15-418/618 Spring 2022

Natural Selection Simulation Final Report

Group: Hridayesh Joshi (hvj), William Giraldo (wgiraldo)

1 Summary

In this project we implemented and parallelized a simple natural selection simulator where agents compete to survive with limited resources, and where the most capable survive to pass on their characteristics to the next generation. Parallelization was done with OpenMP attempt to garner the best performance speedup to allow for the largest possible simulations (with large worlds and many agents). We analyze the best methods possible to parallelize a problem with complex interactions between multiple agents.

2 Background

In general the simulation is run by creating an initial population of agents and placing them in the world, along with some amount of food. The simulation then runs for a certain number of rounds. Each round lasts a certain number of steps, and at the end the most performant agents are reproduced, via a genetic algorithm.

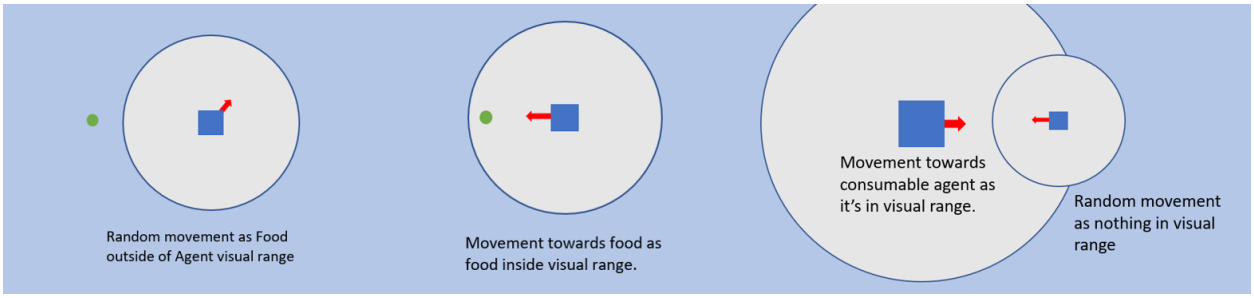
2.1 Data Structures

Each agent is represented by its core properties, which are its speed, size, and vision range, as well as its position. We maintain an array of agents and food, which is modified as the algorithm progresses. Our algorithm is designed to work in the 2D plane, but could easily be adapted to work in 3 dimensions, or even higher if so desired.

The choice to use a array to maintain the array of agents allows us to work on different parts of the array efficiently, as well as focus on the important information to each agent. The array was chosen over some other options such as a grid data structure, where agents and food would be placed on the grid, because it allows for more efficient parallelization over the agents themselves, instead of physical grid space, which is important because agent behavior and location is generally unpredictable.

2.2 Key Operations

2.2.1 Action Phase



On every tick of the simulation, each agent has to go through a decision process to determine its next action. Its decision is primarily based on 2 things, the nearest agent, and the nearest piece of food. In order of priority, the following actions can be taken by the agents at each step.

1. If it is possible to eat the nearest food, then do so.
2. If it is possible to consume the nearest agent (current agent is 20% larger than them, and it's within eating range), then do so.
3. If the closest agent can consume the current agent (they are 20% larger than it), and they are within the current agents visual range, then move away from them.
4. If the current agent has the ability to eat the closest agent, but is too far to do so, and it is within the agents visual range, move towards it.
5. Move towards the nearest food if the agent can see it (it is within the agents visual range).
6. Make a random movement (weighted to be similar last direction of the agents movement).

Each of these actions is affected by the individuals agents properties. Its size allows it to eat smaller agents, and increases the range at which it can consume other agents and food, its speed allows it to move faster (make larger movements each tick), and its visual range increases its sensing distance (a larger visual range means it can begin to act sooner, and won't move randomly as often as it is more likely to have a goal destination).

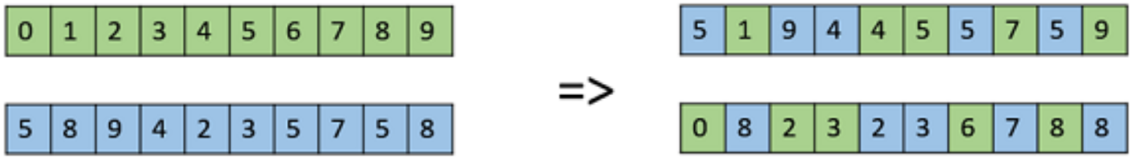
It is also important to note the main limiting factor placed on these agents. Each agent begins each round with the same amount of energy, and adds energy when it consumes food or another agent. Whenever an agent makes a movement, it consumes energy proportional to how large it is, how fast it is, and how far it can see. This means that a very fast, far seeing agent will consume more energy per step than a slower, closer seeing agent.

If an agent runs out of energy, then they die. This creates a natural fitness function from which to judge each agents abilities (surviving agents are better), which is an important criteria for the next phase of our key operations.

2.2.2 Reproduction Phase

The **reproduction** phase first determines the top some percentage (we decided to use the top 25%) of the agents, where the rankings were determined by their fitness scores. We call these the 'fit' agents. As the agents' fitness scores should reflect every agent's ability to survive, the fitness scores were a weighted sum of the number of rounds it has survived since its inception and the amount of remaining energy it had at the end of the round. The select fittest agents were then randomly paired with one other agent. Note that a pairing could consist of two fit agents, but an agent can only procreate once in our algorithm.

There are myriad ways to determine the traits of the new child agent, but for this algorithm, we decided to utilize a uniform crossover. In a uniform crossover, for each chromosome (trait), we flip a coin to decide which one of the two parents' chromosome are we getting. We then fuse the chromosomes to produce the new child agent. The diagram below illustrates how this works with 9 different chromosomes.



Finally, mutations in real life occur all the time. Likewise, we introduce mutations to the newly created child agents. Within the context of our algorithm, for each trait, a mutation occurs with some probability (the default is 40%). If a mutation occurs, new value for that trait is chosen within a small range centered at the original designated value.

2.3 Inputs and Outputs

There are many configurable initial parameters in the simulation. These can range from things such as the number of steps in a round, number of rounds, to the initial starting properties of the agents, the value of the food, and the cost of each agents actions.
However, the two most important inputs (and the only ones customizable from the command line for simplicity) are the initial number of agents, and the amount of food placed on the world each round.

2.4 Benefits of Parallelism

There are two components of the algorithm, which were described in section 2.2, where there is a great benefit to parallelism.
The first is when a round is being run, and each agent has to be iterated over to decide on an action for it. To decide an action, and agent must also know the nearest food and nearest agent, which requires finding this information for each agent. If there is n agents and r food, then this process is $O(n(n+r))$. Additionally there are many mathematical operations which occur to calculate the next movement vector for each agent. With parallelism, each agent can be considered in parallel, and with proper implementation, the time required to find the closest agent/food can also be reduced significantly. Two approaches are taken to this which are discussed later.

The second place to benefit from parallelism is the **reproduction** phase, which is after a round when the surviving agents are assigned fitness scores, and the most capable are used to create the next generation of agents, who'll play in the next round. First, we can perform a simple mapping to compute all the agents' fitness scores. Next, when we have to match the fit agents for procreation, and the worst case cost asymptotically is $O(n^2)$. Thus, we stand to benefit greatly if we can parallelize over the fit agents (which is about a fourth of all agents). Additionally, we can reduce the time it takes to randomly select an agent that has not already been selected.

2.5 Workload and Dependencies

In the simulation, a certain number of rounds are played. In each round, the simulation is run, and the agents interact and move on the grid, this is the **action** phase. After each round, the **reproduction** phase occurs, which is when the surviving agents are used to create the new generation of agents which will play in the next round (alongside the surviving agents). These 2 processes, the **action** and then **reproduction** phases, are inherently sequential, with the **action** phase always needing to come first. There is also inter-round dependencies, in that each rounds **action** and **reproduction** phase must come before the next rounds phases, for obvious reasons. Because of this, there is no opportunity for parallelism across phases and rounds.

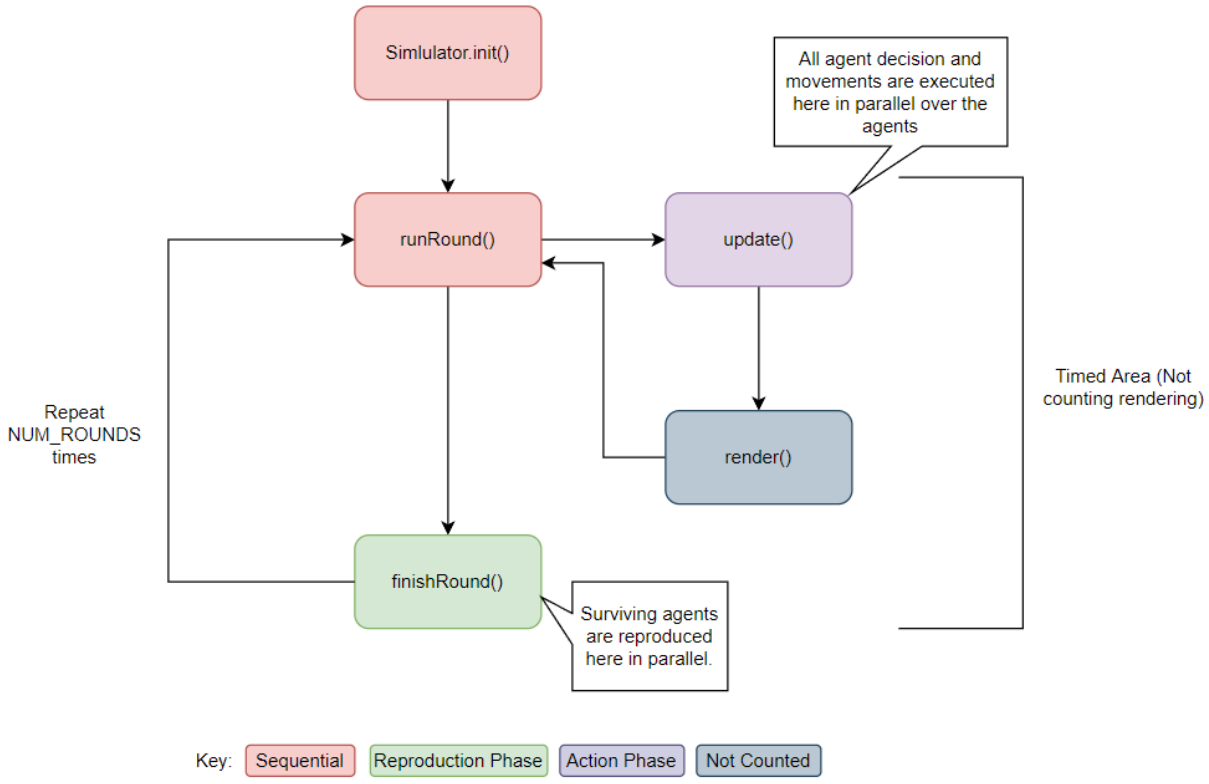
In each round i.e. during the **action** phase, each agent cannot compute its next action until it has found both the nearest food, and the nearest agent, which requires it to look at every other agent and all the food. However, each agents action is independent of the other agents action for a single tick, so it is possible to consider each agent in this step independently. Because the computation is across agents and not grid spaces, most of the locality comes from considering agents which are close in the list, as well as considering food which are near each other while processing agent actions.

During the **reproduction** phase, although, we are parallelizing over the fit agents, it is

imperative that we do not reuse an agent for procreation. Another thing to note is that this type of computation isn't well-suited to SIMD, since there is lots of branching involved in each individual agents decision process based on its own attributes, which may vary significantly from nearby agents, even if they both have the same nearest food and nearest agent.

3 Approach/Program Overview

The general program runs by looping a set number of rounds. In each round, a certain number of steps is executed. Each step consists of processing all the agents and determining what there actions will be based on the environment. Each round consists of this **action** phase, as well as the **reproduction** phase. After the **action** phase occurs where all the agents make there decision and act, the **reproduction** phase occurs in which the surviving agents are used to repopulate the grid for use in the subsequent round. The parallelism in these 2 separate phases will be discussed further below.



3.1 Technology Used

Our project was developed in C++, and parallelized using OpenMP. We opted not to use CUDA at the start because the agents actions are very divergent, and would not work well with the SIMD and warp execution used on GPU's. Determining actions and acting is the main bulk of the program, and thus it is more logical to begin with OpenMP and CPU parallelism. There is no specific target for the program, but if the simulation is to be run with many thousands of agents, then it is better to do it on a high core count machine (such as PSC), so the load will be handled better.

3.2 Configurations and Parallel Mappings

As mentioned earlier, the parallelization occurs separately in each phase. In the **action** phase, it primarily occurs when having each agent determine and check its next action. The most natural mapping (and the one we used) is to take each thread and assign it to an agent, and have it compute its action.

There were a few more possibilities, such as parallelizing over each grid space (or region) and computing for agents which are there. However, this was opted against as in general in these simulations we found that the number of agents might be fairly sparse compared to the amount of grid room depending on the setup, so doing this would be unnecessary work, which wouldn't garner much speedup either.

The **reproduction** phase consists of the top some percentage (we decided to use the top 25%) of the fittest agents mating with other agents. The most straightforward avenue of parallelism was across the fit agents. Additionally, to ensure that each agent is only selected once for procreation, we utilize a reduction over a visited set, indicating which agents have been used and which ones have not. This bodes well for the rejection sampling method we used to sample an agent in which we continuously randomly select an agent until one that has not been used is selected.

3.3 Changes to Serial Algorithm

The overall approach and computation remained the same across the serial and parallel algorithm. However there was one change to account for fairness when parallelizing over the agents.

The changing of the agents positions was pipelined when parallelized, i.e. first we compute the movement that the agent will make, but we don't make the agent actually take this movement until all other agents have also planned their moves. Then all of the planned movements are committed at the same time. This change was made to ensure fairness, so that each agent was working with the same information as the other ones, and didn't gain

an advantage by being computed first.

The **reproduction** phase required no modifications when transitioning from the sequential to parallel version of our algorithm. However, we must ensure that we do not reuse an agent. In particular, a fit agent may get reused if we are not careful since we parallelize over the fit agents.

3.4 Past Iterations

Since the beginning, our general approach has been the same since the axis of parallelization is fairly straightforward, and some approaches were clearly not going to be worth the investment, such as the parallelizing over grid space approach. This approach might seem good because of locality, but we realized that regardless of locality it still makes sense to parallelize over all the agents, because there are many less agents than there are grid spaces. We did however implement a variation of this, the goal of which was to speedup the process of finding the closest agent and food. In this approach, we still parallelized over the agents, but wanted a faster way to compute the closest agent and food. To this end we divided up the agents spatially, and maintained groupings of close agents, so that in each step for an agent it wouldn't need to search through all of the agents and food to find the closest food. However this approach was quickly realized to be very inefficient for several reasons. First of all, it turns out that because agents have very different properties, they can diverge rather often, which requires frequents updates to the groupings. Additionally depending on the round, the distribution of agents over space could become rather sparse, at which point it doesn't make sense to maintain spacial groupings for efficiency, since each agent would be in its own group, or the groups would be very large, which also doesn't help.

The groups were also difficult to hold to correctness because different agents have different visual ranges, which has an impact on which other agents/food they can see, and simply complicates the grouping process.

Additionally, because of the nature of the problem, many agents die, and many new agents are created after each round. Then when a new round begins all agents receive new positions to ensure fairness. Because of this, when maintaining groupings, the groupings would need to be recomputed completely in between rounds, which was not efficient. Because of this this approach was quickly seen to be flawed, simply because of the natures of the problem.

This led to our current approach, which unfortunately has to do the work of checking all the agents. Although this approach may seem inefficient, it was the best we were able to get with the time which we had.

3.5 Starter Code

We did not have any starter code, but do use a library called SDL for rendering.

4 Results

4.1 Metrics

Performance is measured by the wall-clock time comparison and speedup from increasing the number of cores, as well as variations on different inputs. We will keep track of the time that our program takes for different core counts on the same input and compare these, since there is a lot of variation from input to input. Note that the program has lots of randomness, so all times are accumulated averages over multiple runs on the same setting.
We are only timing the time it take to actually run the simulation, i.e. the accumulation of all the **action** and **reproduction** phases, and not any rendering or setup time, since the focus is on computing the simulation itself as quickly as possible.

4.2 Setup and Inputs

While the simulator has many parameters which are available for tuning, the following test will only concern themselves with 2 primary inputs, the initial number of agents and the amount of food per round. These are the 2 parameters which should have the largest impact on the performance of the simulation, and thus are the most interesting to examine in the context of improving performance and parallelization. In particular, we parallelized over the agents and food for both phases of the algorithm, so we are concerned with the speedup as the inputs scale. The other inputs such as the probability thresholds and agent traits, are used for simple mathematical computations, so changing these values will not affect runtime and speedup.

For the other cases that do not involve varying the input sizes, we maintain the same parameter configuration: 1000 agents, 700 food, and 1000 x 1000 grid size.

The benchmark for each test is the serial code running with no OpenMP capabilities. Additionally, since the number of agents will change each round by the nature of the simulation, we decide to maintain the same number of agents throughout the simulation in order to purely and effectively analyze the performance of the code across the various input sizes.

5 Performance Graphs

5.1 Dynamic vs. Static Scheduling

The above graph shows the differences in speedup for dynamic versus static scheduling of threads. We can see that at lower processor counts, they are about even, but there is a significant difference as the processor counts scale up. This is likely because as we get to higher core counts, the overhead of dynamic scheduling becomes more worthwhile as the amount of work done for different agents may differ, and dynamic scheduling may greatly reduce load imbalance when compared to static scheduling.

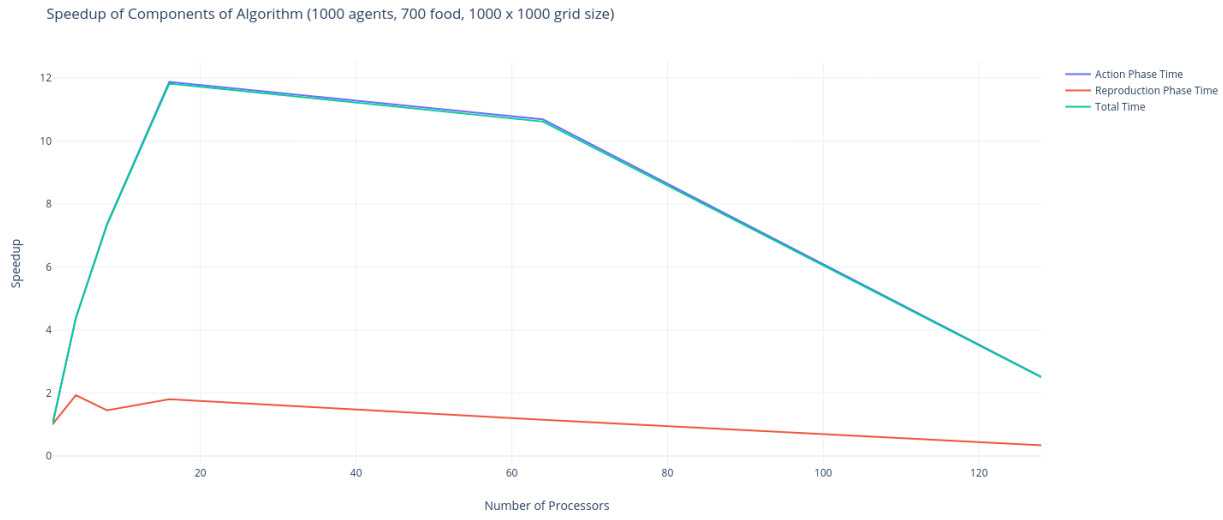
5.2 Runtime vs. Number of Agents

The above graphs shows the runtime as it relates to the number of agents. Number of agents is one way in which the problem size can change. Here we see that this clearly increases runtime. Although it is slightly hard to tell, it is reasonably clear that the runtime seems to be increasing quadratically. This makes sense, as we state earlier that in the **action** phase, each agent must loop over all other agents to find the closest one, which is $O(n^2)$. Although this is flattened out by increasing the number of processors, as long as there isn't an infinite amount of computational power (like a processor per agent), this will continue to be a quadratic curve. We can also see that the total time about halves each whenever the number of threads is doubled, which is very good and as expected, since a large portion of the runtime is computed in parallel.

5.3 Runtime vs. Number of Food Pellets

The second way in which the problem size can increase in this simulation is by increasing the number of food pellets. In this graph we can see a similar relationship as the previous one, but it seems to be more linear in nature. Because now for each agent we still have to find the closest food, and there is increasing amounts of it, but the number of agents doesn't increase. This graph also indicates that at these lower thread counts we are achieving similarly good decreases in runtime as the thread count increases.

5.4 Speedup in Different Program Sections



This graph shows the speedup for different thread counts on the **action** and **reproduction** phase. The focus here is that the **action** phase benefits much more from parallelization than the **reproduction** phase. This is likely due to the fact that there is much more work done in

the **action** phase than in the **reproduction** phase. Additionally the **reproduction** phase uses lots of standard library calls to modify data structures, which are done sequentially, and this places lots of limits on the speedup, so its possible that these significantly limit the speedup which can be obtained in the **reproduction** phase, which is why it doesn't benefit as much from the parallelism. We can see that it actually has a sub-1 speedup for higher thread counts, which makes sense since now there is the additional work needed to deal with more threads, when they aren't actually helping with any more of the computation.
Note that this graph was taken with static scheduling, because initially we'd believed that static scheduling would outperform dynamic because it didn't have the runtime overhead, and all the agents workload should be the same. However, we can see that this wasn't the case from our previous graph. This detail shouldn't take away from the main conclusions of this graph, which are virtually identical when dynamic scheduling is used (graph not included for brevity).

5.5 Limiting Factors on Speedup

There are a few different factors which could be preventing the program from reaching perfect speedup, but likely the main culprits are the parts of the code which cannot be parallelized. This is specifically when each agent needs to find the closest other agent and closest food. To do so, it must look at every single other agent. As a side note, normally a reduction might be possible to compute this more effectively, but on our system and compiler (primarily Windows with MSVC), the OpenMP version was not recent enough to support the custom reductions needed. Additionally these reductions wouldn't work with OpenMP's current capabilities, since to say which of 2 agents is closer to the current agent, it isn't possible to decide by comparing these agents with each other, only with the current agent, which isn't possible to setup in a custom reduction in OpenMP. Because of this, this area of the code was entirely sequential, and although we're able to parallelize over the agents entirely, each agent needs to consider every other agent sequentially.

This same effect is predicted by Amdahl's Law. It essentially says that speedup is limited by the amount of parallelizable code. In our case, while we could greatly parallelize over the agents, each agent still had to do a significant amount of sequential work. Additionally because there are only so many processors, and many more agents, while adding more processors does give speedup, it is comparable to decreasing the number of agents by a constant factor. However, inside each thread of each agent, they still must do sequential work over a large number of agents, which becomes very limiting.

For higher processor counts speedup drops significantly. This is likely due to some of the above given reasons, but also because at these higher counts, it may be the case that having this many processors no longer does enough work, and it costs significantly more to have the overhead of having that many threads in the first place.

We also had to pipeline certain parts of the algorithm to ensure fairness when run in parallel. This could contribute partially to the lack of significant speedup, since it requires that we go through after all the agents have decided what there next move is and actually commit these moves, may require some expensive memory operations since at this point we may no longer have this information cached since we'll have compute other agents and information.

Additionally, by design our program must compute the **reproduction** phase after the **action** phase, and before the next **action** phase. As specified earlier, an important aspect of the **reproduction** phase is that each agent procreates only once. Since we parallelize over the fit agents, we must check if an agent has been utilized inside that loop. Moreover, in order to use our rejection sampling method, we must ensure that there exists at least one other agent that has not yet procreated. Otherwise, the algorithm would be stuck in an infinite loop. We attempt to perform a reduction over the visited set to determine whether or not all the other agents have been used; however, we primarily parallelize over the fit agents (the outer loop), so there may not be enough threads available to perform this efficiently in most instances. The infeasibility of this parallelizing this phase is evident in Section 5.4, where the speedup of the **reproductive** phase is significantly less than the other phase.

5.6 Runtime Breakdown

The vast majority of our program is spent in the **action** phase, and very little is spent in the **reproduction** phase. This is because the **action** phase runs for many steps, while the **reproduction** phase only runs once at the end of each round, and the majority of the round is compromised of the **action** phase because that is where all the main computation for agent decisions is done, and is repeated for the number of steps in a round.

This shows us that a lot of the room for improvement will come from speeding up the **action** phase, and that less focus can be put on the **reproduction** phase. This also goes hand in hand with the earlier conclusions we made by looking at the graph of speedup on these different phases, where we see that the **reproduction** phase gains very little from parallelism and higher thread counts.

5.7 Machine Choice

Although we were unable to fully implement a GPU algorithm and adequately compare its performance, we still believe that targeting CPU parallelism was the better choice, depending on the scale. We avoided GPU computation as the initial implementation of the problem because even from the start it is clear that there is lots of divergence in the computation since agents may take drastically different actions based on their individual characteristics. However, as the scale of the problem scales up, GPU computation may still provide better speedup, simply because of the sheer increase in computational power that it would be able to provide, as long as the divergence is not as problematic as we believe. However, it seems

that for the sizes which we test, CPU parallelism was sufficient, and fairly speedy at lower core counts, although speedup did decrease significantly when too many cores where used.

However, this could be due to different parameters which affect how the simulation plays out, and subsequently the number of agents at each round, and that with different parameters and tuning which specifically affect the number of agents and their interactions, the speedup would have been greater.
Regardless, it definitely would have been interesting to test a GPU implementation, and given more time this is definitely an avenue which we would have explored.

6 Output

Please see the final recording for an example of how the program runs when rendered. Below is a screenshot of the simulation.

7 References

Inspiration: https://www.youtube.com/watch?v=0ZGbIKd0XrM

8 Work Distribution

Equal work was done by both partners.
William - Sequential Algorithm, Action Phase
Hridayesh - Rendering, Reproduction Phase, Benchmarking
Both - Parallelization