GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN IN PUBLICA COMMODA SEIT 1737

HPS

**Report in the Practical Course on High-Performance Computing**

# Simulation of a Simplified Ecosystem to Study the Influence of Environmental Factors on Bee Populations


**Georg Eckardt**
Matrikelnummer: 23112861

**Henrik Jonathan Seeliger**
Matrikelnummer: 2053484

**Georg-August-Universität Göttingen**
Institute of Computer Science

Supervisor: Hauke Kirchner


September 29, 2023

# Abstract

This report examines the viability of sector-based map splitting in simulations for sharing the computational load on multiple processes. The presented work considers honeybees and hives, but our findings can be generalised to other biological fields. Simulations are often limited in size and complexity by computational power. Parallelization of simulations is typically challenging, because elements calculated in different processes have to interact with each other. We aim to present a parallelization approach that offers good performance while still being easily modifiable. We evaluated our program using benchmarks on different scenarios, to test for strong scaling performance and Score-P for tracing analysis. It was found that our current program receives good performance results and we presented ways, how to further increase the performance.

# Repository

We host implementation as well as benchmarking and visualization related code on GitHub. It can be found under:

`https://github.com/HnSee/hpc-2023-bee-simulation`

# Declaration of Authenticity

## 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 Figures

# List of Tables

# List of Listings

# Acronyms

| | |
|---|---|
| **CTD-Array** | Contiguous, two-dimensional array |
| **GCC** | GNU Compiler Collection |
| **HPC** | High-performance computing |
| **MPI** | Message Passing Interface |
| **OpenMP** | Open Multi-Processing |

# 1 Introduction

## 1.1 Motivation

Humans influence ecosystems in a variety of ways, most notably through human-made climate change. The resulting changes in the habitat pose a challenge for all organisms and can lead to diminished populations, in some cases to a local or even global extinction of species. Bees are one of those species that has been recently in the spotlight of media and scientists. The worries around a decline in bee population are enormous because bees are one of the most effective pollinators, and therefore vital for the production of almost all fruits, vegetables and, of course, honey [1], [2]. Unlike for many other species, the reason why bee population has been dropping for years remain unclear and are subject of scientific investigation to this date. There are a number of suspected biotic and abiotic factors like climate change, pesticides, air pollution and numerous more[1]. Especially the reasons for phenomenons like Colony collapse disorder, where all workers of a healthy colony suddenly leave the queen behind, remain completely unknown.

## 1.2 Aim of the Project

The aim of the project is to develop a platform for simulations which could be used to investigate possible causes and find solutions to help bees adapt to the changes, caused by humans. Because the list of possible causes is still vague, we designed our project to be configurable and extensible in a variety of directions. Therefore the concept must be adaptable, while still providing good performance on an HPC-system. Our approach is an agent- and tick-based simulation that splits the map into segments, with each segment being calculated by one MPI-process. The agent, tick-based model provides a high amount of variability because new environmental factors or agents are completely independent of the process of parallelization itself. This should allow for the possibility to add biotic and abiotic entities without or only with minimal change to the rest of the simulation.

## 1.3 Findings

We were able to show that the chosen approach of using agent-based simulations in combination with parallelisation and vector splitting is a efficient way to work the given problem. The benchmarks showed promising results and indicate that our parallelization approach works well for agent-based simulations. We managed to parallelize about 90% of the problem successfully and reduced the run time to just 6-7% of the sequential version. With further optimizations and an implementation of algorithms that selects more optimal sectors sizes, the percentage of parallelized code could have been further increased and the variance in results reduced. The presented concept could be used in a interdisciplinary research project, investigating the recent decrease in bee population in particular Colony collapse disorder.

# 2 Methodology

This chapter describes the stated problem in more detail, as well as the methods and design decisions leading to our solution. Furthermore, we state our method to validate and analyse our solution.

## 2.1 Problem Relevance

Simulations of ecosystems are, when done with enough precision, always resource intensive tasks. Because of the large amount of data and calculations involved, this problem represents a suitable topic for a high-performance computing system. To ensure accuracy and realism of the simulation, a high number of entities and a precise underlying spatial representation of the world is needed. Those aspects can only be processed by a high-performance computing system offering more resources than conventional computers if high accuracy and precision of the simulation is desired.

## 2.2 Solution Approach

To create our simulation of a simplified ecosystem, we have to design a theoretical model representing the ecosystem with its various factors and an architecture supporting the parallelization and adaption towards a high-performance computing system.

### 2.2.1 General Approach

Simulating a system with many interfering entities is not trivial, both on the modelling and the technical side. We needed a way to simulate many entities individually to measure their effect on the ecosystem. Modelling e.g. bee swarms as a whole does not provide a satisfying simulation accuracy and therefore also result accuracy.

To provide a high simulation precision and realistic results, we have chosen an agent-based model as our underlying simulation model as described by various authors [3]. This

term refers to various systems with many differences that rely on the same principle: modelling the application domain by creating individually thinking *agents* that affect each other. Agents may range from very simple to rather complex, depending on the type of agent in the application domain and what an agent refers to in the given context. For example, in the context of distributed cloud applications, agents may be independent smaller applications that implement their own individual functionality and may consume other agents and therefore interact with each other. In our context, every biotic and abiotic environmental factor is represented by an agent. This enables us to model many different agent types, or in our context, environmental factors with different effects on the world and the other agents. The core computation of the simulation is then the calculation of those effects individually for every existing agent. Every agent may behave differently, increasing the result accuracy of the simulation. This system also represents the main aspect of the simulation's extensibility. Using an agent-based approach, new agents with their behaviours and effects may be defined and implemented without great effort or any restructuring of the code.

Besides agents, which form the main part of our simulation, an abstraction of the underlying environment is necessary because aspects like biomes may also affect the agents' decisions. For this reason, we define our underlying *world* as a squared grid consisting of *blocks* with an edge length of one meter each, even though every agent may have real coordinates. Each block is assigned a biome, which results in a *world map* with different biome areas. These biomes can be seen as further environmental factors that drive the simulation. Agent instances and this world map together constitute the current simulation state that we define as the *world state*. Figure 2.1 illustrates this world state in conjunction with the agent-based model.

The simulation process we have chosen is simple. After generating the world map, we instantiate an initial simulation state controlled by the various simulation parameters, which is called *seeding*. The world generation step is randomized and produces a world map as described above. The simulation parameters control various aspects of the world generation like size and biome generation. After the world generation step has been completed, the initial world state is *seeded*. Using the simulation parameters which define e.g. how many initial bee hives to place, the seeding step initiates the initial agents. Then, the main simulation process starts, which is defined as a tick-based process. This means that the computations driving the simulation process happen in small computational and repeating steps, called *ticks*. Every tick, every agent's behaviour and effect on the world state is computed. To prevent logical and technical deadlocks or inconsistencies, one *tick*

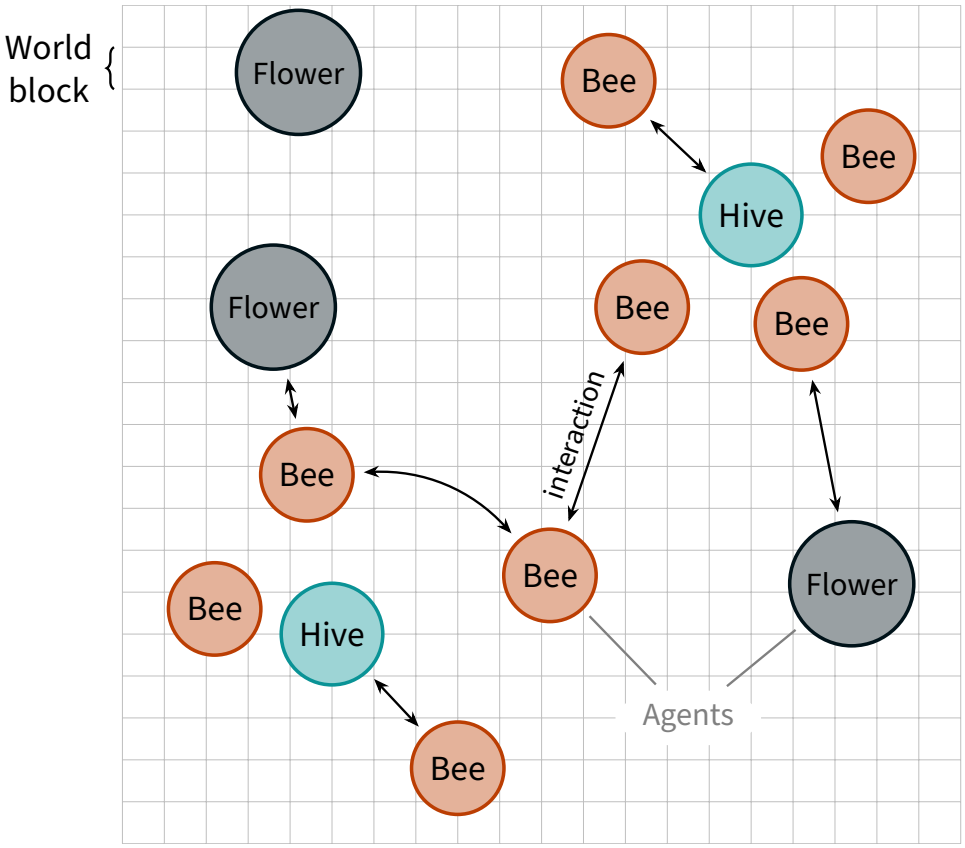



Figure 2.1: Schematic representation of the agent-based model and the world state as the simulation's underlying model

consists out of two phases: the *update* and the *move* phase. This has mainly technical reasons, which will be explained later in further detail on, but also logical problems are solved by using a clear phase separation. If an agent's behaviour defines updates to its internal state e.g. based on the current surroundings, it may use the update phase to update the internal state. Subsequently, it may decide whether and how to move in the move phase. In the update phase, every agent is passed the same world state as the other agents, which solves the question of which agent to update first. If an agent wants to react to or update another agent, it has to happen the next tick. Other possibilities like updating the agents in a randomized order were rejected by us because it may introduce logical issues and additional complexity to the simulation's model and implementation. Figure 2.2 summarizes the process of the simulation.

In order not to make the simulation too complex, we have decided to model and implement three basic agents: bees, bee hives and flowers. Those three basic agents enable observations of basic interactions in a bee-centered ecosystem simulation with the

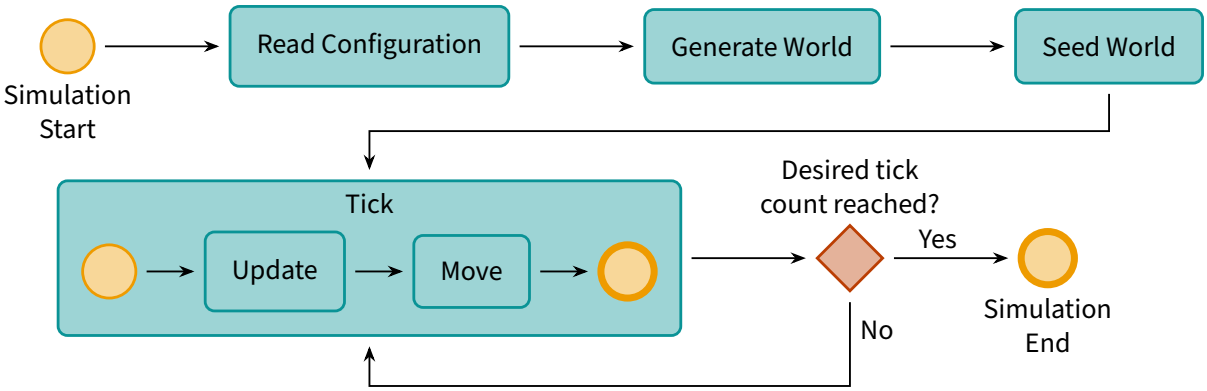



Figure 2.2: Process of the simulation

possibility to simulate bee colony behaviour including foraging and feeding. As described above, one key property of our simulation is its extensibility, meaning further agents to enhance the simulation may be added in future work.

### 2.2.2 Technical Approach

Because of the simulation's desired accuracy and scale, its implementation represents a resource intense application designed to run on high-performance computing systems. To save resources and time, our implementation has to be efficient. This means that we have to choose suitable tools and decide on technical aspects for the implementation in the first place.

The first decision we had to make was the programming language. Being an application to be parallelized later on for increased time and resource efficiency, we had to select a language with support for the Message Passing Interface (MPI) which is a standard for message exchange between application processes, implemented by various libraries like Open MPI [4], [5]. MPI implementations exist for many programming languages, like C/C++ or Python [5], [6]. We had to choose one of those languages for our implementation and decided on C++. C and therefore C++ are programming languages with first-class MPI support [5] and we wanted to use a fast and resource efficient language. Python also has an MPI implementation [6], but it is generally agreed that Python offers inferior execution time and resource efficiency when compared to compiled languages like C or C++ [7]. C and C++ are both efficient and fast programming languages, but C can be seen as operating on a lower level of abstraction than C++. C++ offers many advantages and additions over C, like object orientation using classes and a comprehensive standard library with many built-in functionalities and data structures. With the extensibility of

our simulation in mind, the object orientation was the main aspect that led to our decision. But also the standard library capabilities are an important advantage because they can save a lot of development time by reducing the amount of custom code to write.

Generally speaking, development time efficiency and developer experience are important aspects to us. Thus, we have chosen Meson as our main build tool because of its multiplatform support and simple configuration, while offering a considerable number of features [8]. Besides Open MPI [5] as an MPI implementation, we also use additional external libraries for e.g. logging and specific mathematical functions which will be explained in detail later.

Using these tools, our simulation can be implemented in a fast and extensible way. Implementation details can be found in chapter 3.

## 2.3 Implementation Design

One important aspect of our implementation is the parallelization. Parallelizing our code means adapting our code towards running on multiple processes in parallel instead of running on a single process sequentially. Doing so requires planning beforehand because parallelization solutions may be different for different problem domains. We have to plan what parts of our implementation may be parallelized and how to do so for each part.

### 2.3.1 Sequential

The general process of the simulation was elaborated in section 2.2.1. The sequential approach for our implementation is rather simple: compute everything in every process step in sequence. This applies mainly to the world generation and the ticking steps. Reading the configuration and seeding the world are trivial steps that make up only a fraction of the overall simulation's time and resource consumption.

For the world generation, we use an algorithm based on algorithms and world generation processes from various well-known voxel video games [9]. Without going into too much detail, our algorithm uses a combination of Voronoi diagrams and simplex noise to create a random but realistic world map consisting of various biome areas. As stated previously, every block gets assigned one biome. This biome has to be calculated and assigned for every block on the map. Because of the calculations involved in e.g. simplex noise, this can be computationally intense for larger maps. Our sequential approach is to just calculate every block's biome in sequence, leaving behind potential for optimization by parallelization.

Calculating every agent's behaviour every tick is the main computational step executed in the simulation. The sequential approach for this computation is trivial: every agent is evaluated in sequence.

### 2.3.2 Parallel

The world generation and the agent evaluation every tick are clear parts of our implementation that can be parallelized to gain time and resource improvements. Reading the configuration and seeding the world are the smallest part of the simulation in terms of time and resource usage. This is why we do not consider these steps concerning a possible parallelization.

As described earlier, the world generation's main computation is calculating every block's biome using different mathematical methods. This step uses random values in various calculations to ensure a more realistic world map. The world generation step consists of various smaller steps which include different smaller computations operating on one block each. Besides random values coming from a random number generator that can be made deterministic using *seeds*, those calculations for every block do not have any dependencies. This means that the smaller operations for every block can be parallelized easily by distributing them to the available processes. Because of the missing dependency between the respective blocks, the calculations can run independently on different processes while sharing the same memory space. This enables an easy to implement parallelization for these particular calculations, resulting in a planned performance improvement of the world generation step.

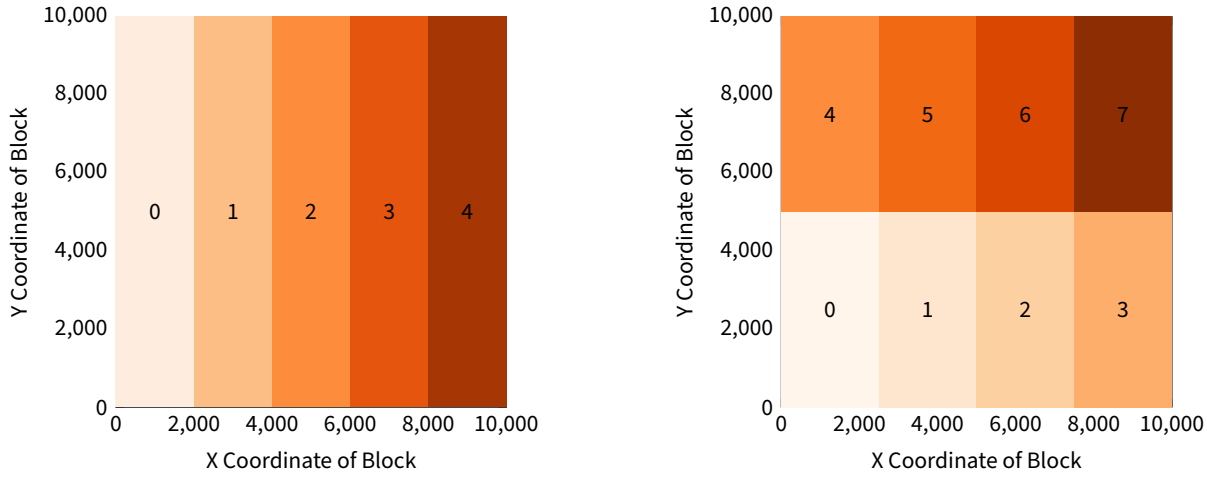
Parallelizing the ticks and agent evaluation is more complicated and our main concern, as the world generation is far less time and resource consuming than the ticking steap of the simulation. For this, we considered different possibilities with different advantages and disadvantages.

The first and simplest solution would be to just equally distribute the agent's evaluation computations among available processes. For example, if we have four processes and one thousand agents, every process would be assigned 250 agents which will be evaluated by just the corresponding process. This solution has two major drawbacks. The first one is the decision on how to distribute the agents in the first place and how to distribute new agents. During the simulation, new agents may come, or old agents may go, e.g. when new bees are hatched. This may result in heavy fluctuations in the overall number of agents. Using a round-robin procedure to distribute agents among available processes would be an easy solution for new agents, but could result in an unbalanced distribution of

agents as removed agents are not balanced out. We could think of a simple algorithm to equally distribute agents among processes, e.g. by checking every process's current agent number first, but this would create additional computation and particularly communication overhead. The second drawback is far more significant than the first. One aspect of agents is that they interact individually with their surrounding environment. For example, a scout bee's main behaviour is to scout for flowers, i.e. it has to search for flowers in its direct environment. This means that every agent has to be provided access to its direct environment to query for necessary information, e.g. query for nearby agents. If all agents are distributed randomly among available processes, this would result in ticks being communication-bound instead of computation-bound, as every process has to query every other process for possibly relevant information: Every relevant agent or information to query for may be on any of the available processes. This would mean huge communication efforts for each tick and each agent. These communication efforts being introduced by these two drawbacks need to be minimized and are the reason for us to use a different approach for parallelizing the tick mechanism.

The second solution we introduce is to distribute the agents with their evaluation computations among available processes in a spatially organized manner. As described earlier, distributing the agents randomly leads to huge communication efforts, which have to be minimized. The main reason for those communication efforts is the fact that every agent has to be provided information about its direct environment. This is why we want to distribute agents with nearby positions to the same process. This means that agents and processes do not have to query other processes to access information about their direct environment in the majority of cases. Only if an agent wants information about agents or the world further away, other processes have to be queried. This reduces the communication effort in comparison to the first solution. To define how to distribute the agents spatially and among processes, we need to define a system that splits the world into specific regions, called *chunks*. For $n$ chunks, this system needs to split the world map into $n$ equally large areas. By finding the closest divisors of $n$, the number of splits along the x and along the y-axis can be calculated. These can then be used to find the chunk bounds of each chunk, i.e. their coordinate bounds which in turn can be used to identify the chunk or process respectively to distribute individual agents to. Figure 2.3 shows example results of this chunking method.

Every process then handles one chunk each, i.e. computing and evaluating the update and move phase of every agent positioned in the chunk. This may lead to an efficient parallelization of the tick system because the communication effort can be minimized,




(a) Areas calculated with $n = 5$ chunks

(b) Areas calculated with $n = 8$ chunks

Figure 2.3: Example chunk constellations with a world edge length of 10000

resulting in a higher computational resource utilization. Besides that, this solution also has implications for the memory distribution across the processes and respective computing nodes. Processes do only need the memory for the agents assigned to them. This means that no memory is shared across nodes, and memory is managed dynamically per process according to the number and types of agents present in a chunk. Additional memory is not necessary, but time and computational overhead for exchanging the data is inevitable.

The chunking approach also has disadvantages which have to be considered. The major disadvantage is that the efficiency of this approach highly depends on the spatial distribution of the agents. If every agent is positioned inside the borders of one chunk, all agents are evaluated by a single process, resulting in no parallel computations at all. Because of the unpredictable and randomized character of an ecosystem and the corresponding simulation, it cannot be guaranteed that all agents are distributed equally and therefore that the simulation is parallelized efficiently at all time. Another disadvantage is that agents with their internal state data have to be passed between chunks and therefore processes if they move into the borders of another chunk. This does not only mean some communication effort, but may also require a complicated implementation. Nevertheless, this parallelization solution still has more upsides than the first proposed solution in our opinion. For this reason, we have chosen it over the former solution.

# 3 Implementation

After describing the methods, processes and models to use, we can explain the most important aspects of our resulting implementation. This does not mean to describe every detail of our code, but to highlight the technical properties of our implementation in the first place. Afterwards we want to give an overview of the implementation's structure as well as some particular technical details regarding our parallelization and performance-related efforts.

## 3.1 Technical Key Properties

As already described, the simulation was implemented using the programming language C++. For us, it represents the appropriate compromise between speed, language features and library availability. With the C/C++ world being cluttered by many different approaches for compilation, configuration and dependency management, we have chosen Meson [8] as our build tool. This tool enables us to reduce the time spent with the configuration of the build tool chain, as it is easy to configure and provides sensible defaults. We designed our whole project to be configurable and buildable using Meson, but after technical issues on the high-performance computing system given to us for running our software[1] we also implemented a *CMake* [10] based build process. These two tools may be used interchangeably to compile our simulation. Compilation instructions as well as prerequisites can be found in the file `README.md` in our repository. To make building our simulation reproducible and easy, we have also included a `flake.nix` file for *Nix* [11] users. Building in a *Spack* [12] environment will be explained later on.

To save time and to profit from expert knowledge in specific fields, our implementation uses specific external open-source libraries developed by third parties. Besides MPI i.e. Open MPI [5], the only libraries we use are:

[1] Meson is currently suffering from a bug causing wrong flags to be passed to the linker when used in a Spack environment.

**jc_voronoi** As already mentioned, our implementation of the world generation uses Voronoi diagrams. Various algorithms exist for creating Voronoi diagrams, including *Fortune's algorithm* [13]. Since we are neither experts in C++ nor Voronoi diagrams or mathematics in general we resort to an external implementation of this algorithm to generate our Voronoi diagrams called *jc_voronoi* [14]. Being a header-only library, the library's code has been integrated in our code base directly and can be found under `src/extern/jc_voronoi.h`.

**Cairo** Cairo [15] is a library for creating and manipulating 2D graphics. Fortune's algorithm gives us the basic and initial boundaries of biomes used in the world generation step. Because these boundaries are straight edges in a real coordinate system and we need to distort those for more realistic biome edges we have to rasterize those edges into a pixel map where every pixel represents one world block. For this, we use *cairo* or more precisely *cairomm* [16], Cairo's C++ bindings. These libraries give us the possibility to implement an important step in the world generation efficiently.

**SimplexNoise** Another mathematical construct we use in the world generation is simplex noise [17]. It is used for randomly but systematically blurring the biome edges. Again, a custom implementation would be outside our filed of knowledge, and we have chosen *Perlin Simplex Noise C++ Implementation (1D, 2D, 3D)* as the implementation to use [18].

**spdlog** Logging in software is an equally simple and old practice to give software users information about the software, while the software is running. It may be used to give immediate feedback e.g. by printing text to the standard output or save information to disk in a long-running process. Depending on the logger, simple text or machine-processable text formats may be used, enabling analysis of the logged information after the software has been terminated. Instead of manually printing text to the standard output, we use *spdlog* [19] for logging. This library gives us a robust logging solution with many features like logging levels and different logging formats while we do not need to deal with the underlying details. We use this logging mechanism for benchmarking and immediate user feedback while the simulation runs.

**cxxopts** Using the `int argc` and `char* argv[]` standard parameters, options can be passed to programs using the command line. But those options are passed as raw character

data and to get meaningful options, the raw character data must be parsed. This can be complex to implement which is the reason why we use the *cxxopts* [20] library to do so. This enables our simulation to be configurable.

**Benchmark** While developing our simulation, we wanted to find the fastest and most efficient implementation for specific tasks. This is why we used the *Benchmark* library [21] to do some micro-benchmarks while developing. It offers many utilities that helped us creating small benchmarks which will be described later on.

**Google Test** Software testing is an important aspect of software engineering. This is why we have implemented unit tests for some of our internal components, including our utility data structures we have created with performance improvements in mind. These utility data structures are also designed on an abstract level so they can be used in other contexts as well. They will be described in detail in section 3.3. To implement these unit tests we have used the *GoogleTest* library [22].

Overall we have tried to maintain a high software quality level by complying to the *C++ Core Guidelines* [23] and using static code analysis tools like *Sonarlint* [24]. This also includes memory management where we used higher-level constructs of the C++ standard library instead of raw memory management known from C like smart pointers instead of `malloc()`. These decisions were made with performance in mind, as we considered memory and runtime overhead introduced by these constructs.

To build and run our software on the High-performance computing (HPC) cluster given to us, we implemented a build definition for the HPC package management system *Spack* [12]. This definition can be seen in the appendix B.1. Using this definition, the software may be built and configured using Spack, allowing users to compile it using different compilers or dependency providers e.g. for MPI. Using our Spack definition it may also be compiled using *Score-P* [25] to gain insights regarding MPI behaviour later on in the evaluation.

## 3.2 General Structure

To give an idea of our implementation's structure, we would like to give a short explanation of object orientation and inner class structure aspects.

Our whole implementation builds upon object orientation principles and follows various coding guidelines like the *SOLID* principles [26]. This means, for example, that most aspects of our implementation are encapsulated inside classes and objects. For example, to represent the world state as described in section 2.2.1, we have implemented a class called `WorldState` and to generate world maps, we have encapsulated the necessary logic inside the `WorldGenerator` class. The most important part of our implemented class hierarchy is the agent hierarchy, as it represents the core of our implementation's extensibility. Figure 3.1 shows this hierarchy.

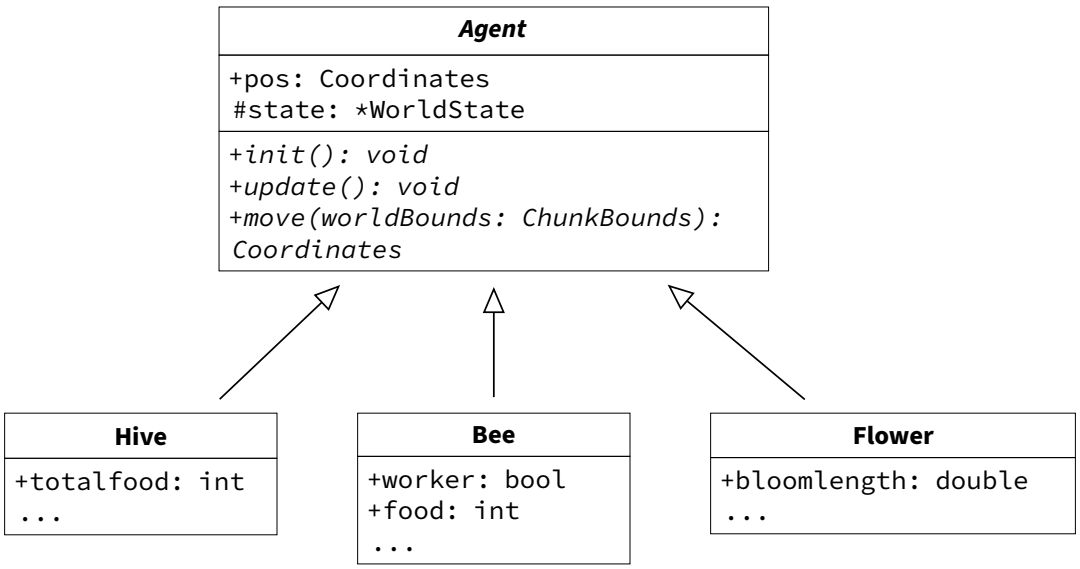



Figure 3.1: UML class diagram of the agents

As already stated, one of our goals was to create an extensible simulation that could be expanded by further agents. This is possible using the class hierarchy design shown in figure 3.1. To create a new agent, the class `Agent` has to be inherited by the new agent's class, which can then be instantiated in the simulation. To do so, additional logic in the seeding process of the application and in the other agent's logic is necessary, but may be added as needed.

## 3.3 Parallelization and Performance-Focused Details

The most important aspect of our implementation is its performance and efficiency. As described in chapter 2.3, we chose to parallelize certain parts of our application to improve those aspects. But parallelization is not the only idea we had to achieve this. Performance

and efficiency improvements may also be observed when using suitable data structures and algorithms. This section gives an overview of certain implementation details regarding this topic.

### 3.3.1 OpenMP

A simple procedure we were able to improve using parallelization is world generation. In this procedure, many calculations have to be done on a grid, which can be seen as a matrix. These calculations are cell-wise and do not depend on other cells. This is why we were able to parallelize some of this procedure using Open Multi-Processing (OpenMP). OpenMP gives us the possibility to parallelize simple for loops using compiler directives. Those for loops are used multiple times in the inner world generation steps and are ideal for OpenMP-based parallelization. Listing 3.3.1 shows the usage of OpenMP directives in the world generation procedure. Using this parallelization technique, parts of the world generation are executed in parallel on multiple cores instead of purely sequentially, which results in a large time efficiency improvement, depending on the cores used.

```
#pragma omp parallel for
for (std::size_t i = 0; i < displacementMap.size(); i++) {
  for (std::size_t j = 0; j < displacementMap[i].size(); j++) {
    displacementMap[i][j] = std::pair<int, int>(
        clip(i + this->edgeDisplacement *
                    noiseGenerator.fractal(this->perlinOctaves,
                                           (i + 0.1) / scale, j / scale,
 ↪  1),
             0, this->config.size - 1),
        clip(j + this->edgeDisplacement *
                    noiseGenerator.fractal(this->perlinOctaves,
                                           (i + 0.1) / scale, j / scale,
 ↪  0.5),
             0, this->config.size - 1));
  }
}
```

Listing 3.3.1: Excerpt from the world generation showing the usage of OpenMP directives

### 3.3.2 CTD-Array

One problem we have faced during development are two-dimensional arrays in C++. Arrays in C/C++ can have a static size by declaration and definition at compile time,

resulting in a stack-allocated array, or may have a dynamically defined size by allocating it on the heap at runtime. To make our simulation configurable, we had to use dynamically defined arrays as the world map data is represented by such an array and the map's size may be given at runtime using command-line parameters. Using one-dimensional dynamically allocated arrays is trivial in C++, but the world map is represented by a two-dimensional array where every dimension represents one coordinate dimension. In C++, multiple ways exist to declare two-dimensional arrays and we chose an efficient and MPI-compatible way: representing a two-dimensional array in a one-dimensional array with special indexing. Representing a two-dimensional array, using an array of pointers where each pointer is initialized with its own independent array has many disadvantages, like the possibility of being scattered across the memory, resulting among other things in cache lookup drawbacks. Besides that, the two-dimensional data would not be contiguous, which would make passing it using MPI very complicated as MPI operates on low-level contiguous memory to send and receive data. This is why we implemented the Contiguous, two-dimensional array (CTD-Array). This two-dimensional array implementation uses the described method for keeping the data in contiguous memory while enabling two-dimensional access to the data, with the only drawback of having a small computational overhead for indexing as the two-dimensional index has to be translated to an one-dimensional index first. This is an important implementation detail that takes part in our MPI communication described later on.

### 3.3.3 *k*-d Tree

One of the most important performance-related implementation details we had to consider was the technique for structuring the agents. In the world state, consisting of the world map and the agents, we must keep the agents in any kind of data structure to access them. One trivial solution would be a simple vector of dynamic size with every agent in it in no particular order. This solution would make it easy to add and remove agents, but it has one major drawback. Our agent's behavior depends heavily on the agent's surroundings, i.e. the surrounding agents. This means that we have to implement an efficient way to store the agents that enables efficient spatial queries.

For this purpose, we have decided to implement a $k$-dimensional tree ($k$-d tree) which is a multidimensional binary search tree offering good time complexity for multidimensional search keys instead of one-dimensional ones [27]. In our case, we use a fixed dimension of 2 because our agents are moving on a two-dimensional plane, which is why we call our implementation *Point Tree* or *2-d tree*. The position, i.e. the coordinates of the agents,

are the index of this tree, while (smart) pointers to the actual agents are the inner value. This enables us to index the agents using their position and query certain information using specific algorithms.

One of these algorithms is the nearest neighbor search. This enables agents to query the nearest other agent in relation to a given point. This can be helpful in a number of situations for example, when a bee wants to know which other agent is closest, to make behavioral decisions. By using a $k$-d tree or more specifically a 2-d tree, the average time complexity for this operation is $O(\log n)$ which is a good complexity when compared to linear search on an unordered multidimensional array.

Another important algorithm that is used by agents is range search. This search queries for all agents in a given range from a given point. This enables, for example, bees to search for flowers in their perceptive range. The algorithm is also much more efficient on a $k$-d tree than a simple vector.

One drawback of the 2-d tree is that we have to keep it balanced, or the algorithms operating on it may become worse in terms of time complexity. This means that we have to rebuild the tree after every tick, which creates a perfectly balanced tree but at the cost of computational overhead. The overhead of this operation is not neglectable, but the time saved by using the $k$-d tree algorithms for, e.g., querying compensates this overhead. Besides that, the tree's inner data structure uses shared pointers, a specific kind of the standard library's smart pointers, to manage the inner data's lifetime and memory. This enables cleaner memory management as the data is automatically cleaned up when no other part of the application references it, but again at the cost of computational as well as memory overhead.

### 3.3.4 MPI

The core of our performance optimization efforts is the parallelization of the simulation's main part using MPI. This main part is the so-called ticking system and was described in detail in section 2.2.1. The idea of parallelizing this step is to split the world map into smaller parts of equal size called chunks. One chunk is then assigned to each process, and each process handles agents positioned in the respective chunk. This partitioning into chunks is an implementation detail and should not have any impact on the simulation's logic, meaning that when agents cross chunk borders, they should get transferred to the corresponding chunk's process. Because of this, we needed to implement the possibility of transferring agents and their data among processes.

We implemented this functionality using MPI. MPI offers functions to send messages

between processes, which may even be distributed among different computing nodes. We use it in the ticking process for exactly the functionality described above. Every chunk has its own world state with its corresponding agents. For each tick, every chunk calls the `update()` and the `move()` methods of all its agents. The `move()` method returns a new position, so the new chunk can be calculated to move the agent to if necessary. Agents that should be moved are collected and then returned to the main function, which is where MPI is used for the data exchange.

The main function is where the simulation's main process is handled. As it is duplicated for every MPI process, every process represents one chunk and has its own world state. The ticks are run in a loop, which runs the configured tick count. In this loop, besides executing the actual tick phase, agents are exchanged. It is important to notice that this exchange has only been implemented for bees yet because of the way MPI handles types and data to send or receive. The exchange process using MPI functionality is straightforward: for every process (the receiver), every other process (the sender) collects all the agents it wants to send to the receiving process. Because of the way MPI is implemented, they first exchange the count of agents and, therefore, the size of data to send or receive. After that, every sender sends its agents to the specific receiver using `MPI_Send()`, and the receiver collects all of this sent data using `MPI_Gatherv()`. Then, the internal world state of the receiver can be updated by inserting the new agents in their desired position. This process happens every tick and enables the processes to change their data, maintaining a consistent and parallelized overall world state. Listing B.2 shows the relevant MPI code that is used for this procedure.

MPI is also used to exchange the world map. Not every process has to generate the world map itself. It is sufficient that one process generates the world map alone, using the parallelization described earlier, and then sends the resulting map to the other processes.

# 4 Evaluation

This chapter aims to answer whether our methodology and implementation achieved the goal of successfully parallelizing most or all of the simulation workload.

## 4.1 Parameters for Benchmarking

The scenario chosen for every benchmarking process can have an immense impact on the results and has to be chosen wisely. Since our parallelize approach relies on splitting the map into sectors and then calculating the sectors individually, the distribution of the hives is very important for performance because if the load is distributed very unequally, some processes take much longer while the others are blocked. This is obviously bad for benchmarking purposes because the choice of scenario might blur the results or even change them. Therefore, we decided to repeat every benchmark 10 times and take the worst, best and average of all results. All benchmarks measured the time it took to perform the first 1000 ticks without the initial creation of the map and the first three ticks. The initial creation of the map wasn't included because it was neither the focus of this project nor did we put much effort into parallelizing it. Furthermore, the map only needs to be created once, so the creation time would become more insignificant the longer the simulation runs. For weak and strong scaling, we measured the execution time for two scenarios each. The large-scale test is where we measure how the application does with many processes, and the small-scale test is to test how big the differences between the different divisions of the map are. For the large scale test, we chose $2^x$ with $x \in \mathbb{N}$ and for the small scale test, the range of 1–10.

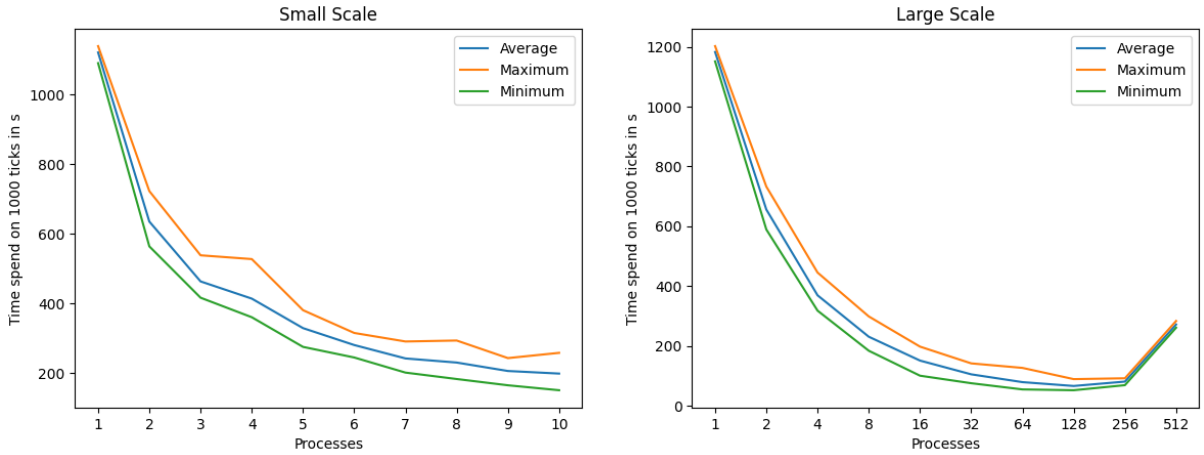
## 4.2 Classification of the Scaling type

Before the benchmarking process, we strongly assumed that the simulation is more bound by computational power than by memory. This was confirmed by the collected data. The memory footprint of our application only consists of the map and the agents. The map is

constant and, depending on the accuracy, between a couple of hundred megabytes and, in extreme cases, up to a gigabyte in size. Every agent in our current scenario is smaller than 30 bytes, and there isn't much reason to expect that agents added in the future will be a lot more complex. Even if we assume the extreme case where a single node had to calculate 100 hives alone ( 50 million agents), the agent footprint would only be 150MB (100 hives · 50000 Agents · 30 bytes = 150MB). This kind of intense workload per task is far from realistic because so many hives would never be in such proximity, and the performance would suffer immensely, to the point where the simulation would be unusable. Benchmarks can be found at **lst:times**, but won't be discussed because they benchmark a non-performance-relevant area of our application.

## 4.3 Strong Scaling

Strong scaling uses a variable number of processes tested against a constant scenario. We tried to use a large, but close to reality setup, which might be used by scientists and creates stable results. Small scenarios generally create more unstable results, because the distribution between the processes will be not ideal, but also can often be calculated by a single core with decent enough speed anyway, therefore opted for using 64 hives and 30000 flowers.



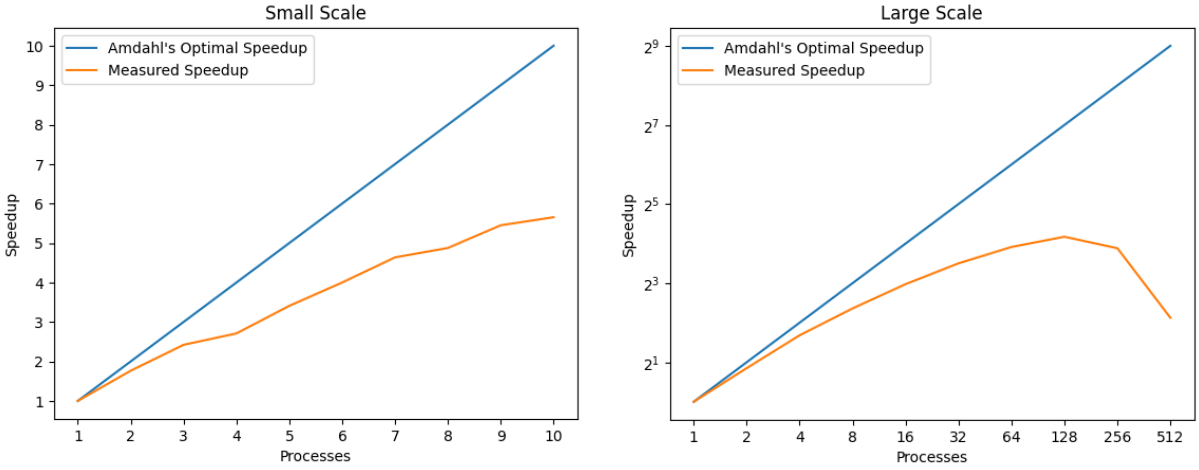

(a) (b)

Figure 4.1: Benchmark time to complete 1000 ticks

The data clearly indicates that our application scales. In this scenario, the run time can be increased by 300% by increasing the number of processes from 1 to 4 and another

250% from 4 to 16 processes (400% would be optimal in both cases). This shows that a high degree of the computations are calculated in parallel; unfortunately, the returns on increasing the processor count diminish a little. An explanation might be that increased communication between processes because of shrinking sector size is driving this decline. Alternatively, it might just be the case that the percentage of sequential execution time is increasing because the parallel part is decreasing (because of more processes).

A very positive result of our benchmark is that the worst and best cases are rather close to the mean, especially if there are more processes than hives. We were worried about the fact that we might have very good performance in ideal scenarios with a very equal distribution but very poor performance in the average scenario.




(a) (b)

Figure 4.2: Speedup ($\frac{t(1)}{t(N)}$)

The speedup graph shows that there must be a part of the problem we didn't manage to parallelize. This can be observed in the "Large Scale" graph, which looks a little like a root function. This happens when some portion of the problem isn't parallelized, and then the graph slowly converges to some value because there can be no further gains in speed by adding more processes. The sequential part is likely due to the increasing need for MPI communication, which decreases the effectiveness of further processes.

## 4.4 Amdahl's Law

Amdahl's Law of Strong Scaling states that the optimal speedup of an application can not surpass the number of parallel processes it uses. This "Law" is based on the equation:

$$\frac{1}{s + \frac{p}{N}} = su$$

where:

$$\begin{aligned} su &= \text{Speedup} \\ s &= \text{Fraction of the sequential code} \\ p &= \text{Fraction of the parallel code} \\ N &= \text{Number of Processes} \end{aligned}$$

From there follows:

$$\begin{aligned} su &= \frac{1}{s + \frac{p}{N}} \\ 1 &= su \cdot (s + \frac{p}{N}) \\ 1 &= s \cdot su + p \cdot \frac{su}{N} \end{aligned}$$

Furthermore logically follows that:

$$1 = s + p$$

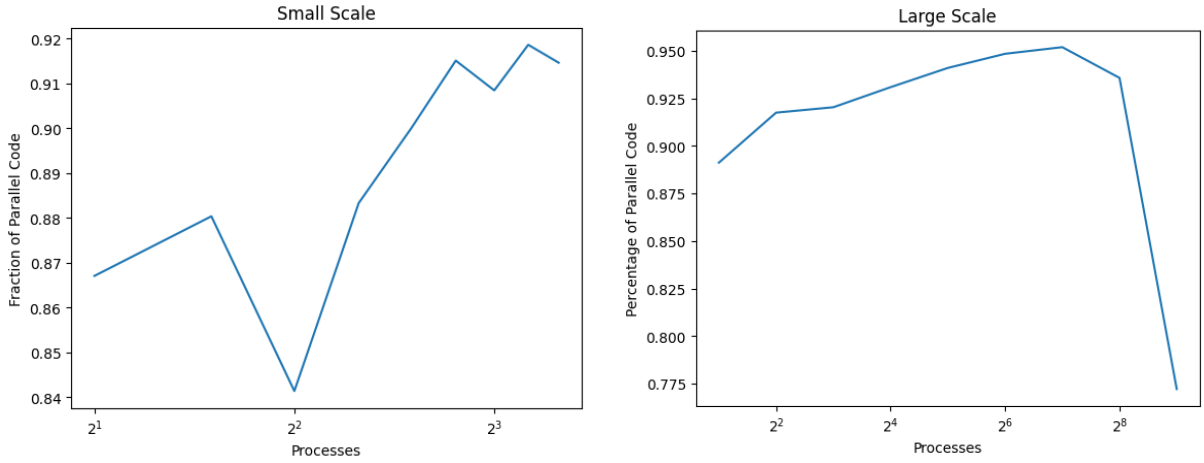
Since we know all variables except s and p, we can calculate them by solving:

$$\begin{pmatrix} 1 & 1 \\ su & \frac{su}{N} \end{pmatrix} \cdot \begin{pmatrix} s \\ p \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

This equation is, in this context, solvable as long as $N > 1$ and $N >= su$. $N$ can't be one because then $su$ equals one, the matrix is singular, and the result is no longer well-defined. Additionally, a single process is necessarily sequential. $su > N$ would not only violate

Amdahl's law, but would also result in $x_s$ being a negative.

We calculated the percentage of parallel code for all values displayed in figure 4.2.




(a) (b)

Figure 4.3: Percentage of Parallel Code

This plot surprisingly shows that the percentage of parallel code increases slowly, with a sharp drop at the end. What we expected to happen is that the percentage of parallel code is constantly dropping because communication between the processes is increasing. It seems like by far the most performance-relevant factor is the distribution of the agents among the processes, not the inter-process communication, and only after the former is solved does the latter impact the results at all. This would explain why the performance suddenly drops so strongly when increasing the processors from 256 to 512. At 256, the processors are already distributed perfectly (4.1 shows that at 256 processes, the worst and best cases are almost the same, unlike at 128), so increasing the processors only results in an increase in communication time, not in any more speedup. What doesn't fit this hypothesis is the big decline at $2^2$ and the smaller decline at $2^3$ (these persisted, after rerunning the benchmark). We couldn't identify the reason why these drops exist, but they seem to get less significant with an increased number of processors. While these drops in the percentage of parallel code don't point towards the distribution being the primary driver of the increase in the percentage of parallel code, they also don't point towards inter-process communication. If inter-process communication was the problem, we would expect the drops to get more severe with an increased number of processors, not better. Assuming the distribution is the primary problem, not the communication, one way to fix this problem could be to adjust sector size to the position of the hives and

therefore the agents and not just split the map into equally large pieces.

## 4.5 Score-P/Vampir

To understand the bottlenecks of our application better, we decided to use Score-P for tracing and visualize the results with Vampir. We were primarily interested in how well the parallelization works in unequal distribution scenarios and how the workload is distributed between the functions. We tested with 2 processes and with 2 hives for 1000 ticks.

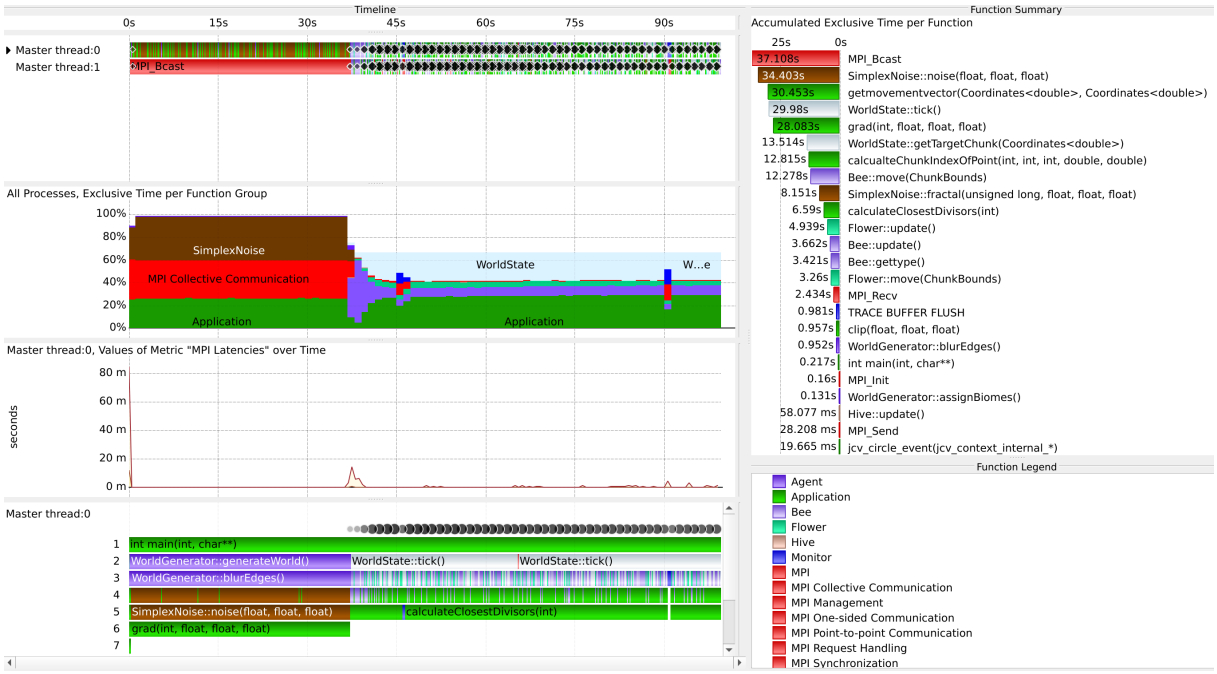



Figure 4.4: Vampir output: 2 hives

The graphic shows, on the top left, a diagram that chooses which family of functions are executed at what time. Below that, there is a graph that shows the percentage time of each family of functions. Below that, there is a graph of the MPI latency. On the right, there is a graphic of how much time each function consumed.

The first portion of the graph can be ignored because it is just the map creation phase. After that, you can see that there is a wider purple portion that gets smaller. These are the initial ticks, which are a lot slower than the others. This happens because the range queries every agent performs return thousands of agents, which slows down their moves significantly. After that, the workload in 4.4 can be roughly divided into 30% “World state”, 20% Agent moves and MPI and 50% “Application”. What is listed on the right

as `tick()` responds primarily to the rebuilding and sorting of the agents. We never even thought of that part as very performance-relevant to begin with. Another huge surprise was the `getmovementvector()` function. This function essentially only returns the new value of the moved agent and is performed once per move by every bee. It is quite unclear to us why this is supposed to be the most performance relevant function. The only thing we can possibly fathom is that the random numbers that are generated in this function are the cause of this performance loss. The latency graph shows that for the balanced version, no real latency's.

# 5 Challenges

This chapter shortly touches on challenges we faced during our project. These include technical as well as organizational aspects.

## 5.1 Validating the Approach

A difficult challenge in our project was that the benchmark and validation process came at the very end. We did not really have any chance to get a feeling for our performance until we were almost done with our project. This creates the constant worry that things might not work, or the performance might not increase with more processes. Since most of the performance, at least in our case, is not in the fine-tuning, but in the general parallelization strategy, this makes the initial planning phase much more important. If your approach to solving the problem is flawed, you will only know after all the work is done.

## 5.2 Compilation and Execution on the Cluster

Another big problem of our solution's implementation and validation phase was the compilation and execution on the HPC-system given to us. We faced multiple challenges while trying to do so, but were able to overcome them with a great investment of time.

The first problem we had to solve were severe compilation errors because of outdated system-level *Binutils*. Binutils is a software collection by the GNU Project that includes various binary tools, particularly *ld*, the GNU linker. As Spack uses the system-level Binutils and therefore ld by default, this caused many compilation errors in various dependency packages of our simulation because the system-provided Binutils are outdated. Using Spack, we were able to install a recent version of the Binutils and ld. This newer version can be used for the Spack compilation using "dirty" compilation environments where those tool versions are used instead of the system-provided tools. Another issue we had in this context was an outdated system-provided GNU Compiler Collection (GCC) which has also been installed and replaced by a newer version. For this, no "dirty"

environment was needed, as Spack natively supports the usage of different compilers for compilation.

Initially, we used Meson as our build system, but Meson currently suffers from a bug which causes wrong linker flags being passed to ld, resulting in our next problem. After investigating a long time, we were unable to find a working solution using Meson. This is why we reimplemented our build system using CMake. In the end, CMake offers similar functionality to Meson, but at the cost of developer experience as CMake build definitions are harder to write. For example, Meson offers exceptional functionality for finding dependencies and linking to them, whereas CMake needs some extra work for libraries that do not offer CMake definitions like cairomm.

## 5.3 Complexity of the Point Tree

The *Point Tree*, a two-dimensional $k$-d tree, is our solution for efficient spatial storage and querying of the agents (see section 3.3). This data structure is no trivial concept to understand or implement and was also a big challenge for us. We needed a high amount of time understanding it in the first place and implementing it. Since sources for an implementation are rare, we had to design and implement most of the Point Tree ourselves, which was not an easy task for us as we had no great experience in C++ and this data structure. We had to consider functional correctness as well as complexity. This complexity includes algorithmic and space complexity, which was also an important aspect to us. To accomplish these requirements, we implemented numerous unit tests and considered many established C++ concepts for memory safety and efficiency, like smart pointers.

## 5.4 Prioritization and Structure

Maybe the biggest struggle and cause for delays and complications during implementation of our project was not setting the right priorities. Especially at the start, we had too big a scope of the project in mind. We started and spent a lot of time on implementing an algorithm that generates realistic looking maps, which could have been a great HPC-project by itself. An aspect of the Project we eventually almost didn't use at all. Then we started developing the structure of the simulation and the point tree in parallel, which meant that the simulation had to be essentially written blind and later verified and revised. Another suboptimal implementation detail that arose from that implementation order or the lack thereof is that the position gets stored twice once in the tree and another time in the

Elements stored in the Tree. It is improbable that this ever causes problems because we rebuild the tree every tick, and the positions to rebuild are extracted from the agents itself. But if this detail is forgotten and should ever in the future create unwanted behaviour, it will be difficult to fix and extremely hard to find.

In general, it would have been very helpful to create a simplified running version at a much earlier stage and then improve upon it, rather than programming essentially finished parts and then assembling them.

## 5.5 Documentation

Many if not all the HPC-tools are not as user-friendly as the tools in many other fields of computer science, for example machine learning. This is likely due to HPC being a smaller and highly specialised field, but for beginners there are many pitfalls and struggles around the tools. For every tool we used, there is surface-level information available, which was partly provided in the lecture. But if, for example, an installation fails or something is incompatible, there are often just the manuals, which are sometimes thousands of pages long and technically written, if existing at all.

# 6 Conclusion

The project has provided a good and efficient approach and illustrates a way how to simulate complex eco-systems. It could be proven that the chosen model works well for the investigation of influence factors on animals. With more time and resources model and methods could have been refined and tailored to the given problem-statement. We presented a number of ways how to improve the simulation setting even though measured performance exceeded our expectations. Especially, the integration of MPI went very smoothly. Part of the reason for that is, that we spent lots of time at the beginning discussing and planning every HPC related detail, which definitely was vital. We got a good overview of available tooling and techniques used in HPC and got to use most of them on our own project. We presented a number of ways, how to improve the performance, but the measured performance already surpassed our expectations.

# Bibliography

[1] D. M. S. Matias, J. Leventon, A.-L. Rau, C. Borgemeister, and H. von Wehrden, "A review of ecosystem service benefits from wild bees across social contexts," *Ambio*, vol. 46, no. 4, pp. 456–467, Nov. 2016. DOI: 10.1007/s13280-016-0844-z. [Online]. Available: https://doi.org/10.1007/s13280-016-0844-z.

[2] R. Rader, I. Bartomeus, L. A. Garibaldi, *et al.*, "Non-bee insects are important contributors to global crop pollination," *Proceedings of the National Academy of Sciences*, vol. 113, no. 1, pp. 146–151, 2016. DOI: 10.1073/pnas.1517092112. eprint: https://www.pnas.org/doi/pdf/10.1073/pnas.1517092112. [Online]. Available: https://www.pnas.org/doi/abs/10.1073/pnas.1517092112.

[3] M. Niazi and A. Hussain, "Agent-based computing from multi-agent systems to agent-based models: A visual survey," *Scientometrics*, vol. 89, no. 2, pp. 479–499, Aug. 2011. DOI: 10.1007/s11192-011-0468-9. [Online]. Available: https://doi.org/10.1007/s11192-011-0468-9.

[4] M. P. I. Forum, *MPI: A message-passing interface standard version 4.0*, Jun. 2021. [Online]. Available: https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf.

[5] E. Gabriel, G. E. Fagg, G. Bosilca, *et al.*, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, Sep. 2004, pp. 97–104.

[6] L. Dalcín, R. Paz, and M. Storti, "MPI for python," *Journal of Parallel and Distributed Computing*, vol. 65, no. 9, pp. 1108–1115, Sep. 2005. DOI: 10.1016/j.jpdc.2005.03.010. [Online]. Available: https://doi.org/10.1016/j.jpdc.2005.03.010.

[7] F. Zehra, M. H. Javed, D. Khan, and M. M. Pasha, "Comparative analysis of c++ and python in terms of memory and time," 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:231630242.

[8] The Meson Development Team. "The meson build system." (2023), [Online]. Available: https://mesonbuild.com/index.html (visited on 09/17/2023).

[9] A. Zucconi. "The world generation of minecraft." (Jun. 2022), [Online]. Available: https://mesonbuild.com/index.html (visited on 09/17/2023).

[10] Kitware, Inc. and Contributors. "Cmake." (2023), [Online]. Available: https://cmake.org/ (visited on 09/18/2023).

[11] NixOS contributors. "Nix & nixos." (2023), [Online]. Available: https://nixos.org/ (visited on 09/18/2023).

[12] T. Gamblin, M. LeGendre, M. R. Collette, *et al.*, "The Spack Package Manager: Bringing Order to HPC Software Chaos," ser. Supercomputing 2015 (SC'15), LLNL-CONF-669890, Austin, Texas, USA, Nov. 2015. DOI: 10.1145/2807591.2807623. [Online]. Available: https://github.com/spack/spack.

[13] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf, *Computational geometry*, en, 2nd ed. Berlin, Germany: Springer, Feb. 2000.

[14] M. Westerdahl, W. Leong, and D. Avedissian. "jc_voronoi." (2023), [Online]. Available: https://github.com/JCash/voronoi (visited on 09/19/2023).

[15] Cairo contributors. "Cairo." (2023), [Online]. Available: https://www.cairographics.org/ (visited on 09/19/2023).

[16] Cairomm contributors. "Cairomm." (2023), [Online]. Available: https://www.cairographics.org/cairomm/ (visited on 09/19/2023).

[17] K. Perlin, "Noise hardware," *Real-Time Shading SIGGRAPH Course Notes*, 2001.

[18] S. Rombauts, J. Belmonte, M. K. Duncan, and A. Gray. "Perlin Simplex Noise C++ Implementation (1D, 2D, 3D)." (2019), [Online]. Available: https://github.com/SRombauts/SimplexNoise (visited on 09/19/2023).

[19] G. Melman *et al.* "spdlog." (2023), [Online]. Available: https://github.com/gabime/spdlog (visited on 09/19/2023).

[20] J. Beck *et al.* "cxxopts." (2023), [Online]. Available: https://github.com/jarro2783/cxxopts (visited on 09/19/2023).

[21] E. Fiselier, D. Hamon, R. Lebedev, *et al.* "Benchmark." (2023), [Online]. Available: https://github.com/google/benchmark (visited on 09/19/2023).

[22] Google Inc. and contributors. "GoogleTest." (2023), [Online]. Available: https://github.com/google/googletest (visited on 09/19/2023).

[23] B. Stroustrup and H. Sutter. "C++ Core Guidelines." (Apr. 2023), [Online]. Available: https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines.html (visited on 09/20/2023).

[24] SonarSource S.A. "Sonarlint." (2023), [Online]. Available: https://www.sonarsource.com/products/sonarlint (visited on 09/20/2023).

[25] Score-P Developer Community, *Scalable performance measurement infrastructure for parallel codes (Score-P)*, 2023. DOI: 10.5281/ZENODO.7817192. [Online]. Available: https://zenodo.org/record/7817192.

[26] R. Martin, *Agile Software Development, Principles, Patterns, and Practices*. 2013, p. 532, ISBN: 9781292025940.

[27] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, no. 9, pp. 509–517, Sep. 1975, ISSN: 0001-0782. DOI: 10.1145/361002.361007. [Online]. Available: https://doi.org/10.1145/361002.361007.

# Appendix

## A Work sharing

Programming work as well as the problem approach and problem solving was done by both authors across the entire Project. The primary working areas are listed below.

- Henrik Jonathan Seeliger
  - Build system (Spack, Meson, CMake, ScoreP)
  - Point Tree
  - MPI
- Georg Eckardt
  - Agent Logic
  - Benchmarking

# B Code samples

```python
from spack.package import *

class BeeSimulation(CMakePackage):
    """Project as part of the course "Practical Course on High-Performance Computing" (summer term 2023) for simulating bees."""

    homepage = "https://github.com/HnSee/hpc-2023-bee-simulation"
    git = "https://github.com/HnSee/hpc-2023-bee-simulation.git"

    version("master", branch="master")

    variant("scorep", default=False, description="Compile using Score-P")

    depends_on("binutils", type="build")

    depends_on("openmpi")
    depends_on("cairomm")
    depends_on("cxxopts")
    depends_on("spdlog")
    depends_on("fmt")
    depends_on("googletest")
    depends_on("benchmark")

    depends_on("scorep", when="+scorep")

    @when("+scorep")
    def install(self, spec, prefix):
        cmake_args = ["-DCMAKE_CXX_COMPILER=scorep-g++"]
        cmake_args.extend(std_cmake_args)

        with working_dir("build", create=True):
            cmake.add_default_env("SCOREP_WRAPPER", "off")
            cmake("..", *cmake_args)
            make("SCOREP_WRAPPER_INSTRUMENTER_FLAGS=--thread=omp")
            make("install")
```

Listing B.1: Spack build definition for the simulation

```cpp
// Transfer necessary agents
for (int receiver = 0; receiver < processes; ++receiver) {
  std::vector<Bee> beesToTransfer;
  // ...
  Bee *beesToReceive;

  if (rank == receiver) {
    std::vector<int> sizes(processes);
    std::vector<int> displs(processes);

    for (int sender = 0; sender < processes; ++sender) {
      int count = 0;
      if (sender != receiver) {
        MPI_Recv(&count, 1, MPI_INT, sender, 0, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);
      }
      sizes[sender] = sizeof(Bee) * count;
    }

    // ...
    beesToReceive = new Bee[sum];
    // ...

    MPI_Gatherv(nullptr, 0, MPI_BYTE, beesToReceive, &sizes[0], &displs[0],
                MPI_BYTE, receiver, MPI_COMM_WORLD);

    // Create Bee structures and insert into world state
    // ...
    delete[] beesToReceive;
  } else {
    auto count = static_cast<int>(beesToTransfer.size());
    MPI_Send(&count, 1, MPI_INT, receiver, 0, MPI_COMM_WORLD);

    int size = count * sizeof(Bee);

    MPI_Gatherv(&beesToTransfer[0], size, MPI_BYTE, nullptr, nullptr,
                nullptr, MPI_BYTE, receiver, MPI_COMM_WORLD);
  }
}
```

Listing B.2: Usage of MPI for transferring agents between chunks

# C Figures


Figure C.1: Vampir output: 3 hives 2 Processes

# D Tables

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1192 | 1197 | 1157 | 1182 | 1194 | 1151 | 1202 | 1193 | 1160 | 1198 |
| 2 | 732 | 620 | 589 | 695 | 692 | 609 | 718 | 643 | 637 | 620 |
| 4 | 388 | 318 | 317 | 445 | 418 | 366 | 429 | 332 | 324 | 351 |
| 8 | 203 | 183 | 217 | 270 | 210 | 298 | 247 | 225 | 184 | 265 |
| 16 | 136 | 160 | 145 | 172 | 131 | 197 | 160 | 116 | 100 | 188 |
| 32 | 115 | 126 | 86 | 100 | 89 | 141 | 97 | 88 | 75 | 128 |
| 64 | 76 | 101 | 84 | 63 | 73 | 126 | 82 | 54 | 60 | 66 |
| 128 | 80 | 89 | 52 | 74 | 66 | 68 | 54 | 51 | 59 | 63 |
| 256 | 75 | 68 | 80 | 88 | 88 | 91 | 83 | 77 | 72 | 79 |
| 512 | 263 | 271 | 283 | 281 | 271 | 270 | 281 | 265 | 267 | 261 |

Table D.1: Strong-Large Data

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1136 | 1137 | 1102 | 1121 | 1121 | 1088 | 1135 | 1127 | 1095 | 1128 |
| 2 | 676 | 589 | 587 | 699 | 629 | 563 | 721 | 640 | 631 | 602 |
| 3 | 415 | 453 | 507 | 537 | 454 | 451 | 462 | 470 | 446 | 424 |
| 4 | 407 | 373 | 376 | 526 | 437 | 383 | 507 | 359 | 383 | 374 |
| 5 | 293 | 364 | 380 | 318 | 314 | 339 | 294 | 343 | 274 | 362 |
| 6 | 244 | 265 | 270 | 275 | 311 | 314 | 251 | 292 | 269 | 304 |
| 7 | 256 | 211 | 272 | 262 | 216 | 290 | 200 | 209 | 220 | 272 |
| 8 | 188 | 187 | 210 | 265 | 238 | 292 | 234 | 234 | 182 | 260 |
| 9 | 239 | 201 | 242 | 201 | 222 | 201 | 164 | 218 | 190 | 172 |
| 10 | 182 | 172 | 242 | 208 | 205 | 210 | 257 | 187 | 150 | 163 |

Table D.2: Strong-Small Data

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 56.9 | 63.1 | 57.0 | 69.2 | 63.0 | 62.2 | 61.9 | 66.6 | 61.9 | 69.7 |
| 2 | 104.0 | 99.2 | 78.2 | 110.0 | 76.2 | 78.4 | 101.7 | 87.5 | 101.3 | 93.6 |
| 4 | 111.0 | 112.7 | 107.7 | 146.6 | 98.8 | 100.8 | 143.5 | 111.6 | 104.8 | 120.3 |
| 8 | 159.5 | 119.6 | 152.7 | 143.4 | 199.7 | 155.5 | 179.0 | 139.0 | 134.1 | 150.6 |
| 16 | 178.5 | 178.1 | 159.4 | 176.1 | 172.7 | 213.4 | 164.3 | 169.7 | 120.3 | 219.1 |
| 32 | 234.1 | 190.0 | 179.3 | 245.7 | 273.0 | 297.4 | 243.4 | 198.5 | 188.4 | 190.8 |
| 64 | 229.3 | 219.9 | 200.5 | 283.4 | 238.2 | 313.1 | 216.8 | 208.4 | 226.9 | 229.7 |

Table D.3: Weak-Large Data

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 65.8 | 72.0 | 64.9 | 77.0 | 70.9 | 70.4 | 68.6 | 76.3 | 69.0 | 76.7 |
| 2 | 89.4 | 109.8 | 72.0 | 121.6 | 78.6 | 82.7 | 88.1 | 99.1 | 87.8 | 92.5 |
| 3 | 107.6 | 115.9 | 85.9 | 86.5 | 84.7 | 98.2 | 91.2 | 141.9 | 112.0 | 84.4 |
| 4 | 120.7 | 108.0 | 98.7 | 122.4 | 107.8 | 104.1 | 109.0 | 103.4 | 86.9 | 109.5 |
| 5 | 179.8 | 134.3 | 92.8 | 111.5 | 116.7 | 118.2 | 110.4 | 121.8 | 100.2 | 99.0 |
| 6 | 210.3 | 125.1 | 123.4 | 137.5 | 130.0 | 140.4 | 160.8 | 107.7 | 136.1 | 165.5 |
| 7 | 150.8 | 103.4 | 101.5 | 121.3 | 160.1 | 124.2 | 101.0 | 110.4 | 95.8 | 140.1 |
| 8 | 185.2 | 101.8 | 125.3 | 125.4 | 177.4 | 143.9 | 179.4 | 114.8 | 120.9 | 145.1 |
| 9 | 135.0 | 136.7 | 154.6 | 154.5 | 148.9 | 151.5 | 138.5 | 132.3 | 117.7 | 156.5 |
| 10 | 141.2 | 186.2 | 127.0 | 137.2 | 147.4 | 190.6 | 136.1 | 146.9 | 138.3 | 172.5 |

Table D.4: Weak-Small Data