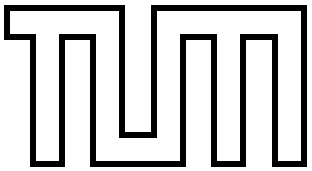


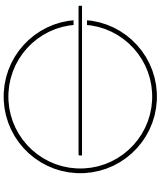
FAKULTÄT FÜR INFORMATIK

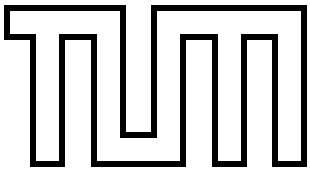
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Implementation and Benchmarking of Adaptive Tree-based Particle Containers for AutoPas

Johannes Spies





FAKULTÄT FÜR INFORMATIK

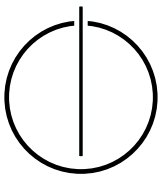
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Implementation and Benchmarking of Adaptive Tree-based Particle Containers for AutoPas

Implementierung und Benchmarking von Adaptiven Baum-basierten Partikelcontainern für AutoPas

Author:	Johannes Spies
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisor:	Fabio Alexander Gratl, M.Sc.
Date:	15.09.2021



I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, 15.09.2021 Johannes Spies

Acknowledgements

Thank you, Fabio, for your great support in all phases of this thesis. I am grateful for the inspiration you gave me, the great discussions and pleasant atmosphere during our meetings. Thank you for replying my mails even at 1am.

Thank you, Jakob, for your excellent expertise in `C++`. Thank you, Fabian, for the great discussions and your help. Thank you for supporting me throughout my thesis, Isi.

Thank you, Isi, Fabian, Martin and Fabio, for proof-reading my thesis.

Abstract

An octree is a space-adaptive data structure. This thesis shall demonstrate the integration of a new octree-based particle container for the `AutoPas` simulation library. The new particle container is intended to serve as an additional option to let `AutoPas`'s special tuner unit choose from in order to find a better, truly optimal simulation configuration. Detailed insight regarding the implementation of the octree container is given as well. Furthermore, the performance of the new container is compared to an existing linked cells implementation. It could be shown that the octree container is working as intended, but it falls behind the existing, optimized containers by a factor of 3 in terms of speed. Tools that were developed during the implementation are presented as well. The outlook provides concrete ideas and inspiration on how the existing implementation can be improved and further analysis can be conducted.

Zusammenfassung

Ein Octree ist eine Datenstruktur, die sich den räumlichen Gegebenheiten anpasst. Diese Arbeit beschreibt die Integration eines neuen, Octree-basierten Parikelcontainers in die Simulations-Bibliothek `AutoPas`. Der neue Container wird von der in `AutoPas` vorhandenen Tuner-Einheit ausgewählt um eine optimale Konfiguration für ein Experiment zu finden. Dabei wird die Implementierung sehr detailliert beschrieben. Desweiteren werden Geschwindigkeitsmessungen des neuen Containers und Vergleiche mit einem bereits existierenden Linked-Cells-Container beschrieben. Insgesamt wird gezeigt, dass der Octree-basierte Container funktionell und damit korrekt ist. Im Vergleich zu bewährten, optimierten Containern ist der neue Container um Faktor 3 langsamer. Die ebenfalls entwickelten Visualisierungsmittel werden erklärt und anhand Beispielen veranschaulicht. Zum Schluss werden einige Ideen und Anregungen zum Ausbau der Implementierung und zu weiteren Analysen dargelegt.

Contents

Part I.

Thesis

1. Introduction

We are living in a rapidly evolving time: Scientists of different research areas are pursuing their need to investigate what atoms, molecules, and other small particles behave like on a microscopic scale. Since performing experiments in a minuscule environment is hard and error-prone, researchers had to come up with better and cheaper solutions. Simulating particles using a computer proved to be a serious alternative. Currently, several toolboxes for so-called molecular dynamics (MD) simulations exist.

Using the computer, MD simulations provide reproducible and exact results. A common issue concerning these tools is the fact that simulating a large number of particles is computationally extremely expensive and takes a long time. It is therefore necessary to think of ways to improve the run-time cost.

During this thesis, one optimization was developed and integrated into the existing MD simulation library `AutoPas`. In the following, an introduction to the theoretical backgrounds is given and fundamental concepts related to MD simulation software are shown. The thesis shall serve as a detailed overview of the implementation of a so-called octree container for `AutoPas`. Octrees are a vastly common data structure used in various computer science-related fields. For instance, [Sam89a] uses octrees to accelerate raytracing by pruning the number of objects inside a scene. Another application is the utilization of the acceleration of collision detection systems as described by [WLZ14]. In `AutoPas`, the octree container is supposed to reduce the time a simulation takes by minimizing the number of particles that have to be considered. Essentially, this is closely related to what octrees achieve in collision detection systems. By inserting objects into an octree, it is much easier to exclude objects that are not eligible for collision. Here, the novel implementation is compared to an existing container data structure. In the end, some starting points for future investigations are given.

2. Theoretical Background

In this chapter, the fundamental theoretical backgrounds, which serve as a foundation for the implementation strategies and optimizations shown in the following chapters, are provided. Here, a general overview of MD simulation is given first. The second topic will be container data structures, which play an important role in the performance of MD simulation tools. Last but not least, the concept of octrees will be introduced since it was used in the implementation.

2.1. Molecular Dynamics Simulation Software

This section shall give an overview of the physical connections that are utilized to model particle interactions in simulations. The focus will then be drawn towards the MD simulation library `AutoPas` that was used during the implementation phase.

2.1.1. Motivation and Theory

This first part will introduce theoretical concepts and optimization techniques.

As mentioned in the introduction, simulating a large number of particles leads to a very large computational cost. Therefore, a whole range of optimizations is used in order to reduce the duration of a simulation. One common model that is used to calculate forces between particles in a fast manner is the Lennard-Jones potential. [Ver67]

$$U_{\mathrm{LJ}}(r) = 4\varepsilon \left[\left(\frac{\sigma}{r} \right)^{12} - \left(\frac{\sigma}{r} \right)^{6} \right] \tag{2.1}$$

Equation (2.1) describes the potential between two particles with euclidean distance r and is used to model forces between particles. The other parameters ε and σ encode how strong and ranging the potential is. Figure 2.1 illustrates how U_{LJ} behaves depending on the particle distance r. A force F_{ij} between two particles i and j with euclidean distance r_{ij} can be obtained from U_{LJ} by taking the derivative as shown in Equation (2.2). [Ver67]

$$F_{ij} = \frac{\mathrm{d}U_{\mathrm{LJ}}(r_{ij})}{\mathrm{d}r_{ij}} \tag{2.2}$$

The force is then applied to each particle at every time step according to the superposition principle. A force F_i acting on a particle i is calculated by summing over all partial forces, which are derived for every time step according to Equation (2.2). The superposition connection can be expressed as Equation (2.3). [Mac13]

$$F_i = \sum_{j \neq i} F_{ij} \tag{2.3}$$

This summation must be done at every step and, theoretically, between every possible pair of particles. This has a huge impact on the overall program performance, since the number of particles is very large and the experiments may run for more than 100,000 iterations. A naive implementation without optimizations gives rise to an enormous computational complexity of $\mathcal{O}(n^2)$, where n is the number of particles that participate in an experiment.

Fortunately, the above-mentioned physical properties can also be exploited in order to drastically improve program performance. Two optimization techniques are described in the following.

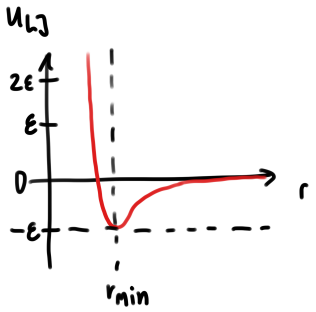


Figure 2.1.: Lennard-Jones potential U_{LJ} depending on the distance r with minimum value $-\varepsilon$ at $r_{\min}$. σ models the distance at which $r_{\min}$ is located on the r-axis and how far the potential reaches. The strength of the potential is defined by the parameter ε.

Convergence of the Lennard-Jones Potential Figure 2.1 shows the typical behavior of the Lennard-Jones potential. One important aspect that can be utilized for optimization is that U_{LJ} converges to zero for large enough r. This means that the potential between two particles that are distant to each other is almost zero. The force between them can thus be considered as zero, since the derivative also vanishes for large enough r. Therefore, interactions between particles that are distant to each other, do not need to be considered. Usually, a cutoff specifies at which distance r_{c} the potential U_{LJ} is considered zero. r_{c} can be tweaked according to the experiment configuration and is assumed to remain constant during one simulation run. This optimization proves to be very beneficial for the program performance as it reduces the average computational complexity from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$ for homogeneous scenarios. [GSBN20]

Newton 3 Optimization There exist two aspects guiding this optimization: First, the observation that the force calculation is a computationally intensive task involving a lot of floating-point operations, which are costly. Second, according to Newton's third law, a force between particles i and j acts on particle i with F_i and on j with $F_j = -F_i$. As a result, the force between two particles does not have to be calculated twice. It can simply be computed for one particle and the opposing force can be applied to the other particle.

A certain form of logic is required to prevent iterating the same pair of particles twice. Using carefully derived orders of iteration or keeping track of already iterated particle pairs solves this issue.

The techniques mentioned above are relevant in the progress of understanding the architecture of the concepts of the following sections.

2.1.2. `AutoPas`

During the practical part of this thesis, parts of the open source `C++` software package `AutoPas` were used and extended. This subsection describes central aspects of the `AutoPas` library and its functionality.

`AutoPas` is a simulation library, which can be used to simulate any kind of short-range particle interactions, for instance using Lennard-Jones. The library aims to find an optimal configuration to run a simulation, while a large range of different helper algorithms and data structures are made available. These provide fast particle gathering and pairwise iteration that are optimized to run in a highly parallel environment using MPI or OpenMP. `AutoPas` tries to identify those algorithms and data structures, which lead to optimal performance during the simulation. A special tuner unit is called in fixed time intervals to determine whether the current program configuration is still optimal for the evolving simulation in order to find the best configuration. The tuner can pick from a range of different so-called containers and traversals. A container is a data structure that can hold a collection of particles. There exist several traversals for a container. Containers may utilize the convergence of the Lennard-Jones potential optimization introduced earlier in order to create data structures that utilize the spatial structure of the environment. These containers are then used by specialized iteration schemes to provide fast access to particle pairs that lay within the cutoff range r_c. Most of the traversals are capable of using the newton 3 optimization to reduce the number of redundant force calculations. Currently, there are six different tuning strategies available for `AutoPas`, which can be picked by the user.

The `AutoPas` library is designed such that the entire functionality is available through one object of type `AutoPas`. Changing the configuration can be done by calling the respective setter method on the `AutoPas` instance. Further, the object allows for adding, removing, updating, and iterating particles. Which data structure and iteration scheme the tuner chooses is completely hidden to the user of `AutoPas`. This allows keeping a clean and easy-to-use library interface.

For testing purposes, the `AutoPas` project offers a program that serves as a reference implementation. It is called `md-flexible` and implements the entire infrastructure needed to run an MD simulation experiment with `AutoPas`. `md-flexible` includes a `.yaml` configuration file parser that is able to obtain the experiment and program configuration for the `AutoPas` library. The `.yaml` file contains information not only about the particle configuration and layout, but also concerning the experimental environment, most importantly the cutoff range, the tuning interval, and the number of simulation steps that should be executed. Furthermore, the range of available containers and traversals can be restricted to enforce configurations that would not always be picked by the tuner.

The default output of `md-flexible` is a periodically written `.vtk` file. It contains the particle positions and velocities at each point in time. One can use this file as a starting

configuration for `md-flexible`, potentially as a checkpoint.

A typical run of `md-flexible` is illustrated in Figure 2.2. A run consists of three different phases: The initialization, during which the program configuration is obtained and passed to the `AutoPas` library; the simulation phase in which the iterations are performed and the output phase, which logs statistics.

The previous subsections already introduced the concept of a container as a data structure that contains particles. One of the challenging tasks of `AutoPas`'s tuner is selecting an appropriate container for an experiment. More information on the tuning strategies is provided in [GSBN20].

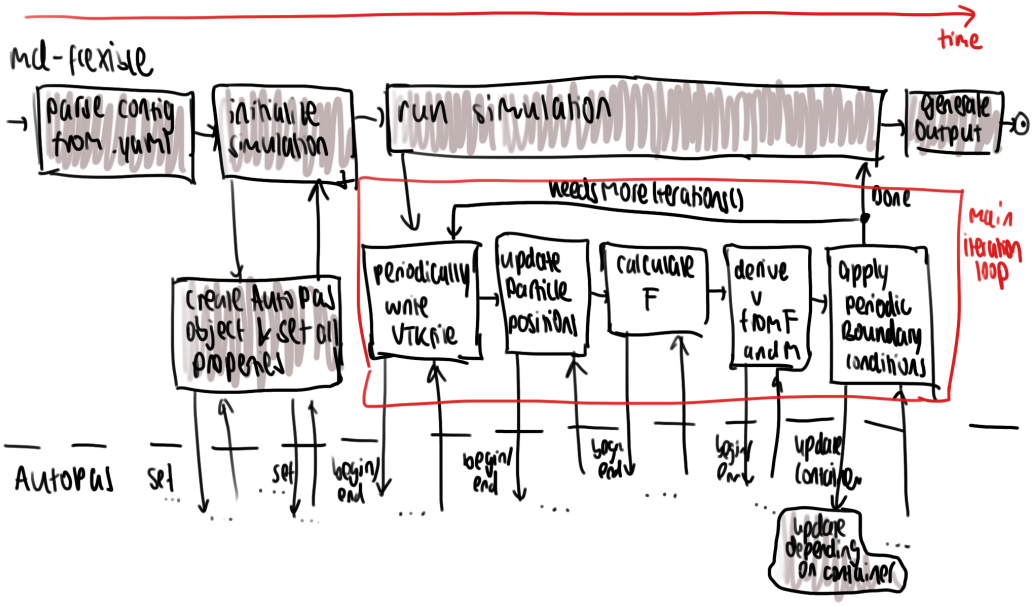


Figure 2.2.: This figure shows a typical progression of a `md-flexible` run. The distinction between the user (`md-flexible`) and the library (`AutoPas`) is indicated by the dashed horizontal separation line.

2.1.3. Skin, Cutoff, and Interaction Length

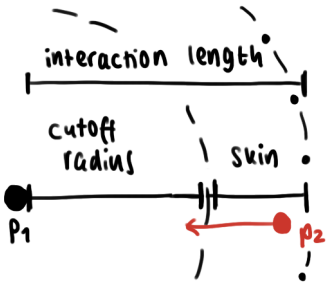


Figure 2.3.: This figure illustrates the relationship between skin, cutoff radius and the interaction length. The dashed line marks the cutoff radius around a particle p_1. The red particle p_2 is in p_1's skin but travels into the circle within p_1's cutoff radius in the iteration. Without an update of the neighbor list or the container structure, p_2 would not be considered a potential interaction partner of p_1, but since it is in the skin, the interaction gets tracked. Furthermore, the relationship between interaction length, cutoff radius, and skin is illustrated.

In Section 2.1.1, the cutoff radius r_c was introduced as the distance beyond which the force acting between two particles can be considered zero. In this section, the concept of a cutoff radius will be refined and extended.

MD simulations are based on iteration loops, as shown in Figure 2.2. Part of such a loop is a periodic update of the underlying data structures. Since those updates are usually very costly in terms of compute time, they are performed as seldom as possible. This is achieved by running the simulation without updating the underlying data structure. An important job of these data structures is to store particles that are within r_c, which is enforced by the update. This constraint could be violated if the update is not called after each particle movement. To solve this issue, the data structures are required to keep all particles accessible that are not only within r_c, but also within a safety margin which is called *skin*. This skin value is chosen such that it is the maximum distance two particles could travel during the period without an update.

The sum of cutoff radius and skin is the maximum distance at which particles could interact with each other without an update. This sum is called *interaction length*. The relationship between cutoff radius, skin, and interaction length is illustrated in Figure 2.3.

2.1.4. Halo and Owned Particles

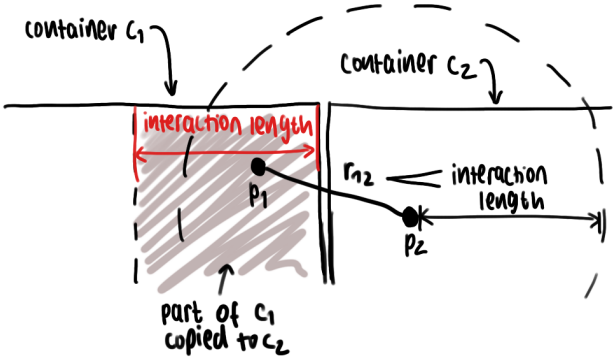


Figure 2.4.: This figure shows how two containers (potentially running on different compute units) exchange particles that are close to the skin. Particle p_1 is within interaction length of p_2, which means that it should be able to interact with p_2. However, p_1 is in a different container than p_2. This is solved by copying a small region of container c_1 to c_2. This piece contains particles that are so close to c_2 that they could interact with particles that are technically inside c_2.

Another essential concept that is important in order to understand the implementation details is the distinction between *halo* and *owned* particles. In general, owned particles belong to a specific container and are strictly inside the container's simulation box. Halo particles, on the other hand, are outside this simulation box and potentially belong to a different container. The reason for this is explained in the following paragraph.

`AutoPas` is optimized to run in highly parallel environments such as supercomputers using MPI. Running a simulation on a distributed system is achieved by splitting the entire domain into smaller chunks that can then be processed by the individual compute nodes according to the distributed memory parallelization principle. Each of those nodes holds its own instance of a container that contains a part of the simulation. The following problem arises from this configuration: The domain is distributed across different nodes and all particles that are inside the domain within reach should be able to interact with each other, but they are not necessarily in the same container. This problem is also illustrated in Figure 2.4. One approach to solve this issue is to copy the neighboring region within interaction length into the container. As a result, particle pairs are correctly generated, even across container boundaries. Copying a fraction of the neighboring container significantly reduces the overhead compared to gathering potential interaction partners on demand.

In `AutoPas`, it is the containers' task to distinguish between halo and owned particles in order to manage both correctly. Therefore, the halo part of the container is by interaction length larger than the owned part in every direction.

2.2. Container Data Structures

This section provides deeper information about containers and outlines different container types, ordered by increasing complexity. Their advantages and disadvantages are also briefly discussed.

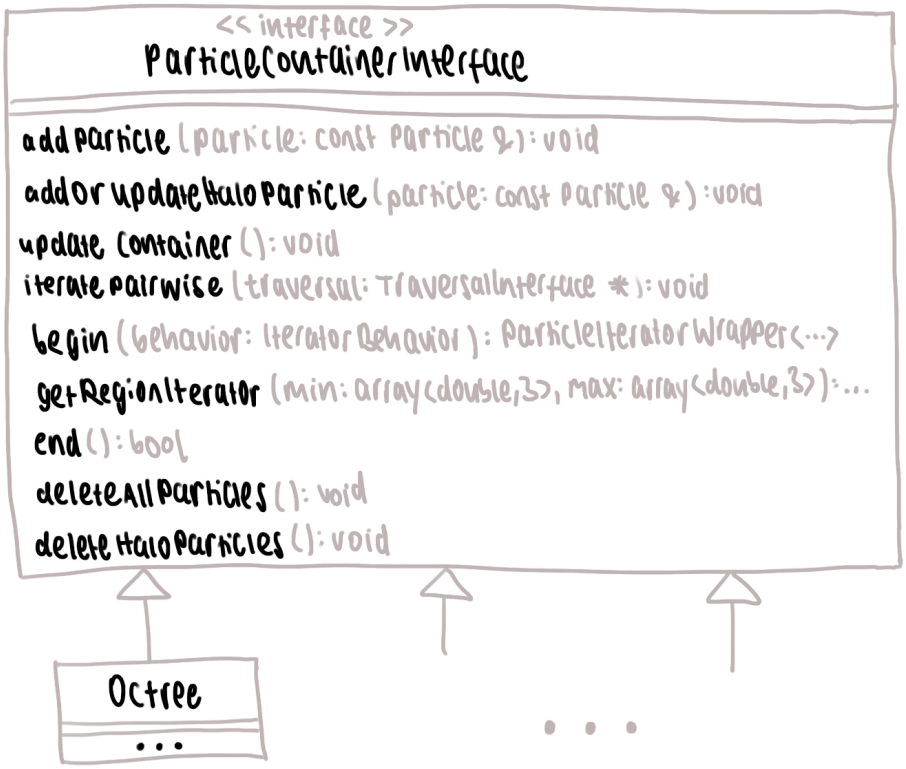


Figure 2.5.: This figure shows the method definitions and one potential usage of the `ParticleContainerInterface`.

2.2.1. Overview

Adding to the previous definition, a container in `AutoPas` is a data structure that supports the following operations. A brief overview of the class hierarchy is shown in Figure 2.5.

Insertion `addParticle(`p`)` adds an owned particle p to the container. A halo particle q can be added using the `addOrUpdateHaloParticle(`q`)` method. In `AutoPas`, a container must be able to distinguish and manage both owned and halo particles.

Deletion The methods `deleteAllParticles()` and `deleteHaloParticles()` can be used to clear the container. Furthermore, the user can mark individual particles as deleted. Those are not removed until the next update to keep the container stable, but excluded in the iteration methods.

Update The `updateContainer()` method triggers an update, that depends on the container type. During the particle position update (visible in Figure 2.2), the positions of the

particles may have changed. Some containers require to be notified about this change and updated periodically in order to maintain stability.

Pairwise iteration A container needs to supply a method of iterating particle pairs, which is required in order to calculate pairwise forces like the Lennard-Jones potential and force derivation. This is done using the `iteratePairwise(`t`)` method, which receives a pointer to a so-called `TraversalInterface` t specifying a method of iterating the container using a callback function. There may be different types of traversals supported by a container. They can be passed to it using the `iteratePairwise(`t`)` method, which passes the call further to the traversal itself. This object is then responsible for iterating the container.

Particle iteration Often, not only an iteration of particle pairs, but also iterating individual particles is required. This can be done using the `begin` and `getRegionIterator(`p_{min}`,` p_{max}`,` b`)` methods. The former allows iterating all particles inside the container. The latter can be used to specify a cubic region, defined by the corner points p_{min} and p_{max}, from which particles should be returned. Further, an `IteratorBehavior` b can be specified allowing to include or exclude the iteration of halo particles.

Other Several other functions are not included in this overview, for instance getter and setter functions. These functions are not relevant for the following discussions and therefore omitted.

All of the operations mentioned above have to be supported by every particle container in `AutoPas`. This summary shall simply serve as a brief overview, one may also consult the `AutoPas` documentation for further details. [1]

2.2.2. Container Types

The simplest container type consists of a list that contains all particles participating in an experiment. On the one hand, this allows for a straightforward implementation. Gathering enclosed particles is also easy, since the list can be iterated trivially. On the other hand, all particles are enclosed in one large box. Neither is the spatial distribution of the particles used, nor does the container include any special tricks to store the particles. As mentioned before, a frequently used operation on containers is pairwise iteration. Due to the lack of optimizations, iterating particles in this container results in a very high computational complexity of $\mathcal{O}(n^2)$. This approach is called *direct sum*.

Because of the large computational cost of direct sum's pairwise iteration scheme, more sophisticated container structures are required in order to reduce the cost of iteration for more than 2000 particles. In Section 2.1.1, the convergence of the Lennard-Jones potential was mentioned as an optimization technique. This property can be used in order to construct containers that allow for much faster pairwise iteration, reducing the complexity even up to $\mathcal{O}(n)$. The two main ideas that can be used are briefly described in the following list and visualized in Figure 2.6.

[1] General information: `https://autopas.github.io/doxygen_documentation/git-master/` and specifics related to the particle container interface: `https://autopas.github.io/doxygen_documentation/git-master/classautopas_1_1ParticleContainerInterface.html`

Linked Cells This approach divides the box that encloses the experiment, into equally sized smaller boxes. Those are called cells in the following. The box sizes are chosen such that only direct neighbors of the cells contain particles sufficiently close that the enclosed particles are within interaction length. When computing particle pairs, each box is iterated and pairs within the box are generated. Interactions between box and direct neighbors are also captured. The iteration itself is then carried out using direct sum, just on much smaller boxes. This reduces the computational complexity to $\mathcal{O}(n)$ for homogeneously distributed particles. [GKZ07]

Verlet Lists The main idea behind this approach is to store potential interaction pairs, which is achieved by maintaining a so-called neighbor list for each particle. This list includes particles that are within the interaction length. When iterating, pairs can be generated from particles and their respective neighbors that are within the cutoff radius. One challenge with this approach is that building those neighbor lists is expensive, thus they should be reused as often as possible. Particles within those lists may be put into their own linked cells container for faster iteration on top.

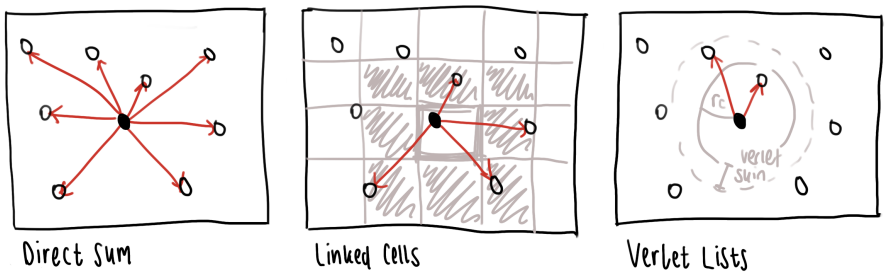


Figure 2.6.: This figure shows the differences between direct sum, linked cells, and Verlet lists regarding pairwise iteration in two dimensions. The red arrows mark interactions that have to be computed between the dark particle in the center and its neighbors. Direct sum does not include any spatial optimization strategy and picks all particles as potential partners to interact with the black particle, linked cells divides the space into a regular grid and Verlet lists only generates interactions for particles within the Verlet skin.

Both of the previously mentioned advanced approaches exploit the Lennard-Jones potential convergence property. Another optimization is to take the distribution of particles inside the box into account. Therefore, the two most prominent techniques are presented in the following. Both are based on the idea of creating a search tree that recursively divides the space into smaller chunks. $d \in \mathbb{N}$ is the number of dimensions in the underlying space. An example configuration for both container types can be seen in Figure 2.7.

k-d trees This data structure is a binary tree, where each node represents exactly one half of the area enclosed by its parent. When inserting a particle, the tree is traversed downwards to find a node that can be split to insert two new nodes. A child node is always split at an axis of the coordinate system that is different to the parent's split axis. Those axes are obtained by the tree depth h at a current node: The split axis

index a always is $a \equiv h \pmod{d}$. Further information and deeper analysis is provided in [Ben75].

Octrees This is another tree-based data structure that, unlike the k-d tree, divides space in eight equally sized smaller boxes. An octree can be thought of as the 3 dimensional equivalent of a 2 dimensional binary tree. The following section provides further detail regarding the layout, creation, particle insertion, and accessing methods in octrees.

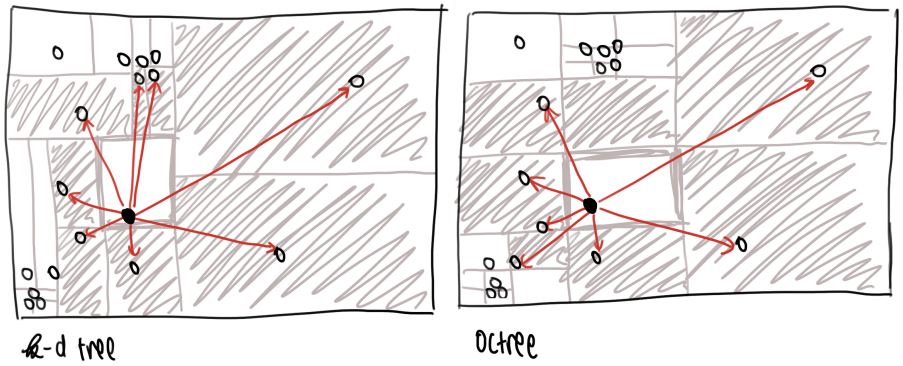


Figure 2.7.: This figure shows the differences between k-d trees and octrees regarding pairwise iteration for $d = 2$ dimensions. The red arrows mark interactions that have to be computed between the dark particle in the center and its neighbors.

2.2.3. Octrees

An octree is a tree data structure that adapts to the particle distribution by dynamically changing its layout depending on its contents. The tree consists of *nodes*, which can be either *inner nodes* or *leaves*. The former always contains exactly $2^d = 8$ for $d = 3$ nodes as *children*. [Mea80] Leaves are nodes that do not have any children, but they may contain particles. Every node within the octree is found inside the sub-tree starting at the *root node*.

How an octree adapts to the particle distribution can be seen by looking at the octree construction procedure: One applies the `insert()` method to the root node for each particle that should be inserted. A node's behavior then depends on its type. Inner nodes recursively pass down the insertion request to an appropriate child. A child is called appropriate if its enclosed region contains the position of the particle to insert. Leaves check the so-called *splitting criterion* and add the particle to its own collection of particles or split themselves up into eight new leaves when the criterion is met. When a leaf splits itself, it applies the `insert()` method to a newly created inner node with eight leaves as children. The leaf then reinserts its particles into the new inner node and also inserts the new particle. Overall, this procedure leads to a tree that is fine-grained at regions with many particles and coarse at locations with fewer particles. This results in a tree that adapts to the spatial distribution of particles. Figure 2.8 illustrates the construction procedure based on an example.

Leaves split themselves whenever the splitting criterion is met. This can be an arbitrary predicate, but two conditions that are applied in the implementation are the following: First of all, octrees restrict the number of particles inside leaves to a certain amount called tree splitting threshold C. Second, leaves may not become smaller than the interaction length

along any axis. The pairwise iteration inside octrees can be done similarly to linked cells: Gather all leaves with their respective neighbors and perform direct sum on them. One problem occurring with leaves that are smaller than the interaction length is that particles may become excluded if they are in a leaf that is too far away, but still within the cutoff radius. Therefore, leaves with a size less than the interaction length are beyond the scope of this thesis.

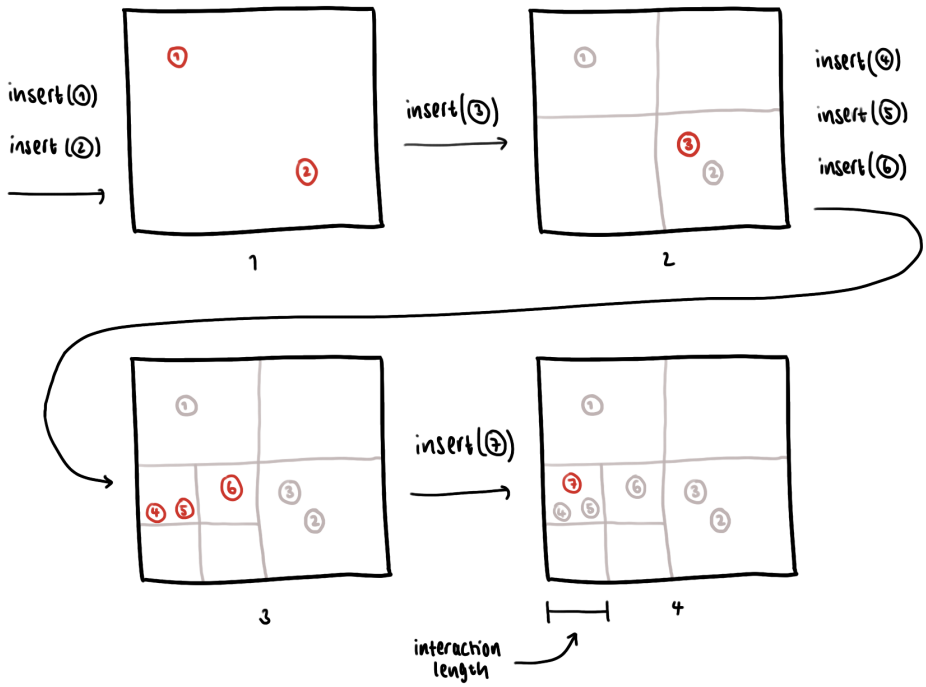


Figure 2.8.: This figure shows four of the steps involved in creating an octree and randomly inserting seven particles into it in two dimensions. The interaction length is set to $\frac{1}{4}$ of the container width. The maximum number of particles per leaf (tree splitting threshold C) is set to two.
Particles p_1 and p_2 are inserted first. Since C is not exceeded, the octree remains to be one large leaf.
Second, p_3 is inserted, which causes the leaf to split into four new leaves in order to regain a particle count less than or equal to C in every leaf.
Third, p_4, p_4 and p_6 are inserted, which causes the bottom left leaf to split itself up even further.
In the last step, particle p_7 is added. The leaf it was inserted into now contains three particles, which is more than allowed by C. As a result, the leaf should split. Since splitting the leaf would generate four leaves whose width and height are smaller than the interaction length, the leaf remains in the current state.

3. Implementation

During the practical phase of this thesis, the `AutoPas` library was extended by adding a new container class that uses an octree to store particles.

The following four sections will provide a detailed overview and explanations of the implemented components. An uncommon approach of interpreting coordinates in a coordinate system that was used to ease the implementation process will be explained first. Second, the architecture used to integrate the octree into `AutoPas` and technical details about the octree data structure will be given. The third chapter will introduce two pairwise traversal strategies that were implemented. Different visualization methods that were developed for debugging and demonstration purposes will be presented in the last section.

3.1. Orientation System for a Cartesian Coordinate System

This section will provide details about the special coordinate indexing system that was used to identify vertices, edges, and faces in three dimensional space. They can be applied to d dimensional cuboids. Figures supporting the claims of this thesis are simplified to the $d = 2$ dimensional space, which also implies that the data structures shown are not octrees but quadtrees.

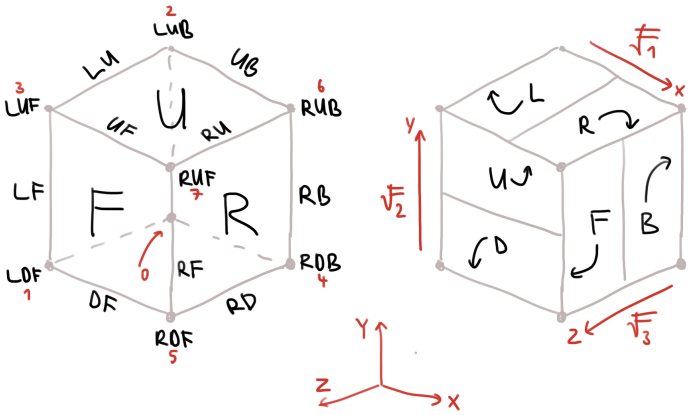


Figure 3.1.: This figure shows the indexing scheme for vertices, edges, and faces introduced by [Sam89b] that is used in the octree neighbor finding algorithm. On the left hand side, all (visible) vertex, edge and face names are visualized. Here, the red numbers on the vertex labels are the indices used to identify corners in the cube. The right hand side shows how the faces are derived and assigned to their respective set $\mathcal{F}_i$ for $i \in \{1, 2, 3\}$.

The indexing system was necessary for convenient implementation of the ideas from

[Sam89b]. Both of the octree pairwise iteration algorithms rely on the tables and algorithm sketches mentioned in the publication. These algorithms are explained further in Section 3.3, it suffices to be aware that they both gather all leaves of the octree and then have to obtain all neighbor leaves to perform pairwise iteration. Gathering the neighbors is carried out using the algorithms `GTEQ_FACE_NEIGHBOR()`, `GTEQ_EDGE_NEIGHBOR()` and `GTEQ_VERTEX_NEIGHBOR()` introduced by [Sam89b]. Those algorithms depend on special predefined tables, which use the indexing system shown in Figure 3.1. This system derives edge and vertex labels from adjacent face names.

In the following, a definition for edges and vertices based on face labels is provided. Let $\mathcal{F}_1 = \{\texttt{L}, \texttt{R}\}$ be the set of all possible face names for the x-axis, $\mathcal{F}_2 = \{\texttt{D}, \texttt{U}\}$ the set for faces from the y-axis and $\mathcal{F}_3 = \{\texttt{B}, \texttt{F}\}$ the set for faces on the z-axis respectively. Then, the following two rules can be used to build edge and vertex labels:

- Two neighboring faces $(A, B) \in (\mathcal{F}_i \times \mathcal{F}_j)$ with $i < j$ for all $i, j \in \{1, 2, 3\}$ form a shared face AB. For instance $\texttt{R} \in \mathcal{F}_1$ and $\texttt{F} \in \mathcal{F}_3$ together build the edge `RF`. The $i < j$ constraint is purely artificial and used to reduce the number of labels for one edge from two to exactly one.
- Likewise, three faces $(A, B, C) \in (\mathcal{F}_1 \times \mathcal{F}_2 \times \mathcal{F}_3)$ adjacent to a vertex form a label ABC for a corner. For instance $\texttt{R} \in \mathcal{F}_1$, $\texttt{U} \in \mathcal{F}_2$ and $\texttt{F} \in \mathcal{F}_3$ build the label for vertex `RUF`. Again, the order constraint ($\mathcal{F}_1$ before $\mathcal{F}_2$ before $\mathcal{F}_3$) is used to reduce the number of vertex labels.

Note that the order of faces that appear inside this string of face labels depends on their axis index: `L` and `R` both lay within the x-axis while being parallel to the yz-plane and are always put first in the label string, then `D` or `U` from the y-axis and last but not least `B` or `F` from the z-axis.

The previous paragraph introduced an indexing system for cube elements, whose translation into code will be covered in the following. `OctreeDirection.h` contains the definitions and functions for working with the system. It contains several `enum`s that provide statically generated labels for faces, edges, and vertices. First, an `enum` has to be declared using predefined values. Those can be seen in the following snippet. Furthermore, an element that does not identify a specific face called `O` is provided. They are also referred to as Ω entries. All other values are integers taken from the range $[1, 6]$, thus, every face requires at most 3 bits to store it.

```
1 enum Face {
2   O = 0,  // omega/unknown
3   L = 1, R = 2, D = 3, U = 4, B = 5, F = 6,
4 };
```

Listing 3.1: Defintion of the base `enum`, `Face`.

As explained above, faces serve as the base class from which edges and vertices are derived. This task is implemented in two functions using `C++`'s meta-programming capabilities: `buildEdge()` and `buildVertex()`. Listing 3.2 shows two simplified versions of these functions. Furthermore, a definition for a datatype that is wide enough to hold a face, edge or vertex is given in form of the `Any` datatype. The maximum width required for a type to index anything within a cube is the vertex, which may be up to 9 bits wide. Therefore, using an `int unsigned` for the `Any` type satisfies the demands completely.

```
using Any = int unsigned;

template <Face f1, Face f2>
static constexpr Any buildEdge() {return (f1 << 3) | f2;}

template <Face g1, Face g2, Face g3>
static constexpr Any buildVertex() {
  return (g1 << 6) | (g2 << 3) | g3;
}
```

Listing 3.2: Definition of an `Any` datatype. Implementation of the compile-time generator functions.

Both functions expect template parameters which are then used to construct a unique identifier for edges or vertices by bit-shifting them to a unique location. As mentioned before, storing a face requires up to 3 bits. Therefore, bit-shifting a face identifier by 3 or 6 bits is sufficient to place the face within an edge or vertex. `buildEdge()` builds the (A, B) tuple by taking $A = \texttt{f1} \in \mathcal{F}_i$ and $B = \texttt{f2} \in \mathcal{F}_j$ with $i < j$ and $i, j \in \{1, 2, 3\}$ and converts them to a 6 bit identifier. `buildVertex()` works similarly, it simply takes $\texttt{g1} \in \mathcal{F}_1$, $\texttt{g2} \in \mathcal{F}_2$, $\texttt{g3} \in \mathcal{F}_3$ and builds a 9 bit identifier. Because of their marking as `constexpr` and the fact that they receive their arguments through template parameters, the functions can be used – as shown in Listing 3.3 – to construct the `enum`s at compile time.

```
enum Edge {
  OO = 0,  // omega/unknown
  LD = buildEdge<L, D>(), LU = buildEdge<L, U>(),
  LB = buildEdge<L, B>(), LF = buildEdge<L, F>(),
  RD = buildEdge<R, D>(), RU = buildEdge<R, U>(),
  RB = buildEdge<R, B>(), RF = buildEdge<R, F>(),
  DB = buildEdge<D, B>(), DF = buildEdge<D, F>(),
  UB = buildEdge<U, B>(), UF = buildEdge<U, F>(),
};

enum Vertex {
  OOO = 0,  // omega/unknown
  LDB = buildVertex<L, D, B>(), LDF = buildVertex<L, D, F>(),
  LUB = buildVertex<L, U, B>(), LUF = buildVertex<L, U, F>(),
  RDB = buildVertex<R, D, B>(), RDF = buildVertex<R, D, F>(),
  RUB = buildVertex<R, U, B>(), RUF = buildVertex<R, U, F>(),
};
```

Listing 3.3: `enum` definitions that are built at compile time using `buildEdge()` and `buildVertex()`.

Both `enum`s also provide Ω entries, `OO` and `OOO`. Since all face identifier values other than `0` are non-zero, this yields a huge benefit: One can infer whether an index given as an instance of `Any` is a face, edge or vertex by checking whether the corresponding bits are zero. Starting from the least significant bit, faces have no bits above the third bit set and edges non above the sixth. Everything else is a label that identifies a vertex.

Furthermore, `OctreeDirection.h` contains several helper functions to aid the user of the indexing system. Those include functions to check whether a given element `isFace()`, `isEdge()` or `isVertex()`. The file also contains functions to obtain lists of all available

faces, edges, and vertices. The tables required by the neighbor gathering are wrapped into the functions, whose prototypes are shown in the following snippet.

```
1 inline bool ADJ(Any direction, Vertex octant);
2 inline Octant REFLECT(Any direction, Octant octant);
3 inline Face COMMON_FACE(Any direction, Vertex octant);
4 inline Edge COMMON_EDGE(Any direction, Vertex octant);
5 inline Any getOppositeDirection(Any direction);
6 inline std::vector<Octant> getAllowedDirections(Any along);
```

Listing 3.4: Function definitions for helper functions.

An `Octant` is a name alias for `Vertex` that is also inspired by [Sam89b]. The functions declared in Listing 3.4 internally use a lookup array to implement the tables required to find neighbors in octree nodes.

3.2. Octree Container for AutoPas

This section will introduce the new container data structure that was developed to fit inside the AutoPas ecosystem. First, the overall code and file layout will be presented. After that, important details about the implementation will be mentioned.

3.2.1. Overview

A container can be added by placing the source code inside the `src/autopas/containers` directory. There must be a class provided that extends the behavior of the base class, `ParticleContainerInterface`. In this case, the class `Octree` was placed inside a file called `src/autopas/containers/octree/Octree.h` and serves as one part of the API that is used by AutoPas to interface with the octree. The second part are the different traversals which are explained in Section 3.3. Additionally, the octree was registered inside the `ContainerSelector` after creating a new `ContainerOption` called `octree`. After that, the new container was available in AutoPas.

3.2.2. Base Data Structure

The most important part of the octree is the recursive data structure that can be seen on the right side in Figure 3.2. Its origin is the base class `OctreeNodeInterface`, which is an abstract class. Many operations on the octree depend on the information whether a processed node is an inner node or a leaf node. As a result, one can declare abstract method prototypes inside the base class that are implemented in the derived sub-classes where the required information about the node type is present. `OctreeNodeInterface` contains the definitions for those abstract methods. An excerpt is given in Listing 3.5. Note that these methods already suffice to implement the operations defined in Section 2.2.1. Iteration can be done by gathering all leaves using `appendAllLeaves()` and iterating the particles inside the `OctreeLeafNodes`.

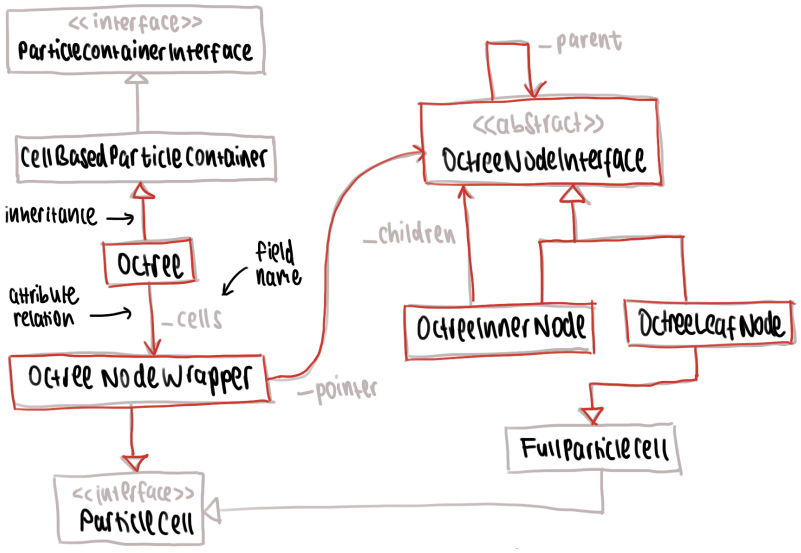


Figure 3.2.: This figure gives an overview of the infrastructure provided by `AutoPas` used to embed the octree implementation into the project. Classes marked red were added and implemented during the practical phase. All other classes already existed in `AutoPas`. The template parameters of all classes are not shown for the sake of keeping the diagram simple. The right hand side shows not only the inheritance graph used to build the octree but also their owner relationships using the field `_parent` and `_children`

```
1  template <class Particle>
2  class OctreeNodeInterface {
3   public:
4    // ...
5    virtual ... insert(Particle p) = 0;
6    virtual void clearChildren(std::unique_ptr<...> &ref) = 0;
7    virtual void appendAllLeaves(std::vector<...> &leaves) const = 0;
8    virtual void appendAllParticles(std::vector<...> &ps) const = 0;
9
10   // Used for logging
11   virtual void appendAllLeafBoxes(
12       std::vector<std::pair<std::array<double, 3>,
13                             std::array<double, 3>>>
14       &boxes) const = 0;
15 };
```

Listing 3.5: Class definition of the base interface for inner nodes and leaves with method prototypes that are implemented in the subclasses. "..." indicates that some parts were left out for brevity.

One `OctreeInnerNode` can contain up to eight children, which can be either inner nodes or leaves themselves. This is required to build the hierarchical structure that divides space. Children are stored using smart pointers, as illustrated in Listing 3.6, which shows a part of the definition for inner nodes. The pointers are initially set to eight leaves at construction

time of the inner node which is explained in further detail in Section 3.2.4. As a result, an inner node is the root of a valid sub-tree at any point in time.

```
1 template <class Particle>
2 class OctreeInnerNode : public OctreeNodeInterface<Particle> {
3   // ...
4  private:
5   std::array<std::unique_ptr<OctreeNodeInterface<Particle>>,
6              8> _children;
7 };
```

Listing 3.6: Class definition of an inner node with array of pointers to children.

Leaves, on the other hand, receive the ability to store, manage and iterate particles by inheriting from `FullParticleCell`. This is a special cell whose sole purpose is to store particles inside a `std::vector` and allow easy access to them. `OctreeLeafNode` combines `OctreeNodeInterface` with the `FullParticleCell`. This is achieved by using `C++`'s multiple inheritance capabilities, which can be seen in Listing 3.7 showing the definition of `OctreeLeafNode`.

```
1 template <typename Particle>
2 class OctreeLeafNode : public OctreeNodeInterface<Particle>,
3                         public FullParticleCell<Particle> {
4   // ...
5 };
```

Listing 3.7: Class definition of a leaf node with inheritance from two base classes.

3.2.3. Environment of the Octree Container

Figure 3.2 gives an overview of how the octree interfaces with the rest of `AutoPas`'s programming environment. `Octree` inherits from `CellBasedParticleContainer`, which is a commonly used base class in `AutoPas` to implement containers that store their particles in different cells. The `Octree` is implemented such that it consists of two root cells, one that only contains the halo particle cell and another one for the owned cell. Those cells store their particles in two distinct octrees. Providing an interface to act as a cell and managing a reference to the root node of the octree it the job of `OctreeNodeWrapper`. Cells inside a `CellBasedParticleContainer` are not allowed to change the reference to themselves, which required implementing a wrapper that stores a pointer to an instance of a sub-class of `OctreeNodeInterface`. The reference to the octree's root node needs to be able to change the pointer since the type of the root node may change from inner to leaf node or vice versa by altering the particle configuration inside the containers.

3.2.4. Algorithmic Details

The previous subsections introduced the overall structure and gave an overview of the implementation. In the following, parts that are considered especially important from the code will be discussed. Those include specific information about particle insertion, container update and clearing.

Particle Insertion and Tree Construction

As introduced in Section 2.2.1, particles can be added to a container using the `addParticle()` or `addOrUpdateHaloParticle()` operation. The `Octree` class decides whether to add a particle to the halo or owned octree and delegates the call to `OctreeNodeWrapper`, which itself passes the request to the root node. Insertion is implemented as specified in Section 2.2.3. Listing 3.8 shows the simplified behavior of an inner node's `insert()` method: It iterates over all possible children and checks whether the particle fits in one of them. The `getR()` method obtains the current position in space of the particle, which can then be checked against the child's enclosing box using the `isInside()` method. In order to reduce overhead, the `for`-loop iterating over all children is only executed until a child is found.

```
1  template <class Particle>
2  class OctreeInnerNode : public OctreeNodeInterface<Particle> {
3   public:
4    // ...
5    std::unique_ptr<OctreeNodeInterface<...>> insert(Particle p)
       override {
6      // Find a child to insert the particle into.
7      for (auto &child : _children) {
8        if (child->isInside(p.getR())) {
9          auto ret = child->insert(p);
10         if (ret) child = std::move(ret);
11         break;
12       }
13     }
14     return nullptr;
15   }
16 };
```

Listing 3.8: Simplified version of an inner node's `insert()` method.

The code contains a `C++` trick, which is required as a result of the following issue: Insertion of a particle into a sub-tree may change the root node's type. Adding a particle to a leaf may convert the leaf to an inner node, once the split condition is fulfilled. Therefore, the `insert()` method needs a way of communicating to the caller (either another inner node or an `OctreeNodeWrapper`) if the leaf splits itself up. It does this by returning a `std::unique_ptr` to a node whose value is `nullptr` if the leaf did not split, otherwise a pointer to the newly created inner node is returned. This behavior can be seen in the code, where the return value of the `child->insert()` call is stored and then checked. If it did change, the child will be assigned to the new reference.

The last statement in its `insert()` routine is `return nullptr`, which signals to the parent that the node never splits up.

Inserting a particle into a leaf, on the other hand, is a little bit more complicated. When inserting a particle into a leaf, it has to distinguish between two cases: First, the splitting condition is not fulfilled. In this case, the leaf node acts like a `FullParticleCell` and adds the particle to itself. Second, if the splitting condition is met, the leaf generates a new inner node with eight new children, which are all leaves. Then, all particles from the original leaf and the new particle are distributed among the children of the new inner node. A simplified version of this routine is shown in Listing 3.9.

```
1  template <typename Particle>
2  class OctreeLeafNode : ..., public FullParticleCell<Particle> {
3   public:
4    // ...
5    std::unique_ptr<OctreeNodeInterface<...>> insert(Particle p)
       override {
6      // Check if the size of the new leaves would be to small
7      bool anyNewDimSmallerThanMinSize = false;
8      // ...
9
10     if ((this->_particles.size() < this->_treeSplitThreshold) or
11         anyNewDimSmallerThanMinSize) {
12       this->_particles.push_back(p);
13       return nullptr;
14     } else {
15       auto newInner = std::make_unique<OctreeInnerNode<Particle>>(
16           this->getBoxMin(), this->getBoxMax(),
17           this->_parent, ...);
18       auto ret = newInner->insert(p);
19       if (ret) newInner = std::move(ret);
20       for (auto cachedParticle : this->_particles) {
21         ret = newInner->insert(cachedParticle);
22         if (ret) newInner = std::move(ret);
23       }
24
25       return newInner;
26     }
27   }
28 };
```

Listing 3.9: Simplified version of a leaf's `insert()` method.

Note, that the method returns `nullptr` when it is possible to add the new particle to the leaf and a pointer to a newly created inner node if the leaf was split up. The splitting condition for a leaf consists of two predicates that must both be fulfilled to split a leaf:

- The number of particles within the leaf is greater than or equal to the tree split threshold C, named `_treeSplitThreshold` in the snippet. It is stored in a field within the `OctreeNodeInterface` and is passed down from the root of the octree. This allows for adjusting C for different octree types. During this thesis, a tree split threshold of $C = 16$ was used exclusively.
- The leaves must be greater than the interaction length. Otherwise, interactions across multiple leaves that technically are in range would not be captured. This is ensured by the `anyNewDimSmallerThanMinSize` variable. Since the region is not necessarily cubic, a loop (that is omitted for brevity in the snippet above) checks whether the domain size split into eight pieces would be to small.

Every time an inner node is generated, it also spawns eight children that are leaves. Each of them represents exactly one eighth of the domain size of the parent node. Those children are then distributed across the inner node's box according to their 3 bit identifier, which is

the respective child index between 0 and 7 in the array. All of the children's new corner coordinates defining their box can be obtained by carefully picking values depending on the axis. Those are obtained from the inner node's corner coordinates $p_{\text{min}}, p_{\text{max}} \in \mathbb{R}^3$ and a calculated center coordinate $p_{\text{center}} = \frac{1}{2} \cdot (p_{\text{min}} + p_{\text{max}}) \in \mathbb{R}^3$. Every bit within the identifier is assigned to one axis in the domain, meaning the most significant bit is assigned to the x-axis, the middle bit to the y-axis and the least significant bit to the z-axis. If an axis bit inside the identifier is set, the corner coordinates for this leaf are picked from p_{center} and p_{max} for the respective axis. If the axis bit is not set, the coordinates for the respective axis are picked from p_{min} and p_{center}. This allows coordinates to be generated for each leaf in the inner node's box. Figure 3.3 illustrates this problem for two dimensions.

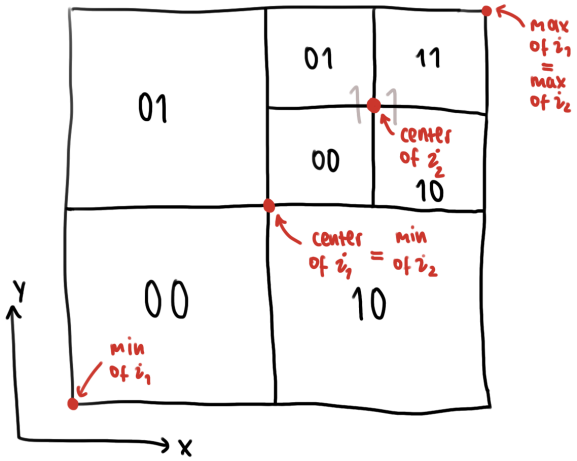


Figure 3.3.: A simplified view on an octree with two inner nodes: A big one and a smaller one in the top right corner. The number labels show the nodes' binary indices relative to their parent nodes. The first number is the x-axis bit, the second the y-axis bit.

Clearing an Octree

The implementation also supports clearing an entire octree using the `clearChildren()` method. Note the function definition in Listing 3.10.

```
1 virtual void clearChildren(
2     std::unique_ptr<OctreeNodeInterface<Particle>> &ref) = 0;
```

Listing 3.10: Definition of `clearChildren()` in the base class `OctreeNodeInterface` with self-reference as method parameter.

The method takes a reference to a pointer to a node as its only parameter. It is a reference to the pointer that the parent holds to the child itself. Depending on its type, a child decides how to clear itself: A leaf simply clears all the particles inside it and keeps the reference untouched. An inner node first recursively clears its children, then generates a new leaf node and finally moves the new leaf node onto the reference. As a result, the inner node is gone and replaced with a leaf node.

Updating the Octree Container

An important concept within AutoPas is the periodic container update that gets called before each simulation step. During this step, some of the particles may have moved outside the container boundaries and become halo particles. One job of the `updateContainer()` method is to identify these out-of-bounds particles and return them. Furthermore, the method is called to allow the container to reorganize itself in order to remain stable.

```
1  template <class Particle>
2  class Octree : public CellBasedParticleContainer<...>, ... {
3   public:
4    // ...
5    std::vector<Particle> updateContainer() override {
6      // 1. Copy all particles out of the container
7      std::vector<Particle *> particleRefs;
8      this->_cells[CellTypes::OWNED].appendAllParticles(particleRefs);
9
10     // 2. Partition the particles into valid/invalid particles
11     std::vector<Particle> particles{}, invalidParticles{};
12     particles.reserve(particleRefs.size());
13     for (auto *p : particleRefs) {
14       if (utils::inBox(p->getR(),
15                        this->getBoxMin(),
16                        this->getBoxMax())) {
17         particles.push_back(*p);
18       } else {
19         invalidParticles.push_back(*p);
20       }
21     }
22
23     // 3. Clear all particles (including the halo particles)
24     this->deleteAllParticles();
25
26     // 4. Insert the particles back into the container
27     for (auto &particle : particles) {
28       addParticleImpl(particle);
29     }
30
31     return invalidParticles;
32   }
33 };
```

Listing 3.11: A simplified version of the `Octree`'s `updateContainer()` method. The container update consists of four phases. Those are marked by the comments.

As illustrated in Listing 3.11, the container update consists of four phases, which are marked in the code snippet. First, all particles that actually belong to the container (owned) are copied out of the data structure into a temporary `std::vector` buffer called `particleRefs`. This buffer then contains pointers to the particles inside the octree. The second step decides for each of the particle references whether it is still within the container bounds or if it went outside of the border, implying that it became a halo particle. Not pointers, but actual particle objects are then stored in one of the respective lists: `particles`

or `invalidParticles`. After that, all owned and halo particles are cleared in the third step. Both cells are empty leaf nodes again. In the last step, all particles that are still within the octree are inserted back into it. The entire procedure clears the octree and reinserts the particles. This ensures that the octree is valid after `updateContainer()`.

3.3. Special Optimized Traversals

Each container in `AutoPas` comes with its own set of traversals. A traversal allows for fast, shared-memory parallel pairwise iteration, which is required for calculating forces between particles. During the thesis, two different approaches were implemented for the octree container. These are discussed in further detail in Section 3.3.1.

3.3.1. C01 and C18 Traversals

The two methods of pairwise iteration in the octree are called C01 and C18 traversal. In both traversals, the owned leaves are gathered in Line 8 of Listing 3.12. Each traversal contains a field called the `_cellFunctor`. This is an object that is capable of computing pairwise interactions either for one cell using `processCell()` or for two cells using `processCellPair()`. Because of the architecture of `OctreeLeafNode`s as sub-classes of `FullParticleCell`s, they can be used as arguments for calls made to the `_cellFunctor`. The code in Listing 3.12 first uses `processCell()` to generate all pairwise interactions from within every leaf and then processes all neighboring cells in Line 19, including those from the halo octree in Line 28, using `processCellPair()`. Neighboring leaves are obtained using two different methods: `getNeighborLeaves()` and `getLeavesInRange()`. The differences between those two functions are explained in further detail in Section 3.3.2.

Note that the code is designed such that it contains one large `for`-loop that loops over all leaves within the owned octree. Therefore, these leaves have to be obtained before `traverseParticlePairs()` is issued, which happens in a setup routine called `initTraversal`. An implementation for this setup routine must be supplied by every sub-class of `TraversalInterface`, which holds true for both octree traversals. Gathering leaves is executed by the `appendAllLeaves()` method that was introduced before. The obtained nodes are stored in a vector called `_ownedLeaves`, which can then be iterated over in Line 8. A very important aspect of MD simulation software is the ability to run in parallel environments utilizing a large number of threads. During this thesis, both the C01 and the C18 traversals were designed for running parallel using OpenMP. Therefore, the outer `for`-loop in Line 7 was marked with the ability to run on multiple threads. Each loop body may be executed on a different thread since the body only depends on the leaf index `i`. As a result, the pairwise iteration can be executed on an arbitrary number of threads. This number is only capped by the number of leaves available, which should not be a problem in practice since the number of leaves usually exceeds the number of threads.

```
1 template <class Particle, ...>
2 class OTC01Traversal : public CellPairTraversal<...>,
3                         public OTTraversalInterface<...> {
4  public:
5   // ...
6   void traverseParticlePairs() override {
```

```
7     #pragma omp parallel for
8     for (int i = 0; i < this->_ownedLeaves.size(); ++i) {
9       OctreeLeafNode<Particle> *leaf = this->_ownedLeaves[i];
10
11      // Process cell itself
12      _cellFunctor.processCell(*leaf);
13
14      // Get neighboring cells for each leaf
15      auto uniqueNeighboringLeaves = leaf->getNeighborLeaves();
16      // Process connection to all neighbors in this octree
17      for (OctreeLeafNode<Particle> *neighborLeaf :
18           uniqueNeighboringLeaves) {
19        _cellFunctor.processCellPair(*leaf, *neighborLeaf);
20      }
21
22      // Process particles in halo cell that are in range
23      auto min = subScalar(leaf->getBoxMin(), _interactionLength);
24      auto max = addScalar(leaf->getBoxMax(), _interactionLength);
25      auto haloNeighbors =
26          this->getHalo()->getLeavesInRange(min, max);
27      for (OctreeLeafNode<Particle> *neighborLeaf : haloNeighbors) {
28        _cellFunctor.processCellPair(*leaf, *neighborLeaf);
29      }
30    }
31  }
32 }
```

Listing 3.12: A simplified version of the octree's C01 traversal.

The previous paragraph introduced the overall structure of an octree traversal with specific focus on the C01 traversal. In the following, the reason for a C18 traversal and the difference to the C01 traversal are discussed. Figure 3.4 shows the main distinction between the C01 and C18 traversal: The former does not allow for using the newton 3 optimization, but all neighbors can be considered when iterating. The latter, on the other hand, is able to utilize the newton 3 optimization. However, only those neighbors, whose identifier numbers are greater than the current identifier, can be included into the pairwise calculations.

Listing 3.12 shows the implementation of the octree's C01 traversal. One important aspect of this traversal is that some of the cell pairs are processed twice: Two neighboring leaves l_1 and l_2 are iterated using the base `for`-loop shown in Line 8. When obtaining their respective neighboring leaves in Line 15, they both are part of the `uniqueNeighboringLeaves` set of their neighbor. This means that both $l_2 \in l_1$`->getNeighborLeaves()` and $l_1 \in l_2$`->getNeighborLeaves()`. As a result, `processCellPair()` is called for both cell pairs (l_1, l_2) and (l_2, l_1) during the whole iteration. Assuming the newton 3 optimization was enabled, this would lead to duplicate force calculations since the forces would be applied to the particles ins l_1 and l_2 twice. A solution for this is to disallow the newton 3 optimization for C01 traversals.

The C18 traversal solves the double-iteration problem. During the previously introduced setup routine `initTraversal()`, every leaf obtained through `appendAllLeaves()` is assigned an integer identifier. The only requirement for the identifiers is that they need to be unique and monotonic. With those identifiers, the C18 traversal can be derived from the C01 by

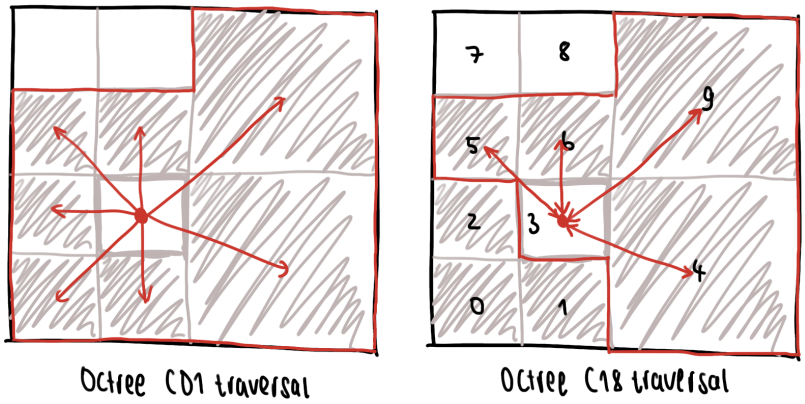


Figure 3.4.: This figure shows the difference between the C01 and C18 traversals in two dimensions. It shows one step of the `for`-loop in Line 8 of Listing 3.12 the leaf with gray border. The red arrows indicate which neighbors are involved in a call to `processCellPair`. Note that for the C01 traversal the arrows only have one tip, which means that the newton 3 optimization is disabled. For C18, the tips of the red arrows are on both sides, which indicates that the newton 3 optimization can be used with this traversal. Also, exemplary ID numbers for each cell are displayed.

guarding every call to `processCellPair` (Lines 19 and 28 in Listing 3.12) with an additional `if` statement. The `if` statement ensures that only cell pairs between the current leaf and a neighbor, where the identifier of the leaf is smaller than the neighbor's identifier, are processed. Listing 3.13 shows the implementation of such a guard.

```
1 if (leaf->getID() < neighborLeaf->getID()) {
2   _cellFunctor.processCellPair(*leaf, *neighborLeaf);
3 }
```

Listing 3.13: Guarded version of `processCellPair()`.

To sum it up, C01 and C18 are simple traversal strategies for an octree without and with support for the newton 3 optimization respectively. Both rely on gathering all leaves upfront and processing cell pairs between each leaf and its neighbors.

3.3.2. Methods of Leaf Gathering

Two different strategies for gathering leaves have already been mentioned: The methods `getNeighborLeaves()` and `getLeavesInRange()`. Both can be called on any sub-class of the `OctreeNodeInterface` and return a collection of unique leaves. The key differences between the two are explained in the following.

Table-based `getNeighborLeaves()` collects all unique neighbor leaves of the node on which the method was called. It iterates over all faces, edges, and vertices in order to collect all neighbors along the leaf's surroundings. This is done by first finding the neighbor that is of greater than or equal size to the leaf (`GTEQ`) and then fetching all touching leaves using an overloaded helper method called `getNeighborLeaves(`d`)`. The method

takes in a direction d as an instance of an `Any` object and returns all leaves that can be found along the face, edge, or vertex specified by d. This is done by first calculating the opposite direction d' of d. The underlying function `getOppositeDirection` is implemented as a table lookup that inverts `Any` direction. (Table C.1 contains the lookup entries.) Second, `getNeighborLeaves(`d`)` obtains all allowed directions using the table lookup function `getAllowedDirections`. (Table C.2 contains the lookup entries.) For `Any` given direction, this function returns a `std::vector` of directions that are all along the given direction d'. This set of directions can then be used to traverse down the octree in the opposite order to obtain leaves that are adjacent to the currently processed leaf. A shortened version of this table-based neighbor gathering scheme can be seen in Listing 3.14.

```
1  std::set<OctreeLeafNode<Particle> *> getNeighborLeaves() {
2    std::set<OctreeLeafNode<Particle> *> result;
3    for (Face *face = getFaces(); *face != 0; ++face) {
4      OctreeNodeInterface<Particle> *neighbor =
5          GTEQ_FACE_NEIGHBOR(*face);
6      if (neighbor) {
7        auto leaves = neighbor->getNeighborLeaves(*face);
8        result.insert(leaves.begin(), leaves.end());
9      }
10   }
11   // ...
12   return result;
13 }
14
```

Listing 3.14: An excerpt of `getNeighborLeaves()` is shown in this listing. The `for`-loop in Line 3 iterates over all possible faces. Then, a node that touches `this` leaf at a `face` is obtained by calling `GTEQ_FACE_NEIGHBOR` in Line 5. If existent, all leaves along the `face` are gathered and added to the result set. In Line 11, the equivalent code for edges and vertices is omitted for brevity.

The three most important methods `GTEQ_FACE_NEIGHBOR()`, `GTEQ_EDGE_NEIGHBOR()` and `GTEQ_VERTEX_NEIGHBOR()` introduced in Section 3.1 were translated to `C++` from the code provided by [Sam89b]. Therefore, several auxiliary functions were implemented to mimic the programming environment described in the paper. The most important helper functions are listed in Appendix B.1. Using this technique, neighboring leaves can be gathered in constant time.

Range Query Another neighbor gathering method is implemented in `getLeavesInRange()` as a range query. This method takes in two parameters: $p_{\min}, p_{\max} \in \mathbb{R}^3$. These specify the minimum and maximum coordinates of a cuboid from which leaves should be taken. `getLeavesInRange()` is implemented differently for inner nodes and leaves: Inner nodes check for every child whether the box of the child overlaps with the given region between $p_{\min}$ and $p_{\max}$. If this is the case, the call is recursively propagated to the child in order to find leaves within the child. The results of these recursive calls are then accumulated in one `std::vector` that is returned. Leaves, on the other hand, return a `std::vector` that only contains themselves if they overlap with the given region,

otherwise they yield an empty `std::vector`. This recursive procedure ensures that all leaves within the given region are captured and returned. Another neighbor gathering method is implemented in `getLeavesInRange()` as a range query. This method takes in two parameters: $p_{\text{min}}, p_{\text{max}} \in \mathbb{R}^3$. These specify the minimum and maximum coordinates of a cuboid from which leaves should be taken. `getLeavesInRange()` is implemented differently for inner nodes and leaves: Inner nodes check for every child whether the box of the child overlaps with the given region between p_{min} and p_{max}. If this is the case, the call is recursively propagated to the child in order to find leaves within the child. The results of these recursive calls are then accumulated in one `std::vector` that is returned. Leaves, on the other hand, return a `std::vector` that only contains themselves if they overlap with the given region, otherwise they yield an empty `std::vector`. This recursive procedure ensures that all leaves within the given region are captured and returned.

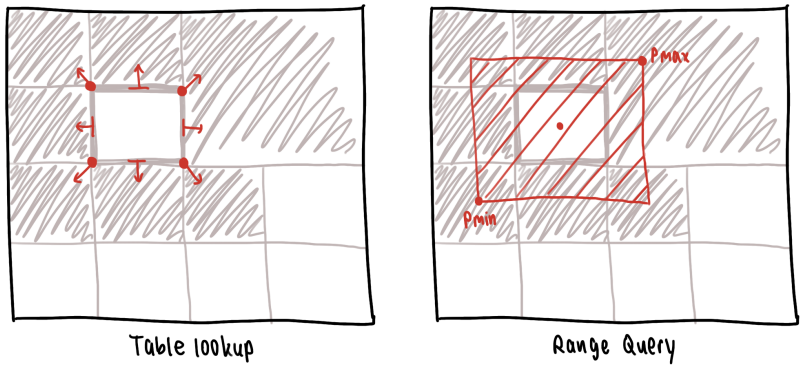


Figure 3.5.: This figure shows the difference between neighbor gathering techniques: Looking up neighbors using tables and obtaining them by range queries in two dimensions. Table lookup in two dimensions can only be done via vertices or edges, which is illustrated by the dots on the corners and the small dash on the edge of the leaf. The region for the range query is determined by the two parameters p_{min} and p_{max}. In two dimensions, this forms a rectangular region that is illustrated by the red dashed area.

Both methods are illustrated in Figure 3.5. As the minimal requirement, `getLeavesInRange()` is necessary for the implementation since the table-based lookup can be substituted with `getLeavesInRange()`. As mentioned before, the halo octree is larger than the owned octree in every direction. As a result, the nodes' boxes of the owned octree do not align with the boxes of the halo octree. Therefore, it is required to gather halo leaves that contain potential interaction partners from the owned leaves using `getLeavesInRange()`.

3.4. Support for Visualization

During this thesis, two methods for visualizing the octree data structure were implemented. These are discussed in the following.

3.4.1. Browser Octree

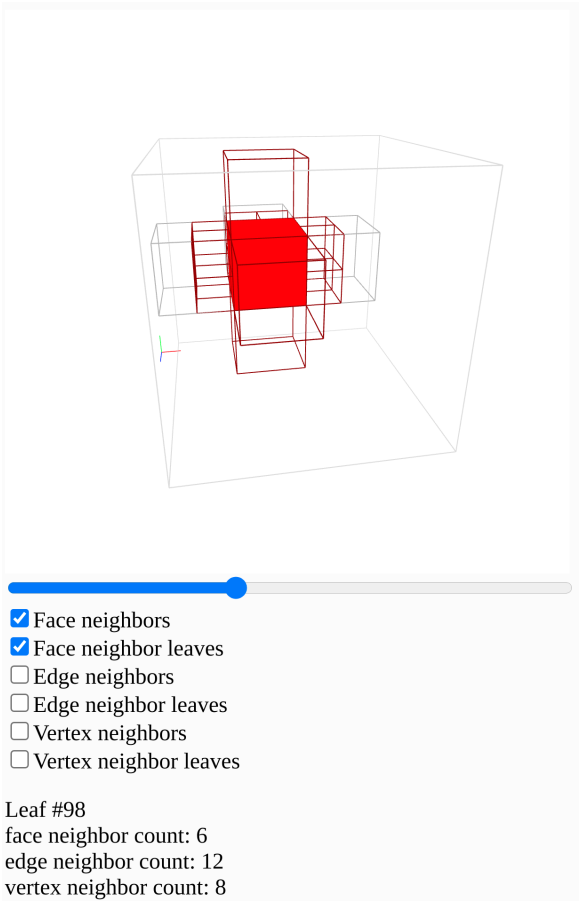


Figure 3.6.: A scene showing the octree boxes in the browser tool. Both the face neighbor and the neighbor leaf options are enabled.

For debugging purposes, it proofed beneficial to have the ability to visualize the octree itself and the table-based neighbor gathering strategy. Therefore, a browser-based tool was written in `JavaScript`. For a rapid implementation, the popular `JavaScript` framework `p5.js` was used. [1] The browser octree visualization is published through its own `git` repository. [2] The tool allows importing specially generated `.json` files, which not only contain box information about the halo and owned octree but also the particle positions. This technique proved itself useful when trying to find bugs inside the program and tables.

`AutoPas` contains several loggers. They were extended by an `OctreeLogger`, which contains static method definitions that are capable of outputting files related to the octree implementation. For instance, the logger contains functions called `octreeToJSON()`, `particlesToJSON()` and `leavesToJSON()` which allow generation of the previously mentioned `.json` files. Those can then be selected within the browser and displayed. The view can be rotated, translated, and scaled using mouse and keyboard. Any octree box can

[1] `https://p5js.org/`
[2] `https://github.com/AutoPas/OctreeVisualization`

be selected and its neighbors (obtained using the `GTEQ_..._NEIGHBOR` functions) can be inspected. Figure 3.6 contains a screenshot of the visualization tool. Pictures of different configurations are shown in Appendix A.

3.4.2. `.vtk` Logger

`md-flexible` outputs its particles in a user-specified interval to a `.vtk` file. [3] Those files can then be visualized using external tools like ParaView[4]. To integrate with the existing tools, the `OctreeLogger` was extended such that it can output both owned and halo octrees in a fixed interval as well. The logger periodically outputs special `owned_`i`.vtk` and `halo_`i`.vtk` files, where i is an ascending record number. Screenshots of the resulting output in ParaView are part of the next chapter. Figure 4.1a shows a screenshot with both the owned and the halo octree.

[3] `https://vtk.org/wp-content/uploads/2015/04/file-formats.pdf`
[4] `https://www.paraview.org/`

4. Analysis

This chapter demonstrates that the implementation is working as intended. Furthermore, the octree container is compared to the existing linked cells implementation. Selected parts are analyzed even further to gain insights in the performance characteristics.

4.1. High-level Comparison

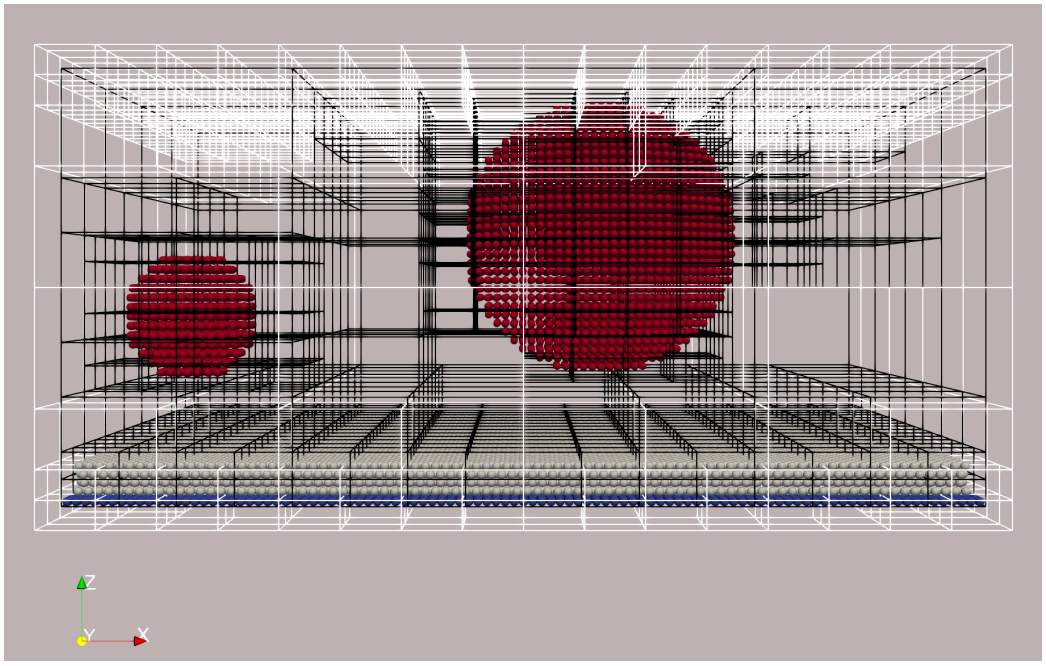
As mentioned in Section 2.1.2, `md-flexible` comes with the ability to load an experiment configuration from a `.yaml` file. The goal of using an octree data structure as a container is that it can utilize cells of different sizes for storing particles. This goal was achieved during the thesis: `AutoPas` now offers containers of this kind. Its behavior is also visualized in Figure 4.2, which shows the simulation state of `fallingDrop.yaml` at selected timestamps. Note that in the beginning, the particle bed on the bottom is wrapped by the container data structure in a particularly fine manner since there is a large number of particles. On the top, only the falling drop is covered in a more detailed grid because the particle density is higher at the drop's location. Furthermore, the region in the top right corner does not contain any particles. As a result, the octree's divisions are very coarse. Overall, this shows that the goal of implementing a space-adaptive container for `AutoPas` was achieved by this thesis.

Another scenario is captured in Figure 4.1. Figure 4.1a shows that the halo octree is also working by displaying the halo cell as an overlay.

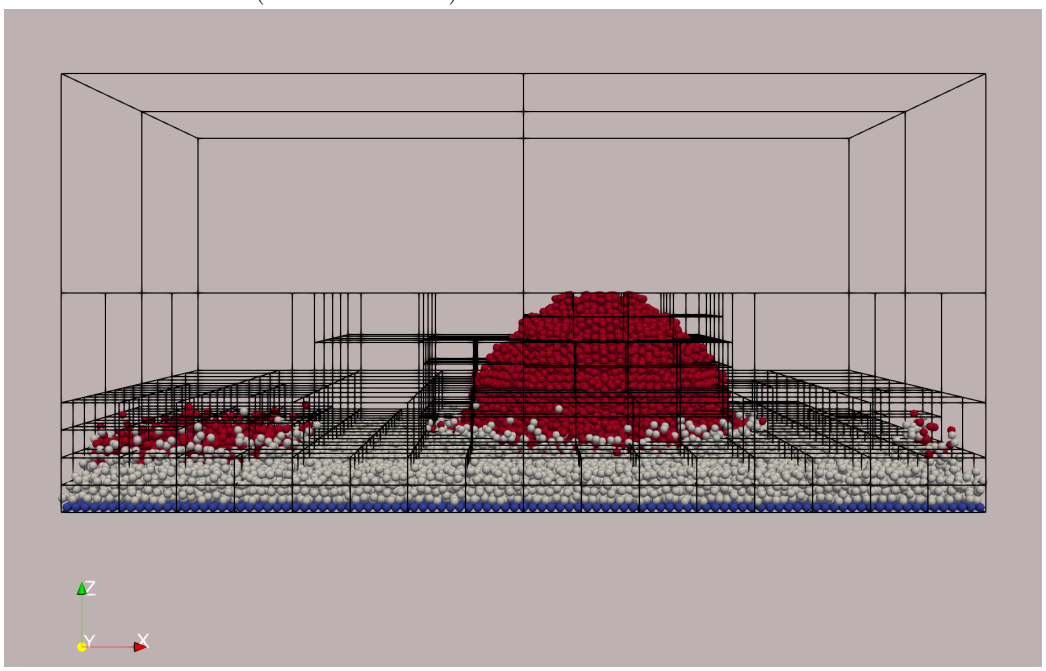
4.2. Octree Convergence

The repository provides several input `.yaml` files that serve as example configurations in order to be used as a starting point to build custom simulations. For evaluation of the octree container, a new scenario was derived from `fallingDrop.yaml`. The following paragraph explains the benefits of this new scenario.

One assumption about the octree container is that it allows for fast iteration of particle pairs, especially for scenarios with inhomogeneously distributed particles. This should be attributed to the fact that an octree is able to adapt its structure to different particle configurations. Octree leaf nodes offer the option of splitting themselves up whenever the number of enclosed particles exceeds the splitting threshold C and the following condition holds: A new leaf must not become smaller than the interaction length along any direction. In other words, there exists a minimum size for octree leaves. If there are too many particles inside a simulation box and an octree container is unable to split itself up because of the minimum size constraint, the structure of an octree converges against the regular cell grid layout of a linked cells container. As a result, the octree will most likely provide slower pairwise iteration performance than linked cells since gathering all nodes that contain

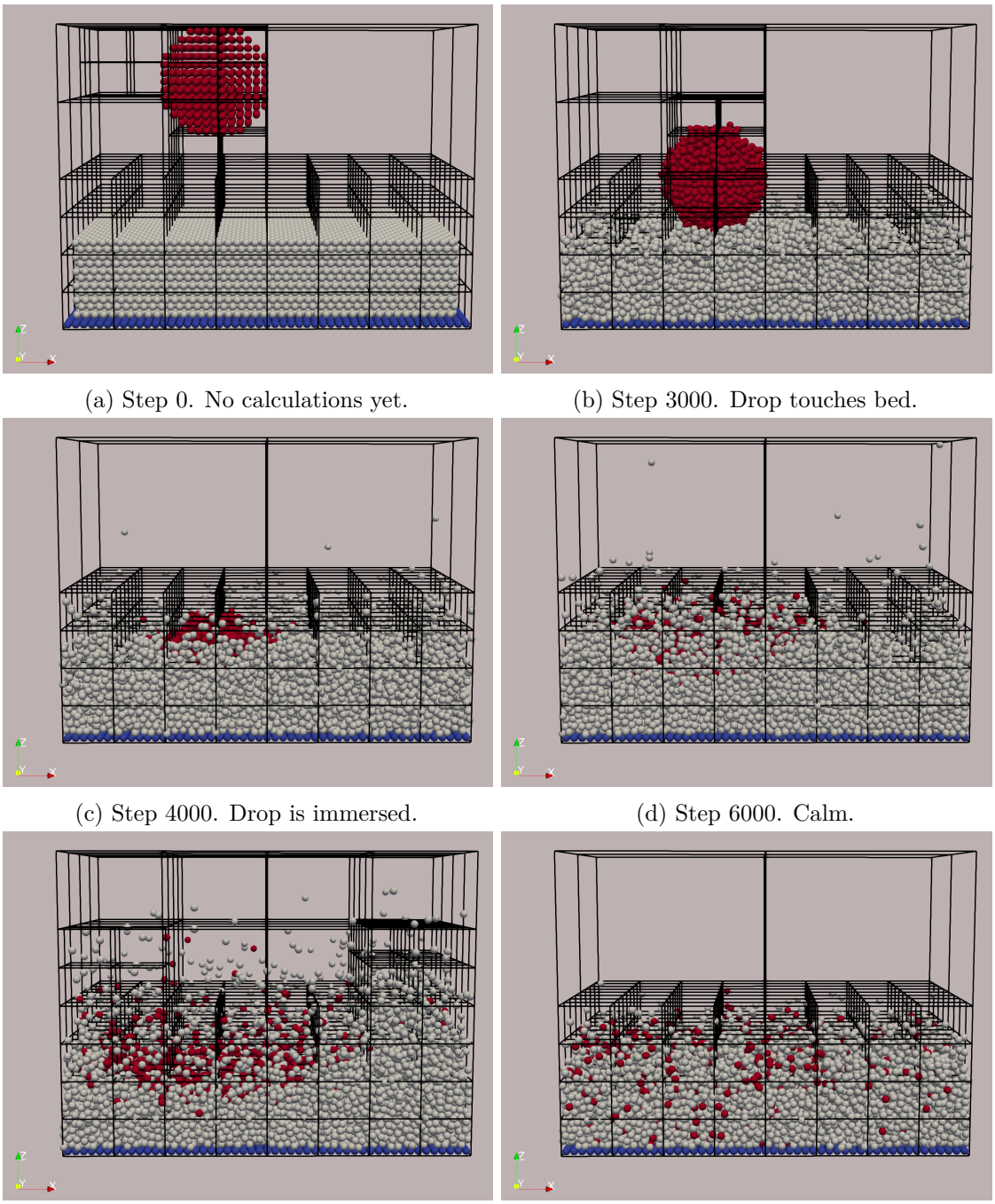


(a) Step 50. The octree containing the halo particles is also drawn (white wireframe) along the owned octree (black wireframe).



(b) Step 4000. The drops touch the bed.

Figure 4.1.: This figure shows the behavior of the octree container when running the inhomogeneous simulation. Finer space divisions compared to Figure 4.2 can be seen around the bigger drop.



(a) Step 0. No calculations yet.

(b) Step 3000. Drop touches bed.

(c) Step 4000. Drop is immersed.

(d) Step 6000. Calm.

(e) Step 8500. Small shock wave.

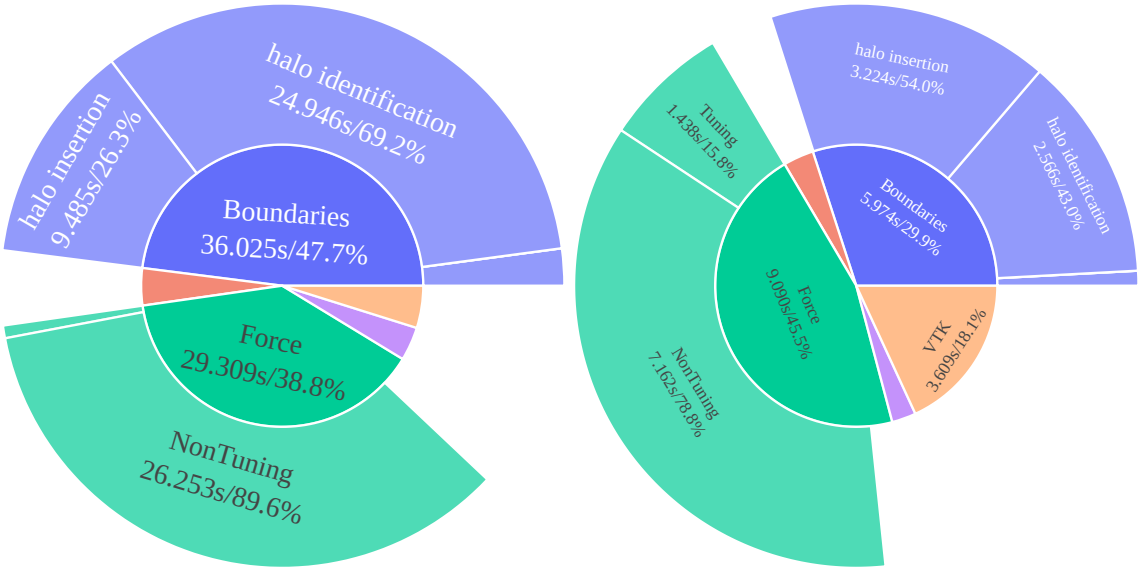
(f) Step 45000. Homogeneous fluid.

Figure 4.2.: This figure shows the behavior of the octree container when running the falling drop simulation. A drop (red particles) falls down into a bed of particles (white) which lies on top of an immovable layer (blue particles). The octree container is shown as a black wireframe mesh. Apart from the initial configuration (Figures 4.2a and 4.2b) and the shock wave (Figure 4.2e), the octree behaves just like a linked cells container on the bottom half since it cannot further split its leaves because the leaves would become smaller than the interaction length. The screenshots were taken using ParaView.

particles – which are the octree's leaves – does not happen in constant time. Figure 4.2 shows this problem based on the falling drop scenario.

A new scenario called `inhomogeneous.yaml` was introduced: It contains drops of different sizes and a thinner bed. The simulation box is almost twice as big as the box of the falling drop scenario. This gives the octree much more freedom to take advantage of its splitting behavior. Many of the performance measurements in this chapter were taken using this scenario. Figure 4.1 shows two screenshots of this experiment. The `md-flexible` configuration for the experiments is shown in Appendix B.2.

4.3. `md-flexible` Performance



(a) Octree, entire run took 75,607 s

(b) Linked cells, entire run took 19,986 s

Figure 4.3.: This figure shows timing information gathered from running the inhomogeneous scenario with `md-flexible`. High-level parts are located in the center of the circles. Those are split up further towards the edge. The charts only include information from the `Simulation` stage. Parts that take a too small portion of time are excluded here, however, the precise data, including the omitted parts, can be found in Tables C.3 and C.4.

A first performance analysis is discussed in the following. Figure 4.3 serves as a foundation for the argumentation. Both figures show which parts of `md-flexible` take specific fractions of the whole run time for different container types. One note beforehand, linked cells is a highly optimized container that has been tested and refined over several years. Due to the novelty of its implementation, the octree's performance is close to factor 4 slower than linked cells.

Nevertheless, some general trends can still be identified. In both runs, roughly 70-85% of the time is spent in the calculation of so-called periodic boundary conditions and force calculations. These are the parts in which the containers' performance becomes the most

evident. During force calculations, the pairwise iteration strategies introduced in Section 3.3 are used. The parts marked as `NonTuning` are those in which the actual simulation is executed. The linked cells container can choose from a large number of traversals available, which can be seen in the relatively large portion of force calculations occupied by `Tuning`. Since there are only two traversals available to the octree, the tuning phase is much smaller for the new container.

4.4. Boundary Conditions

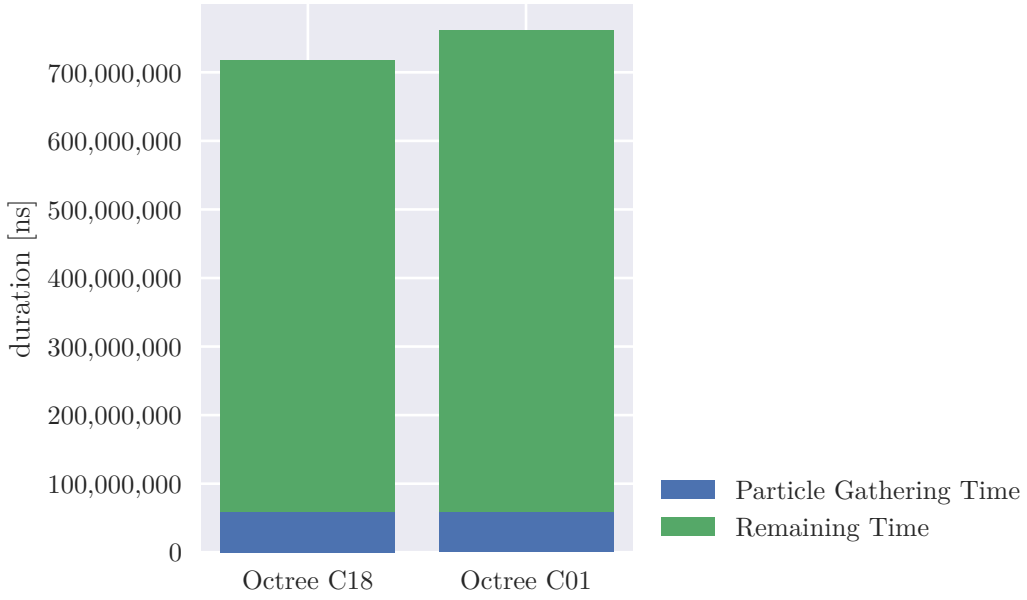


Figure 4.4.: This figure shows the absolute time it takes to execute the `halo identification` step (shown in Figure 4.3) for each of the available octree traversals. The tests were conducted using the `fallingDrop.yaml` scenario and running it for 100 iterations.

The most cost intensive task of the octree is boundary calculations. During the analysis of the new container, one part of these calculations was identified to be especially costly. As mentioned in Section 2.2.1, an important part of every container data structure is fast access to particles inside the container. For instance, identifying particles that enter or leave the simulation box is done by iterating over all particles and checking whether their position is still within bounds after each simulation step. It is therefore mandatory to provide fast access to all particles, preferable in $\mathcal{O}(1)$ for each particle. At the moment, `AutoPas` uses iterators with a special interface to iterate over particles in a cell. Two functions `begin()` and `at()` serve as the interface to obtain an iterator from a cell. `at(`i`)` takes an integer index i that ranges from 0 to $n-1$ where n is the number of particles. The octree solely stores enclosed particles inside its leaves. There is no linear data structure from which the

particles could be accessed directly.

At earlier stages of this thesis, access to particles via `at(`i`)` was implemented by traversing the tree, decrementing i with each particle, and returning a particles whenever the counter i reached zero. This method proved itself very inefficient since it lead to an access complexity of $\mathcal{O}(n^2)$ when iterating over all n particles within the container. After observing that `at(`i`)` was called n times only after one call to `begin()`, the following optimization was implemented: `begin()` fills a `std::vector` with pointers to the particles elements inside the octree upfront. This allows `at(`i`)` to access every particle i inside the octree in constant time. On the other hand, this method has a huge drawback: The copying process is costly. Figure 4.4 shows that the time it takes to copy all particles out of the octree is significant for one example. There are several other places inside `AutoPas` that involve particle iteration. All of them require copying all particles out of the octree as previously described. In other containers such as linked cells, there is no obligation for this kind of copying to happen. Due to their static layout, they can provide a constant mapping from consecutive indices to particles within the container. This is not possible for variably-size cells like the octree cells.

4.5. Pairwise Traversal Comparison

As shown in Figure 4.3, the `Force` calculations account for a big part of the overall run-time cost, both for the octree as well as for linked cells. Therefore, this part is analyzed in the following.

4.5.1. High-level Comparison

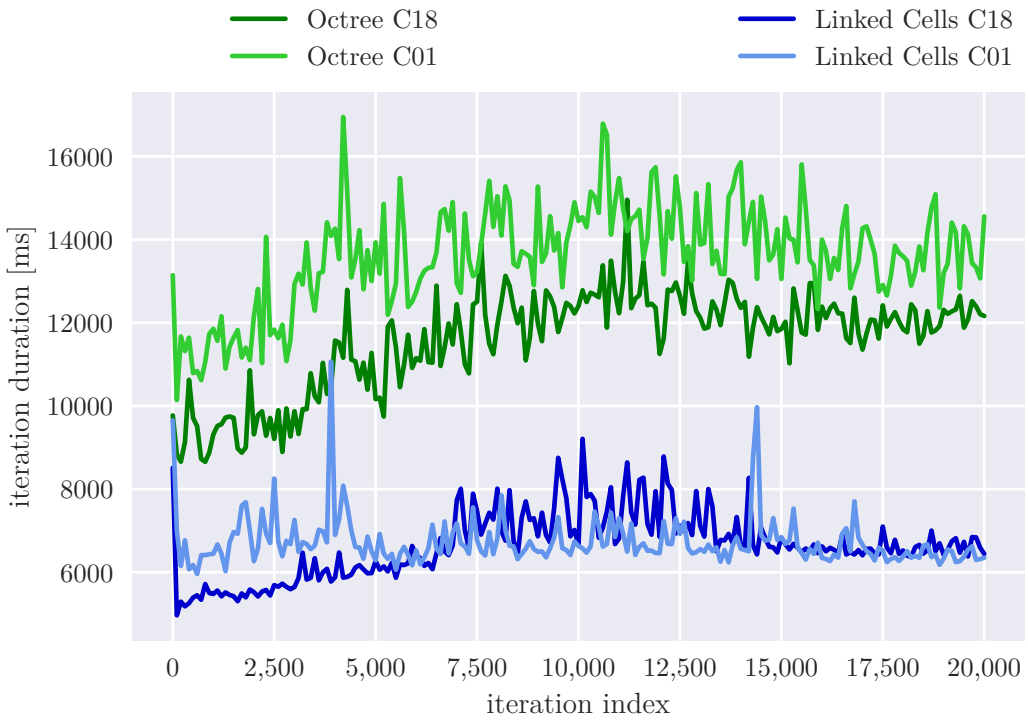


Figure 4.5.: This figure shows how long one simulation step of the inhomogeneous scenario took regarding the pairwise iteration both for linked cells and the octree. For both containers, two traversals are shown. The simulation was run for 20000 steps. To reduce the number of samples, only every 200th sample is shown. This already supports the claim that the octree's traversal techniques are slower.

Figure 4.5 shows how much time each iteration step takes for different configurations. It becomes evident that both octree traversal techniques are slower than the respective linked cells equivalents. This arises from the fact that one pairwise iteration step takes longer in the octree implementation. The amount of time corresponding to one iteration step is shown in Figure 4.5. The figure shows that the simulation becomes harder to execute between steps 2 500 and 12 500. All four traversals follow this pattern. In conclusion, iteration performance of the octree must become faster by a factor of two in order to beat the linked cells traversal speeds.

4.5.2. Work Distribution in Pairwise Traversal

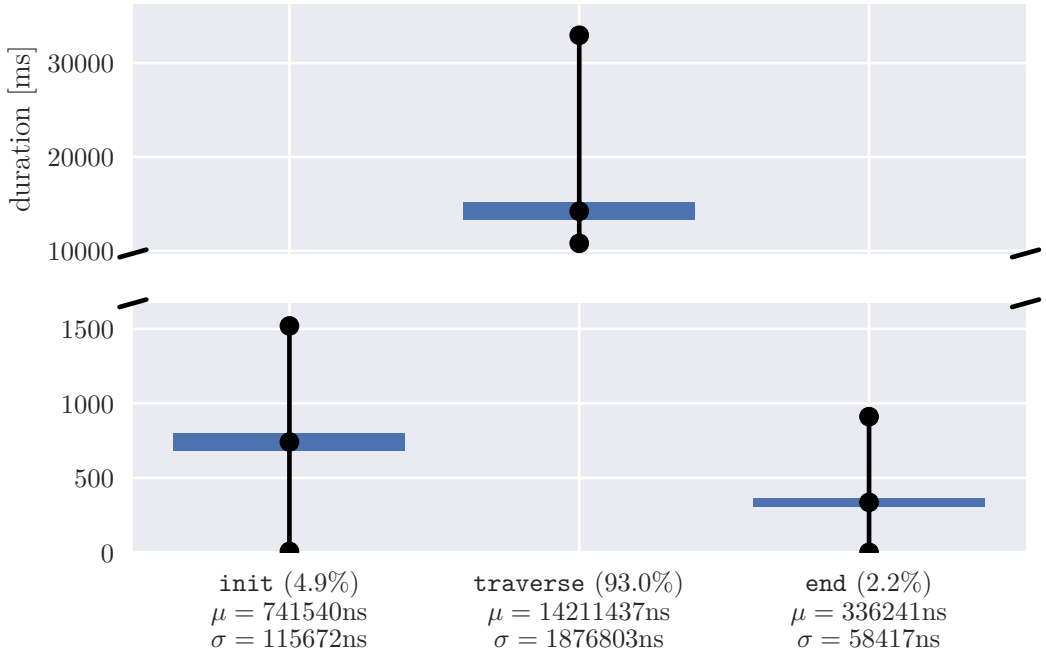


Figure 4.6.: This figure illustrates how much time each phase of pairwise iteration takes on average. Those phases include `init`ialization, the actual pairwise `traversa`l, and an `end` section. `init` includes the time it takes to gather all leaves required for pairwise traversal. Furthermore, `init` and `end` contain code that loads special buffers optimizing the cache-efficiency of particle accesses. The standard deviation σ is shown as the blue bar around the mean μ (black center dot) of the time each phase took in nanoseconds. The black bars show the entire value range, from the minimum measured value to the maximum (these two are also marked by the black dots at the end of the black bars).
Overall, it can be observed that the setup and tear-down cost of the traversal is negligible compared to the cost of pairwise iteration.

The pairwise iteration includes three phases in `AutoPas`: A step function that is used to initialize the traversal and prepare it, the actual iteration phase, and a step in which the traversal is torn down. In the following, those phases are called `init`, `traverse` and `end` respectively.

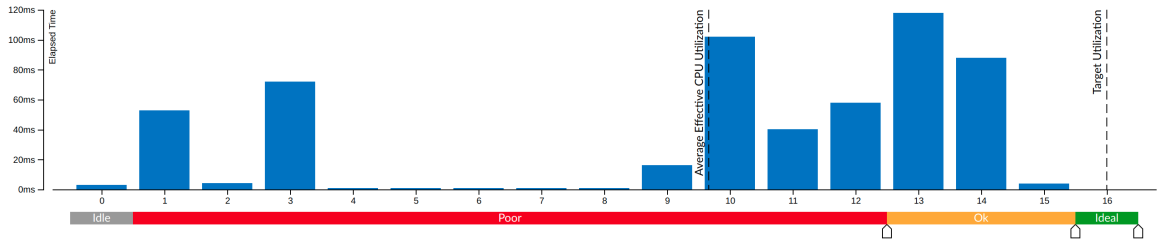
In `init`, the linked cells container loads special buffers for cache-efficiency on demand. The octree, on the other hand, gathers all leaves of the tree. These are required for iteration in the `traverse` step. Figure 4.6 shows the distribution of the time the different phases take. Since the `traverse` step takes up over 90% of the entire iteration time, both of the other steps can be neglected in terms of run-time cost.

The leaf gathering within the octree's pairwise traversal is working single-threaded at the moment. It could be sped up by running it in parallel, which should be easy to achieve.

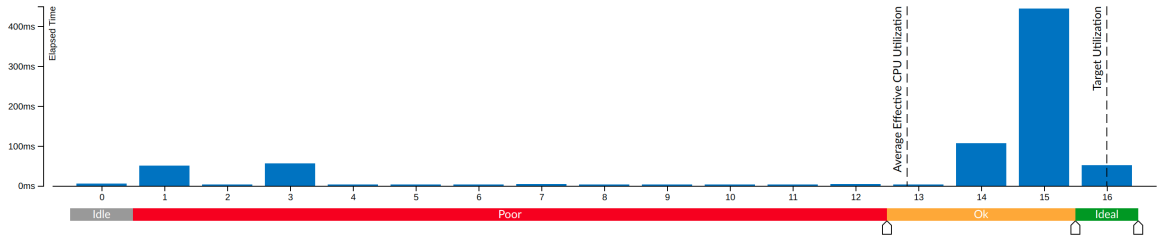
However, since the overhead of `init` is so small, the focus for optimization should be drawn towards the `traverse` phase because here, improvements would have a significantly bigger impact on the overall performance, as explained in the previous paragraph.

4.5.3. Impact of Different Cell Sizes

The octree and the linked cells containers were compared to each other using the popular performance analysis tool VTune[1] by Intel.



(a) Octree



(b) Linked cells

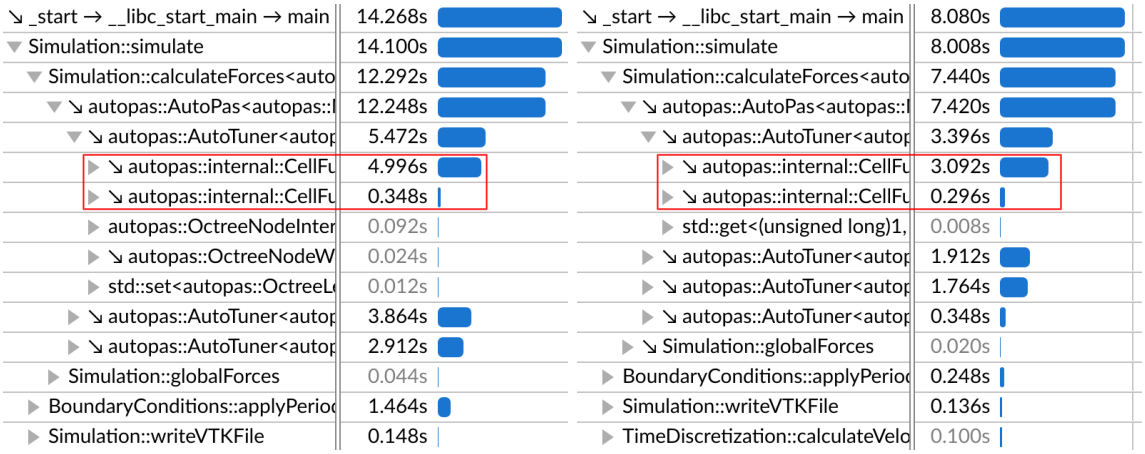
Figure 4.7.: This figure shows two screenshots of the "Effective CPU Utilization Histograms" inside VTune. Both charts display how much of the run-time was spent in which degree of parallelization. Every potential number of cores is shown on the x-axis, whose values range from 0 cores to 16 cores, which is the maximum number the CPU offers with hyper-threading enabled. The y-axis indicates the time that was spent using one specific number of processing units. In the best case, a very high clustering on the right is achieved. This means that the application utilizes the full potential of the CPU. The experiment was conducted using 10 iterations of the falling drop scenario. Both containers used their respective C18 traversal.

One potential explanation for the drawbacks regarding the speed of the octree is that it does not utilize the capabilities of multi-core CPUs. Therefore, a hotspot analysis was executed inside VTune. The results of the measurements can be seen in Figure 4.7. The analysis revealed that the octree lacks potential in terms of parallel performance since it has an "Average Effective CPU Utilization" of 9.7 threads, where linked cells reaches a degree of parallelization of 12.8 threads.

VTune provides more information regarding the individual experiments: It captures the time different statements take in order to identify performance hotspots. These are parts of

[1] `https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/vtune-profiler.html`

the code that take up a disproportionately large amount of the run-time and are therefore great candidates for optimizations.



(a) Octree (b) Linked cells

Figure 4.8.: This figure shows two screenshots of the "Top-down Tree" analysis in VTune of the same experiment as in Figure 4.7, but using only one thread. Both tables list the absolute duration (right column) that each statement (left column) takes during the run. The data is organized hierarchically, `Simulation::calculateForces<...>` is a child of `Simulation::simulate` for instance. The calls to `processCellPair()` and `processCell()` are marked with a red box.

The performance hotspots can be identified by looking at the so-called "Top-down Tree" output of VTune. Figure 4.8 shows the tables for both, a run with the octree and linked cells. One small detail is especially important here: The call to `processCellPair()` of the octree takes 4,996 s whereas the same call takes only 3,092 s for linked cells. What `processCellPair()` does was already explained in more detail in Section 3.3.1. The method takes two cells, computes all particle pairs using the cartesian product, and executes the pairwise force computation. This approach was introduced as direct sum in Section 2.2.2 and has a high computational complexity of $\mathcal{O}(a \cdot b)$ where a is the number of particles in the first container and b the number of particles in the other one. Assuming a homogeneous particle distribution, every cell contains n particles on average. The direct sum computation between two cells then happens in $\mathcal{O}(n^2)$. Thus, the average number of particles per cell hugely impacts the overall performance.

The assumption is the following: The octree container has, on average, more particles inside its cells than linked cells. In order to investigate this issue, the C18 traversals of the respective containers were extended such that they collect data during the iteration. This data is then logged, comprising the following items: The total number of particles participating in the experiment, the number of cells processed, the average number of particles per cell, and the total number of pairs generated in the run. The last one is particularly interesting since it should be directly proportional to the duration of the simulation.

Metrics of C18 Traversals			
Traversal	Cells Processed	Average Particles per Cell	Total Pairs
Octree C18	9 653	28.18	20 668 517
Linked Cells C18	28 464	10.97	6 690 503
Factor		2.57	3.08

Table 4.1.: This table shows different metrics for the C18 traversals of the octree and the linked cells container in every row. The last row contains factors by which the numbers generated by the octree's C18 traversal are higher than by linked cell's C18 traversal. All data was obtained from two runs of the falling drop scenario.

The following conclusions can be drawn from Table 4.1: The average number of particles per cell is significantly higher in the octree. This results in a larger computational cost for the direct sum computations between the cells. The claim is supported by the number of total pairs: The C18 traversal of the octree computes 3.08x as many pairwise interactions as the linked cells equivalent.

These findings make it easier to understand why the octree lacks potential in terms of speed with these parameters. Having almost thrice the average number of particles per cell drastically increases the run-time overhead of the pairwise traversal. The reason for this high average number of particles per cell is the size of the octree leaf nodes, which are the cells that contain the particles. Table 4.2 shows the lengths of the smallest cells in each container. It becomes evident that the cells of the linked cells container are very close to the interaction length, which is the minimum size requirement in this experiment. If cells are exceedingly full of particles, they need to be as small as possible in order to keep the number of potential pairs, occurring between neighboring cells, down. The octree is not able to achieve this in this scenario since its cell geometry is determined by the simulation box. All octree nodes are scaled versions of the simulation box. As a result, a minimum-sized octree leaf node is almost twice the interaction length long on the x-axis. This leads to a much larger number of particles per cell compared to the compact cells of the linked cells container, which then yields the high average number of particles per cell for the octree.

Minimum Cell Dimensions			
Container	x-axis	y-axis	z-axis
Octree	6.375	3.875	4.724
Linked Cells	3.400	3.444	3.436

Table 4.2.: This table lists the dimensions of the smallest cell for the octree and linked cell containers. The interaction length for this experiment was set to 3.3. All data was obtained from two runs of the falling drop scenario.

To conclude, it is extremely desirable to build cells whose dimensions are as close as possible to the interaction length, for areas with a high particles density. For the octree cells, this is the case if the simulation box is cubic and the dimensions of the box are a multiple of the interaction length.

5. Future Work

5.1. Investigating Further

Chapter 4 already provided detailed information about the octree's performance. Compared to linked cells in terms of speed, the octree has not yet reached the degree of optimization of the linked cells container. However, there is still potential for further investigations in order to improve parts of the octree.

The first question to be answered is: Why is the pairwise iteration slower? Figure 4.5 shows the speed problem the octree traversals currently have. It was shown in Section 4.5.1 that the problem does not lie within the leaf collection since this only takes a small fraction of the pairwise iteration time. Thus, there must exist an issue within another part of the code. Linked cells is optimized to provide cells in a very memory- and cache-friendly way. The octree, on the other hand, does not embody any of those optimizations yet. Even though this might pose a problem at some point during the `AutoPas` project, there may be other issues that prevent the octree from unleashing its full potential yet.

The boundary calculation part mentioned in Section 4.4 is much slower than in other containers. Figure 4.3 illustrates the difference. Calculations with the octree container take up to 6x more time than the reference data structure. Section 4.4 tries to give reasons why the process is so slow, but the biggest cost factor is still hidden. Therefore, it is necessary to investigate this issue further: Why is the particle gathering so slow? Does this impact the performance in parts other than the boundary condition calculations?

Plans for `AutoPas`'s future involve moving away from the `begin()`- and `at(`i`)`-based iteration scheme towards a callback-based technique for particle gathering. The octree implementation would benefit a lot from this step since the copying described in Section 4.4 could be substituted with an approach that keeps the particles inside the octree.

During this thesis, every experiment was conducted with a tree split threshold of 16. It is also possible to set different thresholds using a newly introduced command line or `.yaml` file parameter for `md-flexible`. Using this parameter, the impact and quality of the chosen threshold could be evaluated.

As discussed in Section 4.5.3, the cell size plays a central role in the performance of the octree traversals. It would be extremely interesting to find out whether the size can be decoupled from the simulation box geometry. One potential fix is to split the simulation area into regions that are as cubic as possible. Multiple octrees could then be instantiated in order to optimally fill the entire simulation space.

5.2. Building on the Existing Implementation

At the moment, the `updateContainer()` method is implemented in a naive way: It copies all particles, destroys the tree, and rebuilds it every time. One extension of the current

implementation might be to introduce different update strategies for the octree container that allows for reusing of the existing tree as long as possible. This can be achieved by only rebuilding parts of the octree where particles went outside of the leaves. Nodes can then be split or united depending on the number of particles enclosed.

It is important to note that this optimization is only necessary if empirical evidence, showing that `updateContainer()` takes up a large portion of the run-time cost, is gathered first. Otherwise, the optimization focus should be drawn towards the pairwise iteration strategies.

5.3. Implementation of further Approaches

The octree merely represents one way of implementing a space adaptive container data structure. In general, there exists a wide variety of tree-based data structures that can be used for efficient particle storage. One of those was already mentioned in the beginning in Section 2.2.2: The k-d tree. However, there exist several ideas for other tree-based space-adaptive container data structures, for instance the ball tree or the M-tree. [Omo89, CPZ97] Various implementations of those approaches could be developed and compared, which would allow the tuner unit to choose from a wider range of container types for different use-cases.

6. Conclusion

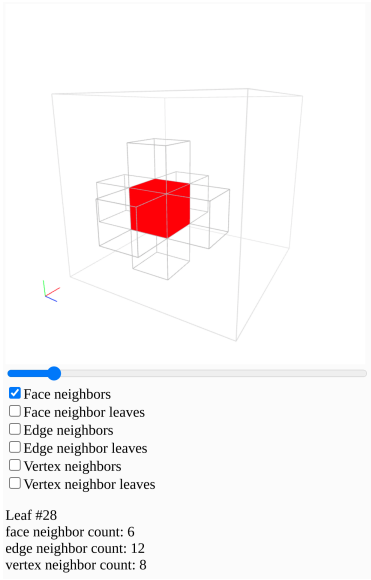
In the following, the most important aspects of the discussion are summarized. Each point refers to one major chapter of this thesis.

1. At the beginning of this thesis, the theoretical background about particle physics and the software used was introduced. The first chapter concludes with an overview of different container data structures and draws the focus towards octrees, which are special data structures that contain dynamically sized boxes.
2. The second part gives an in-depth explanation of the software that was developed during the practical part of this thesis. It starts by introducing a new orientation system that is used in the implementation of the octree container. Subsequently, a detailed explanation of the new container itself is provided. The octree is modeled using three classes, an abstract base class and two derived children. These serve as tree nodes, one for inner nodes with pointers to eight children and one for leaf nodes storing actual particles. Support for common operations like inserting of particles, clearing, and updating is explained. Last, the different visualization techniques, one for the browser and one for an existing software package, are shown. These were especially helpful for analysis and debugging.
3. In the analysis part, the validity of the implementation is made evident by providing screenshots of the visualization. After that, the performance of the novel octree implementation is evaluated by comparing the performance to the existing linked cells container. It could be demonstrated that the octree's pairwise iteration speed is slower by a factor of two than the reference container. Overall, the runs that were utilizing the octree container are slower by a factor of three than with linked cells. The two main reasons behind this behavior are the following: First, a lot of copying is involved in the octree's pairwise iteration algorithms. Second, the octree traversals are not as optimized as the reference container's. It is shown that the dimensions of the octree leaves, implied by the simulation box geometry, highly impact the iteration speed of the container. With a well-chosen simulation region, the octree iteration performance could be improved.
4. The Future Work section shows further optimizations, for which starting points are suggested. For instance, different tree split thresholds could be chosen to allow for optimizing the octree configuration. Other than that, additional analysis on the octree traversals could lead to improvements in terms of memory- and cache-friendliness. Last but not least, alternative approaches for space-adaptive containers, such as k-d trees, could be implemented and evaluated.

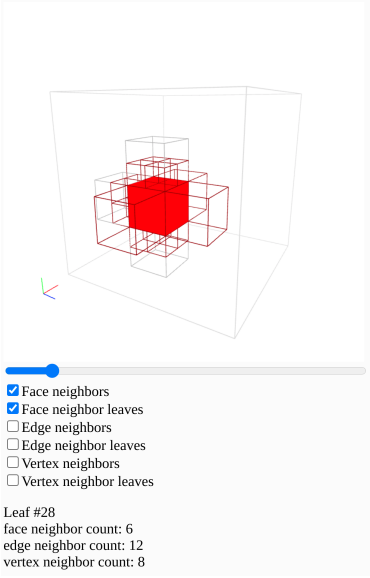
Part II.

Appendix

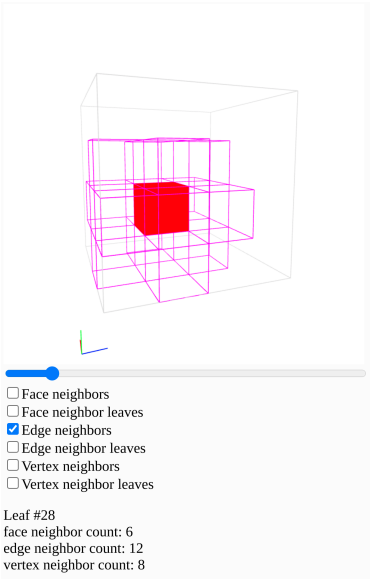
A. Browser Octree Screenshots



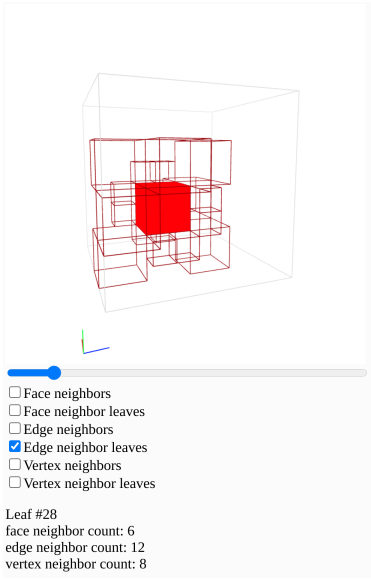
(a) Image of the browser octree tool with face-neighbor-view enabled



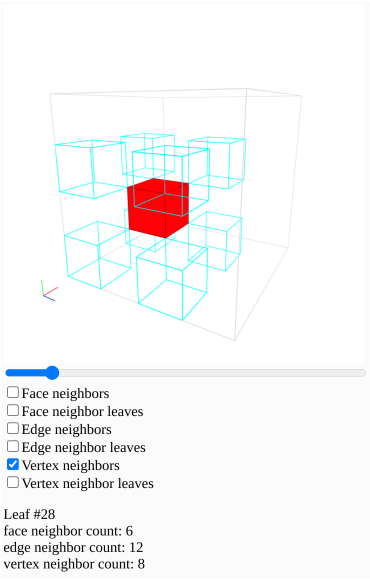
(b) Image of the browser octree tool with both face-neighbor-view and face-neighbor-leaf-view enabled



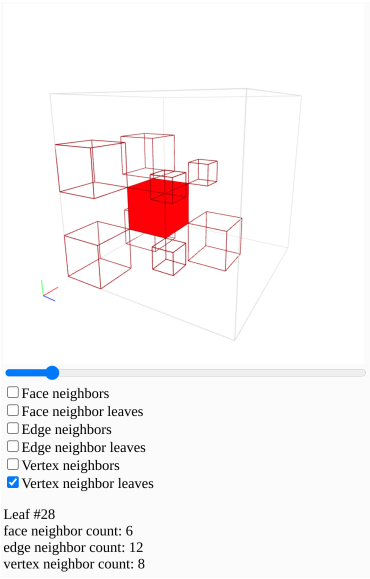
(c) Image of the browser octree tool with edge-neighbor-view enabled



(d) Image of the browser octree tool with edge-neighbor-leaf-view enabled



(e) Image of the browser octree tool with vertex-neighbor-view enabled



(f) Image of the browser octree tool with vertex-neighbor-leaf-view enabled

Figure A.1.: Screenshots of the browser octree visualization tool with different selections. In all screenshots, leaf number 28 was chosen randomly. The selected leaf is highlighted as a red box. The blue slider can be used to select other leaves. Different neighbor visualization configurations can be selected using the pickers below the view. Neighbors are visualized as wireframes around the current box.

B. Algorithms and Configuration

B.1. Helper Functions for Lookup-based Neighbor Finding

The following explains functions used in Chapter 3. All of the functions mentioned were introduced in [Sam89b].

`GRAY(`n`)` This function yields `true` if and only if the given node n has children.

`FATHER(`n`)` This function yields a pointer to n's parent, or `nullptr` if there is no parent. Every `OctreeNodeInterface` maintains a pointer to its parent, which is `nullptr` in the root node. This can also be seen in Figure 3.2.

`SONTYPE(`n`)` This function yields the `Octant` in which a given node n is, relative to its parent node.

B.2. Configuration for the Inhomogeneous Scenario

The following output was generated by `md-flexible` when running the inhomogeneous scenario.

```
container                    :  [Octree]
verlet-rebuild-frequency     :  4
verlet-skin-radius           :  0.3
selector-strategy            :  Fastest-Absolute-Value
data-layout                  :  [AoS, SoA]
traversal                    :  [ot_c01, ot_c18]
tuning-strategy              :  full-Search
mpi-strategy                 :  no-mpi
tuning-interval              :  1000
tuning-samples               :  3
tuning-max-evidence          :  10
functor                      :  Lennard-Jones (12-6)
newton3                      :  [disabled, enabled]
cutoff                       :  2
box-min                      :  [-0.5, -0.5, -0.5]
box-max                      :  [100.5, 80.5, 47.3982]
cell-size                    :  [1]
deltaT                       :  0.0005
iterations                   :  100
periodic-boundaries          :  true
```

```
# object definitions are omitted for brevity

globalForce                     :  [0, 0, -12]
vtk-filename                    :  inhomogeneous
vtk-write-frequency             :  25
log-level                       :  1
no-flops                        :  false
no-end-config                   :  true
no-progress-bar                 :  false
```

C. Used Tables

r = `getOppositeDirection(`d`)`	
`Any` d	`return` value r
L	R
R	L
D	U
U	D
B	F
F	B
LD	RU
LU	RD
LB	RF
LF	RB
RD	LU
RU	LD
RB	LF
RF	LB
DB	UF
DF	UB
UB	DF
UF	DB
LDB	RUF
LDF	RUB
LUB	RDF
LUF	RDB
RDB	LUF
RDF	LUB
RUB	LDF
RUF	LDB

Table C.1.: This table shows the mapping from a direction d to its opposite direction r using the lookup table-based function `getOppositeDirection()`.

r = `getAllowedDirections`(d)	
`Any` d	`return` value r
L	{LDB, LDF, LUB, LUF}
R	{RDB, RDF, RUB, RUF}
D	{LDB, LDF, RDB, RDF}
U	{LUB, LUF, RUB, RUF}
B	{LDB, LUB, RDB, RUB}
F	{LDF, LUF, RDF, RUF}
LD	{LDB, LDF}
LU	{LUB, LUF}
LB	{LDB, LUB}
LF	{LDF, LUF}
RD	{RDB, RDF}
RU	{RUB, RUF}
RB	{RDB, RUB}
RF	{RDF, RUF}
DB	{LDB, RDB}
DF	{LDF, RDF}
UB	{LUB, RUB}
UF	{LUF, RUF}
LDB	{LDB}
LDF	{LDF}
LUB	{LUB}
LUF	{LUF}
RDB	{RDB}
RDF	{RDF}
RUB	{RUB}
RUF	{RUF}

Table C.2.: This table shows the mapping from a direction d to its set of allowed directions r using the lookup table-based function `getAllowedDirections()`.

Timing			
Action	%	[s]	[ns]
Time total	100.00	75.607	75 607 329 788
Initialization	0.01	0.009	8 622 394
Simulation	99.98	75.595	75 595 177 117
Boundaries	47.66	36.025	36 024 901 945
updateContainer	4.42	1.593	1 592 906 858
entering particles	100.00	0.000	140 889
halo identification	69.25	24.946	24 945 861 604
halo insertion	26.33	9.485	9 485 021 519
Position	4.24	3.204	3 203 978 252
Force	38.77	29.309	29 308 531 786
Tuning	1.86	0.546	545 614 827
NonTuning	89.58	26.253	26 253 296 594
Velocity	3.83	2.898	2 898 251 874
VTK	4.81	3.633	3 633 258 586

Table C.3.: This table shows the timing information when running the inhomogeneous scenario using an octree container on a workstation.

Timing			
Action	%	[s]	[ns]
Time total	100.00	19.986	19 985 906 258
Initialization	0.05	0.009	9 264 309
Simulation	99.93	19.973	19 972 713 391
Boundaries	29.91	5.974	5 973 546 776
updateContainer	2.83	0.169	169 176 661
entering particles	0.00	0.000	93 599
halo identification	42.96	2.566	2 565 995 597
halo insertion	53.97	3.224	3 224 125 846
Position	3.49	0.697	696 758 828
Force	45.51	9.090	9 090 213 997
Tuning	15.82	1.438	1 438 358 001
NonTuning	78.79	7.162	7 162 264 433
Velocity	2.76	0.551	551 050 255
VTK	18.07	3.609	3 609 412 878

Table C.4.: This table shows the timing information when running the inhomogeneous scenario using a linked cells container on a workstation.

List of Figures

List of Tables

Listings

Bibliography

[Ben75] Jon Louis Bentley. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM*, 18(9):509–517, September 1975.

[CPZ97] Paolo Ciaccia, Marco Patella, and Pavel Zezula. M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. pages 426–435, 1997.

[GKZ07] Michael Griebel, Stephan Knapek, and Gerhard Zumbusch. *Numerical Simulation in Molecular Dynamics: Numerics, Algorithms, Parallelization, Applications*. Springer Publishing Company, Incorporated, 1st edition, 2007.

[GSBN20] Fabio Alexander Gratl, Steffen Seckler, Hans-Joachim Bungartz, and Philipp Neumann. N Ways to Simulate Short-Range Particle Systems: Automated Algorithm Selection with the Node-Level Library AutoPas. *Computer Physics Communications*, December 2020. Preprint submitted.

[Mac13] Ernst Mach. *The Science of Mechanics: A Critical and Historical Exposition of its Principles*. Cambridge Library Collection - Physical Sciences. Cambridge University Press, 2013.

[Mea80] Donald Meagher. Octree Encoding: A New Technique for the Representation, Manipulation and Display of Arbitrary 3-D Objects by Computer. 10 1980.

[Omo89] Stephen M. Omohundro. Five Balltree Construction Algorithms. Technical report, 1989.

[Sam89a] Hanan Samet. Implementing ray tracing with octrees and neighbor finding. *Computers and Graphics*, 13(4):445–460, 1989.

[Sam89b] Hanan Samet. Neighbor finding in images represented by octrees. *Computer Vision, Graphics, and Image Processing*, 46(3):367–386, 1989.

[Ver67] Loup Verlet. Computer "Experiments" on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules. *Phys. Rev.*, 159:98–103, Jul 1967.

[WLZ14] Tsz ho Wong, Geoff Leach, and Fabio Zambetta. An adaptive octree grid for GPU-based collision detection of deformable objects. *The Visual Computer*, 30:729–738, 06 2014.