SwarmFlocking

Praktikum Parallele Programmierung

Fabian Besner, Dominik Lohmann, Jakob Rieck
{2besner,2lohmann,2rieck}@informatik.uni-hamburg.de

2014

Contents

1 Introduction

Swarm behaviour is something humans have long admired for its elegance and beauty, efficiency and seemingly perfect self organisation. The emergent behaviour that results from each individual animal following some very basic rules is simply astonishing. Amongst the most prominent examples of swarm behaviour (also often simply called "swarming") are "flocking" for birds or "schooling" for fish. Often times essential for survival, swarms form to guard against attackers, to keep warm in rough conditions and to harness aerodynamics effectively.

Biologists are not the only people professionaly invested in understanding, modeling and simulating swarms. Swarm simulations are also used for entertainment purposes, in animated movies, TV shows and even computer games.

Swarm simulations for us are a great problem to work on for lots of reasons: The results produced can be visualised in a multitude of ways leaving room for unique and uncommon approaches. The simulation, while computationally intensive, can be broken down into smaller pieces and is thus a prime candidate for parallelisation.

The foundation of our work was laid by Craig Reynolds in his paper titled "Flocks, Herds, and Schools: A Distributed Behavioral Model" in 1987. In this report, we will follow the terminology he introduced closely.

With our project, we set out to create a functioning simulation of the boid flocking algorithm, while learning to use libraries and tools like *MPI* and *OpenMP* that faciliate parallel programming.

2 The model

In this section we are going to present our model, including our world, boids, predators and forces that govern the interactions between boids, boids and predators and predators among themselves.

2.1 Boids

In our model, a Boid is a massless particle representing one organism in a swarm. Its two important properties are *position* and *velocity*, but there is also a third field, *force*, which is used to store the force calculated in each step.

The actual structure definition reads as follows:

```
struct Boid {
  Vector3f position;
  Vector3f velocity;
  Vector3f force;

  // ...
};

typedef Boid Predator;
```

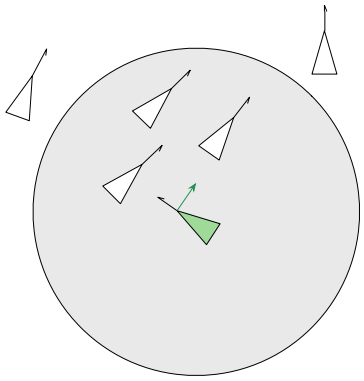
Each boid's position is confined to our world boundaries. Velocities are also bounded by a maximum velocity.

2.2 Forces

Each boid follows three simple steering behaviours called *alignment*, *cohesion* and *separation*.

For every force we present both a short text describing the basics and a graphic for visualisation: A green triangle represents the boid that is currently being observed, every other triangle represent neighbours of this boid. Each triangle inside the grey circle is a direct neighbour. It influences the boid that is being observed. The arrows point in the direction each boid is heading. The green arrow symbolises the force of the next discrete time step.

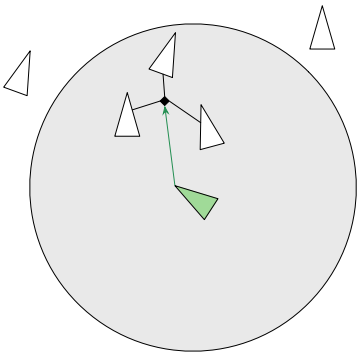
2.2.1 Alignment



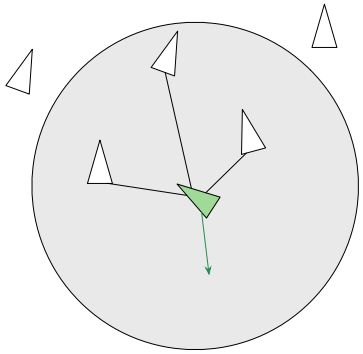
The first force is *alignment*, which results in each boid flying towards the average heading of its neighbours.

2.2.2 Cohesion

Cohesion results in each boid flying towards the centroid of all surrounding boids. This helps keep the swarm close.



2.2.3 Separation



In our model, just like in the real world, organisms in swarms rarely, if ever, collide. To model this, Reynolds introduced *Cohesion* which makes sure a boid steers away from other boids that get too close.

Each of these forces can be weighted differently and, by carefully choosing parameters, the resulting sum of those three forces results in swarm behaviour. If these forces are assigned indifferent weights, for example if *alignment* and *cohesion* are not considered at all, the resulting model does not resemble swarm behaviour, but instead it looks like a static particle grid. In this case, since all boids try to steer away from all the other boids, we expect to see a uniform distribution of boids in our world.

In Reynolds original proposal, the direct neighbourhood was defined in terms of a specific distance and a specific angle to model the fact, that any organisms perspective is limited. In our implementation we decided only to consider the distance and disregard the angle component. This was mainly done to keep the implementation as simple as possible. Nonetheless, it should be easy to take the angle under consideration as well.

2.3 Predators

Predators were added to make the simulation more dynamic. They are modelled the same way boids are, but interact differently with each other and with boids: A predator tends to avoid other predators and tries to get close to boids, which, on the other hand, are trained to evade predators and, if a predator gets too close, steer in the opposite direction to maximize the distance between the attacker and themselves. In our model, predators never actually catch boids, since there is no collision detection.

2.4 The world

We use a three-dimensional cube to model the world. All endings are connected: *Left* wraps around to *Right*, *Up* to *Down*, *Front* to *Back*. Because no boid leaves the world, we do not have to manage spawning new ones.

3 Serial implementation

Our earliest prototype, written in *C*, made it clear we needed operator overloading, mainly to work with three-dimensional vectors, so we chose *C++* for our programming language. *C++11* allows us to use many advanced language features like extended for-loops, enum classes for better readability and anonymous lambda functions to encapsulate common tasks inside of functions. As for libraries, we use *Boost C++ Libraries* for parsing of command line parameters and *SFML* for visualising the data.

We built seperate programs for producing the data and for consuming the data. This was done because we knew we only had to parallelise the actual simulation, not the visualisation. For initialisation, we populated our world with boids generated with pseudo random position and velocity. To allow easier testing and to ensure each process in the parallel version would always generate the same data during initialisation, we seed *srand* with a constant.

After initialisation the main simulation loop begins: In each iteration, forces for all boids and all predators are calculated, new positions are determined - based on the forces calculated previously - and the new data is written to disk.

To fully implement and create our idea, we had to change our *calculate Difference()* function that is used to calculate a difference vector between two boids' positions. Starting off, we used a simple vector subtraction to calculate the distance between two boids. Obviously this behaviour is different from the behaviour we originally wanted to create in our model, so we had to redo this function. Our new, more complicated version fixed this discrepancy, but introduced a performance loss by more than factor two.

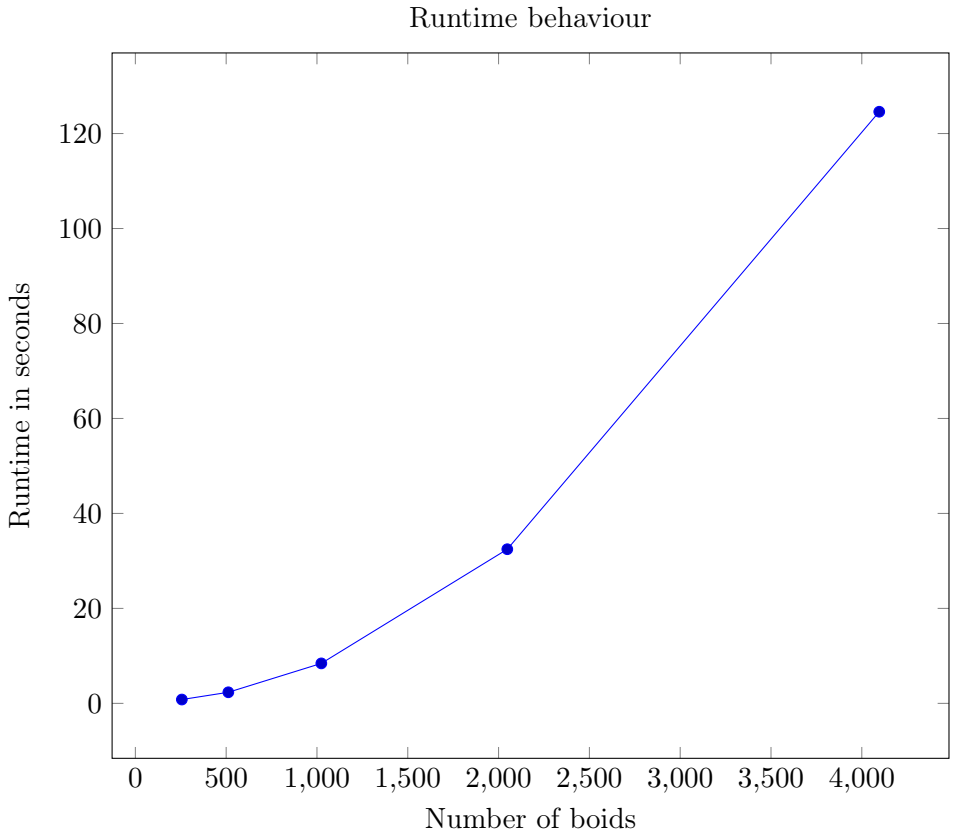
3.1 Performance optimisations and hotspot analysis

Profiling the application using *gprof*, we realized our more complex *calculateDifference()* function took up roughly 50% of our runtime. (Up from approximately 1%). By rewriting *calculateDifference()* and our vector class using SSE2 intrinsics, we were able to improve performance by roughly 80%, compared to our slower version. Because SSEs load instructions tend to be faster for 16 Byte aligned data, we chose to add a padding field to our *Vector3f* class.

For our main loop, we added an *OpenMP* statement to execute the loop in parallel. This resulted roughly in the projected speedup.

3.2 Runtime behaviour

The underlying algorithm is of complexity $\mathcal{O}(n^2)$. Because we also added predators, our new complexity for n boids and m predators is $\mathcal{O}((n+m)^2)$.

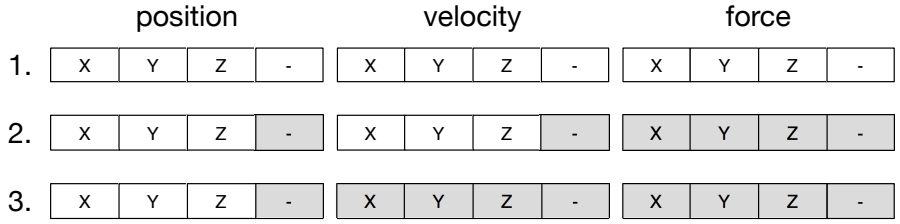


The graph above is a diagram showing the runtime of our serial program. Plotted on the x-axis is the number of boids, plotted on the y-axis is the runtime in seconds. We simulated 1000 steps - no predators were included. The diagram shows the general runtime behaviour as n gets large: For twice as many boids, there are roughly four times as many calculations to be done. This fits perfectly with the projected complexity of $\mathcal{O}((n+m)^2)$.

4 Parallelisation

Over the course of our work, we ended up creating two different versions, both with a unique approach. We ended up using the same MPI datatypes for both versions, so they are introduced first. Boid, predator and vector datatypes are used to send data to other processes without relying on implementation dependent details like *sizeof(float)*, as well as to save bandwidth, as we do not need to send padding data in *Vector3f*.

The graphic shown below visualises our design. Grey fields are skipped when sending or receiving data.



1. is the datatype *Boid* used in our code. Each field is a *Vector3f*, and each vector consists of 3 *floats* and one padding field (also a *float*)

2. is an illustration of our *MPI_BOID* datatype: The force is recalculated in each step, so we do not need to send it. The padding field does not contain relevant data so it does not need to be sent. This datatype is used during synchronisation.

3. is *MPI_BOID_THIN*. It is used to store data to disk using MPI I/O routines. We decided to only save the position.

4.1 Trivial approach

Our first version is the logical continuation of the obvious approach to threading. Just like with t threads, where we made sure each thread updates $\frac{n}{t}$ boids, we now use p processes, each with t threads, and make sure each process updates just $\frac{n}{t \cdot p}$ boids. Thus we do not divide the world but rather the individual boids. The algorithm works as follows:

1. Process setup: MPI initialisation, create pseudo randomly positioned boids.

2. Synchronisation: Receive each other processes data, send local data to each other process.

3. Serialisation: Save local data to disk (in order of MPI process number).

4. Calculation: Calculate velocities for locally stored boids; Update boid positions.

5. If $steps = 0$ then exit else $steps := steps - 1$ and *goto* 2.

4.1.1 Hiding communications

For this version, we went through trouble trying to hide our MPI communications. In short: We extended the obvious threading approach by distributing all data to all processes and selectively updating a subset of that data in each process. After that we would redistribute the data using collective MPI operations such as MPI_Allgather to all other processes and repeat that process. Thus, after each local update, we end up with parts of the data updated and other parts not yet updated because different processes are responsible for those parts. In order to hide our communications, during synchronisation we initiated the data transfer and then started computing partial forces for the boids in our reach. Since we already had part of the data from the next step, namely boids that we had previously updated, we could use those to start the process. After we consumed all data from our local cache, we would wait for the communications to end, before continuing the calculation and computing the new positions. This works well if the local cache of boids is rather large, so this approach favors fewer processes and larger boid counts. If there are few boids in each local cache, the partial force calculations are not enough to cover the time to fetch data from the other processes and the overall runtime is dominated by collective MPI operations.

We used vampirtrace to profile our application after we had made the aforementioned change. Here is the data we gathered:

sample	acc. application time (s)	acc. MPI time (s)	Walltime (s)
1	449.610	19.720	29.404
2	450.458	26.233	29.912
3	450.832	19.886	29.477
4	451.061	17.206	29.317
5	449.324	17.506	29.228

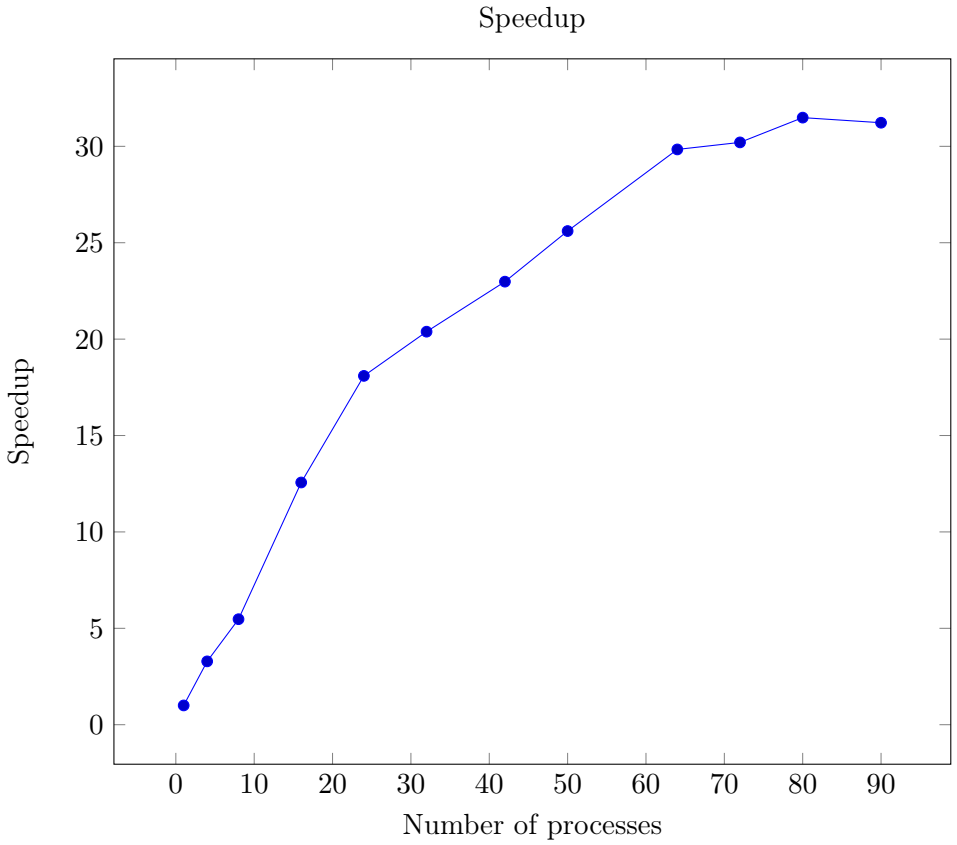
Data for asynchronous version:

sample	acc. application time (s)	acc. MPI time (s)	Walltime (s)
1	451.798	23.218	29.766
2	451.193	21.321	29.634
3	450.731	22.415	29.649
4	450.419	18.096	29.331
5	450.264	20.151	29.484

As you can see, there are absolutely no improvements in our runtime. From this data, you can even argue the opposite, it seems like the performance degraded slightly. We think this is due to two reasons: First of all, the message size each process contributes to the pool is comparatively small: For each boid, only its position and its velocity are distributed, and for each vector we only need to send 12 Bytes (3 Floats). Thus, for $n = 2^{16}$ and $nprocs = 16$, each process contributes just 96 kB data per step. Sending this data should not take long, perhaps not even warranting the use of asynchronous collective operations. Secondly, since the time it takes a process to calculate its chunk is not fixed but depends basically on the average number of boids in each boids influence range, each processes' runtime is slightly different. Thus, the additional call to *MPI_ Wait* to finish synchronisation actually introduced new waiting times.

Since the flocking algorithm scales so poorly, it is unlikely we could ever produce enough data for this optimisation to work out in practice. Furthermore, since we have no load balancing, we actually end up spending more time in MPI routines than before: Most of that time is spent in MPI_Wait, waiting for longer working processes to finish.

4.1.2 Scalability



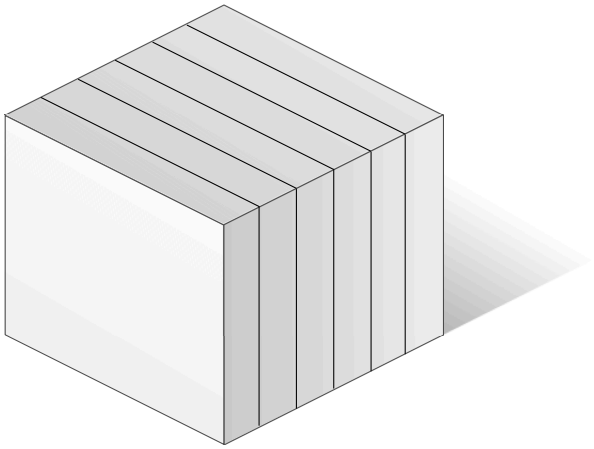
For the diagram above, we simulated 200 steps with 65536 Boids, no predators and a varying count of processes, plotted on the x-Axis. The speedup on the y-Axis is relative to the runtime for the simulation with $p = 1$ processes. As you can see, the runtime is increasing for $p > 80$. We believe this is due to the fact that each process receives the whole dataset in each step. Calculating with $p > 80$ processes, each one updates $\frac{65536}{80} \approx 819$ boids in each step, which is probably the threshold when the communication costs outweigh the calculation costs. At this point, adding more processes does not have a positive impact. If we were to use an even higher boid count, this scaling would most likely continue until this technical limitation is reached once again.

4.1.3 Conclusion

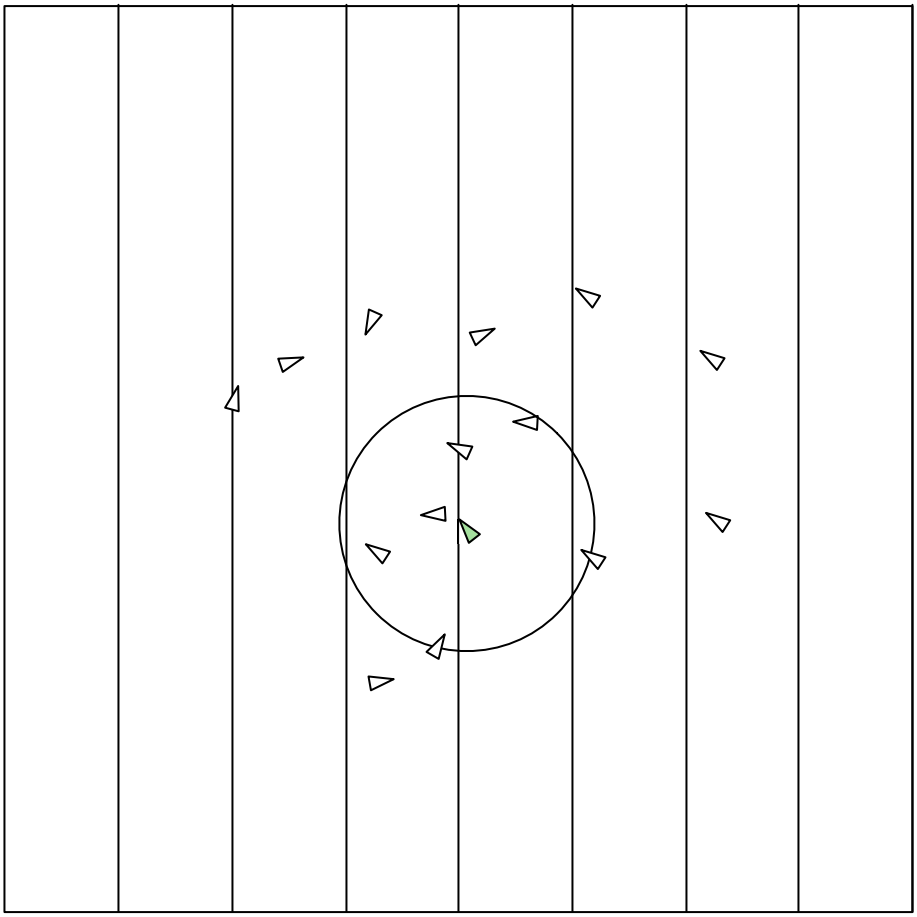
To conclude, the "trivial approach" was great in two regards: No matter how many processes one uses to compute the data, the result never changes, allowing us to test new versions simply by checksumming the produced output files. This is worth mentioning because floating point math is *neither* associative nor commutative, so when trying to optimize floating point code one almost always ends up changing the result slightly. Furthermore, the implementation is straightforward and offers a playground for further experiments.

4.2 Segmented approach

Our second approach works by segmenting the world into equidistant slices along the x-axis:



Each process is responsible for calculating the forces in exactly one slice, send these updated boids to its neighbours and receive updated boid data from its neighbours. A slice a is a neighbour of another slice b if and only if $d = distance(a, b)$ is smaller or equal to the maximum range of any force. This definition ensures that each process gets just the information it needs to calculate new positions for boids in its bucket. Every process has at least two neighbours: the leftmost processes neighbour are the last process (rightmost process) and the second process. More neighbours are possible if the maximum range of any force extends over more than one slice, say if the width of a slice is 25.0 and the maximum range is 50.0, then each slice has four neighbours, two on each side.



In the case depicted above, the segment containing the green boid (the boid located in the center of the circle) has four neighbours, because the maximum range extends for two segments on each side.

The general algorithm is as follows:

1. Process setup: MPI initialisation, create pseudo randomly positioned boids.

2. Calculate velocity for boids in processes bucket, taking boids gathered from neighbours in consideration.

3. Calculate partial predator forces by looping over local boid data only. Update boid positions.

4. Synchronise: Send and receive partial predator forces to and from all other processes; Compute the aggregated forces. Update predator positions (each process updates all predators as they are only stored locally). Send processes local boid data to all neighbours; Receive boid data from all neighbours.

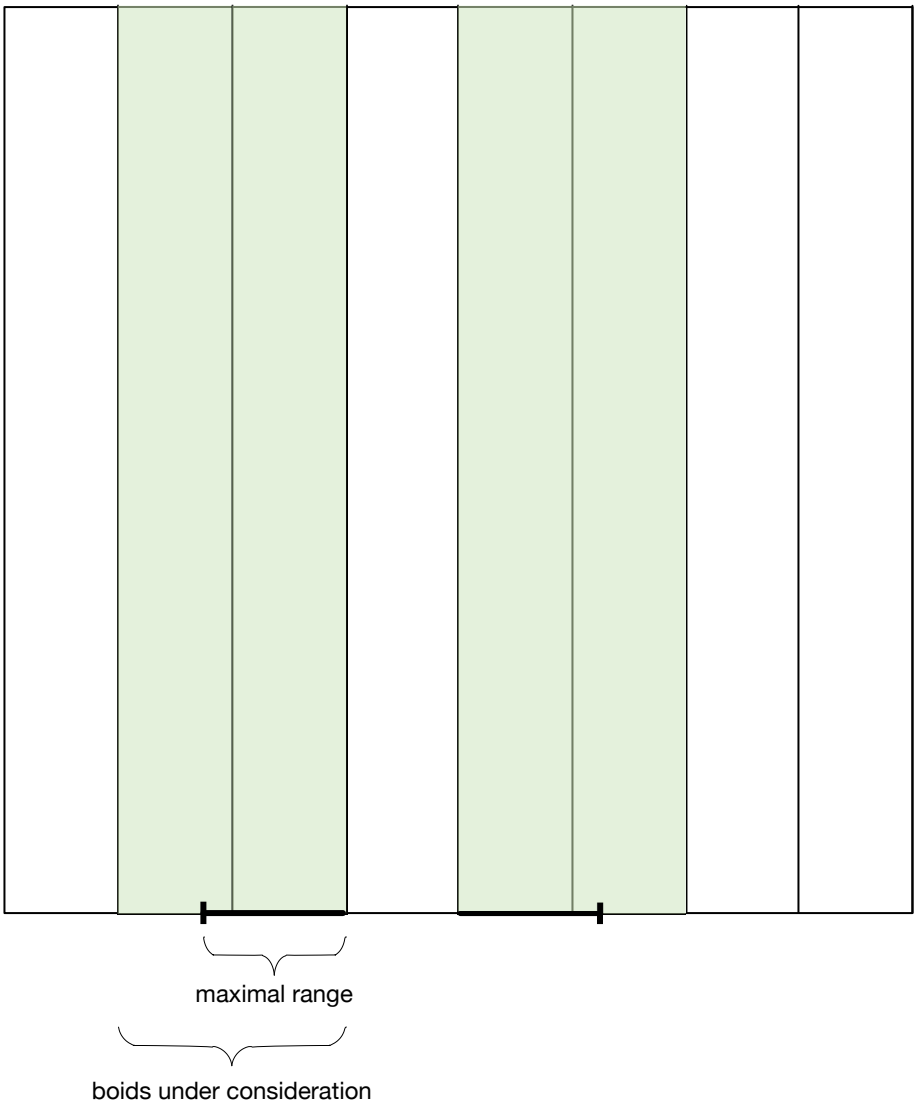
5. Serialisation: Save data to disk, using MPI I/O. Each process stores its local data, there is no process dedicated just to save data to disk.

6. If $steps = 0$ then exit else $steps := steps - 1$ and goto 2.

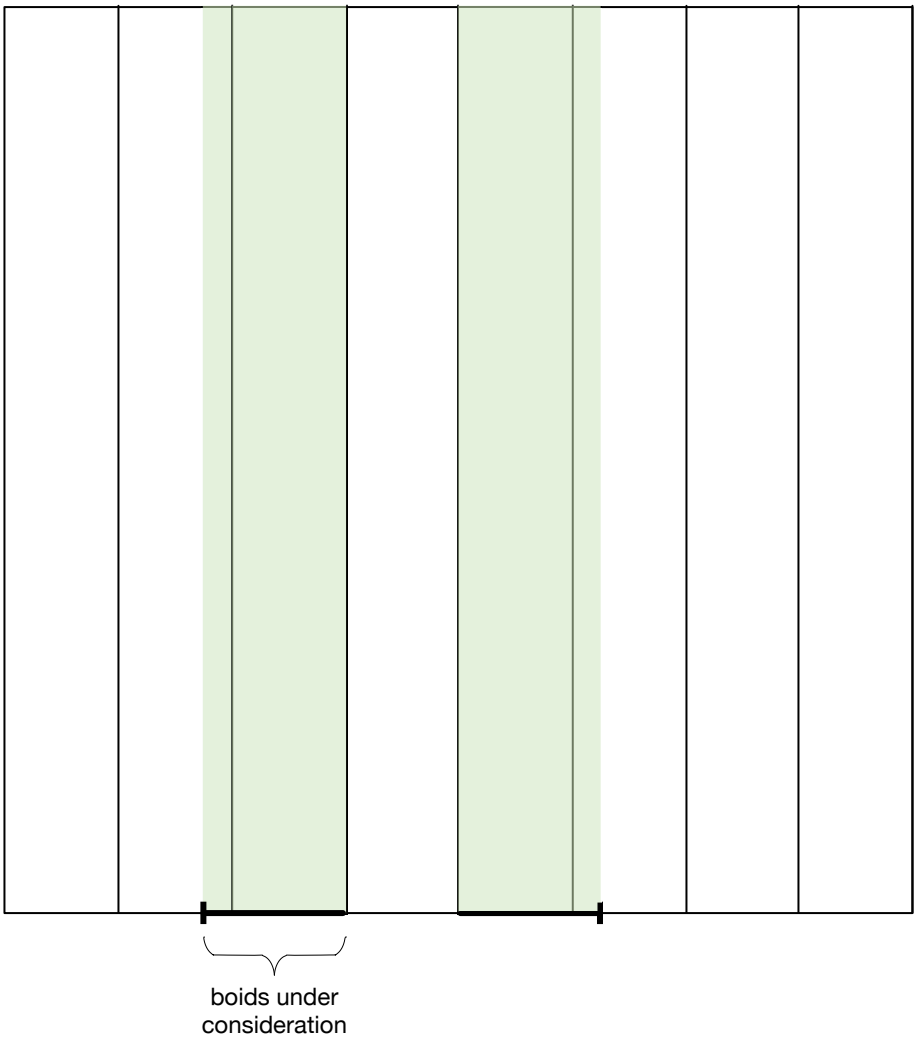
4.2.1 MPI Optimisations and tweaks

Finally, we had two additional performance optimisations we wanted to work on. Firstly, we wanted to use all available processes to calculate the next predator positions, rather than just using one designated process as we had done previously to do all the work. According to benchmarks, this practice of ours accounted for a large percentage of all MPI work in our application, as we used a Gather, followed by a Scatter to first send all data to the root process and then distribute the updated predator positions, calculated by the root, to all other processes. In order to improve this behaviour, we originally wanted to use MPIs Reduce functions, but soon realized, that a requirement for a user supplied reduce function in MPI is associativity, which cannot be guaranteed when working with (vectors of) floating point numbers. In order to retain some kind of testability, we instead chose to distribute every processes information to every other process using MPI_Allgather and to then let every process compute the final predator positions on its own. Unfortunately, this only solves a part of the associativity problem, as we are still computing partial forces on a per process basis. We changed the final result slightly, again due to floating point arithmetic on modern architectures. We think this change in the output is small enough to justify the performance improvements.

Secondly, we have reduced the number of boids we send from each process to its neighbours. Previously, we sent each processes' local data to every other process that could theoretically be influenced by any boid in the sending processes' segment. The graphic below visualises this behaviour. Even though the maximum influence range of any force only extends to about a third of the second neighbour, we still receive all local data from that neighbour.



If only parts of a segment are needed to perform all calculations, now we only send that part:

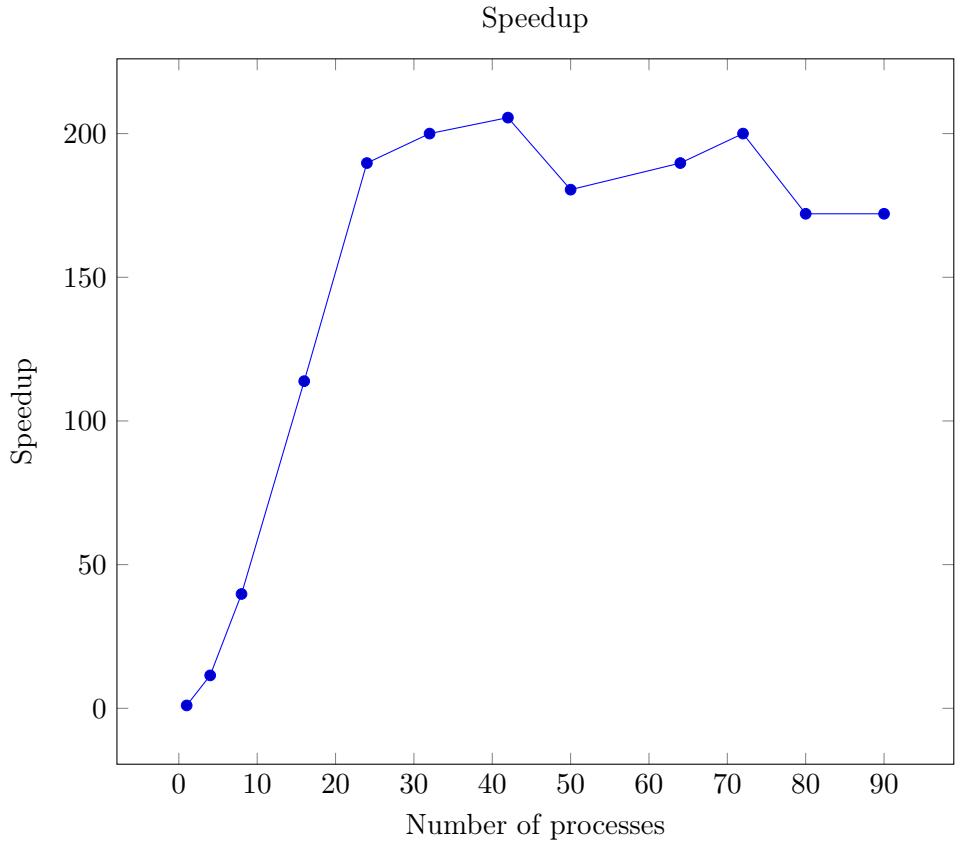


The performance improvements of this change depend in large parts on the configuration, mainly the world size, number of processes and distribution of boids in each bucket. If we assume a uniform distribution of boids (in terms of their position) in each bucket, a world size w and n processes are working together, then the potential savings depend on $\frac{w}{n}$. If this fraction works out to be less than the maximum influence range, almost nothing is won, because we are still sending all of our data to our neighbours. After all, every piece of our information could be used by any of the adjacent processes. On the other hand, if said fraction is greater than the maximum

influence range, we only send part of our data to our neighbours. Assuming uniform distribution of boids in our bucket, we send $\frac{maxInfluenceRange}{\frac{w}{n}}$ % to each of our neighbours. Obviously, our simulation should never approach a uniform distribution, but this can be used to estimate the savings. In real life examples, the cost tends to be a lot more suddle, even if the world size is large enough and we use comparatively few processes, and, while the average message size still decreases, we end up sending lots of data to some processes and little to others. This is a suboptimal assessment, since the speed of our synchronisation is determined by its slowest part. Nevertheless, this change saves bandwidth and may allow the usage of our program on a cluster with more limited connectivity.

For this second version, we did not try to hide communications because of the added complexity of the code in our main branch, as well as the considerably smaller local datasets that could be used to fill the time to effectively hide asynchronous MPI operations and the nature of our application requiring synchronisation after every step. This decision was also informed by our previous findings, but in theory, the same method we described for the trivial implementation would also work here.

4.2.2 Scalability



For this graph, we used the same parameters as before in the discussion about scalability of the "trivial approach": We simulated 200 steps for $2^{16} = 65536$ boids and no predators in a world of size 3200. The most interesting piece of information in this graph is the superlinear speedup for programs with $p \leq 32$ processes. This happens because for $p > 1$ processes, a process does not need to loop over all boids, but only those in its segment (and over some of the boids from his neighbours). When $p > 64$ and default weights are used, each process has more than two neighbours, so the communication overhead gets larger. We attribute the short downs for $32 < p < 64$ to missing load balancing, which ends up favoring some configurations over others.

5 Visualisation

While calculating data is an important task, it is also important to check for data validity. This can easily be done by visualising the output data and checking whether there is noticable swarm behaviour. Visualising the swarm behaviour also helps showing the ideas behind our calculations and the effect of each individual force.

5.1 Output file format

For the file format used to channel data from the simulation into the visualisation we decided to use a simple binary format. We chose not to include any metadata, but just to dump the positions of all boids and predators to disk. To process this data, a user has to supply the visualisation process with the correct number of boids, predators and steps used in the process that generated the file.

5.2 Data visualisation

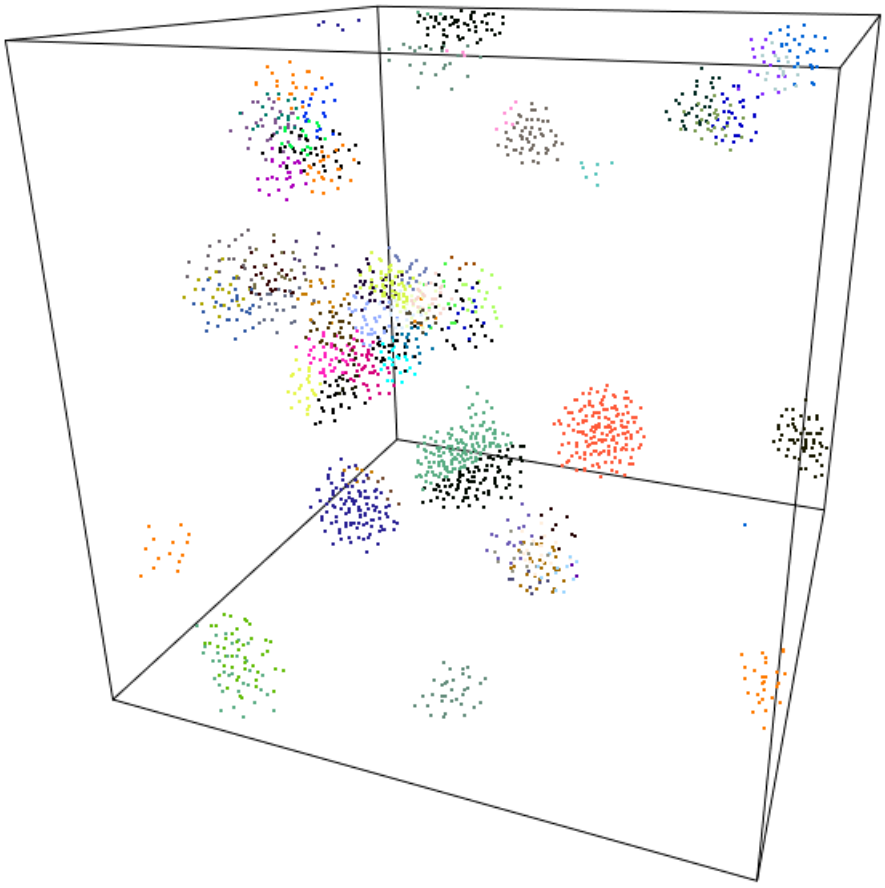
Given an input file create by the simulation program, our visualisation binary is capable of showing the boids and predators in a rotatable three-dimensional cube. Apart from the input data file, the visualisation expects the number of boids, predators and steps simulated and optionally, an upper bound to the number of frames rendered per second. On the implementation side, we chose to use *SFML* to create a window and draw the boids and predators using *GL_POINT* on *SFML*s underlying *OpenGL* context. We used SFML for its ease of use to create windows and handle inputs, and for its support for X11 forwarding via SSH.

The visualisation can be paused by pressing space at any given time and the cube can be rotated with the arrow keys. The rotation defaults to 2° per frame, but can be accelerated using Numpad1 and decelerated using Numpad2. Escape closes the application window.

5.3 Cluster detection

For visualisation purposes, we ended up adding an algorithm particularly popular in the machine learning community: clustering using *k-means*. Because its results are not (yet) perfect, we only enable clustering for single stepping mode. In the section "Remarks and future work" we share an interesting idea to use this clustering approach as the basis for a completely new implementation of the simulation.

Here is a screenshot of our visualisation taken while running in single stepping mode. As you can see, the clustering is certainly not perfect, but already works reasonably well.



6 Remarks and future work

We can think of a few ways to potentially improve performance in the serial implementation that we have not had the time to implement by ourselves: As we have already discussed, we added a padding field to our vector class in order to get the benefits of loading aligned data. Looking back, additional cache misses introduced by this change may end up being costlier than simply loading from unaligned addresses. Furthermore, our boid data structure may not be optimal: Most loops access only the position and velocity fields, so it may be advantageaus to move from an array of structs (*aos*) to a struct of arrays (*soa*) for this key data structure. For our project, we chose readability over potentially minor performance improvements in our code.

One obvious problem with the segmented version is load balancing. Right now, segments that contain more boids or have a huge number of neighbours do more work compared to other processes, whose buckets might only contain very few boids. So far, we have not made efforts to dynamically resize the buckets, which could help in balancing the workload. In practice, at least for relatively few steps (less than 10000) we have not observed large discrepancies, but this is again dependent on various parameters, including the weights of all three forces, which effect the flocking behaviour and are directly responsible for the distribution of boids in the space domain, the size of our world, which limits the positions the boids can be at (remember that a wraparound is implemented), number of predators, which again affects the swarm behaviour, and various other factors. Despite this, even small discrepancies can dramatically limit the achievable speedup, so this is an important problem to solve.

Benchmarks show the slowest MPI calls are related to I/O. These take up a considerable amount of time and could possibly be queued up in a smart way, wherein the individual data buffers are buffered for a longer time so that the main simulation would not have to wait for the I/O operations to complete. This is not trivial to implement, as each process holds its own chunk of the data, so it is not directly possible to just dedicate one worker process to this task. Another solution could be to make each process write its data into a local file and then to combine them after program execution has terminated.

We can also conceive an entirely new implementation that does not rely on a segmentation of the world as is the case in our implementation, but rather uses different means to distribute whole swarms to individual processes. We implemented the first step for this in our visualisation: In single-stepping mode, a clustering algorithm is used to calculate clusters, roughly

corresponding to swarm, and display them using different colours. It is conceivable that the root process could calculate these clusters identifying individual swarms, build up a graph encapsulating both each swarms centroid and diameter and the relative distances between the swarms. Then send each available process a specific subproblem, a swarm and the swarms influencing that particular swarm. Afterwards, the process would either send all calculated forces to the root process, or just the swarms new centroid and diameter in cases where the nearest swarm is still so far off that it does not influence the current processes data. In that case, communications between processes could potentially be reduced drastically. It is worth noting that in general, it is not easy to compute clusters, so perfect results cannot be expected. Despite that, the resulting algorithm should work reasonably well.

7 Conclusion

If there is one thing developing this simulation has taught us, it is how to properly work in a team. Coming from different programming backgrounds and having different knowledge about the tools needed, we learned to complement each others set of skills. Obviously there are a few things we wish we had implemented differently, like simply ignoring boids that "fall out" of our world and spawning new ones. Summarising, it took us a lot longer to develop, debug and benchmark our programs than we would have expected. Still, looking back, we met our goal: Our simulation is working and produces data that can be fed into our visualisation to examine the results in detail.

In the last section we shared our thoughts about future improvements and changes we think could improve the performance. If you would like to continue working on this project, feel free to contact us. We would love to hear about your efforts!