Distributed Boids Simulation: Performance Analysis and Implementation Challenges

Brett M. Martin, Ryan D. Winz, Luke J. McFadden, Tor J. Langehaug
Air Force Institute of Technology, WPAFB, OH, USA
email: brett.martin.4@us.af.mil, ryan.winz.1@us.af.mil,
luke.mcfadden.2@us.af.mil, tor.langehaug@us.af.mil

***Abstract*—Boids is a computer methodology to simulate animal flocking behavior. Each boid within the flock is modeled using simple geometric steering rules to define how it moves. Past research uses a boids model in areas such as animation, simulation, and drone swarms. While most approaches utilize a fully centralized architecture where all components are hosted on one computer, this research encapsulates individual boid behavior and distributes it among several computational devices. This is primarily achieved through the use of middleware client software that facilitates the distributed aspect. For our implementation, Docker was used to containerize the program component; RabbitMQ and Redis were used to facilitate communication and storage. Each component of this distributed system was tested and evaluated on a single host machine.**

***Keywords*-Modeling and Simulation; Boids; Flocking;**

I. Introduction

Boids is a popular method for modeling flocking behavior in birds, fish, and other organisms. This simulation involves creating virtual agents (i.e., boids), that interact with each other and the environment to mimic natural movement patterns. While the boids simulation is widely used in various fields, its computational complexity can pose a challenge for real-time rendering and other applications that require high performance.

To address this challenge, we have explored the use of distributed systems for boids simulation. In a distributed system, the computational workload is spread across multiple machines to improve processing speed and efficiency. One such approach involves using a middleware client to coordinate the communication between the boid objects and worker clients that perform processing on these objects. The results are then collected by a host machine that updates the database with the updated boid positions and velocities. We compare the performance of a distributed boids simulation approach with a standalone model.

Our distributed system leverages middleware clients to coordinate the processing of boid objects by multiple worker clients. We evaluate the performance of this approach by measuring the simulation's frame rate and scalability under different numbers of worker clients. We also discuss the challenges and limitations of creating a distributed system for boids simulation.

The rest of this paper is organized as follows: Section II provides a literature review of Craig Reynolds' original paper on boids and related work in similar simulation models and distributed systems. Section III describes the systems architecture of the distributed boids simulation and the experimental setup. Section IV presents our performance evaluation results and analysis. Section V discusses the challenges and limitations of creating a distributed system for boids. Finally, Section VI concludes the paper and discusses future work.

II. Related Research

This research builds upon the work of Craig Reynolds, who introduced the concept of boids and flocking simulations in his paper *Flocks, Herds, and Schools: A Distributed Behavior Model* [1]. Reynolds aimed to create a realistic model of flocking behavior in animals such as birds and fish by describing three simple rules that guide the behavior of the entities: separation, cohesion, and alignment.

Separation is based on the idea that entities should try to avoid collisions with other entities by keeping a minimum distance between themselves and their neighbors. Cohesion refers to the tendency of entities to move towards the center of mass of their neighbors, thereby maintaining the cohesion of the flock. Alignment involves aligning the heading of an entity with the average heading of its neighbors, thus promoting coordination within the flock.

Reynolds' approach to modeling flocking behavior using boids has since become a widely used method in computer graphics and animation. It has been applied in various contexts such as crowd simulations, robotics, and even video game AI. The Air Force Institute of Technology (AFIT) has also employed boids in their research, utilizing it for Unmanned Aerial Vehicle (UAV) guidance in a paper titled *Flight Test Results for UAVs Using Boid Guidance Algorithms* [2]. Similarly, Hungarian researchers boasted the first-of-its-kind drone flock using the boids model as the foundation in their paper *Autonomous Drones Flock Like Birds* [3]. However, the computational cost of simulating flocking behavior can be high, particularly when dealing with large numbers of boids.

DOI 10.1109/CSCE60160.2023.00133

To address this issue of performance, Reynolds explored different algorithmic approaches. He noted that a naive approach, in which each boid needs to consider the position and velocity of every other boid in the simulation, has a time complexity of $O(n^2)$. This means that the computation time grows exponentially as the number of boids increases, making it impractical for large-scale simulations.

Reynolds proposed a solution to this problem by parallelizing the calculations needed to update each boid's position and velocity. This reduces the time complexity to $O(n)$, making it more practical to handle larger numbers of boids. However, the efficiency of this approach depends on the number of processes available and the amount of time it takes to pass messages between them. In this paper, we build upon Reynolds' work by implementing a distributed flocking simulation and testing its performance under different conditions.

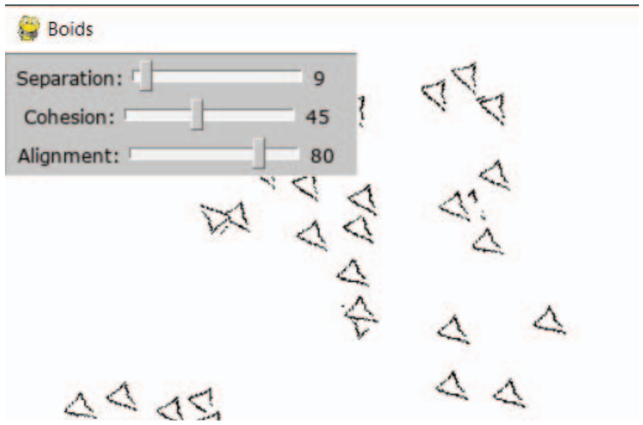
III. System Architecture

The design we followed for our simulation mirrors a simple distributed system. At the start of the simulation, a predetermined quantity of boid objects populate a Pygame (Pygame is a Python package) screen at random positions with random starting velocities. Boids are represented here as an object class containing a unique identifier, position, and velocity. Each frame of the simulation consists of calculating the position and velocity of each boid using a set of three "steering" rules, communicating these new translation values to the other boids, and drawing each boid to the screen using the updated translation values.

In his original paper, Reynolds suggests that each boid, since they calculate their own respective translation values, could be likened to a virtual computer [1]. In our boids simulation, each boid object calculates new translation values using a set of three rules that govern its flocking behavior: alignment, cohesion, and separation. The rule of alignment orients a boid's facing direction and velocity towards the average of its neighbors. The cohesion rule dictates that boids within a certain proximity of each other will tend to stay together as a group. This is calculated by determining the centroid, or average position of nearby boids, and steering the velocity vector of the boid under calculation towards it. Lastly, the separation rule ensures that each boid will keep a reasonable distance from these neighboring boids to avoid collisions.

In our implementation, each of the rules has a variable range of effectiveness. Using user interface (UI) sliders, the effective radius for each of the rules can be attenuated to achieve varied behavior patterns. When each rule is calculated, the steering velocities are summed, applied to the boid, and used to determine the appropriate heading value, which is used to orient the image.

In our standalone model, this is all achieved from a single script. By using the distributed model, we demonstrate how the simulation can be implemented through the fragmentation of its constituent elements, enabling them to operate separately and over a network.



Separation: 9
Cohesion: 45
Alignment: 80

Figure 1: Screenshot of the Boids simulation, including UI sliders to control effective range of steering velocities.

A. Standalone Model

The standalone model of our simulation consists of a single Python script, as the name suggests. The Pygame package is used to draw our simulation to the screen, Thorpy is used in the implementation of UI sliders, and the entire simulation operates exactly as described earlier in this section. The standalone model was developed to assist in benchmarking the performance of our distributed model.

The visual representation of the simulation in action is depicted in Figure 1. It is worth noting that while the standalone model and the distributed system implementation differ significantly from each other in their underlying architecture: the front-end graphical user interface (GUI) for both models remains the same. This ensures consistency in the way the user interacts with the simulation and ensures an easy comparison of the results obtained from both models.

In this simulation, the behavior of the boids can be modified by changing the ranges of the three steering rules. This offers a great deal of flexibility and enables the user to explore different scenarios and analyze the impact of varying parameters on the overall behavior of the boids in the system. For instance, if the separation range is narrowed, the boids tend to flock too close together, whereas a wider separation range leads to more evenly-spaced and distinctly-rendered boids. As such, this feature offers a powerful tool for investigating the complex dynamics of flocking behavior and understanding its underlying principles.

It is important to note that in order for the intended emergent behavior to occur in the simulation, it is essential that the ranges of alignment and cohesion be greater than that of separation. If the separation range is too large and the cohesion range is too low, the boids will not form flocks and instead move away from each other. Similarly, if the alignment range is lower than the separation range, the boids will turn away from each other before they can come into alignment. This demonstrates the delicate balance between the different behavior ranges that must be maintained to achieve the desired flocking behavior in the simulation. Therefore, a careful selection of these parameters is necessary to ensure that the boids display the desired emergent behavior, while still avoiding collision with each other.

B. Distributed Model

For the distributed model, the simulation front-end is separated from the boid calculations, much like you would see in a real-world distributed system. The system is comprised of several components, including a game client, middleware client, worker nodes, a Redis database, and a RabbitMQ backbone. Redis, a database and message-broker library for Python, is used to store boid values that are accessed by both the worker nodes and the boids client [4]. Additionally, RabbitMQ, a message broker service, facilitates message-passing between nodes in the simulation [5]. Each of these components, with exception to the game client, which runs on the local machine, are containerized using Docker, and run simultaneously using the Docker Compose framework.

The game client uses Pygame to draw boids to the screen, but does not perform any calculations [6]. Instead, it sends delta time values to the middleware client over a RabbitMQ exchange. A remote procedure call in the middleware client then determines which worker nodes are responsible for which boids and sends this information, along with the delta time value, to the worker nodes via a RabbitMQ Fanout Exchange.

When a worker node receives a message from the middleware client, it performs all of the necessary calculations for the updated position, velocity, and heading of the boid based on values in a Redis database via another remote procedure call. The worker node then updates the Redis database with the new values.

Finally, the game client pulls the updated values from the Redis database and draws the boids to the screen using these new values. This process is repeated continuously, resulting in the simulation of a flock of boids. Ideally, all of this message passing and updating happens before the next clock tick signals the Pygame game client to draw the resulting frame to the screen. This system, which is outlined in Figure 2, operates using a distributed batch processing model, where the indices of active boid objects are partitioned between each worker node. This approach allows for efficient processing of large amounts of data and demonstrates the potential of using distributed systems for simulations.

The evaluation criteria for testing each of these models include several factors, such as the performance of the

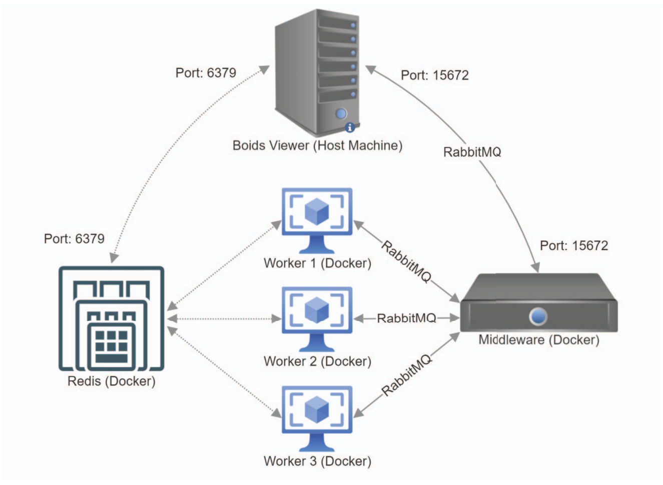


Figure 2: Visual Representation of Distributed System Architecture

simulation in terms of the number of boids, the frame rate of the simulation, and the number of worker nodes for the distributed model. Additionally, the scalability of the distributed model is also be evaluated by measuring its ability to handle an increasing number of boids while maintaining performance. Another essential evaluation criteria is the reliability of the system and its ability to handle errors and exceptions gracefully, although this is not addressed within the scope of our research.

IV. Results and Analysis

After distributing the workload and simulation state among different entities, we tested the performance to determine how well a spread-out flocking algorithm performed under varying quantities of boid objects and worker nodes. The experiment involved changing the number of total boids in the system between quantities of 25, 50, 75, and 100 and the number of workers available to perform calculations between quantities of 1, 3, and 10. The test was run on a Windows machine with an AMD Ryzen 5 1600 six-core processor with a clock speed of 3200 Mhz and consistent start-up procedures. The dependent variable measured was the time between frames which acted as a measure of how well the simulation could actively update the system based on the distributed nature of the algorithm when compared to the standalone model. Faster times suggested that the simulation was updating more frequently and with a greater frame rate, while slower times indicated that the simulation was being bogged down by the large amounts of calculations and message passing.

The information in Tables I through IV provide insights into the performance of the simulation with varying quantities of boid objects and worker nodes. The standalone model shows decreasing frame rates as the number of boid objects increases. This is expected as the computation required to update the position and velocity of each boid scales linearly with the number of boids. The frame rate drops significantly when the number of boid objects reaches 200 and 300, indicating that the system is unable to effectively handle the computation required for these many boids.

The data obtained from the distributed model with a single worker node in Table II also shows decreasing frame rates as the number of boids increases. However, the frame rates are much lower than the standalone model for all configurations. This is because the distributed model adds additional overhead due to communication between the game client, middleware client, worker nodes, Redis database, and RabbitMQ server. The time taken by the worker node to perform its computations is also much higher than the time taken by the standalone model to perform the same computations.

The performance of the distributed model with three worker nodes, as shown in Table III, is slightly better than the single worker node configuration, with higher frame rates and varied worker node computation times. However, the frame rates are still significantly lower than the standalone model for all configurations.

The data for the distributed model with 10 worker nodes in Table IV shows the highest frame rates for all configurations, with significantly lower frame rates than the standalone model only when the number of boids is 300. This indicates that the distributed model with a greater number of worker nodes can handle a larger number of boids more efficiently, as the computational workload is distributed across multiple worker nodes. However, adding more worker nodes beyond a certain point may not result in significant improvements in performance due to overhead and communication costs.

N	Delta time (ms)	FPS
25	32.580	30.693
50	33.209	30.113
75	34.233	29.211
100	35.170	28.433
200	61.144	16.355
300	108.208	9.241

Table I: Time between frames and frames per second of the standalone Boids simulation where n is the number of boid objects.

N	Delta time (ms)	FPS	Avg work (ms)
25	63.618	15.719	47.868
50	106.081	9.427	54.769
75	136.415	7.331	30.745
100	176.821	5.655	30.866

Table II: Time between frames, frames per second, and average time a worker node spends processing data with one worker node for the distributed Boids simulation where n is the number of boid objects.

N	Delta time (ms)	FPS	Avg work (ms)
25	73.715	13.566	43.499
50	110.231	9.072	41.583
75	162.884	6.139	50.729
100	217.682	4.594	52.575

Table III: Time between frames, frames per second, and average time a worker node spends processing data with three worker nodes for the distributed Boids simulation where n is the number of boid objects.

N	Delta time (ms)	FPS	Avg work (ms)
25	86.261	11.593	217.696
50	164.989	6.061	141.580
75	259.299	3.857	212.821
100	219.701	4.552	107.939

Table IV: Time between frames, frames per second, and average time a worker node spends processing data with ten worker nodes for the distributed Boids simulation where n is the number of boid objects.

There are a few trends to note given the experimental results. The first is that the standalone version performed much better than the distributed one, overall. The improvements over the standalone model that we were aiming to accomplish did not materialize, most likely due to the overhead of managing multiple scripts on the same device. The built-in latency of message passing may have also played a large role as the time to compute updated boids on the standalone system was much lower in comparison to the time it takes to move messages between the Docker containers used to run our back-end scripts. It is difficult to say for certain which effect had a greater impact on the overall performance, but this could be studied further by separating the containers into distinct processors such as on a cluster computer. If computing in parallel was an effective strategy, then a configuration like this would have run more efficiently than the standalone version.

Another takeaway to note is how the average frame rate decreased as the number of workers in the system increased. This supports our aforementioned hypothesis that adding multiple Docker containers created unnecessary overhead. Distributed systems thrive in their computational improvement over centralized systems when the algorithms they run are complex but easily separated into discrete buckets to process. Breaking down each worker to calculate a different set of boids checks the box for the latter, but the evidence suggests that the geometric calculations used in determining steering velocity and position values of boids are not elaborate enough to benefit from a distributed implementation.

In addition to the quantitative results, there were also qualitative observations that can be made about the performance of the standalone and distributed models. Based on the visual feedback from the simulation, the standalone model appeared more intuitive and natural, as the birds looked like they were actually demonstrating the emergent behavior of birds. On the other hand, the distributed model was choppy and had a low frame rate, making it difficult to tell that flocking behavior was occurring easily. This could be due to the added latency of message passing between the Docker containers used to run the backend scripts. These observations suggest that while the distributed model may be able to handle a larger number of boids more efficiently with more worker nodes, there are other trade-offs that must be considered, such as the potential impact on the visual quality and perceived naturalness of the simulation.

V. Challenges

When implementing a distributed system, several common challenges occur. The biggest ones that come to mind are undoubtedly consistency and availability. While one central computer knows for certain that what is stored in its own memory is the ground truth for its programs, when multiple computers operate in a networked configuration, this may not always be the case. Additionally, if a single process within a distributed system is reliant on another, measures must be put into place to ensure the original process remains running when the other process fails.

The paper *A Compendium on Distributed Systems* suggests that apart from consistency and availability, there are generally numerous difficulties that emerge while developing a distributed system [7]. Distributed systems are complex and have several challenges that need to be addressed for successful implementation. These challenges include heterogeneity, transparency, openness, concurrency, security, scalability, and failure handling.

This article also presents solutions to these challenges faced by distributed systems. In terms of consistency, one solution is to use transactional event handlers. Transactions enable the appearance of a group of operations as if they were executed simultaneously or not at all, providing a robust and influential capability. For the issue of availability, replication transparency can be implemented. If the system provides repetition (for availability or performance reasons), it should not affect the user. Also, scaling transparency requires the ability of the system to expand without impacting the application algorithms. The capacity to grow and evolve is crucial for many businesses and the system should also be able to reduce in size when necessary and allocate necessary space and/or time.

In another paper titled *Some Issues, Challenges, and Problems of Distributed Software System*, similar distributed software system implementation challenges are analyzed [8]. The authors suggest that different task scheduling algorithms should be evaluated on available task evaluation parameters for a specific task graph. A practical approach to evaluating the performance of these algorithms is to inject faults, such as communication or computation delays, and observe their performance under these circumstances. This approach will minimize the challenges inherent to the implementation of distributed systems and improve overall performance.

Admittedly, with our design, we took a shortcut by managing everything on a "simulated" distributed system where each component of the system was containerized on a local machine using Docker. Failures and consistency problems were not nearly as prominent in this case, but the tradeoff is severe latency issues due to each container competing for system resources and communications overhead. If we were to set up separate computers on a network, the greatest challenge would most likely be connection consistency, which would be the prime driver for delays in frame updates. To alleviate this, a "dead-reckoning" or similar algorithm could be employed such that the simulation could estimate updated positions until newer, more reliable data returns.

VI. Conclusions and Future Work

The boids algorithm is a simplistic yet exciting behavioral simulation to study. Its framework provides an easy way to learn simulation and understand basic computer graphics and game engine physics. Furthermore, it provides for an enticing distributed systems case study as we have outlined in the paper. Although our implementation using Docker containers on a single computer did not yield the desired performance improvements, we believe that exploring alternative architectures, such as using separate hardware, could provide promising results.

Future work could also focus on implementing more advanced features to the boids simulation, such as obstacles and terrain, which could introduce new challenges for distributed processing. Additionally, exploring alternative parallelization techniques, such as using GPUs or multi-node clusters, could potentially yield better performance improvements than our current implementation. In order to better identify the difference in performance between the standalone and distributed models, further research could be done in analyzing how implementing our distributed model over an actual network with distinct machines compares to the performance of our standalone model. Lastly, the logging messages we used to debug aspects of the distributed model could be removed to reduce the volume of console output.

VII. Acknowledgements

We would like to acknowledge the contributions of Brett Martin for building the standalone Boids model, Ryan Winz for solidifying the communication and database protocols used in the final implementation, and Luke McFadden for setting up Docker and configuring the distributed system components to run in Docker Compose. Additionally, we would like to thank everyone who contributed to building the final distributed Boids system.

VIII. Disclaimer

The views expressed in this paper are those of the authors and do not reflect the official policy or position of the United

States Air Force, the Department of Defense, or the U.S. Government.

References

[1] C. Reynolds, "Flocks, herds, and schools: A distributed behavior model," *SIGGRAPH '87: Preceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques.*, vol. 21, no. 4, p. 25–34, 1987.

[2] J. Clark and D. Jacques, "Flight test results for uavs using boid guidance algorithms," *Procedia Computer Science*, vol. 8, p. 232–238, 12 2012.

[3] E. Yong, "Autonomous drones flock like birds," *Nature*, vol. 9, Feb 2014.

[4] [Online]. Available: https://redis.io/

[5] [Online]. Available: https://www.rabbitmq.com/

[6] [Online]. Available: https://www.pygame.org/news

[7] A. Khole, A. Thakar, A. Kulkarni, H. Jadhav, S. Shende, and V. Karajkhede, "A compendium on distributed systems," 2023. [Online]. Available: https://arxiv.org/abs/2302.03990

[8] K. Mishra and A. Tripathi, "Some issues, challenges, and problems of distributed software system," *(IJCSIT) International Journal of Computer Science and Information Technologies*, vol. 5, no. 4, pp. 4922–4925, 2014. [Online]. Available: https://ijcsit.com/docs/Volume%205/vol5issue04/ijcsit2014050420.pdf

Author Biographies

BRETT M. MARTIN is a 1st Lieutenant in the United States Air Force, currently stationed at Wright Patterson Air Force base. He received a B.S. in Computer Engineering from the United States Air Force Academy in 2020, a M.S. in Computer Engineering from the Air Force Institute of Technology, and is currently a doctorate student at the Air Force Institute of Technology studying Software Engineering. Lt Martin is currently researching human-autonomous systems at the 711th Human Performance Wing, Wright-Patterson AFB, Ohio USA.

RYAN D. WINZ is a 2nd Lieutenant in the United States Air Force, currently stationed at Wright-Patterson Air Force Base. He received a B.S. in Industrial Engineering from North Carolina State University in 2021 and is in the M.S program for Computer Science at the Air Force Institute of Technology. Lt Winz currently works in the Cyber Resiliency Office for Weapons Systems at Wright-Patterson AFB, Ohio USA.

LUKE J. MCFADDEN is a 1st Lieutenant in the United States Air Force, currently stationed at Wright-Patterson Air Force Base. He received a B.S. in Computer Engineering from the United States Air Force Academy in 2019 and is in the M.S program for Computer Science at the Air Force Institute of Technology. Lt McFadden is currently studying machine learning with regards to nuclear non-proliferation efforts.

TOR J. LANGEHAUG received the B.S. degree in computer science from California State University, Sacramento, CA, USA, in 2009, and the M.S. degree in information security from Carnegie Mellon University, Pittsburgh, PA, USA, in 2017. He completed his Ph.D. in 2022 at the Air Force Institute of Technology investigating applications of machine learning to low-level microarchitectural data sources.