Seminar Report

Predator-Prey Relationship in a Closed System

Vincent Hasse and Kimia Taba

MatrNr:
Vincent Hasse: 12019510
Kimia Taba: 29019649

Supervisor: Dorothea Sommer

Georg-August-Universität Göttingen
Institute of Computer Science

September 30, 2023

Abstract

Predator-prey simulation is a critical research area in ecology and computational biology, try to understand the dynamics of predator and prey interactions in natural ecosystems. Beside that, High-Performance-Computing(HPC) is one of the top fields of computing and it is used in many fields for improvement. So we tried to do the simulation of predator-prey population by using the advantages of HPC and using MPI. We compare the performance in sequential way and parallelized way by using benchmarking tools and show how it boosts in the parallelized approach. In this report, we introduce our approach and the idea behind it in more details and introduce the tools that we used. Then the limitation of our implementation and the challenges that we had, and also further optimization opportunities are discussed. All in all, we introduce our solution that could do the simulation in parallel successfully.

Statement on the usage of ChatGPT and similar tools in the context of examinations

In this work I have used ChatGPT or a similar AI-system as follows:

- [✓ Not at all]
- ☐ In brainstorming
- ☐ In the creation of the outline
- ☐ To create individual passages, altogether to the extent of 0% of the whole text
- ☐ For proofreading
- ☐ Other, namely: -

I assure that I have stated all uses in full.
Missing or incorrect information will be considered as an attempt to cheat.

Contents

List of Tables

List of Figures

List of Abbreviations

HPC	High-Performance Computing
BFS	Breadth-First Search

1 Introduction

Simulators have lots of usages in science. Often models of reduced complexity compared to the real world problem are used to get an estimate for scenarios with certain parameters.

The Predator-Prey model shows the growth of two interdependent populations. Preys get their energy by eating plants but the preys serve as food for the predators and the growth of one population is influenced by the size of the other population. As prey numbers go up, predator numbers also go up. When there are many predators, the number of prey declines, causing a decline in the prey population ultimately leading to timely shifted populations of predators and preys. This is also described by the first Lotka-Volterra rule. This is a concept in ecological modeling and population dynamics that provides a framework to understand the complex interactions between predator and prey species within ecosystems.

Due to a high number of dynamic variables in large ecosystems with thousands of agents on it, simulating a world like that needs lots of compute resources and time. In this project, we created a simulation of an ecosystem which takes advantage of multiprocessing. This allows to compute more simulation steps with more agents in a bigger world, compared to a sequential approach.

The biggest challenge of environment simulations is that for every step in the simulation every agent has to have knowledge of every other agent and/or part of the simulation environment that may have influence on its behavior, in our case especially the movement. This requires a lot of communication between compute nodes when the simulation is parallelized. If no assumptions are made and restrictions are applied, it can be very inefficient to parallelize a predator-prey environment. This is due to the fact that if agent 1, let it be "prey 1" moves, the world-state has to be updated in order to calculate the movement for the next agent, lets call it "predator 1". Otherwise it could lead to e.g. predator 1 to move to the old position of prey 1 and trying to eat it, while in the new world state prey 1 has moved to a new position already. We have to decide for some way of handling these mismatches in world-states while still allowing for a parallelization of the simulation without ramping up the needed communication between nodes to an absurd level that would limit the scaling of the solution.

We propose a solution in which the configurable environment is split up into various parts that then allow for a parallel calculation of agent-movement which constitutes the biggest computational load of agent behavior in our simulation. The solution is implemented using MPI for inter-node communication avoiding the overhead that would normally cripple performance resulting from pickeling Python-objects.

We evaluated the correctness of our approach and simulation by using predictions of the first Lotka-Volterra rule. Based on that, we found out that our approach works properly. Also for evaluating the performance of the parallel approach, we used profiling packages, that show using multi processes can increase the simulation performance significantly but not linearly; because of the communication overhead for example.

The presented solution offers an implementation of a research-related simulation in Python - a wildly known programming language also in the non-computer-science community - while avoiding the Python-specific overhead that would normally result from pickeling Python-objects in order to be able to send them using MPI.It provides a simulation environment in which different agents with different behaviors could be added

This report is organized as follows: In chapter 2, we talk about the general idea of the project in more detail. Chapter 3 presents our implemented environment and the sequential approach as well as the steps that had to be taken in order to implement a parallelized solution. Chapter 4 describes the steps for running the project in different approaches. Chapter 5 is all about evaluation and performance; the benchmarking strategy and the results. Chapter 6 is about the challenges and borders that occurred during the development process. In chapter 7, some ideas on further optimizations are described, which were not implemented in the current version. Chapter 8 is the conclusion of the report: The whole project is reviewed and learning achievements are discussed.

2 The general idea

The general idea for the project was to build a simple ecosystem that simulates predator-prey relationships. In a later stage, machine learning "brains" should be added to the agents to develop individual survival strategies, therefore Python was chosen as a programming language. As being part of a practical course for High-Performance Computing (HPC), the simulation should be run in parallel on the cluster computer of the GWDG. The simulation is run in simulation-steps in each of which plants can spawn and agents take their actions like moving.

2.1 Simulation Configuration and Logging

Many parameters of the simulation can be specified in the *config.py* file. For every simulaiton, the configuration-parameters are logged in the *sim.log* file as well as the population for every simulation step in the *population.csv* file. We also implemented a basic visualization: If turned on, the program will generate a *.png* image out of the world state for every simulation step. While slowing down the program significantly, it helped a lot with debugging and making sense out of the data. The size of the images scales with the world-size as otherwise not every cell was drawn by PyPlot.
The *population.csv* file can be used by the *createPopulationGraph.py* script in order to create a population graph after the simulation.

3 Implementation

In the following the implementation of the idea as a sequential approach as well as the parallelization will be described.

3.1 Outline

For the simulation several parts of an environment need a technical counterpart in the program. This will be presented in the following, a visual representation can be seen in figure 1.

1. The World
The world is the frame that the simulation is run in. It provides spaces for agents to move in and plants to grow. Technically it is represented by a $n * n$ NumPy array of

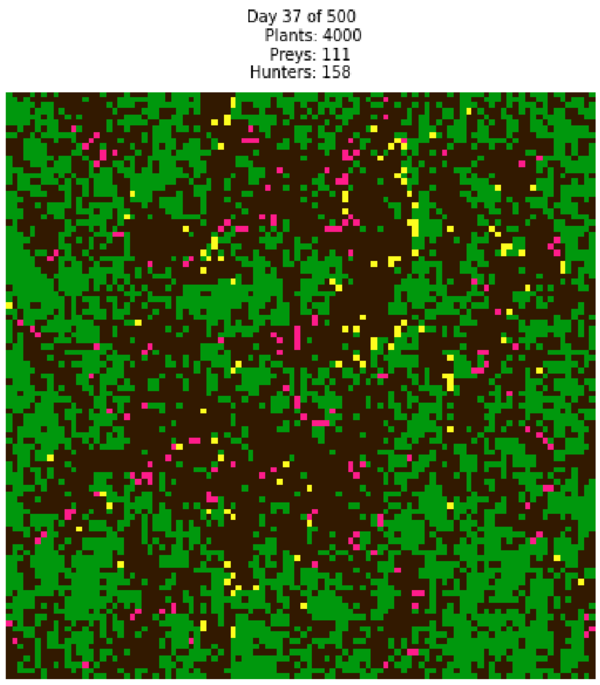


Figure 1: Example of the World-Visualization: Free spaces in dark brown, plants in green, prey in yellow and hunters in red. Also the Day (Simulation Step) and the population sizes are displayed.

dtype uint8. Hence, every cell of the array can store a value between 0 and 255. With 0 representing a free space in the world, theoretically 255 different types of entities - e.g. plants, preys, hunters - can be added to the simulation. For our project we only used three generic ones: plants, preys and hunters. As the array can only hold one value, each cell can only either be free or occupied by one entity. The world is "looped", meaning that an agent that leaves the world to the right side will appear on the very left of the world at the same height - leaving the world in any other direction will make them appear on the opposite side respectively.

2. Plants
Plants, represented by the numerical value 1 have no simulated behavior. They spawn at random, free positions at a rate that can be specified in the config-file (see chapter 2.1) of the simulation. This includes the max spawn rate as well as the maximum percentage of the world that can be populated by plants.

3. Agents
There are two types of agents in the simulation: preys and predators that have a specific position in the world. Preys have plants as a food source, hunters eat preys. While they normally avoid each other, when two hunters meet, one hunter will kill the other and consume it. Preys and Hunters both have an energy pool that gets depleted when moving and gets renewed when food is consumed. If it drops to zero, the agent dies. Agents also have a maximum age and can breed within the interval of 25% and 75% of their maximum age. Breeding currently does not require searching a mate but is just implemented as spawning another individual nearby, with a certain chance at every simulation-step, if the age of an individual is within the interval mentioned above and the energy of the

agent is above a certain threshold. When offspring is produced, the energy level also drops significantly. Movement is implemented as a Breadth-First Search (BFS) where only free cells as well as cells containing plants are considered valid points of the movement path. Plants are destroyed when hunters move over them (for preys they are the food source and hence the target of the path-finding algorithm) in order to avoid the encirclement of hunters by spawning plants. Hunters consume double the energy than prey but in return are allowed to move two cells per simulation-step.

3.2 Sequential Approach

The first thing we did was to implement a sequential implementation of the simulation as a first prototype. This helped to get deeper into the idea and find implementations and for parts of the simulation we would need as well as easily debugging these solutions.
In the squential approach, every prey and hunter was contained in one of two dictionaries with a key-value pair of $(ID, Prey)$ or $(ID, Hunter)$ respectively. For every simulation step the program loops over these dictionaries, handling the movement and every other function of the agent one after the other before finally spawning plants and continuing with the next simulation step (see Figure 2).

Sequential Simulation

```
for step in range(config.SIMUALTION_STEPS):
    for everyAgent in dicts:
        addAge()
        moveAgent()
            letItDie(energy, age)
            letItEat(isFoodAtNewPosition)
            letItBreed(energy)
    spawnNewPlants()
```

Figure 2: Pseudo-Code for the simulation in the sequential approach.

This has the benefit of the world state being always up-to-date when calculating movements, spawning offspring etc. For example when the movement of the second agent is calculated, the position of the first agent has already been updated and hence its new position is blocked for movement while its old position is free.
While the solution was working, the goal was to implement a parallel solution in order to make the simulation as fast as possible, even for large worlds and hence many agents. The parallel implementation with the changes that had to be made to the sequential approach as well as the problems that arose and how they were tackled will be presented in the following.

3.3 Parallel Approach

The parallelization of the program was implemented using the mpi4py library. The first step was to split up the world in chunks according to the number of available processors N. Generally there should be one master process, rank 0, that collects the calculation results

from the worker-nodes and combines them in order to prepare for the next simulation step and also potentially take care of some other functionalities like logging, visualization and so on. So the number of slices is $N-1$. If the world size is not divisible into even chunks, remaining, smallest chunk is allocated to rank 0. A graphical representation of this can be seen in Figure 3. This assures, given a close-to even distribution of agents in the world, that rank 0 finishes its calculations first and is ready to receive callbacks from the other nodes.
If we wanted to parallelize the sequential solution as implemented, for every simulation-step each process needed access to the current world-state in order to calculate updates for the individual world-parts. This is because the closest food-source for an agent may not be in the same slice as the agent itself. Also the agents that are inside of a certain part of the world change between simulation-steps as the individuals move requiring the full dictionaries of preys and hunters in order to have access to the current state of the agent and be able to make changes accordingly. While increasing wold sizes lead to bigger files that need to be broadcasted for every step, the bigger issue is the overhead resulting from pickeling python objects, i.e. the dictionaries, in order to be able to send them via MPI. To avoid this overhead, changes had to be made to the program.
The goal was to only use the (capital letter) Send and Receive functions that only work with NumPy arrays but are avoiding pickeling and hence the connected overhead and are therefore much faster than the lower-case functions. For this we changed the key-value pairs of the dictionaries to $((x, y), Prey)$ and for hunters respectively, identifying them by their position on in the world. This works as only one individual can be at a certain cell of the world-array at a given time. This change then allowed us to, instead of sending dictionaries back and forth in order to identify individual agents, to just use the positions that are already included in the world-state-array. This, however, also means that only the movement calculation can be divided between ranks and thereby be parallelized - updating the individual agents regarding their energy etc. as well as the world-state is left to rank 0 and remains sequential. As the calculation of movements was expected to be the most compute intense (most of the rest is mainly simple value-manipulation) it seemed to be an acceptable tradeoff.
For reporting back the results of the movement calculations we use a NumPy that contains the total number of changes, the old position of every agent in the respective slice for identification and the new position as the calculation result. Rank 0 then uses these arrays for updating the dictionaries (e.g. the age, energy etc. of the agents) and updating the world state including spawning offspring and setting the respective values in the world-array concurrent to the new positions of the agents. As the changes-array, due to MPI specifications, always has to have the same size, it includes the total number of changes in the first row in order to only run updates for meaningful rows of the array. Afterwards the new world-array is broadcasted to all nodes for the next calculation step.
In order to process the changes in the fastest way, rank 0 (after finishing own calculations if necessary) waits for the fastest process to finish and already applies the changes. This avoids unnecessary waiting times that would happen if one wanted to apply the changes arrays in any particular order.

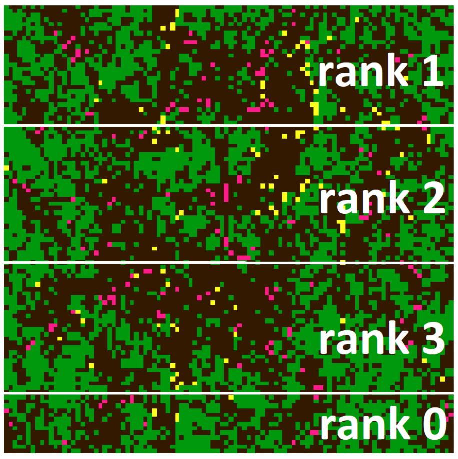


Figure 3: Visualization of the allocation of world-chunks to the different ranks of processes. Ranks one to 3 get even slices, the remainder is assigned to rank 0.

3.4 Memory

One important aspect regarding MPI and the program in general is the amount of memory needed to store all important information regarding the world as well as the agents in it. The array for storing the world state is a NumPy Array with dtype=unint8 as it is the lowest memory heavy option available and still allows for 255 different types of agents. The memory needed for a world of the size $n * n$ is hence expected to be approx. $8 * n * n$ bytes. Keeping the size of the world array as low as possible is key, as this is the array that is sent between nodes for every simulation step.
Additionally rank 0 needs memory space for the dictionaries. According to the getsizeof() function it seems that every entry takes up between 20 and 56Bytes of memory depending on the number of objects in the dictionary. For calculations we assumed an average of 40Bytes per entry. In the worst case (all the wold is filled with hunters/prey) this would result in $n * n * 40$ Bytes of memory.
Also memory is needed for the changes-array. It heavily depends on the world size, as also the dtype is dependent on it. As it stores positions of the agents, the dtype needed is calculated at the beginning of the simulation (see Figure 10). The size of the array is then $(world\_size * number\_of\_rows\_in\_one\_slice + 1) * 4$. The required memory is dependent on the dtype.

An example calculation for required memory can be found in Table 1. With increasing sizes of the world array as well as the changes array the problem of memory access speed comes into play. When updating the world, the CPU needs to access both arrays. With the arrays not fitting into Working Memory the calculation speed will drop significantly. Hence, the memory size that should be taken into account when scheduling a job on the HPC cluster.

World-Size	World-Array	Changes-Array	Dictionaries	Total
1000x1000	7,63	2,44	38,15	48,23

Table 1: Theoretical memory requirement for 51 nodes, worst case, in MB

4 How to run the Program

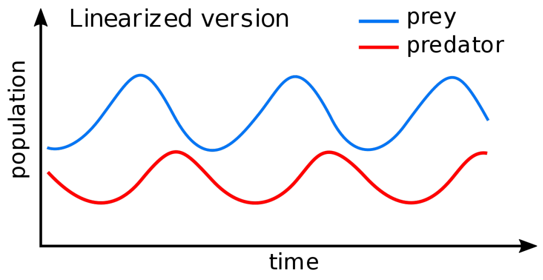
1. Change the simulation parameters in the config file to your likings
2. Change the filepath in the config file to the desired output directory (for logs and images). Make sure that you have writing access for the folder from the runtime environment. On the GWDG HPC cluster you could use
   $FILEPATH = f"/scratch/users/os.environ.get('USER')/$
   $simulation/os.environ.get('SLURM\_JOBID')"$
3. Run the program using MPI. On the Cluster you can use the slurm_job.sh script. Change the parameters such as they fulfill your needs. Keep in mind the memory your simulation will take up and make sure there is enough headroom for keeping the world-array and a change array in memory at the same time. Using the -d (--discardImages) flag, the program will not create and save images of the individual simulation-steps but only save population sizes in the .csv file. This speeds up runtime significantly and should be used when no graphical representation of the experiment is needed.

5 Evaluation and Benchmarking

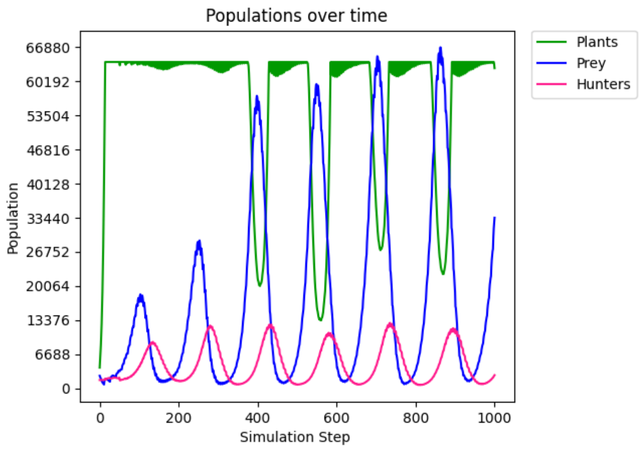
In this section we are going to see, our parallelized approach really works and then present the acceleration results because of the application of MPI.

5.1 Sideeffects

There are some sideeffects from parallelization regarding the behavior of agents.
One is, that the calculation of the next step of an agent is calculated only based on the world-state as it was at the beginning of a simulation step. Movement calculations that are already finished and would lead to a different behavior in a sequential approach are not taken into account. For example predator 1 and predator 2 are both one cell away from the same food source (prey 1). In a sequential approach, predator 1 would move towards it and eat it, thereby making the food source unavailable for predator 2. Predator 2 would then search for a different food source (e.g. prey 2). In a parallel approach, the calculations for predator 1 and 2 have the same outcome: move towards prey 1. When rank 0 then updates the world, it results in an invalid movement for predator 2 as it moves on top of a prey. This is a problem that, to our understanding, cannot be fully avoided when parallelizing a simulation where the action of every agent potentially affected by the behavior of every other agent and the changes that it leads to in the environment without sending back and forth considerable amount of information thereby diminishing the benefits of parallelizaiton. A possible solution that is not implemented in our version is sketched out in section 7.
Another sideeffect is that the "preferred" movement direction is downwards. This is a direct result of the implementation of the BFS algorithm that checks for the closest food source in a circle around the agent, beginning at the 6 o'clock position. It is important to be aware of that and randomize it, if the simulation would be used to analyze the behavior of individual agents. A more important aspect may be the movement order of



(a) Populations according to the first Lotka-Volterra rule [1].



(b) Populations in our simulation.

Figure 4: Development of populations over time.

preys and predators: currently hunters move first, then the prey moves. Also the order of movement of agents within the same type is based on their position in the world: the ones in the upper left corner of a slice move first, followed by the ones in the first row but farther to the right, continuing in this manner row by row within a slice.

5.2 Plausibility

We evaluated the correctness of the simulation by comparing it to the predictions implied by the first Lotka-Volterra rule. It states that, given a stable environment and enough food, the populations of predators and preys will fluctuate periodically and with a temporal shift. A simplified version of the resulting populations over time is shown in Figure 4a. In Figure 4b the populations of plants, prey and predators are shown from a run of our simulation. Comparing the two graphs one can clearly see the similarities regarding the populations of predators and prey: the population of predators is on average smaller than the one of the prey. Also the time shifted fluctuation of the populations can be seen. However, the maximum number of prey does not seem to be stable: it is rising in every population cycle.
Yet, overall the populations in our simulation seem to represent a plausible predator-prey relationship.

5.3 Benchmarking

We analyse the run time and evaluate strong and weak scaling behaviour and visualize the result. The benchmarking can be found in the branch "benchmark" of the provided GitLab repository.

In strong scaling, we keep the problem size the same; set the world size to 1000 and do the simulation step in 100 steps and turn the visualization off. Then we calculate the run time by increasing the number of processors. But we could not get results of sequential approach with this configuration, so we just try a lighter configuration with world size = 500 and do the simulation in 10 steps and it takes 47 minutes. By line

[1]https://en.wikipedia.org/wiki/Lotka%E2%80%93Volterra_equations

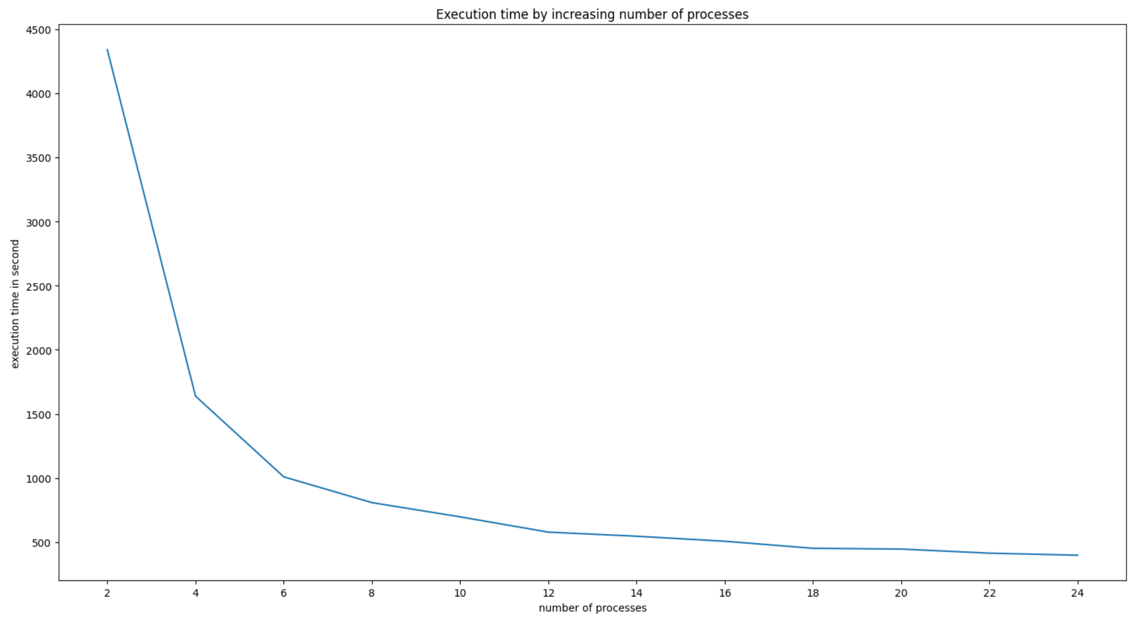


Figure 5: Strong-scaling. Run time performance by using the same configuration in different number of processors.

performance analysis (adding time function in the code), we find out, moving function in sequential approach is so time consuming. However, in parallel approach, as you can see in figure 5, at first with more processors, run time decreases sharply, but with higher number of processors, the run time decreases gradually. We can find out, maybe using 8 to 12 processors is enough for this configuration and by increasing that, we do not have significant performance improvement in every segment of our world.

Weak scaling is the ability of a parallel software to maintain efficiency while jointly increasing the problem size and parallelism. As you can see in figure 6, we calculate run time when we increase the number of processors and world size with the same scale.

To see how our software behaves in parallel execution, identify performance problems, and find sections for optimization and identify bottlenecks, we use code profiling. Code profiling is a method that is used to detect how long each function or line of code takes to run and how often it gets executed. For cpu profiling, we use the cProfile, which is a built-in python module. For visualizing the results of cProfile, we use gprof2dot[2], which is a python script to convert the output from cprofile into a colorful directed call graph that makes it easy to understand the statistics and you can use the command in 7 for getting result.

We also use Snakeviz[3], which is a viewer for python profiling data that runs as a web application in the browser and it gives detailed information about functions. As in our parallelization method, we just calculate new positions in other ranks and send it to rank 0, and then apply all of them in rank 0, so we mostly analyse the performance in rank 0. In figure 8, you can see the output of gprof2dot for rank 0 in different number of processors (4, 10, 20, 24). We can find out that with few number of processors, we wait most of the timefor receiving changes from other processors. But by increasing number of threads, waiting time (Recv time) decreases, in contrast, we spend more time to apply changes that we get from others.

[2]https://github.com/jrfonseca/gprof2dot
[3]https://jiffyclub.github.io/snakeviz/

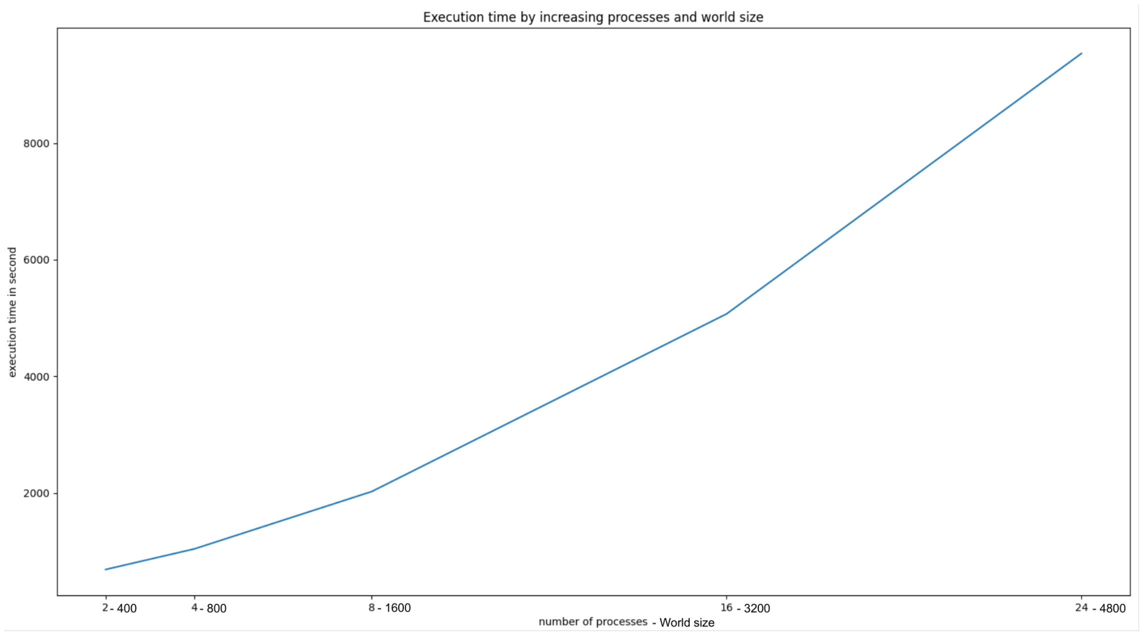


Figure 6: Weak-scaling. Run time performance by increasing number of processors and world size in with the same scale.

Visualizing by gprof2dot

```
gprof2dot -f pstats [.prof file] | dot -Tpng -o [output.png] && eog [output.png]
```

Figure 7: Command for get visualization result from profiling output

And also in figure 9, you can see the visualization by snakeviz. In here we visualize the profiling output of rank 0, when we use 24 threads. You can see that with snakeviz, we have detailed information like total number of calls to the function (ncalls), total time spent in the function (tottime), time per call that is just tottime devided by ncalls (percall), cumulative time spent in function and all its subfunctions (cumtime), cumtime devided by ncalls, and function definition at the end.

6 Discussion

During coding the sequential version, we have some problems regarding the visualization, like not showing plots and not saving images of simulation steps and after resolving them, we found that the moving function works really slow and did some optimization to that but did not gain significant improvement.

Before starting to parallelize, we invested a lot of time to find our way of parallelizing the problem. At first we had 2 options, parallelizing the world or the agents and at the end, we decided to segment our world. For doing that, at first, we tried a complicated solution, which can be found in the branch "parallelization" in GitLab repository. we tried to segment our world based on the number of available processors -1, and send information about objects of each segments to the responsible rack and their neighbours' ranks. Then do all the calculation and changes inside the rank and also handle moving

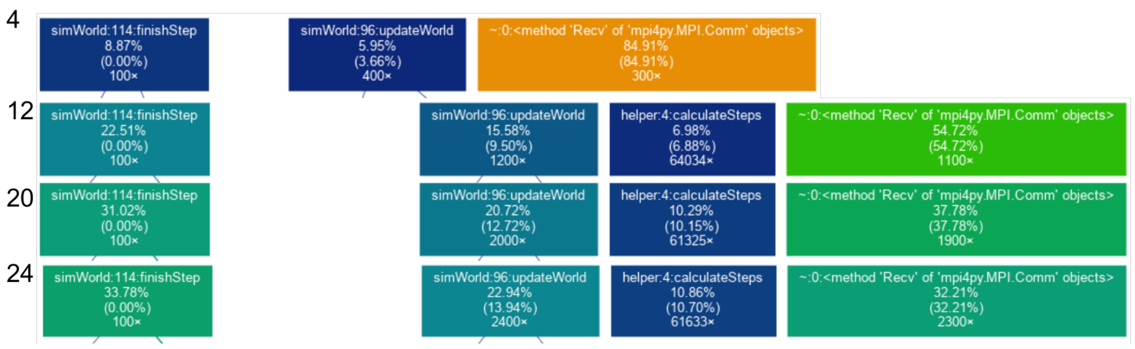
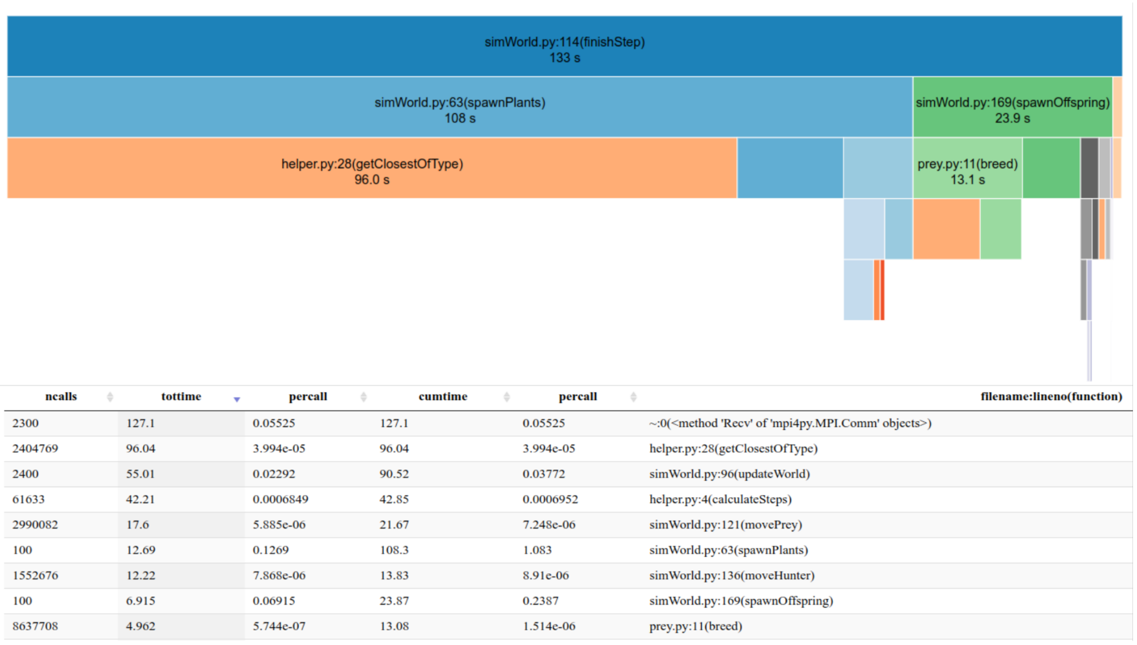

4
simWorld:114:finishStep 8.87% (0.00%) 100×
simWorld:96:updateWorld 5.95% (3.66%) 400×
~:0:<method 'Recv' of 'mpi4py.MPI.Comm' objects> 84.91% (84.91%) 300×
12
simWorld:114:finishStep 22.51% (0.00%) 100×
simWorld:96:updateWorld 15.58% (9.50%) 1200×
helper:4:calculateSteps 6.98% (6.88%) 64034×
~:0:<method 'Recv' of 'mpi4py.MPI.Comm' objects> 54.72% (54.72%) 1100×
20
simWorld:114:finishStep 31.02% (0.00%) 100×
simWorld:96:updateWorld 20.72% (12.72%) 2000×
helper:4:calculateSteps 10.29% (10.15%) 61325×
~:0:<method 'Recv' of 'mpi4py.MPI.Comm' objects> 37.78% (37.78%) 1900×
24
simWorld:114:finishStep 33.78% (0.00%) 100×
simWorld:96:updateWorld 22.94% (13.94%) 2400×
helper:4:calculateSteps 10.86% (10.70%) 61633×
~:0:<method 'Recv' of 'mpi4py.MPI.Comm' objects> 32.21% (32.21%) 2300×

Figure 8: Visualizing the output of cprofiler by gprof2dot. Profile output of rank 0 in different number of processors (4, 10, 20, 24).


simWorld.py:114(finishStep) 133 s
simWorld.py:63(spawnPlants) 108 s
simWorld.py:169(spawnOffspring) 23.9 s
helper.py:28(getClosestOfType) 96.0 s
prey.py:11(breed) 13.1 s

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
2300	127.1	0.05525	127.1	0.05525	~:0(<method 'Recv' of 'mpi4py.MPI.Comm' objects>)
2404769	96.04	3.994e-05	96.04	3.994e-05	helper.py:28(getClosestOfType)
2400	55.01	0.02292	90.52	0.03772	simWorld.py:96(updateWorld)
61633	42.21	0.0006849	42.85	0.0006952	helper.py:4(calculateSteps)
2990082	17.6	5.885e-06	21.67	7.248e-06	simWorld.py:121(movePrey)
100	12.69	0.1269	108.3	1.083	simWorld.py:63(spawnPlants)
1552676	12.22	7.868e-06	13.83	8.91e-06	simWorld.py:136(moveHunter)
100	6.915	0.06915	23.87	0.2387	simWorld.py:169(spawnOffspring)
8637708	4.962	5.744e-07	13.08	1.514e-06	prey.py:11(breed)

Figure 9: Visualizing the output of cprofiler by snakeviz. Profile output of rank 0 with 24 processors..

objects to another segments. We started implementation of this approach, but we faced with lots of problems like handling deadlocks which happened because of communication between ranks and complicated things like handling neighboring relations. So, we decided to switch to a simpler version that just has communication through rank 0 and described in section 3.3.

Assigning all the work regarding updating the world-array and the dictionaries to rank 0 is a significant downside and leads to the overall work only being partially parallelized. Yet, in our solution we use BFS in order to calculate the movements of agents which takes up most of the computational resources, justifying the decision.

The other sideeffects, mentioned in section 5.1, seem tolerable to us as they play a minor role in the overall behavior of agents as it. The comparison of the population development in our simulation compared to the one to be expected by the first Lotka-Volterra rule show that overall the populations have a valid progress.
However, there is some room for improvements, some of which will be described in the following section.

For benchmarking, we have lots of problems to find a suitable profiler. we initially tried VAMPIR and Score-P, and we tried to use this package of scorep[4], which allows tracing of python scripts using Score-P. It was easy to install but we could get results by following its instructions and the same thing happened, when we tried to use tau [5]. At the end we decided to use cProfiler for the analysis, and gprof2dot and snakeviz for visualizing. It does a decent job, but unfortunately it also has its own problems. For example, we could not have visualize and analyse all the ranks next to each other and compare them together. Also, we tried to do the memory profiling, but we could not find any tools for doing the memory profiling for python and support MPI, so we tried to add some lines inside the code to log the memory usage of each rank in different timestamps, but we could not get proper results that helps us to analyse memory usage of the program. Resulting plots show, in different timestamps, each rank uses same amount of memory and it does not change (straight line), but the memory usage of different ranks is different.

7 Ideas for improvement

There are some smaller improvements that could be implemented in our current solution without needing to change the functionality significantly. That includes for example:

1. Directly apply changes that result from agents' movement to the local world-array in the worker-nodes.
   This would eliminate the problem of a predator moving towards a foodsource that has already been claimed by another predator predator in the same simulation step (compare 5.1). It would be an improvement but is limited to agents that are within the same slice. For agents whose calculations are made by two different nodes the problem would persist. It would also be important to check for impacts on performance as the world-array would have to be loaded into memory.

2. Randomize the order in which hunters/prey move.
   This would make sense on the worker nodes if change 1 is implemented. Other-

[4]https://github.com/score-p/scorep_binding_python
[5]https://www.cs.uoregon.edu/research/tau/home.php

wise the order could be randomized when rank 0 processes the change-array. It would make the simulation more close to nature and "soften" the results of time quantization into simulation-steps.

3. Don't transform the world-array into images but save the arrays as such.
   This should be less computational overhead and hence a speedup when you not only need overall population data but also information about the distribution of agents in the world for every simulation-step. The computational resources needed for creating images from the arrays could be reserved either at the same time but on a dedicated node with a different process taking care of it (process the data in-time) or be requested at a completely different time and possibly only for some parts of the data.

4. Don't send the number of changes in the changes-array but send some "EndOfFile" code like 2 succeeding rows containing only zeros and check for that in the update function.
   By avoiding having to store the overall number of changes within a slice which is in the worst case $WORLD\_SIZE * num\_rows$, the dtype of the changes-array would be only dependent on the $WORLD\_SIZE$. In some cases that could lead to much smaller (memory wise) arrays that have to be sent between nodes.

For possibly conflicting movements (the ones that are at the borders of slices) the concerning nodes could calculate the movements of agents at the borders first, broadcast them and then calculate the rest. E.g. rank 2 calculates the changes at the border to rank 3, sends the updates to rank 3 and then the calculation of the rest starts. For this, rank 2 also has to calculate the changes on the side of the border that rank 3 is normally responsible for.

We also had an idea for avoiding the need to broadcast the whole world array. Hence it would lead to less memory required for worker nodes. For this, we would assign a "field of view" to agents. Only in this range they would be able to locate a food source and move actively towards it. Otherwise they would move randomly. We would then only need to send a slice of the world to a respective node that consists of the "original slice" that they work on plus a number of additional rows to the top and bottom of the slice that are equal to the largest field of view of any agent. The individual nodes besides rank 0 would then only have to keep the smaller chunk of the world in memory.

Another, completely different idea would be to not separate the world into chunks but assign every agent to a node that is then responsible for its movement regardless of the position in the world. This is a completely different approach to parallelization than the one we chose and would involve a lot more communication between individual nodes in order to resolve conflicts etc. However, it might be a more efficient one when the fraction of computational load for calculating movements gets smaller than it currently is (for example when introducing a field of view or machine-learning algorithms).

8 Conclusion

In this project a prey-predator model was implemented in sequential and multi-processing ways. For multi-processing approach, We implemented a simple version of parallelization that communications happened through one single rank, and it is a high-performance solution that based on benchmarking results, it outperforms sequential approach and also works correctly and represent a plausible predator-prey relationship when comparing it to the predictions of Lotka-Volterra rule.

In our opinion, the project was successful and we learned a lot about process and usage of MPI and different performance measurement methods and tools. We learned a lot about the importance of parallelizing and based on the performance analysis, we saw how it can exponentially increase the performance of the programs that we implemented. On the other hand, we had a lot of challenges with parallelizing and had to work a lot more on that to resolve them and still, we think that it has a lot to improve, but the process and the code is so simple to catch and find what is going on, and it is wonderful that this less complicated solution, can improve performance significantly.

A Work sharing

If you worked in a group, describe here how you distributed the work and the actual contributions of each peer.

A.1 Vincent Hasse

- Implementation of the sequential approach
- Parallelization of the sequential approach with the changes mentioned in the report
- Sections 2, 3, 4, 5.1 as well as parts of 1, 6, 7 and 8

A.2 Kimia Taba

- Implementation of the sequential approach
- Parallelization of the sequential approach in complex way as was described in section 6
- Benchmarking
- In writing reports: Section 5, 6 and parts of 1, 7 and 8.

B Source Code

The complete source code can be found on:
`https://gitlab.gwdg.de/hpc-practical/hpc-simulation`

C Additional Figures, Code etc.

Calculating the datatype of the changes array

```
# smallest possible datatype for changes array
if config.WORLD_SIZE * numRows > 255:
    changes_dtype = np.uint16
    if config.WORLD_SIZE * numRows > 65535:
        changes_dtype = np.uint32
        if config.WORLD_SIZE * numRows > 4294967295:
            changes_dtype = np.uint64
else: changes_dtype = np.uint8
```

Figure 10: Calculation of the datatype of the changes-array based on the simulation parameters.