GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN IN PUBLICA COMMODA SEIT 1737

HPS

Seminar Report

Parallel Animals

Louis von Leitner & Thomas Hay

MatrNr: 26337291 & 21769229

Supervisor: Martin Paleico

Georg-August-Universität Göttingen
Institute of Computer Science

September 23, 2025

Abstract

Wildlife density estimation with camera traps increasingly relies on simulation to compare methods applicable to unmarked populations, yet realistic, large-scale simulations are computationally demanding. We address the need for a fast, ecologically grounded simulator that can support replicated experiments required for method benchmarking and sensitivity analyses. Existing comparisons evaluate only subsets of estimators or use simplified movement models, and prior implementations are too slow to generate statistically robust data at scale. This limits rigorous, side-by-side assessment of density estimation framework under common comparison designs. We develop an object-oriented Python simulator that couples a correlated random walk parameterized from roe deer GPS data with a detailed camera-trapping module. Sparse spatial indexing reduces unnecessary geometry calculations, and the output supports the application multiple estimators (REM is directly applied). To overcome serial bottlenecks, we design and implement several HPC parallelization schemes: two agent-based variants and a family of master–worker designs, including an optimized version that overlaps work (precomputing step candidates, splitting relocation updates) and a "God photographer" to offload rare, expensive photo-taking. Performance was profiled with Score-P/Vampir and evaluated on the GWDG cluster. The serial configuration (100 animals, 2000 steps, 30 cameras) averages a model runtime of 3696 seconds ($\sim$1 hour); profiling shows $\sim$70% of time is spent in step location generation and selection (with $\sim$97% of that in weight computation) and $\sim$30% in detection. Agent-based parallelization scales nearly linearly up to one node but saturates due to integer division of 100 animals; master–worker optimized exhibits the best strong scaling on-node. The fastest run reached 18.13 s with 297 processes ($\approx$ 204x speedup), with scaling presumably limited by inter-node communication and variability from rare photo events. These results enable large replicated studies and provide a practical foundation for comprehensive, fair comparisons of camera-trap density estimators.

Declaration on the use of ChatGPT and comparable tools in the context of examinations

In this work I have used ChatGPT or another AI as follows:

- ☐ Not at all
- ☐ During brainstorming
- ☐ When creating the outline
- ☑ To write individual passages, altogether to the extent of 5% of the entire text
- ☐ For the development of software source texts
- ☐ For optimizing or restructuring software source texts
- ☐ For proofreading or optimizing
- ☑ Further, namely: Debugging code and Compute Cluster errors -

I hereby declare that I have stated all uses completely.
Missing or incorrect information will be considered as an attempt to cheat.

Contents

List of Tables

List of Figures

List of Listings

List of Abbreviations

HPC High-Performance Computing

1 Introduction

Adaptive management of wildlife is essential to enable purposeful land management that simultaneously can meet the requirement of maintaining desirable wildlife populations (Walters, 1986). In Germany alone, estimates of damage caused by excessive game populations lie in the three-digit million euro range (Clasen and Knoke, 2013), while some wildlife conservationists and hunters are calling for high game densities (Ammer et al., 2010). Objective recording of game populations is essential for conflict resolution between the various parties and is as such a prerequisite for adaptive management (Marcon et al., 2019). Classically, abundance of game species is indirectly derived from the relationships between different proxy indicators, such as surveys of browsed plants (*regeneration inventories*) and hunting bags (amount of harvest) (Bödeker et al., 2021). Although these methods can help to quantify game damage, they are associated with considerable costs (Bödeker et al., 2021). Furthermore, they can only assess the effects of management actions on wildlife density with a time lag and are their accuracy and precision can be difficult to quantify (Bödeker et al., 2021).

This work will focus on abundance and density estimation via camera traps using the European roe deer *Capreolus capreolus* as an example, because it is a highly abundant and well-studied species contributing to game damage in central Europe (Ammer et al., 2010). Common methods directly aimed at quantifying population sizes of deer include sign- or direct counts by human observers besides harvest data (Forsyth et al., 2022). However, resulting accuracy and precision of these methods is rarely even assessed (Forsyth et al., 2022). Therefore, their usefulness in the context of adaptive management can be questioned. Camera traps, on the other hand, could enable an objective recording of game densities and thus contribute to conflict regulation (Marcon et al., 2019).

In general, density estimation methods using camera traps can be split into two groups: Methods aimed at (at least partially) marked species and those methods aimed at unmarked (not individually identifiable) species. Under the methods for marked animals, estimators from the spatial capture-recapture class (Royle et al., 2014) have emerged as the gold standard with regard to accuracy and precision (Palencia et al., 2021). However, roe deer as the target taxon of this work is an example of an unmarked species. Methods for unmarked animals are comparatively new but are highly relevant for ecologists as many species are generally unmarked and artificial marking can be both invasive and expensive. To date, there is no established favourite method amongst practitioners (Gilbert et al., 2021, Palencia et al., 2021). Gilbert et al., 2021 list all frameworks that have been proposed to estimate abundance or density of unmarked populations: Unmarked Spatial Capture-Recapture (USCR, Royle et al., 2014), the N-mixture model (Royle, 2004), the Random Encounter Model (REM, Rowcliffe et al., 2008), the Random Encounter and Staying Time model (REST, Nakashima, Fukasawa, and Samejima, 2018), camera-trap Distance Sampling (DS, Howe et al., 2017), as well as Time-to-Event (TTE), Space-to-Event (STE), and Instantaneous Sampling (IS, all by Moeller, Lukacs, and Horne, 2018).

These methods have been applied to very different degrees, with older methods such as the N-mixture model being generally more popular with practitioners and newer methods such as STE seeing use only very recently (Gilbert et al., 2021, Lyet et al., 2023). Empirical studies comparing the performance of different estimators (Palencia et al., 2021, Doran-Myers, 2018, Anile et al., 2014) in the past were only concerned with small subsets of about three to four estimators out of the seven options identified by Gilbert et al., 2021

above. This is also true for the most comprehensive simulation-based comparison to date by Santini et al., 2022.

Therefore, there is need for a thorough analysis of all available methods within one comparison framework in order to identify the most promising estimators and compare newer methods like STE with the more established older ones. This work tried to close this gap by expanding upon the work of Santini et al., 2022 and guidance by Gilbert et al., 2021, who called for exactly such a comparison framework in their method review. In this work, we will focus on the development of a simulation model for the animal movement and camera trapping process. This model will output data to which the different abundance or density estimation methods can be applied to. For the purposes of this report, we limited ourselves to the direct application of the REM model to generate a meaningful output, as it can be applied without the need of further in-depth statistical inference on the model outputs. However, our model also outputs data to which the *distance sampling* estimator can be applied later, which significantly increased the complexity of the data generation process. In principle, our model allows for a comprehensive comparison of all mentioned estimators.

Such a comparison requires a simulation-based rather than an empirical approach, because some of the estimators are incompatible to each other with regard to their required sampling designs. This means that an empirical study would have to set up multiple camera trap arrays and/or employ multiple camera trigger modes (motion-triggered, timelapse) simultaneously to measure the data needed, which would not only be very expensive but also technically difficult. Besides this, the true density of real populations is rarely known, which is obviously of no concern in a simulation environment. Therefore, a simulation-based approach allows for ranking of estimators against an objective standard that is the true density.

In contrast to Santini et al., 2022, the animal movement model was implemented in Python using an object-oriented programming style. Our model could account for more ecological realism compared to the one by Santini et al., 2022, as it it is designed to reproduce movement patterns observable in the real world. The model was parameterized by means of analyzing GPS tracking data from a roe deer population in southern Germany (this analysis is not contained in this report). Overall, the structure of the modeling approach was chosen to avoid violations of estimator assumptions, which are nicely summarized in multiple sources besides the original papers introducing the methods (Gilbert et al., 2021, Doran-Myers, 2018).

Existing simulation studies to date have focused on optimal sampling design or assumption violations (Santini et al., 2022, Howe et al., 2019, Nakashima, Fukasawa, and Samejima, 2018). We designed our movement model to facilitate investigations of either one of these analysis pathways, or even both at the same time through flexible parameterizations

2 Methods

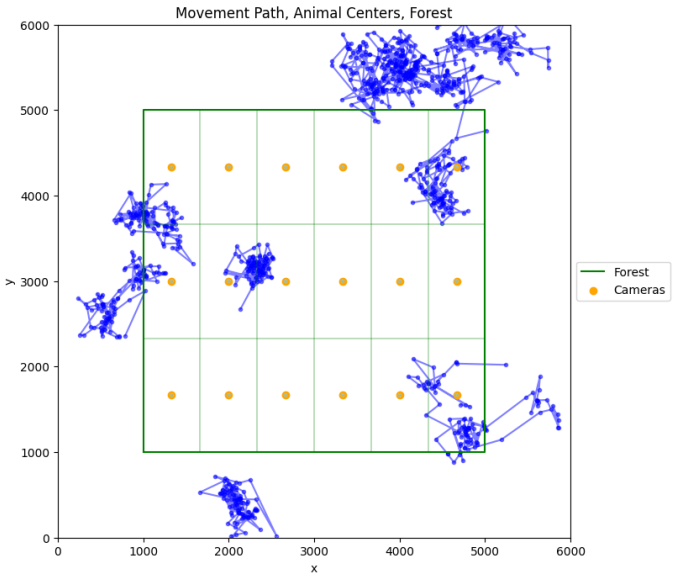

Figure 1: Simulation of 100 steps of 100 animals. Movement paths of animals in blue, forest border in green, forest sectors in light green, cameras in yellow.

2.1 Solution approach and Objective

The general solution approach can be split into two sub-problems containing the main features of the overall assignment: Firstly, animal movement (that is an ordered series of planar coordinates) has to be generated in a way that is parameterizable using real-world GPS tracking data. Secondly, the camera trapping process must be accounted for. This means identifying that a relocation between two points intersected a camera view shed (which has the geometry of a circle sector). If this happens, more nuanced calculations need to be carried out in order to generate the actual model output. This output is a table of "photos", mainly containing a simulation timestamp, the distance of the animal to the camera, and the total time the animal spent on this occasion in the detection view shed. This data can be used to get results from all above mentioned estimators. However, photo generation actually happens very rarely (on average, ca. 1 photo for every 200-400 relocations).

We call a simulation run the execution of this procedure for defined numbers of animals, cameras, and animal relocation steps. In order to estimate statistical uncertainty of density estimators or to conduct sensitivity analysis or optimization of input parameters of the model, many repeated simulation runs will be necessary when actually using our model to answer a real-world question.

Therefore, our goal is to reduce the runtime of one simulation run as much as possible. We defined 2000 steps of 100 animals with 30 cameras to be a reasonable model configuration based on existing studies (Santini et al., 2022, Rowcliffe et al., 2008) and discussions with practitioners.

2.2 Sequential implementation design

2.2.1 Model initialization

At the beginning of every model run, some things need to be initialized before the actual processes (animal movement and camera trapping) can start:

- The *World* and *Forest* are set up. These are a set of nested rectangles (the forest lies inside the world). The purpose of the World is to provide an area where animal home ranges can occur. The forest is the actual "study site" where the camera gird tries to estimate the animal density. Such a two-stage setup is required to allow animals to move across the forest borders. If they were trapped inside the forest (world = forest), this would result in different movement patterns of animals whose home ranges lie closer to the forest border compared to those in the forest center. This is undesirable, as it could lead to a heterogeneous density pattern, which can be difficult to account for during density estimation.

- Animal start positions and home range centers are set up. The start position of every animal is chosen randomly inside the world, i.e. according to a homogeneous Poisson point process. Then, three home range centers are found for every animal around this start location. Home range centers have a uniformly distributed distance between the overall centroid (animal start position) between a minimum and maximum value defined via the GPS tracking analysis and a uniformly distributed direction angle. Overall, the animal initialization can thus be seen as a form of Thomas cluster process. Home range centers serve the purpose of acting as gravitational centers of the animal movement, which is a form of correlated random walk.

2.2.2 Process scheduling

During the model run, two processes are continuously executed after another a predefined number of times. Firstly, animals "move" one step (a new coordinate is found). Then, this relocation from old to new position is evaluated for whether it contributes to the model output, i.e. whether a photo is taken. These two steps happen for every animal one by one. The design of the program is thus serial in two different ways: On the animal level, as the program is essentially cloned for every animal instance, and on the process level, as the per-animal program is serial as well. Starting form this observation, we will present parallelization schemes based on either of these aspects later.

2.2.3 Animal movement

A singular animal relocation is realized in the following way: First, 100 values for the length of the relocation are drawn from an exponential distribution. Using these 100 step lengths and a uniformly distributed direction angle, 100 potential target coordinates around the current position are found. Coordinates are calculated via:

$$x_{target} = x_{current} + sin(\theta) \cdot d \tag{1}$$

and

$$y_{target} = y_{current} + cos(\theta) \cdot d, \tag{2}$$

where x_{target} and y_{target} are the coordinates of the potential target, $x_{current}$ and $y_{current}$ are the momentary x and y coordinates of the agent, θ is the direction angle towards the target against north (drawn from a uniform distribution), and d is the distance to the target (drawn from the exponential step length distribution).

For each potential target, the distance between this target and the nearest home range center of the respective animal to the target is determined. This distance is then transformed to a selection weight via:

$$w = a^{-1} \cdot e^{d \cdot b}, \tag{3}$$

where w is the selection weight, d is the distance between potential target and nearest home range center, and a and b are empirical parameters that were estimated during GPS data analysis.

Finally, one of the targets is selected as the new current position based on the selection weights. All parameters of the movement model were calculated in such a way that one step represents 2 hours in real time, because this was the acquisition interval of the GPS tracking data used for parameterization.

2.2.4 Camera trapping

Sparse modeling
We realized early that the camera trapping process, while triggered rarely, is computationally costly due to the necessity of many vector geometry calculations. Therefore, we employed sparse modeling techniques to reduce the amount of unnecessary calculations as much as possible. The goal was to always identify as early as possible if the camera trapping process can be omitted or stopped. As a first part of this approach, we subdivided the forest into a grid, where every grid cell contains a camera and therefore represents the area a camera is "responsible for". As part of the relocation process, we track for every animal the grid cells the relocation crossed. This allows to narrow down the number of cameras for which an intersection of the relocation with their view sheds has to be checked. It also allows to identify cases where no further checks are necessary (such as relocations happening completely outside of the forest), in which the program can move on to the next step immediately.

Photography
For relocations that could feasibly have crossed a camera view shed, the intersection between the view shed geometry and the relocation line is assessed. This is not completely trivial, because the view shed is a circle sector and all possible relocations need to be accounted for (such as relocations approaching the camera straight on into the "curved" part of the view shed and ending inside the view shed). If an intersection exists, Points along the part of the relocation that lies inside the view shed are generated, spaced one second apart from each other. Cameras used in real-life studies commonly have a cool-down time of 8-10 seconds, so photos can happen maximally on this interval. This is why we decided that the 1 second spacing would be accurate enough. Starting from the first point on the line inside the view shed, following the movement direction of the relocation, a so-called detection function is applied to the distance between the point and the camera. This detection function is a half-normal probability density function with a standard deviation of 5m. This type of function is common for modeling the trigger behavior of the camera motion sensor (Howe et al., 2017), where movement further away

from the camera is significantly less likely to trigger it compared to movement close to the sensor. If the camera is triggered according to this function, a "photo" is generated, i.e. the corresponding information is collected in the output photo table. Then, the "cooldown timer" is activated, i.e. only the 8th point after the one that was just assessed will be tested next. Otherwise, we proceed with the immediately following point.

2.3 Parallel implementation designs

The reader is advised to read about the performance of the serial program in subsection 3.1 before continuing to read about parallelization methods.

As described in the runtime analysis subsection 3.1, the simulation works by doing a sequential execution of `add_relocation`. So, parallelization can speed up the program in two different ways. Either, by running sets of iterations in parallel or by decreasing the time needed for one execution of `add_relocation`.

When running sets of iterations in parallel, there is an important dependence between steps that cannot be violated in order to preserve simulation accuracy: Naturally, to compute the next location in `add_relocation`, the previous location must be known. In the context of our simulation model, this means that it is not possible to compute later steps of an agent before finishing the computation of the previous ones. For example, computation of the steps 30-60 of an animal before knowing the steps 1-30 of the same animal is not possible. Therefore, one cannot run sets of iterations of the same agent in parallel, as one set of iterations will depend on the other and must wait for the computation of those to finish, before it starts its own computations.

A similar dependence can be found in the parallelization of the function `add_relocation` itself. One cannot do `detection` before knowing the new location and thus the relocation path of an agent. A more detailed breakdown will be provided in the respective parallelization approaches subsubsection 2.3.3, subsubsection 2.3.5.

2.3.1 Agent-based parallelization v1

Fundamentally, this parallelization scheme is a very simple master-worker setup, however, there is no practical difference between the master and the workers. The core idea is to leverage the fact that there is no interdependence or interaction between animals during runtime, which is why the simulation can be split up on a per-agent/animal basis. Each worker simply computes his share of animals. The master only defines the number of animals each worker and he himself should compute. After all movement model computations finished, the master only has to collect all the photos from the workers and apply the estimator. We implemented two variants of this idea. They differ in the way how the work sharing is distributed, resulting in different locations of model initialization.

In this first version, every process/rank initializes its own instance of the model. The master rank only specifies how many animals should be contained in each specific model instance to match the overall number of animals specified by the model scenario. Every worker and the master carry out all the (serial) calculations for all of their respective animals independently. In the end, the master rank collects the individual model outputs from his workers in his own model instance in order to apply the density estimation method.

2.3.2 Agent-based parallelization v2

Note that in the overall performance comparison later, the "Simple parallel agents" approach refers to the first one (v1), not this one (v2). We abandoned the v2 approach as we found no real difference in runtime between the v1 and v2 during testing. Still, it contributed to some of the later ideas so it is presented here as well.

The core change compared to the previous one in this second variant of an agent-based parallelization scheme, the Model object is only instantiated once. This has the advantage that further development of the animal model itself can be separated better from the parallelization aspect. Such developments could include e.g. interactions between animal home range centroids during model setup (representing e.g. territoriality of certain individuals or groups). An instantiation of a Model object would thus contain a unique spatial arrangement of animals which depends on the number of animals instantiated.

This parallelization scheme contains three hierarchically ordered types of threads, again forming a cascading master-worker type setup. The `God` thread initializes the model and passes it on towards the `Master` threads. `Masters` form `Workgroups` of equal sizes together with their respective `Slaves`. Each work group is responsible for computing the model for a subset of animals. Animals are mapped to their computing ranks/work teams through their respective indexes. This means that only specific combinations of number of available ranks and number of animals to compute are supported by this design. `Masters` determine which of their slaves should compute the model for which animals by modifying the received model instance and pass the model on to them. `Slaves` only have to receive the model, execute its `let_them_run()` method and create the outputs. In the end, the God collects the outputs from all other ranks and does the final estimation.

This three-stage design may appear unnecessarily complex. We hoped to speed up the model distribution among the ranks using a cascading structure. Also, we had plans to let the God assist the masters or slaves with some specific calculations. In the described setup, he is idle while the model is being computed by all other ranks. These plans were not realized in the end.

Both of the agent-based parallelization designs are conceptually very basic, as they are a simple top-down distribution and collection scheme. No communication between ranks during the actual calculation is needed. Given the agent-based nature of the original modeling problem, these designs come to mind very intuitively. We included them in the project as a from of reference version to compare the more complex designs to.

2.3.3 Master Worker Parallelization

The idea for this parallelization approach was born through the analysis of the performance bottlenecks of the sequential solution subsection 3.1. As described, about two thirds of the iteration runtime is spent computing weights, while the rest is primarily spent computing camera detection. The underlying assumption already stated in subsection 3.1 is that reducing iteration runtime reduces total runtime because the core of the simulation is calling iterations sequentially a lot of times. Potentially, by speeding up the iteration by 1 time unit could speed up the whole program by 2.000.000 time units, because the iteration is called 2000 times per animal for 100 animals.
The core idea behind the Master Worker Parallelization approach is to assign the processes running the program into work groups with so called *Workers* and one work leader called the *Master*. The *Master* distributes work to the *Workers*, and while they complete

it, does some other work. Once the *Workers* have finished their work, the *Master* gathers the results, gives out new work and processes the results. To maximize efficiency, the time *Workers* or the *Master* spend waiting for the other processes to finish should be minimized so that all processes are working at all times. This describes a state of full capacity where all of the compute power is used at all times and thus runtime is minimized. Similar to the Agent-based parallelization methods, the number of steps to be computed per work group is assigned at the beginning of the program by distributing animals equally onto work groups.

A practical starting point for this approach is to divide the processes into work groups of 3, such that one *Master* works together with two *Workers*. The *Master* generates the possible steps and distributes them equally among the *Workers*. While they compute the weights, the *Master* does `detection`. This way, in theory as explained above, the two *Workers* do about two thirds of the work while the one *Master* does one third. Thus, efficiency would be maximized.

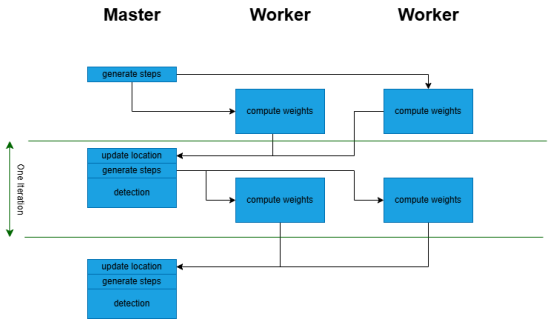

Figure 2: Flow Chart of Master Worker Parallelization Logic

2.3.4 Master Worker Optimized

The Master Worker Optimized Parallelization Approach is closely related to the original Master Worker approach subsubsection 2.3.3. The idea for the optimized version came from looking at a Master Worker Parallelization program run in Vampir subsubsection 3.2.3. To understand the following descriptions and reasoning, it is important to read subsubsection 3.2.3 and specifically have Figure 10 and its description in mind.

To determine speedup possibilities of the *Master*'s work in Figure 10, it is important to be aware of dependencies of processes on each other. The `get_step` function requires knowledge about the weights of the generated points. The `add_relocation` depends on the next location determined during `get_step`. The `generate_possible_steps` needs the new location of the animal to generate 100 possible locations around it. However, there are two key ideas to reducing the *Master*'s work in this section.

(i) All the work in `add_relocation` could be done alongside the `detection` while the *Workers* compute weights. This is except for the updating of the animal's location because the location is necessary for `generate_possible_steps`. Thus, one could divide `add_relocation` into the location update, which is to be done after choosing the next step and the rest, which is to be done while *Workers* compute weights.

(ii) `generate_possible_steps` generates 100 possible locations around the animal's location based on probability distributions. The probability distributions themselves are however independent of the animal's position. Therefore, one could divide the `generate_possible_steps` into two sub parts: (a) generating values from probability distributions to generate 100 points around the origin of the coordinate system and (b) adding the animal's location to those values. This way, (a) would be independent of the animal's location and could be precomputed by the *Master* during

the weight computation of the *Workers*. Then, while the *Workers* wait, the *Master* only has to add the animal's location to the generated points. The end result is the same, but compute during the *Worker*'s wait is decreased.

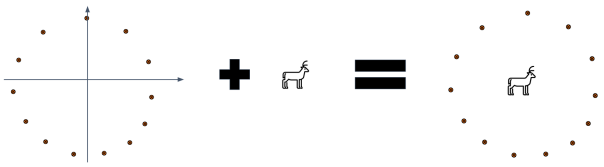

Figure 3: Possible Step Generation broken up into two parts

Implementing this marks the transition in approaches from Master Worker to Master Worker Optimized. The process flow chart has become a bit more complex and less intuitive.

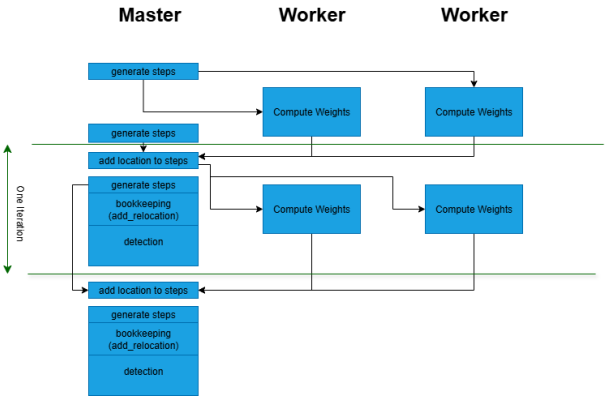

Figure 4: Flow Chart of Master Worker Optimized Parallelization Logic

2.3.5 Master Worker God Photographer

This parallelization approach was invented to counter the long wait times of *Workers* in the Master Worker Optimized approach, which occur when the *Master* does necessary computations for taking photos. It is to be noted that this photo taking happens very rarely, but keeps all the *Workers* idle while waiting for quite some time as can be seen in Figure 13.
To counteract the idling of the *Workers*, a so called *God* process will be designated, which is just responsible for taking pictures. It could be seen as the overlooker and photographer of the whole Program. *Masters* still check if a photo has to be taken to some extent and if the probability of a photo becomes high, they give all the necessary information to the *God* and continue with the next iteration step. The *God* takes over the photo taking process while the work groups can continue computing steps.
In case there are multiple cases of likely photos around the same time, the *God* keeps track of work requests in a queue.
Naturally, this approach only makes sense if there are a lot of work groups, otherwise the *God* will be idle for most of the time, possibly leading to more idle time among processes than without the *God*.
Of course, one could determine the optimal amount of work groups for one photographer, but as we have usually only worked with less than 100 processes and pictures are very unlikely, one photographer will be sufficient. The optimal ratio of *Workers* per *Master* should be the same as in the Master Worker Optimized approach, as most of the step iterations happen without photos and we optimized the work group size over all steps.

2.3.6 Improvement expectations

We expected runtime of the agent-based parallelization to decrease linearly with the number of available computation processes, up to the point where there is one process per agent. Beyond this point, adding further processes is meaningless as the design does not support to offload work onto these resources. However, in the area where speedup is possible, we expect the method to utilize the resources rather well, since overhead is minimal and processes do not have to wait for each other at any point apart from the very end. This is why a slight efficiency loss with high number of processes might be possible (such as one process per animal plus master), as the master does the final stretch of work on his own while many processes are idle.

For the more complex parallelization schemes, it was difficult for us to formulate a hypothesis for their performance. It was clear that some parts of the more elaborate designs would require optimization of some of their parameters (such as the number of workers per master for weight computation) in order to minimize the time that every process spent idle. In general, we expected (or rather hoped) for a potentially weaker performance gain for numbers of processes up the point that is still supported by the simple agent parallelization due to the larger overhead and increased communication. The main point of these designs is however to enable the use of significantly more processes than the simple agents can allow for. This is why we expected significantly shorter possible overall run times, while be it slower at the same number of processes relative to the simple agents.

2.4 Performance analysis setup

For comparability of different methods it is important to use the same environment and setup for result gathering for each method.
The computations were done on the scientific compute cluster of the GWDG at Göttingen.

We analyzed the performance of the sequential implementation by timing the execution of the various methods with Python's built-in timing functions.

For each parallelization approach, a program execution was done with the minimium amount of processes up to 100 processes at one time. Those processes were distributed among the fewest nodes possible. Nodes on the GWDG cluster have 96 processes with 2 CPUs each, so runs with less than 97 processes were executed on just one node. For each of those configurations, three runs were executed and the average runtime of the runs calculated to countersteer external factors interfering with results. Due to number-of-processes constraints given by some approaches, for example the divisibility of the number of processes by a given integer, not all approaches were done with exactly the same number of processes. Therefore the speedup graphs are to be considered for general speedup curve geometry, but not exact runtime method-to-method comparison in specific data points. The speedup itself is calculated by the formula Equation 4 further described in the discussion subsubsection 3.3.1.
Furthermore, for the more complex parallelization methods, in order to get a more detailed picture of what is going on, the Vampir Performance Analysis Tool was used. To use this tool, a Score-P trace file was generated using the Score-P binding for python *Score-P binding for Python* 2025. This allows for visual analysis of processes and their work at all times during the run.

2.5 Implementation

2.5.1 Sequential

The code is structured around object-oriented principles, with classes representing different entities such as animals, forests, cameras, and the world. We made heavy use of numpy and pandas, numpy for all the vector math and pandas for the input/output management.

The Animal class represents an individual animal with attributes such as sex, movement parameters, and home range centers. The class contains methods for realizing animal movement such as for center and step generation, and most importantly `add_relocation`, where most of the program happens. World and forest are contained in the animal class as instance variables, which makes an animal instance fully independent in terms of "its" computations. This allowed for simple parallelization schemes later, where it was sufficient to e.g. distribute the animals among computation ranks.

The Forest class represents the forest environment, including its boundaries and holds the camera grid and forest sector grid. The world is also an argument to the forest (it is constructed relative to the forest, because the forest is the main area of interest), as such the forest can be seen as the main organizational structure for everything that is not the animals. The forest class provides methods for determining which forest sector / potential camera an animal is in and which camera is responsible for a given point.

The Camera Class represents a camera on the camera grid inside the forest, including its position, detection zone, and detection function. The camera class provides methods for determining whether an animal is in its detection zone and for taking photos of the animal along the relocation line.

There exist further classes completing this hierarchical structure, such as the World, ForestSector, and CameraGrid class. They mainly provide instance variables for a consistent naming scheme and helper methods, e.g. for generating unique IDs.

The Model Class is the top-level class that orchestrates the simulation of multiple animals. It provides methods for "users" to interact with the model, allowing to initialize a simulation run, execute it, and gather the outputs. The model class creates instances of the world, forest, and animals depending on the scenario specifications. The `let_them_run` method sequentially accesses the different animals and executes their movement and camera trapping processes. After the simulation is executed, the Model class can collect the photo data from the different animals and apply the *random encounter model* to in in order to generate a meaningful model output. Since the number of animals generated is also known to the class, the estimate can also be compared to the true value.

The script `run_model.py` demonstrates this functionality of the Model class and is the main entry point to the project.

2.5.2 Parallel

All parallel implementations rely on the `mpi4py` package (Rogowski et al., 2023). In the simple agents variant, all communication is handled over `COMM_WORLD`. Blocking send/recv. and gather calls are used to distribute the work and collection of the results via simple python object pickling. The top-level model class remained untouched, the entire parallelization happens from "the outside".

The more advanced master-worker schemes also required changes to this class. In the top-level `run_model` script, the fixed-size MPI teams are setup and, animal counts and

full Model objects are distributed to team members via pickling; it later gathers outputs at root. The `Model` now also implements a per-team master–worker pipeline inside the `Model.let_them_run` method that splits 100 candidate steps among workers, uses NumPy buffers with Isend/Recv to exchange candidate steps and weight blocks, and keeps all state updates on the master. This critical inner loop uses buffer-based MPI (NumPy arrays) and separates the heavy weight-computation across workers while the masters keep animal state updates and geometry/detection locally. Precomputing of "direction vectors" (`possible_steps_next`) separates location-independent and -dependent work, which allows some overlap between candidate generation and weight computation.

The final "God as photographer" design keeps these main features from the initial master-worker scheme. In addition, the God now also does some centralized event processing. He runs a loop: while `done_counter < n_workgroups`, he `comm.recv(source=MPI.ANY_SOURCE, tag=42)`. The events can be:

- Data tuples for a relocation, which God applies by calling `model.animals[animal_number].data_collection(...)` (this can include camera detection and time accounting).
- The string `'all_done'` signaling a group finished all its animals; increments `done_counter`.

This design features the most complex messaging patterns. Six messaging tags are used for the different message contents:

- 5: God to Master (initial payload (slice, model)).
- 0: Master to workers (model object).
- 7: Master to worker (Per-worker count of candidate steps).
- 10: Master to workers (candidate steps (NumPy buffers)).
- 11: Workers to masters (weight results (NumPy buffers)).
- 42: Master to God (relocation events and completion signals).

The sources and destinations in this communication are explicit. Masters infer their worker set as contiguous ranks in the workgroup, and workers compute their master as `rank - ((rank - 1) % threads_per_workgroup)`. Using explicit MPI datatypes for candidate steps and weights in the performance-critical inner loop should minimize serialization.

3 Results

3.1 Performance Sequential

Analyzing the sequential program's performance allows a deeper understanding of how the program works and what its performance bottlenecks are. We will begin by inspecting the different functions' behaviors of the program and then discuss the performance of the simulation program as a whole.

3.1.1 Runtime distribution among program parts

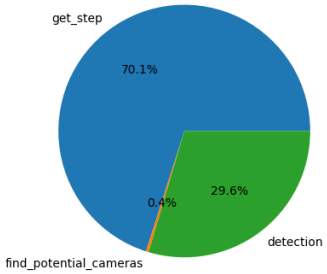

Figure 5: Workload of sub processes for one full step of one animal, including determining next location and Camera Trapping

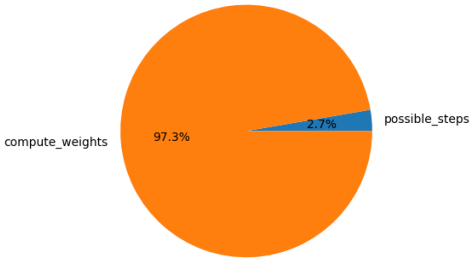

Figure 6: Workload of sub processes for determining next location (get_step)

Figure 5 depicts the time spent by the serial program subsection 2.2 to complete different sub tasks of one full step. Such a full step includes determining the next location that the animal will move to (`get_step`), determining what camera zones and forest sectors to execute detection on (`find_potential_cameras`) and `detection` itself. This full step will be referred to as `add_relocation`, because this is its name in the python implementation. One `add_relocation` call can be seen as a discrete basic iteration measure for the full program. This is because the full program is a sequence of iteratively calling `add_relocation` 2.000 times per animal. Thus, in a run with 100 animals, this full step would be executed 2.000.000 times.
In Figure 5 we can see that the compute time of determining the next location takes on average about 70% of the compute time of `add_relocation` while `detection` takes about 30%. `find_potential_cameras` is the main part of making the model sparse and takes a negligible amount of compute time.
In Figure 6 one can see the workload distribution of determining the next step. As described earlier, to determine the next location of an animal, 100 possible locations are generated based on given probability distributions (`possible_steps`) and for each of those possible locations, a weight is computed which is in relation to the distance to the closest home range of the agent (`compute_weights`). Those weights are then considered as the weighted probabilities for each location and the future location is drawn at random. Figure 6 shows that `compute_weights` makes up for about 97% of compute time of `get_step` while generating the possible steps only makes up about 3%. Thus, overall weight computation is about two thirds of compute time.

3.1.2 Performance behavior full program

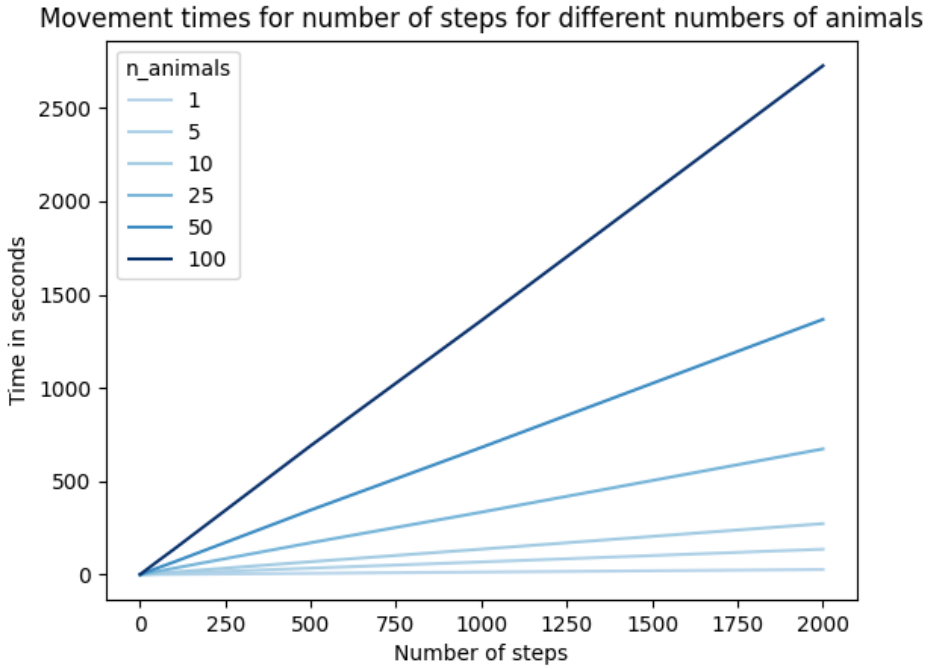

Figure 7: Movement model execution time for different number of animals (n_animals).

The average runtime of the serial implementation (across 5 runs) for the discussed model configuration is 3696 seconds (62 Minutes). Figure 7 shows the total runtime of the model for various numbers of generated steps and amounts of animals. As can be expected based on the repetitive execution of `add_relocation` for every animal, the base execution time of this method can simply be multiplied by the number of animals and number of steps to predict the overall runtime of the model. Model setup and output generation are essentially neglectable constant terms that are not impacted by these two variables.

3.2 Results of Parallelization Methods

3.2.1 Agent-based parallelization v1

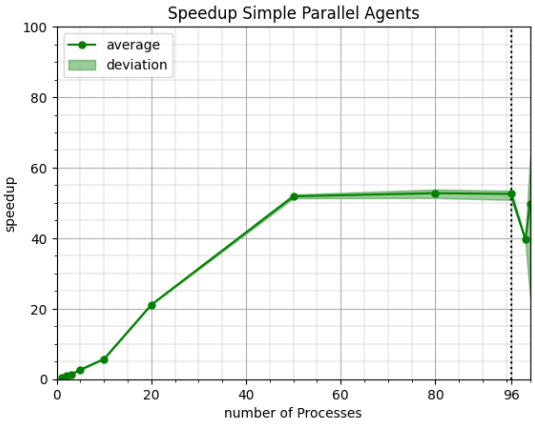

Figure 8: Simple Parallel Agents Speedup Diagram

Figure 8 shows the speedup curve over the amount of processes used for computation on a compute cluster for the Simple Agent parallelization approach subsubsection 2.3.1. Over the numbers of processes lower than 50, the speedup is almost linear. In the interval of 50 to 96 processes, the speedup is constant around 130. Then, at 99 processes, there is a decrease in speedup and at 100, the speedup is higher again, but lower than in the interval between 50-96. Deviation in compute times for the same amount of processes is negligible for less than 96 processses.

The non-linearities are likely caused by noise interfering with the experiments. This noise might have different forms, among which are temperatures of used CPUs before the start of the program and other cluster related workloads on the CPUs.
The reason for the stagnation in the interval between 50-96 processes can be traced back to the way compute is distributed between processes. One agent/animal cannot be computed by two or more processes and agents to be computed are assigned to processes as equally as possible. But, because there are 100 agents, for any number of processes between 50-99, there will always be at least one process which has to compute the steps for 2 agents. For 99 processes this is exactly one process, so the 98 other processes wait idle after having finished the computation of their one animal. In other words, the program does not run quicker when using 99 processes instead of 50, because in this interval, increasing the number of processes just increases the number of processes which have to compute one animal, but there is always a process that has to compute the steps for two animals. This changes at 100, because 100 animals can be divided cleanly onto 100 processes.
So why does the program run longer with 100 processes than with 50?
This is because the GWDG scientific compute cluster's nodes consist of 192 CPUs and the use of 2 CPUs per process. Once more than 96 processes are used, the program runs on two nodes, which drastically increases communication time between processes. This is because communication within one node is quicker than communication from one node to another.
This explains why there is a decrease in speedup/increase in compute time for 99 processes. In theory, using 100 processes should faster than using 99, because animals can now be divided onto processes such that each process has to compute just one animal. One can see that compute time variance is high when using two nodes. This might be because there are now even more external factors influencing runtime, like for instance general cluster communication load.

3.2.2 Agent-based parallelization v2

3.2.3 Master Worker Parallelization

The Master Worker parallelization approach in this original form was not tested for runtime. This is because after writing the code, a Vampir Performance Analysis was conducted, mainly to gain a better understanding of what exactly happens during the program run. This analysis led to the optimized version. Said Vampir Performance analysis yielded the following:

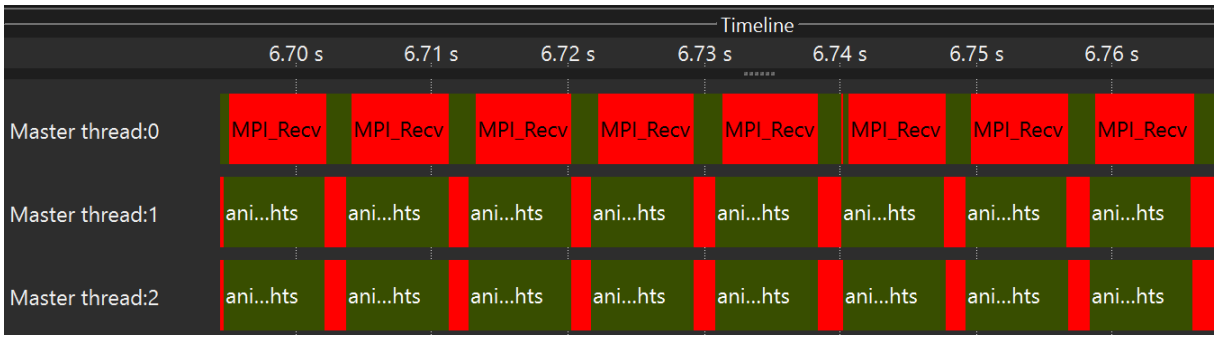

Figure 9: General Overview of Master Worker Parallelization in Vampir

This excerpt from the Vampir Performance Analysis tool shows a general overview of what happens in the program. Thread 0 is the *Master* and Thread 1 and 2 are the *Worker* threads. Time sections colored in red are `MPI` functions that are being executed, while green sections represent functions of the program. All red `MPI` functions visible here are blocking `MPI_Recv` calls. The `MPI` send functions take too little time to be visible here and are usually implemented to be non-blocking. This means that every red segment signifies the waiting of a process to receive information from another process. It is immediately visible that 1. the *Master* spends a long time waiting for the *Workers* to compute the weights and 2. during the time that the *Master* works, the *Workers* wait. They rarely work at the same time, which contradicts the purpose of parallelization. There are different ways of addressing these waiting periods:
2. is arguably easier to improve by increasing the amount of *Workers* per *Master*. This way, the time the *Master* spends waiting decreases, because the compute of weights speeds up. More on this in the Master Worker Optimized approach analysis.
1. is trickier, but more fundamental and requires more knowledge of what happens during the period that the *Master* works and the *Workers* wait, which we acquire by zooming into one such time frame in Vampir:

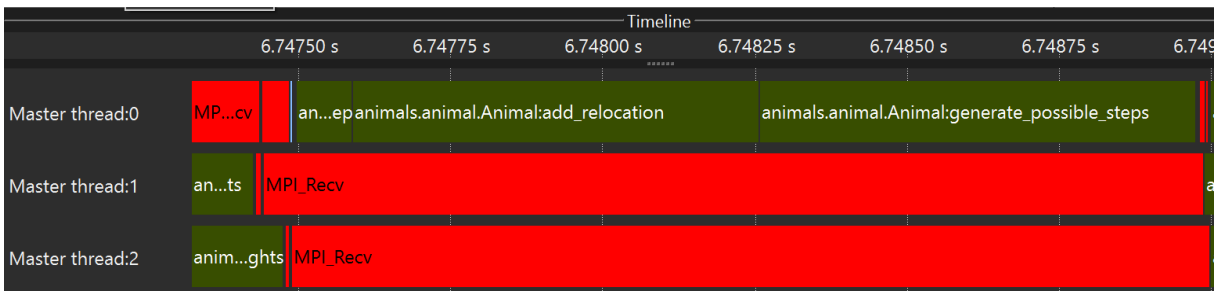

Figure 10: *Master*'s work in detail in Vampir

In this time frame of the program in Vampir, we can see the process's operations in the time during which the *Master* works and the *Workers* wait. The *Master*'s work is as follows: 1. `get_step` 2. `add_relocation` 3. `generate_possible_steps`

1. `get_step`, the *Master* gathers the weights computed by the *Workers* and draws the next step to be done based on the weight probabilities.
2. `add_relocation`, the *Master* updates the animal's location as the new location and does more tasks associated to that like determining what new Forest Sector the animal is in, what Forests Sectors it crossed and how much time it spent inside the Forest during its trajectory.
3. `generate_possible_steps`, the *Master* generates 100 new possible locations and divides them according to the amount of *Workers*.

After this, on the very right of the *Master*'s timeline, the red `MPI` function represents the sending out of the generated steps to the *Workers*. After, while the *Workers* compute weights, the *Master* does the computation of `detection`.
This structure analysis gives rise to the Master Worker Optimized Parallelization Method subsubsection 2.3.4.

3.2.4 Master Worker Optimized

To measure efficiency and scaling of this approach, test runs have been executed and timed. The results are gathered in a speedup diagram:

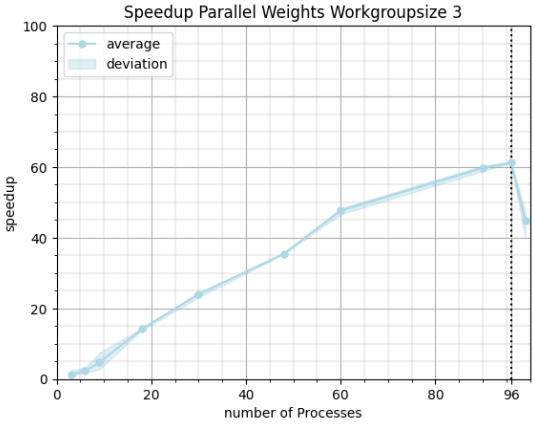

Figure 11: Speedup Diagram of Master Worker Optimized Parallelization with work groups of size 3

Figure 11 shows the speedup of the simulation over different amounts of processes using the Master Worker Optimized parallelization with work groups of size 3. Visible from the plot is that deviation in compute time for the same amount of processes used is negligible and that there is a decrease in speedup, which directly indicates an increase in compute time for 99 processes opposed to 96. The general shape indicates that speedup is not quite linear, but slightly sublinear, especially visible in higher numbers of processes.

As for the performance of the parallel agents approach subsubsection 3.2.1, noise comes up due to cluster conditions.
The sublinearity is probably caused by growing communication overhead with higher number of processes. A lot of communication is necessary, which takes up some of the compute resources. Then because the proportion of runtime of such communication increases in comparison to the one of the runtime for the actual program's functions as the latter decreases.

To come back to the point mentioned in subsubsection 3.2.3 as 2., a so far unadressed inefficiency lies in the waiting of the *Master* on the *Workers* while they compute the weights. We will adress this as suggested by trying different work group sizes and determining the optimal one.

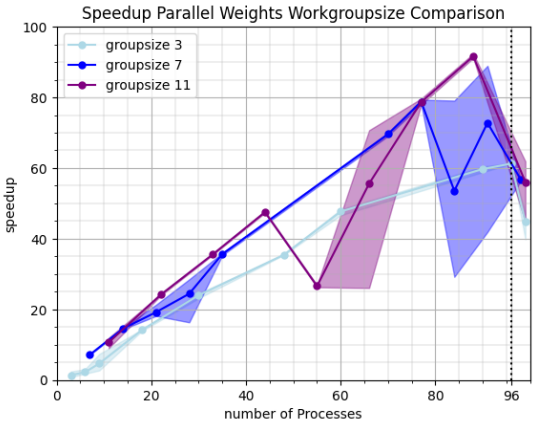

Figure 12: Speedup Diagram of Master Worker Optimized with workgroupsizes 3, 7, 11

Figure 12 shows the speedup diagrams of the Master Worker Optimized approach with different work group sizes. You can see that work group sizes 7 and 11 are more efficient than work group size 3 with 11 being the slightly most efficient one in general. However, it is also visible that those more efficient ones have more volatile execution times, because the span between the minimum and the maximum speedup for a given work group size is large. For all three, using more than 96 processes leads to a decline in performance.

The increase in speedup rate between work group size 11 and 7 is minimal and might also just be a cause of runtime deviations. This volatile runtime probably originates from inconsistencies in program runs. The simulation is a sparse simulation and if in one run, one work group has to take a lot of pictures, runtime can increase drastically. This theory however is opposed by the fact that for work group size 3, this volatility is not measured even though it is running the same program. To further investigate this behavior, more executions would be necessary to get statistically representative data.

For this approach, it is noteworthy that one can employ more processes than animal agents to be simulated. In fact, one can theoretically use one work group per animal, thus the number of processses is capped by *work group size* $\times$ *number of animals*. For a work group size of 11 and 100 animals as usual, this works out to 1100 processes that could be employed. Testing with 297 processes, distributed over 5 nodes lead to a runtime of 18.13 seconds, which is staggeringly fast in comparison to the serial runtime of more than one hour. Speedup at this runtime is 203.84, thus scaling behaviour is not linear, but still somewhat good.

There is one more thing to be noticed when analyzing the trace file of the Optimized Master Worker approach in Vampir:

When a camera takes a picture, which happens very rarely, all *Workers* wait for a "long" time while *Master* does the necessary computations of detection and photo taking. In Figure 13 on the right of all the red waiting of the *Workers*, two steps are done without pictures, visible in green. The big red sections represent two steps during which the *Master* takes photos. They take about 14 times as long as iterations without photos. Even though this happens rarely, it leads to a "lot" of unused compute resources. In addition to that, an animal which just crossed a camera zone is more likely to cross a camera zone again during the simulation because it is close to a camera.

Figure 13: Photo Taking by *Master* in Master Worker Optimized with work group size 11

This means that if one work group takes a picture, it is more likely to take more pictures than other work groups. This leads to inequal compute times, as taking photos is costly in compute. Most work groups will finish their designated animal batch to compute in similar time, but some work groups might work a lot longer. Thus, at the end of the program, many work groups will have waited for some time for other work groups to finish, meaning loss in efficiency. To countersteer this inequality and avoid long wait times for *Workers*, the final parallelization approach subsubsection 2.3.5 has been created as an extension of the Optimized Master Worker approach.

3.2.5 Master Worker God Photographer

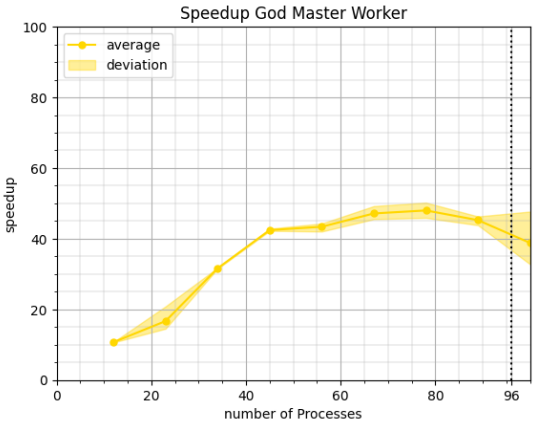

Figure 14: Speedup Diagram Master Worker God Parallelization

Figure 14 shows the speedup diagram of the God Master Worker parallelization approach. While in lower process numbers, scaling seems linear, for process numbers over 50, scaling seems to slow down. There is in general a constant small amount of deviation in runtimes, except for at the use of 100 processes. At the use of 89 processes and 100 processes, speedup declines. In the interval of 45 processes to 89 processes, speedup is quite similar, showing stagnation.

As this parallelization approach is very similar to the Master Worker Optimized approach subsubsection 2.3.4, similar speedup behavior is expected. However, speedup seems to scale worse and there is weaker deviation in runtimes. The decline for 100 processes can be explained the same way as for the other approaches. More confusing though, is the decline in speedup at the use of 89 processes. We cannot find a reason for this behaviour and since

the decline is small, believe that this is due to measurement imprecision. This however, does not explain the stagnation in speedup in the interval of 45 to 89 processes. As more processes are used, there are more work groups among which workload is distributed, thus an increased number of processes should lead to an increase in speedup, as visible for 45 and less processes.

3.3 Discussion

3.3.1 Baseline Expectation Assumptions

To begin the discussion of the results, it is helpful to set a framework of what the best case and expected gain in efficiency from parallelization are. This gain in efficiency will here forth be called *speedup* and measures how much quicker a parallel program finishes compared to a serial version. The speedup S is computed as

$$S = \frac{\text{serial runtime}}{\text{parallel runtime}}. \tag{4}$$

Different theoretical boundaries and estimates for speedup exist, but the simplest and most general one goes back to Amdahl's law and is sometimes referred to as Gustafson's law for parallel computing *Gustafson's Law* 2025.
It states that the estimated speedup S of a program gained by using parallel computing is

$$S = s + p \times N, \tag{5}$$

where s and p are the fractions of time spent executing the $s$erial parts and the $p$arallel parts of the program respectively. We assume that $s+p = 1$. N is the number of processes used.
This rather simple looking formula can become complex very quickly when posing the question of how much of the program's runtime falls under p the parallel part and s the serial part respectively. As runtime distribution might differ from one set of hardware to another and there are lots of factors to consider, it becomes evident that the formula is just useful for a rough estimation of speedup. However, it gives us an upper boundary for speedup. Assume that $s = 0$ and $p = 1$, then $S = N$. Therefore, we can see that speedup can only be linear in the number of processes used for parallelization. It is not possible to get quadratic or even exponential speedup. This means that there is an upper boundary for speedup for any parallel program, which is given by N. The goal is thus to design a parallel program whose speedup is as close to this upper boundary as possible.

3.3.2 Comparison

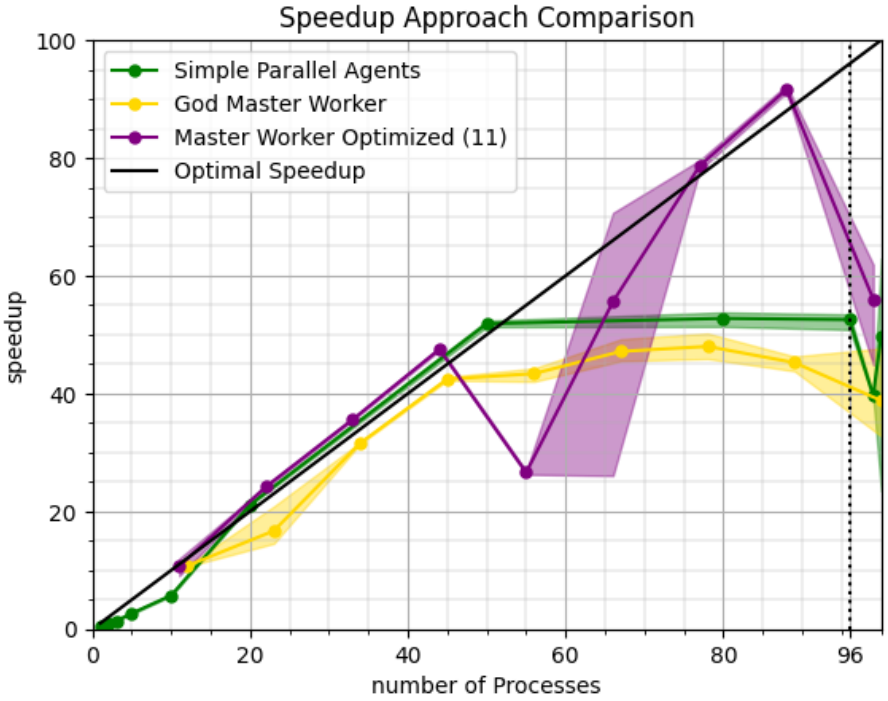

Figure 15: Speedup Diagram Comparing all Parallelization Methods

Figure 15 shows the speedup of all measured parallelization approaches, as well as the theoretical optimal scaling derived in the previous paragraph subsubsection 3.3.1. One can observe that speedup of some approaches sometimes exceeds optimal speedup. This is not possible and likely due to measurement imprecision. Most of the noteworthy, approach specific observations have been made in the previous subsections. This diagram illustrates very well how the different approaches' speedups compare. For number of processes lower than 50, broadly all approaches perform similarly well. From there on, differences in speedup are strongly noticeable. The simple parallel agents speedup stagnates without much variance, while the Master Worker Optimized speedup decreases and increases strongly with a lot of variance. The God Master Worker approache's speedup stagnates with a bit of variance and eventually decreases again. The only approach to scale close to the optimal speedup from there is the Master Worker Optimized one. Even though its runtime varies quite strong, it shows good speedup overall, outperforming the other approaches. Unlike expected, the God Master Worker method, which works very similar to the Master Worker Optimized one, does not share similar speedup behavior.

If one had to rank the different parallelization approaches, it would be important to specify how many processes the program would be run on. If run on 50 processes, using the Simple Parallel Agents approach would be the most favorable as it shows similar strong scaling behavior with little uncertainty. If run on more processes, one would have to weigh reliability vs. potential speed, choosing either the Simple Parallel Agents or Master Worker Optimized approaches. In that case, if the Simple Parallel Agents approach is chosen, one might aswell just run it on 50 processes, because runtime will be similar for 50 or more processes.

For any approach, if one had 100 processes available, it would be best to use at most 96, if they are all available on a single node to avoid node-to-node communication, which strongly slows down runtime.
In the case of unlimited resources, it would be reasonable to use the Master Worker Optimized approach with as many processes as possible, which will be more than for Simple Parallel Agents.
The minimum runtime achieved was 18.13 seconds with 297 processes and 594 CPUs, which is a nice result considering that the serialized version runs for about one hour.
Assuming one has access to this much compute power, the simulation could now be run a lot of times in a short period of time. This way, the simulation could be used for the generation of statistically viable scientific results. It should also be said that the simulation would be run in parallel with every process running one seperate simulation. This would probably be the most common approach for real world use, but it is nice to be able to simulate half a year of wildlife in just a few seconds.

4 Discussion

4.1 Introduction

Overall, we are very happy with the outcome of this project. It became more extensive than we had first anticipated, but this was due to our own curiosity and not some annoying outside factor. Because it was so big, there was a lot of room for learning and we especially enjoyed being able to go our own chosen direction.
When we started with the serialization, the program took between 45 minutes and one hour to run. Using this to generate statistically viable data seemed unthinkable at first, so we got a bit hung up on trying to optimize the serial approach. After the implementation of our first parallelization approach however, we realized how much leverage parallel computing can have.

4.2 Challenges

Along the journey of coming up with, implementing and testing our approaches, we have faced a lot of challenges of varying size. We would like to present the most noteworthy ones.

4.2.1 Generating trace files with Score-P binding for python

There is a Score-P binding for python. It is available on github *Score-P binding for Python* 2025. We are very thankful for the creation of this tool. Once one has learnt the ropes with this tool, it is a charm to use. However, the really hard part, which cost us multiple days of time to master was setting it up on the cluster and learning the basics of use. To use the Score-P binding for python, one has to import its github and install it via pip inside a clean virtual environment. For the installation with pip, the openmpi and scorep modules should be loaded on the cluster such that pip can link dependencies properly.
This is it. It is not very hard. However, if one tiny detail is done wrong, the installation goes wrong. In some cases, it is immediately visible that something went wrong during the installation. In others, one only notices a day later, when wondering why the produced

trace file is not a correct one. After trying a lot of different options inside the tool and giving up multiple times, we tried a hard reset of the system because of a lack of ideas to fix it. The problem in general is the lack of thorough documentation for the software, which leaves the unexperienced user a bit lost in the multitude of possible problems.

4.2.2 Communication Complexity

When writing a serial program, one has to remember what variable saved the information that one is interested in. In comparison to that, when writing a parallel program, one has to constantly have in mind what process or even processes have a specific, sometimes not identical, piece of data that one is interested in. This is obviously much harder. What we often struggled with when coming up with a new parallelization approach was keeping track of what process rank is responsible for what.
We would use modulo arithmetics to assign ranks to groups, for example "every third rank is a *Master*". But this really decreases code readability, especially for the other group member who tries to understand what is going on in this unknown parallelization approach. We tried different systems, and found that assigning boolean values according to the groups, for example called *Master*, *Worker* or *God* to ranks and working with those really facilitated this part of parallelization. A similar approach is good for describing relations between ranks, which could for example be "`master = (rank) 3`". This way, if one wants to communicate data from all *Workers* to their respective *Masters*, it is easy.
This system is similar to object oriented programming, because one builds groups with specific properties and variables. Without this system, one would have to do a lot of arithmetics in for loops and if statements and get lost along the way.

4.2.3 No Data Loss

In our simulation, a lot of data is important as later results. When parallelizing the programs with multiple processes, it is possible that different data points end up on different processes. It was therefore very important for us to pay attention to communicating all of this data. We did not always end up with all the data wanted and following the pipelines through the code is an exhausting task, especially when data is sent from process to process.
To be able to confidently say that our simulations generate the correct results, we decided to plot and output our data regularly. This way, we would see if some unusual results were produced after a change and could find the missing or wrong data before implementing the next change. Of course, if the plots look legit, this is not a guarantee, but still a practical, simple indicator of problems.

4.2.4 Cluster Errors

Sometimes, when developping a new approach or implementing a change, we would get Slurm errors. As we are not used to working with slurm and the cluster, at first we had to do a lot of research every time these things happened. We were often not sure if the problem was a setting/feature of the cluster or a configuration limitation by slurm or if it was a problem in our code. In these cases, it was often very helpful to consult the internet as we could gather more information on the problem than through just reading the manual.
This happened especially during our earlier phases of using the Score-P binding for python

and cost us a lot of time looking for the problem. To find the actual problem faster in such situations, probably the only way is to spend more time and gain experience such that one can develop an intuition for the work environment.

4.3 Learnings

4.3.1 Parallelization Approaches

The key lesson for us during this course was the following:

The 80 // 20 rule holds true for parallel programming!

We have developped quite sophisticated parallelization approaches over the course of this class. The easiest to implement was the Simple Parallel Agent approach, taking only a couple of hours to build and get running on the cluster. Compared to this, the Master Worker approach took multiple full work days to build and refine until it could be used with reliable output. While the Master Worker implementation performs better generally, it was a hassle and by the time it was running, the Simple Parallel Agents approach would probably have been able to produce a more than sufficient amount of simulation data. So, before coming up with a highly complicated way to parallelize a program, one should see if the simplest parallelization is enough to get the job done as desired. But, of course, in some cases, one really needs the most optimal parallel solution. And for those cases, one can put the 80% of extra work.

4.3.2 Every CPU has multiple cores nowadays

When buying a new laptop or computer, one is always presented with how many cores a CPU has. So far, we never really understood this and wondered if this was good for anything else than having multiple applications open at one time. Opposed to this, one might wonder if they would ever use parallelization skills learnt during the course if they do not work with compute clusters.
And this is where it is really satisfying to realize that one can use this skill of parallelization in any discipline related to programming and on any machine. Not only because the concepts learnt can be transferred, but because one can use the actual skill to speed up computation in almost any scenario. Further, it is becoming more and more evident to us that without parallelization, humankind would by far not have made as much progress with computers, because computing with just one process is simply not fast enough.
Moreover, one comes to appreciate how parallelization is a core building block of computer science, because it is already indispensable on transistor level hardware.

4.4 Other Solution Options

Our type of simulation is a sparse simulation. Another sparse simulation example is gas simulation. In such simulations, agents (molecules) which are far apart have an insignificant impact on each other, thus one can negate it. Therefore, a parallelization technique often used for such simulations is location based parallelization. An example would be to divide the world into cubes in 3-dimensional space and have one process compute the attractive of gas molecules within one cube.
A similar approach could have been used to parallelize our simulation aswell. We believe however that it would have been less efficient and not really reasonable, because

our animal agents roam independent from one another. Thus, often, one process would be computing a lot of animals' steps while another one would only be computing a few, leading to idling and thus inefficiency.

There is also another fundamentally different approach, which we considered, but did not have the time to pursue further. Instead of simulating animals live, while computing camera detection and more, one could do these two parts of the program seperately.
For example, one could simulate and save enough different movement trajectories. Those would of course have to be a lot to be statistically viable. Then, random sampling 100 of those movement trajectories, one could compute camera detection and the other necessary data metrics based on those and get very fast runtime on this.
However, this is a fundamentally different way of approaching the problem and was not really aligned with our goal to master parallelization techniques on the cluster.

4.5 Regrets

We are very happy with our own learning progress, aswell as our project's progress. We achieved our goal of making the program run in under one minute and along the way gained understanding of the fundamentals of parallel programming. Of course, we can only claim to have learned the basics and learning more advanced concepts would take a lot more time. But for those reasons, we do not really see a reason for having done anything differently.

5 Conclusion

This project was about building a simulation to simulate animal's movement behaviours and the camera capturing of this. A similar program had existed in a programming language called NetLogo. This program however was too slow to generate statistically meaningful results for further research.
After implementing a similar program in python serially with different data flows and camera capturing, the simulation ran significantly shorter, but still in an unuseable hour. Therefore, parallelization was a very promising way of decreasing runtime drastically.

Our first parallelization approach, which was agent-based parallelization, proved to be a very stable approach. Runtimes compare to more sophisticated parallelization approaches for low to medium numbers of processes.
The most efficient, but least stable approach uses work groups of processes working together. It scales almost linearly on a single node. Furthermore, it allows for a large number of processes to be used, which can result in seriously low runtimes. Our shortest run was close to 18 seconds, using 594 CPUs.
This shows that we were able to far surpass our goal of running the simulation in under one minute.
If used to generate statistically viable results for research, one would probably use the agent-based approach for its stability with a small to medium amount of processes. For example for 50 processes used, one could expect a runtime of around 70 seconds per simulation run. This offers for quick runtime while still having nearly perfect efficiency in

terms of compute use.

So, one could say that our project was highly successful.
Moreover, in terms of learning we were also successful. Among others, we learned to work in the environment of a compute cluster, parallelize a python program, analyze its runtime and function behaviour from a trace file in Vampir and most importantly, to master problems along the way.
A key take away for us was the learning that writing a simple parallel program is possible in a short time and can already lead to surprisingly good runtime improvements. Sometimes this is all that is needed to get the job done. Another learning is the shift in perspective on parallel computing. One does not need a big compute cluster, but can already parallelize programs on a local machine. Parallelization is omnipresent and this realization is very valuable for our further programming.

Finally, to summarize, we learned a lot on our path to achieving our project's goal. We had a lot of fun in this project and are very grateful for the learning opportunity offered by the course on high performance computing. In the future, we will surely use the parallelization techniques learnt and continue our learning journey in this field.

References

Ammer, Christian et al. (2010). *Der Wald-Wild-Konflikt*. 1st ed. Universitätsverlag Göttingen. ISBN: 978-3-941875-84-5. URL: https://univerlag.uni-goettingen.de/bitstream/handle/3/isbn-978-3-941875-84-5/GoeForst5_Ammer.pdf?sequence=4&isAllowed=y.

Anile, S. et al. (Aug. 2014). "Wildcat population density on the Etna volcano, Italy: a comparison of density estimation methods". In: *Journal of Zoology* 293 (4), pp. 252–261. ISSN: 0952-8369. DOI: 10.1111/jzo.12141.

Bödeker, Kai et al. (Aug. 2021). "Determining Statistically Robust Changes in Ungulate Browsing Pressure as a Basis for Adaptive Wildlife Management". In: *Forests* 12 (8), p. 1030. ISSN: 1999-4907. DOI: 10.3390/f12081030.

Clasen, C. and T. Knoke (Apr. 2013). *Die finanziellen Auswirkungen überhöhter Wildbestände in Deutschland*. URL: https://mediatum.ub.tum.de/doc/1100538/document.pdf.

Doran-Myers, Darcy (2018). "Methodological Comparison of Canada Lynx Density Estimation". In: URL: https://doi.org/10.7939/R3Q815805.

Forsyth, David M. et al. (May 2022). "Methodology matters when estimating deer abundance: a global systematic review and recommendations for improvements". In: *The Journal of Wildlife Management* 86 (4). ISSN: 0022-541X. DOI: 10.1002/jwmg.22207.

Gilbert, Neil A. et al. (Feb. 2021). "Abundance estimation of unmarked animals based on camera-trap data". In: *Conservation Biology* 35 (1), pp. 88–100. ISSN: 0888-8892. DOI: 10.1111/cobi.13517.

Gustafson's Law (2025). URL: https://en.wikipedia.org/wiki/Gustafson%27s_law.

Howe, Eric J. et al. (Nov. 2017). "Distance sampling with camera traps". In: *Methods in Ecology and Evolution* 8 (11), pp. 1558–1565. ISSN: 2041-210X. DOI: 10.1111/2041-210X.12790.

— (Jan. 2019). "Model selection with overdispersed distance sampling data". In: *Methods in Ecology and Evolution* 10 (1), pp. 38–47. ISSN: 2041-210X. DOI: 10.1111/2041-210X.13082.

Lyet, Arnaud et al. (July 2023). "Estimating animal density using the Space-to-Event model and bootstrap resampling with motion-triggered camera-trap data". In: *Remote Sensing in Ecology and Conservation*. ISSN: 2056-3485. DOI: 10.1002/rse2.361.

Marcon, Andrea et al. (Oct. 2019). "Assessing precision and requirements of three methods to estimate roe deer density". In: *PLOS ONE* 14 (10), e0222349. ISSN: 1932-6203. DOI: 10.1371/journal.pone.0222349.

Moeller, Anna K., Paul M. Lukacs, and Jon S. Horne (Aug. 2018). "Three novel methods to estimate abundance of unmarked animals using remote cameras". In: *Ecosphere* 9 (8). ISSN: 2150-8925. DOI: 10.1002/ecs2.2331.

Nakashima, Yoshihiro, Keita Fukasawa, and Hiromitsu Samejima (Mar. 2018). "Estimating animal density without individual recognition using information derivable exclusively from camera traps". In: *Journal of Applied Ecology* 55 (2), pp. 735–744. ISSN: 0021-8901. DOI: 10.1111/1365-2664.13059.

Palencia, Pablo et al. (Aug. 2021). "Assessing the camera trap methodologies used to estimate density of unmarked populations". In: *Journal of Applied Ecology* 58 (8), pp. 1583–1592. ISSN: 13652664. DOI: 10.1111/1365-2664.13913.

Rogowski, Marcin et al. (2023). "mpi4py.futures: MPI-Based Asynchronous Task Execution for Python". In: *IEEE Transactions on Parallel and Distributed Systems* 34.2, pp. 611–622. DOI: 10.1109/TPDS.2022.3225481.

Rowcliffe, J. Marcus et al. (Aug. 2008). "Estimating animal density using camera traps without the need for individual recognition". In: *Journal of Applied Ecology* 45 (4), pp. 1228–1236. ISSN: 0021-8901. DOI: 10.1111/j.1365-2664.2008.01473.x.

Royle, J. Andrew (Mar. 2004). "N-Mixture Models for Estimating Population Size from Spatially Replicated Counts". In: *Biometrics* 60 (1), pp. 108–115. ISSN: 0006-341X. DOI: 10.1111/j.0006-341X.2004.00142.x.

Royle, J. Andrew et al. (2014). *Spatial Capture-Recapture*. Academic Press. DOI: 10.1016/B978-0-12-405939-9.00022-0.

Santini, Giacomo et al. (June 2022). "Population assessment without individual identification using camera-traps: A comparison of four methods". In: *Basic and Applied Ecology* 61, pp. 68–81. ISSN: 14391791. DOI: 10.1016/j.baae.2022.03.007.

Score-P binding for Python (2025). URL: https://github.com/score-p/scorep_binding_python/.

Walters, C. (1986). *Adaptive Management of Rewnewable Resources*. Macmillan Publishers Ltd. URL: https://pure.iiasa.ac.at/id/eprint/2752/1/XB-86-702.pdf.

A Work sharing

In general, work was distributed according to availability. Louis is doing the 9C version of the course, and worked more according to the credit distribution.
Thomas is doing the 6C version of the course.

A.1 Louis von Leitner

1. Serial Program
    - Camera Detection Logic
    - Deer Movement
    - General Structure and Logic
2. Parallelization
    - Master Worker Approach
    - Vampir Performance Analysis
    - Master Worker Optimized Approach
    - God Master Worker Approach
3. Presentation
    - Performance Measurements
    - Slides Detection
    - Slides Parallelization
4. Report
    - Performance Measurements
    - Master Worker
    - Master Worker Optimized
    - God Master Worker
    - Results & Discussion
5. Animation

A.2 Thomas Hay

1. Serial Program
    - Initial design
    - Model setup
    - Step generation
    - Sparse modeling things
    - Output handling

- Visualization

2. Parallelization
    - Agent-based approach
    - Error-hunting
3. Presentation
    - Problem outline
    - Solution approach
    - Performance Serial
    - Agent-based parallelization
    - Performance of this
4. Report
    - Introduction
    - Abstract
    - Sequential Implementation design
    - Parallel designs: Agent based v1 + v2/ Improvement expect.
    - Implementation
    - Performance Sequential

B Code samples

The entire code of the project can be found under:
https://gitlab.gwdg.de/hay/parallel-animals

The different implementations of the various parallelization schemes etc. are contained in their respective git branches.