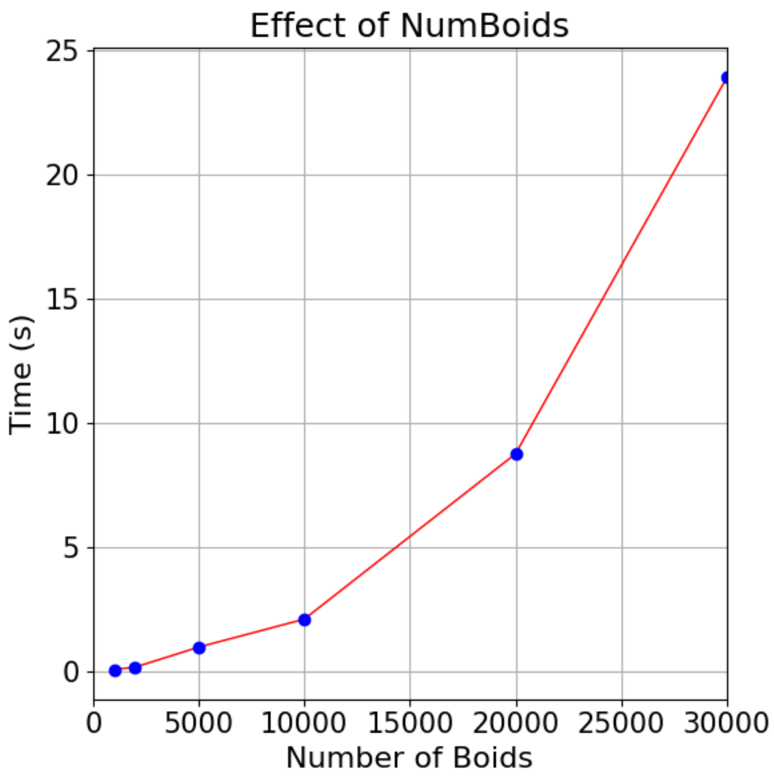


Figure 10: Illustration of the effect of increasing the problem size on the overall runtime

6.6 Potential Limits on Parallelism

It is clear that our performance plateau's quickly after 12 processors, this is primarily due to Amdahl's Law. Amdahl's Law states that our parallel performance will be bottlenecked by the critical path in our parallelism, which is commonly denoted as our slowest sequential portion. In the Boids problem, each and every boid must at least consider the possibility of all the other boids being potential neighbours, and therefore has to check them all, this is an $O(n^2)$ operation. However, we can use the concept of flocks to somewhat mitigate this by filtering out groups of boids at a time, it is still costly to traverse all the flocks, and this is a bottleneck on our potential parallelization speedup.

One way we tested for a lower bound in performance is by measuring an arbitrary boid's computation time throughout the entire program and generalizing that for all boids with perfect parallelism. This should present an approximation for the lower bound determined by Amdahl's law, though it is important to note that not all Boids require the same amount of computation, and this can differ greatly between boids.

The following table denotes the measured time for a single boid in the scene and how we might estimate a lower bound for our computation based off this sample. This is done once we begin plateauing (past 16 processors) with our best overall configuration: Parallel Flocks + Local Neighbourhood.

Num Processors	Measured Time For Boid 0	Lower Bound Estimate	Actual Time
16	0.004092	$0.004092 * 20000/16 \approx 5s$	$5.83s$
24	0.007716	$0.007716 * 20000/24 \approx 6s$	$6.678s$
32	0.008429	$0.00842 * 20000/32 =\approx 5s$	5.47

We also speculate that the main limitation for the CUDA implementation is the branch divergence due to the conditional execution in the **sense** stage. Our intuition for this is derived from the fact that memory accesses to every boid in this stage are unavoidable. This hypothesis also stems from the fact that the layout of boids in the boid array is random in terms of their position, which can mean that the boids in a warp are very disparate in terms of their position in the boid plane. This situation will frequently lead to branch divergence since far-away boids will not share any neighbours.

6.7 Machine Choice

As evidenced by the results from our initial CUDA implementation, we may have been able to achieve better results had we focused our efforts on better utilizing the GPU. We were initially skeptical about starting with the GPU implementation since the sense stage implicates a semi-rare conditional that could result in significant branch divergence. Since the boids in a thread can have very different positions in the boid plane, it is likely that only a few boids will be executing the conditional at a time. Solving this problem would require reorganizing the boids and their associated data items in all of the device arrays by proximity, which is an operation that does not lend itself naturally to the GPU.

However, further study could show that this can be done and improve upon our initial solution. This would have been another direction for our project, and if we had more time we likely would have explored it more.

7 Outputs

The result is not visible unless the parameter `render=true` is set to `true`, in this case we'll do additional sequential rendering (not measured in final time) to create illustrations that look like the following:

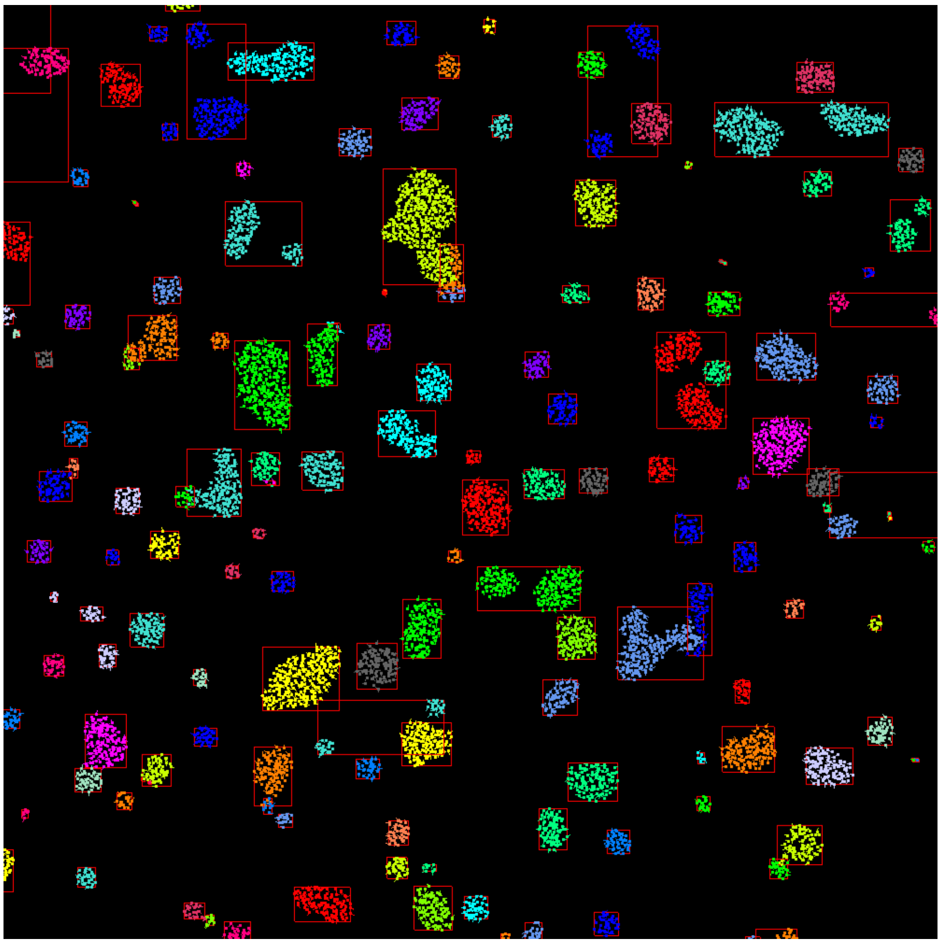


Figure 11: Output of the boids algorithm coloured by flockID and with a rendered bounding box around them

8 References

PSEUDOCODE:	http://www.vergenet.net/~conrad/boids/pseudocode.html
OVERVIEW:	https://cs.stanford.edu/people/eroberts/courses/soco/projects/2008-09/modeling-natural-systems/boids.html
EXAMPLE:	https://eater.net/boids

9 Division of Work

Equal work was performed by both project members.