SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY - INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

An Exploration of Different Approaches for Implementing Verlet Lists in AutoPas

Luis Gall

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY - INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

An Exploration of Different Approaches for Implementing Verlet Lists in AutoPas

Eine Untersuchung verschiedener Herangehensweisen zur Implementierung von Verlet Lists in AutoPas

Author:	Luis Gall
Supervisor:	Bungartz Hans-Joachim, Prof. Dr. rer. nat. habil.
Advisors:	Newcome Samuel, M.Sc.; Markus Mühlhäußer, M.Sc.
Submission Date:	16.08.2023

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

Munich, 16.08.2023 Luis Gall

Abstract

In molecular dynamics, the most expensive part regarding the computational effort is calculating the pairwise force interactions between particles. Most improvements aim at efficiently identifying the nearby particles for the calculation of forces, which usually converge to zero with a growing distance. The Verlet Lists algorithm stores neighbor particles in a list and updates them every few iterations. This thesis includes several implementations for reducing the time spent on the neighbor list rebuilds. The first improvement aims to reduce the number of rebuild iterations. To achieve this, the rebuild criterion is enhanced to be more dynamic by being coupled with the particle movement since the last rebuild. The conducted runtime measurements in different experiments show an advantage of this improvement for scenarios with a high velocity difference over time. Second, the effort for the rebuilds is decreased by implementing a partial strategy to only update the lists which are invalidated by particle movement. This partial rebuilding procedure introduces much computational overhead but can beat the reference implementation in large simulation domains for a small Verlet skin size. Furthermore, a new rebuild traversal is presented, which can reduce the time spent on updating the neighbor lists by achieving a lower CPU spin time. These algorithms are implemented as a part of the open-source C++ library AutoPas, which provides a wide portfolio of optimizations. In conclusion, the results show that every improvement can perform better than its respective reference implementation in at least one scenario and is therefore a valuable extension of AutoPas.

Kurzfassung

Im Bereich der Molekulardynamik besteht der rechenaufwendigste Teil in der Bestimmung der Kräfte zwischen Partikeln. Diese Kräfte konvergieren meist gegen null mit wachsender Distanz zwischen den Molekülen. Das Ziel der existierenden Algorithmen besteht für jedes Partikel in der Identifizierung der naheliegenden Partner, für welche sich der Aufwand der Kräfteberechnung lohnt. Der Verlet Lists Algorithmus speichert die naheliegenden Partikel in sogenannten Nachbarlisten und aktualisiert diese in bestimmten Abständen. Die vorliegende Arbeit umfasst mehrere Implementierungen um den Zeitaufwand für die Aktualisierungen der Nachbarlisten zu reduzieren. Die erste Verbesserung zielt darauf ab, die Anzahl der Iterationen des Neuaufbaus zu verkleinern. Um dies zu erreichen, wird die Notwendigkeit der Aktualisierungen der Nachbarlisten mit der Partikelbewegung seit dem letzten Neuaufbau gekoppelt. Die durchgeführten Laufzeitmessungen im Rahmen verschiedener Experimente zeigen einen Vorteil des dynamischen Kriteriums für Szenarien mit einer hohen Geschwindigkeitsdifferenz über die Simulationszeit. In einer weiteren Verbesserung wird der Aufwand für die Aktualisierungen verringert, indem nur bestimmte Listen aktualisiert werden, welche durch Partikelbewegungen ungültig geworden sind. Das partielle Neuaufbau-Verfahren verursacht jedoch einen hohen Zusatzaufwand, ist dennoch bei kleinen Verlet Skin Größen in der Lage die Referenzimplementierung mit Hinblick auf die Laufzeit zu übertreffen. Zusätzlich wird eine neue Strategie entwickelt, um die Traversierung des Simulationsgebiets für den Neubau der Nachbarlisten zu beschleunigen. Diese Verbesserung kann eine geringere CPU-Wartezeit erzielen und damit den Aufwand für die Aktualisierungen der Nachbarschaftslisten weiter verringern. Alle Verbesserungen wurden als Erweiterung der open-source Bibliothek AutoPas implementiert, welche ein breites Portfolio an Algorithmen und Optimierungen bereitstellt. Da die in dieser Arbeit vorgestellten Implementierungen ihre jeweilige Referenzmethode in mindestens einem Szenario übertreffen können, stellen alle Verbesserungen eine wertvolle Erweiterung von AutoPas dar.

Contents

1. Introduction

In a computer simulation of particles, scientists are interested in the interaction of molecules with different physical properties under certain circumstances like temperature or pressure. Particularly interesting is how particles collide, attract and repel each other, or group into structures. Simulating experiments for protein folding, the formation of membrane structures, or the determination of binding energies are becoming interesting research topics. A concrete application lies in material science, where particle simulations are used to investigate e.g. the crack propagation behavior of nanocrystalline metal. [1] [2]

In molecular dynamics, the most time-consuming part of the simulation is the calculation of the forces between particles. Usually, the force model is defined by the sum of pairwise interactions between each particle, resulting in a complexity of $\mathcal{O}(N^2)$ with N as the number of particles in the simulation. Using Verlet integration for Newton's Laws of Motion, the forces are translated into velocities. These are used to calculate movement for the discretized time steps of the simulation. [3]

To speed up particle simulations by reducing the complexity to $\mathcal{O}(N)$, many different algorithms were developed over the last decades, like Linked Cells or Verlet Lists. Each of them has its strengths and weaknesses, with no algorithm performing best under every possible circumstance. The open-source C++ library AutoPas tackles this problem by providing various algorithms and optimizations. To find the optimal configuration at runtime and react to a changing simulation environment, AutoPas employs dynamic auto-tuning to select the best-performing configuration. [3]

The AutoPas library implements many different features for optimizing the force calculation by parallel traversals or improved data structures. However, relatively less work was done on optimizing the Verlet neighbor list rebuilds.

Therefore, the main goal of this thesis is to decrease the time spent on the list updates and achieve more rebuild flexibility in scenarios with changing circumstances. First, a dynamic implementation of the Verlet list-based containers is discussed, which adapts to a changing simulation environment. This is achieved by coupling the necessity of Verlet neighbor list rebuilds to particle movement. Additionally, a second improvement is presented, which is aiming to rebuild the parts of the simulation domain only which contain invalid neighbor lists. At last, a new traversal is introduced that allows for better parallelization of the neighbor list rebuild routine.

This thesis is structured as follows. First, the theoretical background of molecular dynamics and AutoPas are explained in Chapter 2. The following Chapter 3 provides a short overview of related projects. In Chapter 4, the improvements are described in detail regarding their algorithmic idea and concrete integration into the AutoPas library. Afterward, in Chapter 5, their execution times in chosen experimental scenarios are reported and compared to other existing algorithms. In the end, Chapter 6 discusses some upcoming research possibilities, and the concluding Chapter 7 summarizes the results of this thesis.

2. Theoretical Background

To understand the environment in which this thesis is conducted, the following chapter provides an overview of some helpful concepts of computational molecular dynamics.

2.1. Molecular Dynamics

In the field of molecular dynamics, the motion of particles is governed by Newton's Laws of Motion. For a molecule i with mass m_i, position x_i, and an external force F_i acting on it, its motion according to Newton is given by Equation 2.1. [4]

$$m_i \frac{d^2 x_i}{dt^2} = F_i \tag{2.1}$$

If we want to advance particles over time, we need to calculate the movement of the molecules for the discretized time steps of the simulation. The difference between two time steps is referred to as h. Position of the molecule at the following time step $x_i(t+h)$ can be obtained by using Equation 2.2. [4]

$$x_i(t+h) = x_i(t) + h v_i(t) + \frac{h^2}{2m_i} F_i(t) \tag{2.2}$$

With the help of a Taylor series expansion and some algebraic simplifications, the velocity at the upcoming time step can be calculated by Equation 2.3. This procedure is called the "Velocity Verlet method" and provides good stability and accuracy. [4]

$$v_i(t+h) = v_i(t) + \frac{h}{2m_i}(F_i(t) + F_i(t+h)) \tag{2.3}$$

2.2. Short Range Potentials

A crucial part of every molecular dynamics program is the force computation for every of the N bodies of the simulation. Typically, the force acting on a single particle is the sum of pairwise forces exerted by all other particles as stated in Equation 2.4. [3]

$$F_i = \sum_{j \in particles} F_{i,j} \tag{2.4}$$

To reduce the computational complexity of $\mathcal{O}(N^2)$ resulting from this procedure, characteristics of the commonly used potentials can be exploited. Many force potentials quickly converge against zero with an increasing distance between two particles. Because of this, they are called short-range potentials. This property can be used to reduce the computational complexity to $\mathcal{O}(N)$ by introducing a cutoff radius r_{cut}. Every interaction distance which goes beyond this threshold can be omitted without adding a significant error because these forces are assumed to be zero. [3]

In most cases, the interaction between two particles i and j is described by the Lennard-Jones potential. Either in the concrete formula in Equation 2.5 or in the function graph provided by Figure 2.1, it is visible that the convergence property of short-range potentials holds.

$$V(r_{ij}) = 4\epsilon((\frac{\sigma}{r_{ij}})^{12} - (\frac{\sigma}{r_{ij}})^{6}) \tag{2.5}$$

The necessary parameters are the distance between both particles r_{ij}, the minimum of the potential ϵ, and the zero crossing σ where no force is exerted. The first part of Equation 2.5 models a short-range repulsion and the second is an induced dipole-dipole interaction. [5] [6]

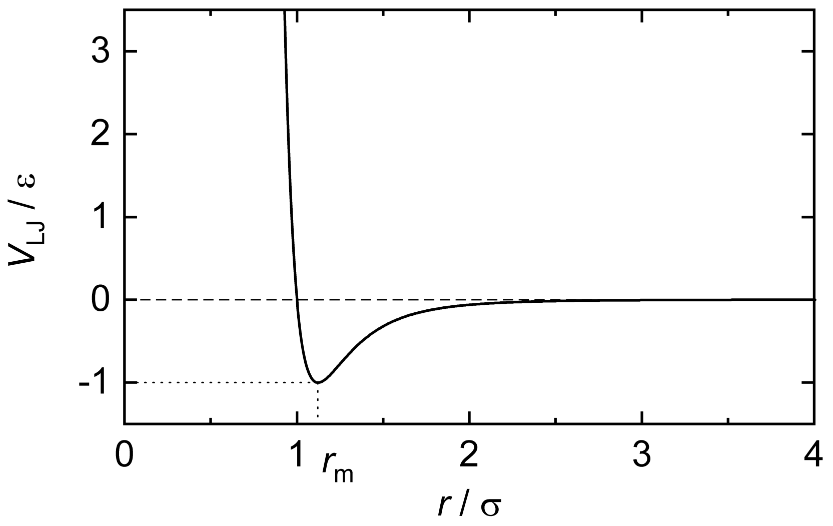


Figure 2.1.: Graph of the Lennard-Jones potential. The energy between two particles V depends on the distance r between each other. [7]

2.3. Newton's Third Law

Another important method that can be used for optimizing the force calculation is Newton's third law of motion. It states that for every force, a force with an equal magnitude but an opposing direction exists. This means that for a force F_{ij}, that is exerted by particle j on particle i, the force $F_{ji} = -F_{ij}$ is exerted on particle j. With this in mind, the amount of force calculation can be halved. [3] In the following, the usage of this law is referred to as `Newton3` optimization.

2.4. AutoPas

There exist multiple algorithms and approaches for computing the interactions between particles efficiently, e.g. Verlet Lists or Linked Cells. However, none of those algorithms performs best under every circumstance and the optimal configuration might even change during the simulation time. The open-source library AutoPas tackles this problem by providing a wide range of algorithms and optimizations. It also employs dynamic auto-tuning for selecting the optimal algorithmic configuration during runtime. [3]

In the following, the features which are most important for this thesis are presented.

2.4.1. Containers

The main part of every molecular dynamics simulation is identifying the particles for which the calculation of the short-range potential is relevant. The goal is hereby to find the particles which are within r_{cut} and should therefore be considered in the force calculation for a certain particle. Depending on the algorithm used, the data structure in which the particles are stored is chosen. In AutoPas, the handling of the underlying data structure for the respective algorithmic approach is done by so-called containers. [3]

In the following, some of the underlying algorithms for the AutoPas containers, which are relevant to this paper, are explained.

Direct Sum

The naive approach is to calculate the distances to all particles, which is visualized by the blue area in Figure 2.2a. All pairwise distances are calculated with particles beyond the cutoff distance being omitted from the force calculation. The advantage of this procedure lies in simplicity and no algorithmic, computational, or memory overheads. As all distances need to be calculated, this procedure scales with $\mathcal{O}(N^2)$ with N as the number of particles. Therefore, this routine is unsuitable for a large number of particles. [3]

Linked Cells

The idea of the Linked Cells algorithm is to split up the simulation domain into a Cartesian grid of cells with a mesh size $\geq r_{cut}$. The particles are stored in these cells and exchanged between the cells based on the particles' current positions. When calculating the interactions for one particle, meaning the distance and, if necessary, the force, only the particles in the own and neighboring cells are taken into account. This technique decreases the complexity from $\mathcal{O}(N^2)$ to $\mathcal{O}(N)$ in the case of homogeneous particle distributions. [3]

AutoPas uses a vector of particles for each cell as a data structure, which has the benefit that particles with small distances between each other also end up close in memory. This facilitates efficient iterations of sub-regions of the domain as well as efficient vectorization and parallelization. [3]

A comparison of the volume of the resulting search space of this approach and the cutoff volume derived from r_{cut} shows that in a 3D scenario, only 15.5% of the distance calculations are necessary as calculated by Equation 2.6. [8]

$$\frac{Cutoff\_Volume}{Search\_Space} = \frac{\frac{4}{3}\pi r_{cut}^3}{(3 \times r_{cut})^3} \approx 0.155 \tag{2.6}$$

The ratio between the red circle visualizing the cutoff volume and the blue area representing the total search space in Figure 2.2b highlights this disadvantage for a 2D scenario.

Verlet Lists

To tackle the problem of irrelevant calculations, the Verlet Lists container is introduced. Loup Verlet describes his idea as a "bookkeeping device" computing the distances between all of the particles only every Nth timestep. All particles within a certain distance of a particle are stored in a list. For the next $N-1$ time steps, when computing the interactions of a particle, only the particles of the particle's neighbor lists are taken into account. [9]

To determine the distance within which a particle is stored in the neighbor lists, a skin factor s is introduced by which r_{cut} is multiplied. The search space results in the red area representing r_{cut} and the blue area visualizing the Verlet skin in Figure 2.2c. With this method, the likelihood that a calculated distance is within the cutoff radius for a 3D domain is given by Equation 2.7 and therefore depends on s. [8]

$$\frac{Cutoff\_volume}{Search\_Space} = \frac{\frac{4}{3}\pi r_{cut}^3}{\frac{4}{3}\pi(r_{cut} \times s)^3} = \frac{1}{s^3} \tag{2.7}$$

Based on this, it can be concluded that with a skin factor $s \leq 1.8$, the Verlet Lists approach needs fewer unnecessary distance calculations than the Linked Cells container. Figure 2.2 visualizes this relation in a 2D scenario.

As the particles might move during the simulation time, these neighborship relations can get outdated. If the distance between particles decreases, it has to be prevented that a particle moves inside of the cutoff radius of another particle without being in its neighbor list. On the other hand, if the particles' distances grow, the list might become larger as necessary leading to a degenerated Direct Sum procedure. Consequently, there is the necessity to update these neighbor lists every N timesteps which will be referred to as "neighbor list rebuild" for this thesis. [9]

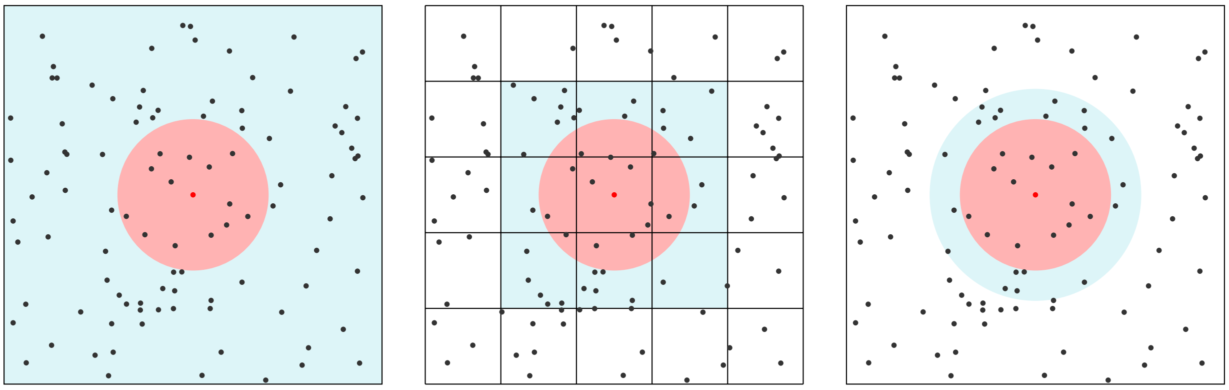
For the efficient rebuilding of the neighboring lists, AutoPas stores the particles in an instance of the Linked Cells container. When computing the distances of the particles every N timesteps, only particles in their own and neighboring cells are taken into consideration. In Figure 2.2b, this area of interest is highlighted in blue. [3]

With this improvement, the complexity for searching nearby particles decreases from $\mathcal{O}(N^2)$ to $\mathcal{O}(N)$. This is due to the fact that the number of interactions for each particle does not depend on the total amount of particles but on the number of particles in neighboring cells which is assumed to be constant. [10]

2.4.2. Traversals

AutoPas supports different algorithmic approaches to iterate over all pairs of particles in a parallel manner to apply a given operation to them. As the focus of this work is not improving the force calculation but rebuilding the Verlet neighbor lists, only some traversal types are interesting in this scope. As mentioned in 2.4.1, all Verlet-list-based containers store their particles in an instance of the linked cells container and use this instance for rebuilding the neighbor lists. Therefore, only traversals for the Linked Cells container are explained further in this subsection.

In a shared memory parallelization for the traversal of the Linked Cells container, the cells are in most cases distributed over threads. To avoid race conditions in the case of using the


(a) Direct Sum, own illustration inspired by [11]

(b) Linked Cells, own illustration inspired by [12]

(c) Verlet Lists, own illustration inspired by [12]

Figure 2.2.: Comparison of search space in blue and cutoff volume in red of selected AutoPas containers for a 2D domain. The Direct Sum algorithm applies the distance checks to the whole simulation domain and therefore requires the highest search space. The Linked Cells procedure can reduce the search space by only considering the neighboring cells. The Verlet List algorithm can further decrease the search space by only using an additional skin around the cutoff radius.

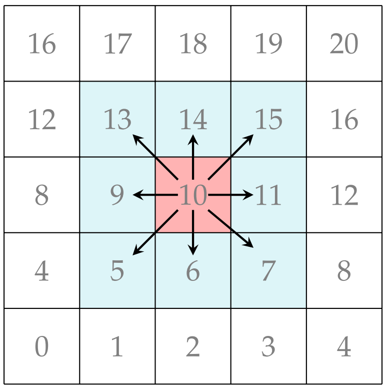
`Newton3` optimization, AutoPas incorporates different synchronization techniques via domain coloring or locking mechanisms. [3]

The idea of the coloring approach is that the cells are painted in different colors so that cell interactions of the same color are not overlapping. All cells within each color can then be processed in parallel. Every cell-based domain traversal is divided into two parts: the traversal of the cells and the operations carried out on each cell. The operations on a cell will be called the base step in the following. Within this base step, all pairwise interactions between particles within a cell and the particles in the neighbor cells are computed. Up to now, AutoPas supports three different base steps which can be seen in Figure 2.3 and are described further in the following. [3]

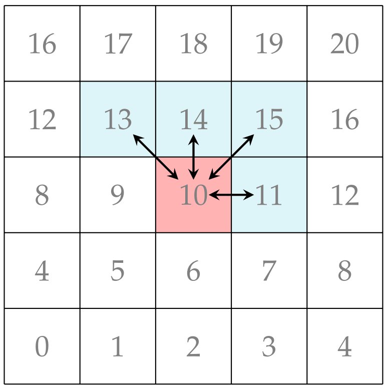
The C01-base step computes every neighborship interaction as visualized in Figure 2.3a and does thereby not use the `Newton3` optimization. Therefore, as no potential concurrent access on the same memory location takes place, no more than one color is required. This means that the traversal of the cells can be distributed over all threads directly. [3]

The C18-base step uses `Newton3` and only computes the cell-wise interactions between forward neighbors with a higher index which is depicted by Figure 2.3b. When using domain coloring to avoid race conditions, this traversal type needs six colors in 2D and eighteen in 3D. [3]

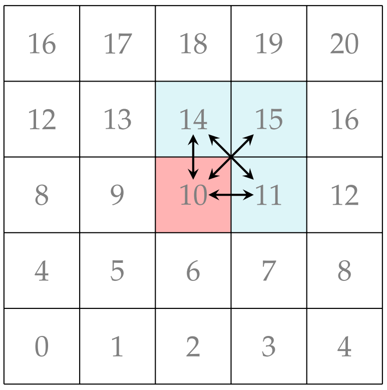
The C08-base step improves the idea further and replaces the computation of the forward diagonal interaction with the forward computation of the next cell as demonstrated in Figure 2.3c. As the number of cells taken into account in this base is decreased further, this traversal only needs four colors in 2D and eight colors in 3D. [3]



(a) C01 Base step, own illustration inspired by [3]



(b) C18 Base step, own illustration inspired by [3]



(c) C08 Base step, own illustration inspired by [3]

Figure 2.3.: The red cell is the currently processed cell by the domain traversal. The blue area and the black arrows indicate the pairwise interactions between particles in the base cell and their neighbor cells [3]

2.4.3. Auto Tuning

AutoPas uses an exhaustive search strategy for auto-tuning. It is assumed, that consecutive iterations of the simulation are comparable concerning the pairwise force calculations. That is because the distribution of particles does not change drastically within a few time steps. During the tuning phase, different configurations are used in successive iterations and the execution times are measured. This approach has the advantage that no execution is wasted and the tuning overhead is minimized. The best-identified configuration concerning a certain evidence, e.g. average, mean, or absolute minimum is chosen for a configured number of iterations. Afterwards, the tuning process starts again to find the optimum again which might have changed since the last tuning run. [3]

3. Related Work

This work is motivated by a previous thesis related to tuning Verlet Lists, where Daniel Asch analyzed the effect of different rebuild frequencies for Verlet Lists. His results show that the optimal configuration for the Verlet Skin and, therefore, the frequency differs for diverse scenarios and might even change over the simulation time. AutoPas was extended with the possibility to auto-tune the frequency during runtime. Finally, the additional tuning phase was very time-consuming and an educated guess might outperform AutoPas concerning the Best-Case runtime. [13]

However, this educated guess might not be optimal, as it was only sufficient to beat the auto-tuning reference implementation. There already exist publications about tracking the movement of the particles since the last rebuild and triggering the next list update when the particle displacement exceeds a certain limit. This threshold is often chosen related to skin size. Pedro Gonnet [14] describes the use of this method in a paper that explored a combination of Verlet Lists and Linked Cells for improved memory locality. However, as the primary goal of his work was comparing memory locality, he did not work out the new rebuild criterion improvement further. Additionally, he does not provide measurements for comparing this approach with a static rebuild frequency.

Other particle simulators also exist, implementing the algorithms discussed by Chapter 2. Most of them support only one algorithm and are optimized for specific use cases, distinguishing them from AutoPas. However, they are a good source for more optimizations in their respective discipline and some ideas on how to improve AutoPas have their origin in those simulators. [3]

A well-known particle simulator is the software GROMACS[1]. It is, among other aspects, famous for distributing the computations among the CPU and GPU. For identifying the interaction partners, it employs a Verlet-List-based cluster pair algorithm which is claimed to suit better for SIMD registers. Its updates of the pair lists are based on a fixed interval. This is because the chosen cluster-pair algorithm allows picking the frequency freely and independently from problem-specific properties, as this only influences the size of an internal buffer and not the accuracy of the simulation. [15]

Another frequently used code package for molecular dynamics problems is the software LAMMPS[2]. It uses the idea of Verlet Lists to identify nearby particles and MPI for distributed memory parallelization. The lists are optimized for short-range potentials and scenarios with a moderate particle density. The code triggers the neighbor list update when any atom has moved half the skin distance since the last rebuild. As LAMMPS already uses the idea of this dynamic rebuild criterion, it serves as the basic idea for the first improvement of this thesis. [16]

[1]`https://www.gromacs.org/index.html`
[2]`https://www.lammps.org`

4. Implementation

This thesis aims to evaluate different approaches for implementing Verlet lists. In this chapter, the considered improvements are explained in detail by looking at the algorithmic idea and the concrete implementation and integration into the AutoPas library.

4.1. Initial Problem

The main goal of the existing improvements for Verlet neighbor lists in AutoPas focuses on the ease of vectorization, memory locality, and cache efficiency when traversing the particle pairs in the force calculation. This led to interesting ideas like the Pairwise Verlet List algorithm, which stores the neighbor lists in pairs of cells. When iterating over all pairs of particles, this results in fewer cache misses as the access to neighbor particles is coupled with information about their storage location. [14] However, maintaining these neighbor lists in the rebuilds may be more time-consuming as the data structure is more complicated. To address this issue, the motivation of this work was to reduce the time spent on the rebuilds.

So far, the neighbor lists in every Verlet Lists-based container are periodically rebuilt every Nth iteration, as described in 2.4.1. As a result, $\frac{1}{N}$ is referred to as rebuild frequency (RF), with N being determined by the user of AutoPas by following some rules. For the rest of this thesis, the RF is considered as high whenever N takes small values. It needs to be guaranteed that no particle traverses the Verlet skin and moves inside the cutoff radius before the lists are rebuilt [9]. Therefore, N has to be small enough that this condition holds even for high particle movements. On the other hand, smaller values for N guarantee that this constraint is met but lead to more rebuilds than necessary which increases the time spent in the neighbor list rebuilds.

Guessing which frequency is optimal might be challenging and, in most cases, this leads to under-approximations as the provided value has to be a lower bound. This problem manifests in rebuilding the neighbor lists more often, as the particle movement might be lower than expected or even change over the simulation time.

4.2. Dynamic Rebuilding

The idea to tackle the issue of too many rebuilds is to update the neighbor lists not depending on a fixed rebuild frequency. Instead, a new rebuild criterion is defined that is expected to be more flexible. This approach is called dynamic rebuilding and the underlying algorithmic idea and its concrete integration in the AutoPas library is presented in more detail in the following.

4.2.1. Algorithmic Idea

As the original Verlet rebuild frequency depends on a guess about the maximum movement [9], the new condition to determine the necessity of list rebuilds is based on a particle's movement since the last rebuild. In every iteration, the displacement of the particle from its rebuild position is computed and compared with a certain threshold. This threshold is chosen as half of the Verlet skin size, as suggested by [16]. The underlying idea is that when two particles move exactly toward each other, the cumulative distance difference is higher than the skin size in such cases. This movement might be enough that a particle is within the cutoff radius of another particle without being tracked by the neighbor list and, therefore, a rebuild is necessary.

If this condition is met for at least one particle, the rebuild procedure is initiated. A description of this procedure in pseudocode is provided by Algorithm 1.

Algorithm 1 Dynamic rebuild criteria - pseudocode

```
listsInvalid ← False
for particle ∈ particles do
    if distance(particle.currentPosition, particle.rebuildPosition) > skin/2 then
        listsInvalid ← True
    end if
end for
return not listsInvalid
```

Figure 4.1 provides a visual example of the molecules' movement behavior. In each iteration, a particle travels a certain distance, yielding a new position within the simulation domain. This position is then used to calculate the difference between the rebuild position, here the position at iteration 0. The particles' displacements after two iterations are visualized by the red dotted lines with their respective labels.

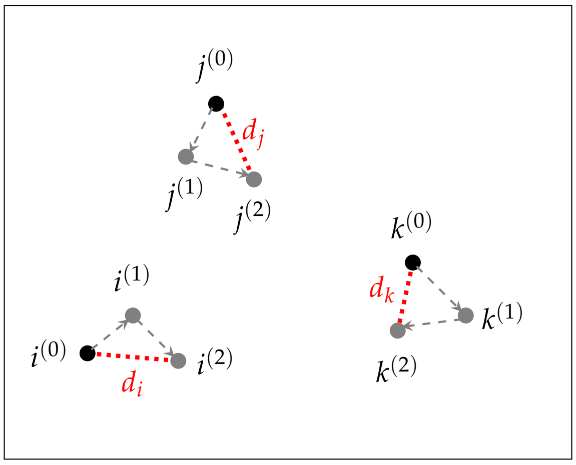


Figure 4.1.: Particle movement around the domain of three particles i, j, and k. The gray arrows visualize the movement of a particle since the last iteration, $i^{(t)}$ describes the position of particle i at iteration t. The displacement of a particle from the rebuild position is highlighted by the dotted red line and the red label $d_{particle}$.

For the actual rebuilding procedure, meaning the deletion and refilling of the neighbor lists, the existing code can be reused as the corresponding rebuild routine itself does not change. After the completed update, the rebuild position of every particle is updated to the current position.

The advantage of this procedure lies in the ability to respond to a rapidly changing simulation environment. As it keeps track of the particle displacement, the dynamic algorithm can react to a changing particle velocity over time quickly. Suppose the particles start with high movement and the velocity decreases strongly over time. Then, at the beginning, the criterion is met more often which leads to more frequent list rebuilds. In the end, as the particle speed is lower, the list updates become less frequent as the condition is fulfilled less often. When using a fixed RF, it would have to be configured with the high movement at the beginning in mind which leads to more neighbor list updates than necessary at the end of the simulation.

4.2.2. Implementation

In the following section, it is examined how the described algorithmic ideas are integrated into the AutoPas library.

Changing the ParticleContainer interface

The described procedure for determining the necessity of list rebuilds requires knowledge of the particles' current positions and their positions at the last rebuild time. Therefore, a suitable place to integrate the algorithm is the particle container which serves as the storage structure for the particles.

The main goal of the adjustments to the program frame is to be as least invasive as possible. To achieve this, the `ParticleContainer` interface is extended by the `neighborListsAreValid()` function which will handle the relevant comparisons regarding the particle displacement as described in Algorithm 1. It is declared as a pure virtual function to force the concrete container subclasses to override it. Every container which does not support a dynamic rebuild routine yet, such as the Linked Cells container, returns `TRUE`.

Subclasses for the existing container implementations

All Verlet lists-based containers have to be able to implement a dynamic rebuilding routine and, therefore, in some ways need to override the `neighborListsAreValid()` method. The Verlet List algorithms relying on a fixed RF should still be available and will be referred to as `static` containers for the rest of this work. Therefore, the best approach to achieve as much code reuse is to declare a new subclass for every relevant Verlet lists-based container. These subclasses only override the functionality which changes and declare their necessary new elements.

In the case of the dynamic rebuilding extension, the concrete methods which need to be overridden are the `neighborListsAreValid()` and the `rebuildNeighborLists()` functions. The first one implements the routine described in Algorithm 1 to determine the necessity of list updates. The latter initially invokes the corresponding base class method to delete the existing lists and fill them again based on the new particle positions. Afterward, it calls a new function named `generateRebuildPositionMap()` to update each particle's rebuild position.

The main addition of every dynamic container is the buffer for storing the pairs of particle pointers and rebuilding position and the corresponding method to update it. The most suitable data structure for this purpose is a C++ standard library vector. The reason for this is that the elements are stored contiguously in memory which is beneficial for the parallelization of Algorithm 1. Additionally, changing the buffer size when particles leave the domain is easier than in an array. The class diagram in Figure 4.2 visualizes the changes with the added classes, methods, and attributes in orange.

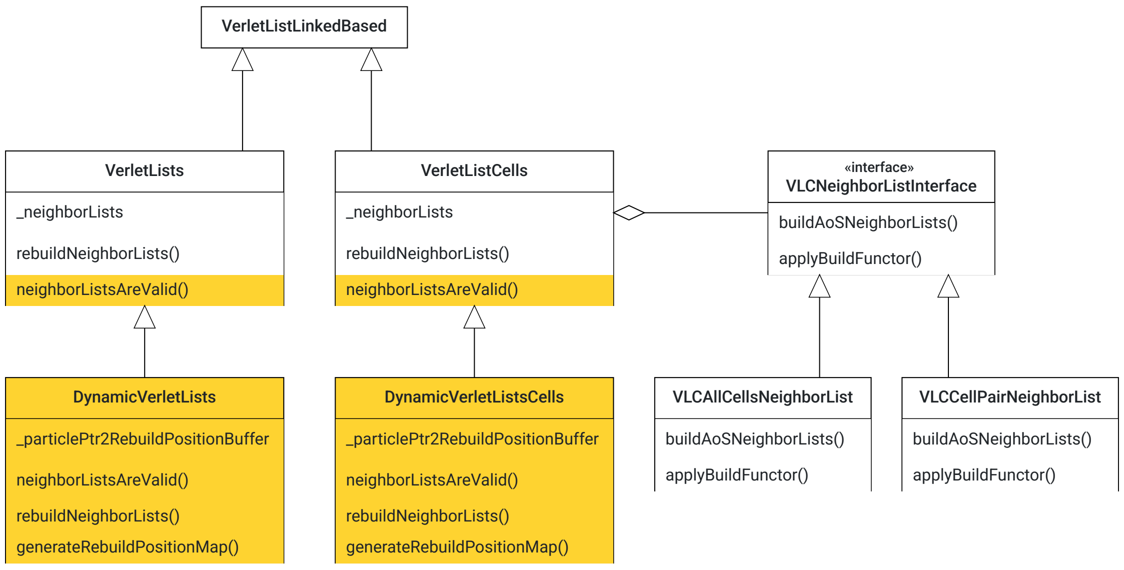


Figure 4.2.: Class diagram visualizing the hierarchy of the new dynamic subclasses. The added classes, attributes, and methods are highlighted in orange.

Integration into AutoPas control flow

Finally, the new rebuild criterion is integrated into the AutoPas control flow. So far, the neighbor lists of every Verlet Lists-based container are periodically rebuilt based on the configured RF. Currently, in every iteration the `neighborListsAreValid()` method of the `LogicHandler` class determines whether a rebuild of the neighbor lists is necessary. This function checks if the number of iterations since the last rebuild exceeds the configured rebuild frequency. This condition is extended by a call to the added `neighborListsAreValid()` method of the `ParticleContainerInterface` class. This resulting function of the `LogicHandler` class is described more formally by Algorithm 2. If the underlying container does implement a

Algorithm 2 neighborListsAreValid - LogicHandler

```
listsValid $\leftarrow$ true
**if** stepsSinceLastRebuild $\geq$ rebuildFrequency or not container.neighborListsAreValid() **then**
    listsValid $\leftarrow$ false
**end if**
**return** listsValid
```

dynamic routine, it will be delegated and performs the relevant comparisons. Additionally, the rebuild criterion with using the RF is verified which is still necessary for the `static` containers without a dynamic rebuild condition.

4.3. Partial Rebuilding

The improvement described in the previous section focuses on dynamically triggering the rebuilds whenever necessary, hence reducing the number of rebuild iterations. However, when one particle exceeds the allowed distance, every neighbor list in the whole simulation domain is updated. This leads to much more lists being rebuilt than necessary, potentially causing a performance disadvantage in scenarios with a few very high-moving particles and a lot of slow particles. Therefore, the second improvement aims at only rebuilding the particle's neighbor lists that need to be rebuilt.

4.3.1. Algorithmic Idea

Identifying the necessity for rebuilding stays the same as described in the previous section with the condition about tracking a particle's displacement since the last rebuild. To reduce the number of neighbor lists being updated, the area of interest i.e., the part of the domain with the high-moving particles, has to be identified.

Yao et al. suggest using the spatial decomposition for the simulation domain provided by the linked cells algorithm to solve this problem. Their idea is to identify the two highest-moving particles within each cell in every timestep. Both displacements are summed up and if this value exceeds the skin size, their residing cell is marked as dirty. This "dirty flag" indicates that the neighbor lists of molecules in the corresponding cell have to be rebuilt. Therefore, during the update phase of the neighbor lists, only the dirty cells and their neighbors are considered. [10]

We integrate the dirty cell identification routine into the procedure to determine the cumulative displacement from the dynamic rebuilding algorithm. Whenever a particle's movement exceeds half the skin size, its residing cell is marked as dirty, as described more formally in Algorithm 3. This criterion differs from the one suggested by Yao et. al. because their idea needs knowledge about neighboring cells during the displacement detection routine. This is because when two particles in adjacent cells move towards each other, the lists also need to be rebuilt when their cumulative distance exceeds the skin size. The current displacement detection routine does not have access to the neighborship information of cells because and for this reason the idea of Yao et. al. is adjusted.

Figure 4.3 illustrates an application of the procedure with the high-moving particles colored in blue. Their residing cells are highlighted in red and the neighbor cells in light red color. The rebuild procedure has to be carried out only for the particles in the red cells, including their references to the neighbor cells' particles. These rebuild interactions are visualized by the black arrows between the dirty cells and their partners. As can be seen in this scenario, the area in which the update has to be carried out is much smaller than the whole simulation domain.

Algorithm 3 Identification of the dirty cells - pseudocode

```
listsInvalid ← False
for particle ∈ particles do
    if distance(particle.currentPosition, particle.rebuildPosition) > skin/2 then
        listsInvalid ← True
        residingCell ← getResidingCell(particle)
        residingCell.setDirty()
    end if
end for
return not listsInvalid
```

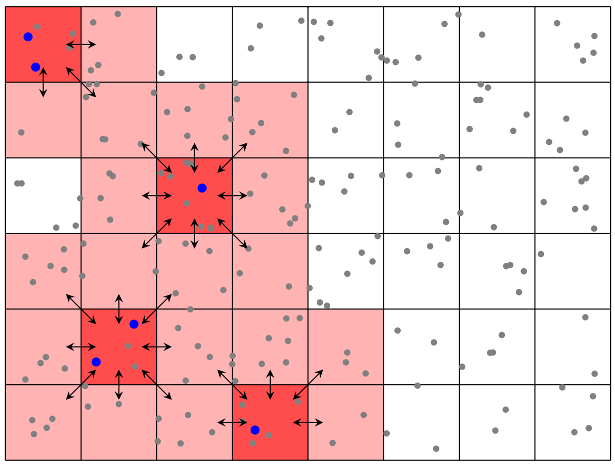


Figure 4.3.: Visualization of the high-moving particles in blue which reside in the red dirty cells. The necessary rebuild interactions between the dirty cells and their neighbors in light red color are highlighted by the black arrows.

4.3.2. Implementation

To integrate the idea of partial rebuilding into AutoPas, some prerequisites have to be determined. As the procedure requires knowledge of the cells of neighbor particles, partial rebuilding does not apply to every Verlet List-based container without significant changes.

In the case of neighboring cells, it is not necessary to rebuild the whole neighbor list but only for the interaction with the dirty cells. Therefore, the partial rebuilding procedure needs information about the cells of a particle in the neighbor list. To evaluate which container might be valid, the respective existing implementations of the Verlet List-based algorithms are described briefly in the following.

Integration into the existing container algorithms

In the basic `VerletLists` (VL) container, the neighbor lists are realized as a global map from particle pointer to a vector of pointers, representing the neighboring particles [3]. As the neighbor lists themselves do not include any information about the cells in which the individual particles reside, this container is not suitable for partial rebuilding.

The `VerletListCells` (VLC) AutoPas container does store the neighbor lists in a cell-wise way. The lists are associated with the cell where the base particle resides. However, the lists do not store information about the cells of the neighbor particles as well [3]. Because of this missing distinction to which cell a neighbor particle belongs, partial rebuilding is not applicable without significant changes to the existing container.

A suitable container to start with the integration of the partial rebuilding idea is the `PairwiseVerletLists` (PVL) container. The essence of the underlying algorithm is that it creates neighbor lists describing the interaction only between a given pair of cells. Particle $p1_E$ in a given cell E stores all its neighbor particles that reside in neighbor cell F in its own vector and all particles from neighbor cell C end up in a different vector. [17] Figure 4.4 provides a visual example of this behavior. A more detailed explanation of the underlying data structure for the AutoPas implementation of the Pairwise Verlet List algorithm can be found in [17].

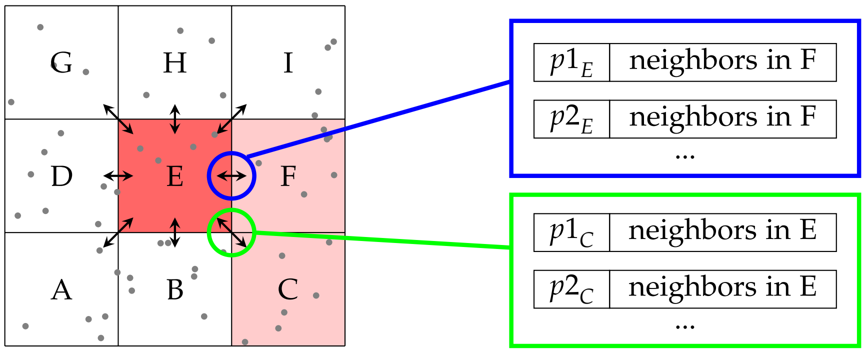


Figure 4.4.: Sketch of the cell-wise data structure. On the left-hand side a 2D linked cells structure with base cell E in red and its neighborship relations visualized by the black arrows. On the right-hand side, the cell-wise neighbor lists for cell pair (E, F) and (C, E). Own illustration, inspired by [17].

Extension of the Verlet List Cells structure

To understand the integration of the partial rebuilding idea, the following subsection provides an overview of the existing class structure and hierarchy of the `PairwiseVerletLists` container.

The structure for every cells-wise neighbor list container is given by the `VerletListCells` class. It resembles common functionality and abstracts the concrete neighbor list implementation with a template parameter. This allows for the easy addition of new neighbor list types. The neighbor list classes are responsible for storing the actual neighbor lists and functionality related to their construction and filling, which differs between the underlying data structures. [17]

The rebuild procedure for the VLC-based containers can be divided into two phases. First, the neighbor lists are cleared, initialized again, and their memory is reserved, making the actual filling more efficient. Second, with a call to `applyBuildFunctor()`, the neighbor lists are populated with actual particle pointers by traversing all cells and their respective particle interactions. As an argument, this method takes a functor that defines the procedure for handling a pair of particles and how to store them in case they are within the interaction distance. [17]

These parts have to be changed for the partial rebuilding procedure, as only specific neighbor lists should be cleared and updated. In addition, the procedure for identifying the dirty cells needs to be integrated into the process flow as well. To make these changes as least invasive as possible and reuse as much existing functionality from the existing containers, two subclasses are implemented.

The first subclass, `PartialVerletListCells` derives from the `DynamicVerletListCells` class. This base class is chosen because the partial rebuilding routine can reuse some functionality from there. For example, the `rebuildNeighborLists()` method can stay the same as it only controls the procedure of actual rebuilding and the generation of the rebuild position map. It serves as a control method to delegate the functionality to smaller methods which can be overridden to allow a partial update of the rebuild position map.

The second subclass is the `VLCPartialCellPairNeighborList` which derives from the `VLCCellPairNeighborList` class. This is because only the methods for preparing the neighbor lists have to be changed. Attributes for storing the actual lists or functions for extracting information about e.g., the number of stored particles, should remain unchanged.

The class diagram in Figure 4.5 highlights the new classes and overridden methods in orange. Furthermore, it provides a visual overview of the described hierarchy.

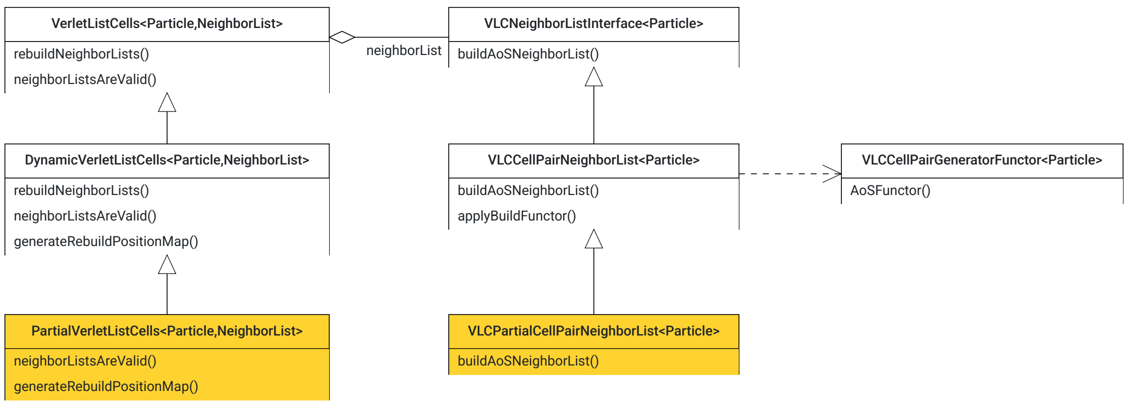

VerletListCells<Particle,NeighborList>
rebuildNeighborLists()
neighborListsAreValid()
neighborList
VLCNeighborListInterface<Particle>
buildAoSNeighborList()
DynamicVerletListCells<Particle,NeighborList>
rebuildNeighborLists()
neighborListsAreValid()
generateRebuildPositionMap()
VLCCellPairNeighborList<Particle>
buildAoSNeighborList()
applyBuildFunctor()
VLCCellPairGeneratorFunctor<Particle>
AoSFunctor()
PartialVerletListCells<Particle,NeighborList>
neighborListsAreValid()
generateRebuildPositionMap()
VLCPartialCellPairNeighborList<Particle>
buildAoSNeighborList()

Figure 4.5.: Class diagram visualizing the hierarchy of the new partial subclasses. The added classes, attributes, and methods are marked with orange color.

Rebuilding procedure

The advantage of the existing cell-wise neighbor list structure is that there is the possibility to only focus on certain cell-to-cell interactions. For a given dirty base cell A, every neighbor list should be rebuilt. But for the neighbor cells, only the lists pointing to this base cell should be updated. The relations to other cells should not be rebuilt as they are still valid.

Therefore, the preparation of the neighbor lists in the overridden `buildAoSNeighborLists()` method must not be carried out on every cell. When iterating over all cells, there are two cases where an action is needed. First, if the current cell is dirty, every neighbor list has to be deleted, initialized, and reserved again. Or second, if the current cell has dirty neighbor cells, only the relation between the base cell and the dirty cells has to be prepared. Algorithm 4 explains this procedure further in pseudocode.

Afterward, in the actual filling of the neighbor lists, only the dirty cells and their neighbors need to be considered. The iteration over all cell pairs is therefore guarded by a condition that checks the dirty flags of the cells. A cell pair is only processed further if at least one of both

Algorithm 4 Preparing the neighbor lists for dirty cells - pseudocode

```
for $cell \in cells$ do
    for $neighbor \in cell.neighbors$ do
        if $cell.isDirty()$ or $neighbor.isDirty()$ then
            neighborLists[cell][neighbor].clear()
            for $particle \in cell.particles$ do
                neighborLists[cell][neighbor].add(particle,vector<ParticleType>())
            end for
        end if
    end for
end for
```

cells is dirty. Processing in this scope means calculating the distances between the particles of both cells and, if necessary, adding a particle to the neighbor list. Algorithm 5 provides a pseudo-code realization of this procedure.

Algorithm 5 Processing of the cells during rebuild - pseudocode

```
for $cell \in cells$ do
    for $neighbor \in cell.neighbours$ do
        if $cell.isDirty()$ or $neighbor.isDirty()$ then
            processCellPair(cell, neighbor)
        end if
    end for
end for
```

When updating the particles' rebuild positions after the completed neighbor list update, only the dirty cells' particles are taken into account. This routine is described in pseudo-code by Algorithm 6. This is because all other particles' neighbor lists were updated only partially or not at all. During this routine, a cell's dirty flag has to be deactivated again to avoid unnecessary cell updates in the next list rebuild.

Algorithm 6 Partial update of the rebuild positions - pseudocode

```
for $cell \in cells$ do
    if cell.isDirty() then
        for $particle \in cell.particles$ do
            particle.rebuildPosition $\leftarrow$ particle.currentPosition
        end for
        cell.setNotDirty()
    end if
end for
```

Particle migration between cells

As the `PairwiseVerletLists` Container is built on top of the Linked Cells structure, there are some dependencies that have to be considered in the partial rebuilding procedure as well.

As already mentioned in subsubsection 2.4.1, storing the particles in an instance of the `LinkedCells` container decreases the complexity of the neighbor list rebuilds. Because the particles might move during the simulation, the particles sometimes have to be transferred to another cell. Otherwise, the adjacency relations and locality information for the rebuilds would not be valid. Exchange in this context means deleting a particle from the original cell's particle vector and adding it to the target cell's list. When performing this migration, the neighbor lists stay unchanged at first, meaning a migrated particle's list is not deleted by the `LinkedCells` routine. The reason is that the algorithm can be used without a Verlet-List-based container on top.

Therefore, this exchange was only triggered on every rebuild of the Verlet List-based container, meaning the neighbor list update. In other words, the RF for the neighbor list updates controls the particle migration as well because the exchange also invalidates some neighbor lists. The existing routine for the update of the Verlet-List-based containers is visualized in Figure 4.6. In orange, the parts are highlighted that need to be changed during the partial rebuilding implementation.

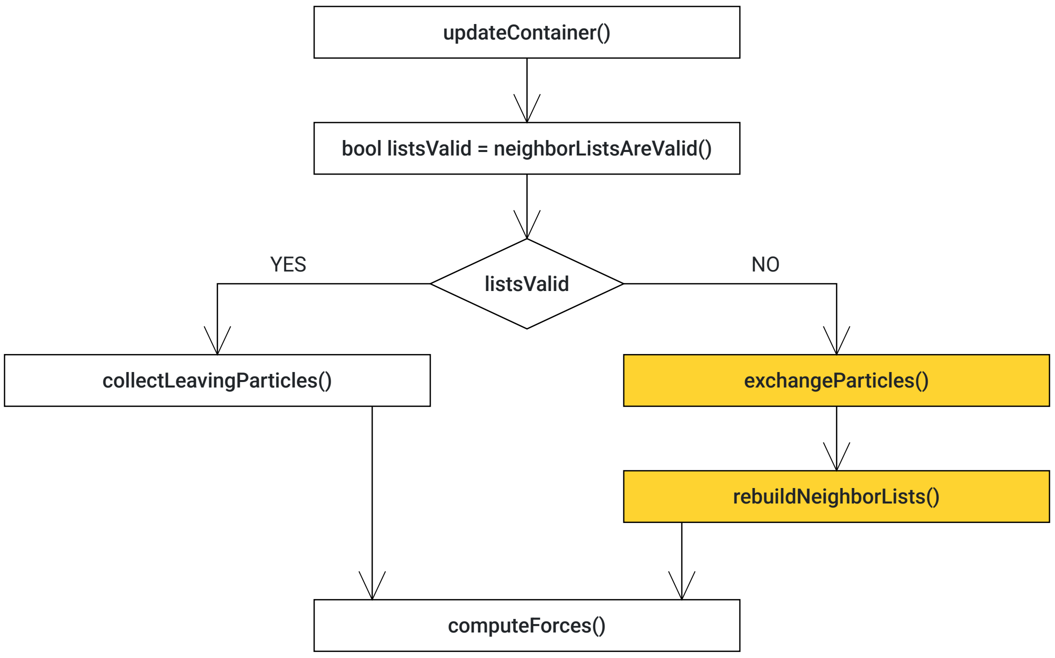


Figure 4.6.: Flow chart of the routine for the update of the neighbor lists for the Verlet-List-based containers. Parts that have to be changed for the partial rebuilding algorithm are highlighted in orange.

The invalidation expresses in pointers which still reference the old memory location of the transferred particle in the vector of the original cell. If the particle exchange is not limited, lists that are not updated contain invalid pointers as the particles were deleted from the original cells vector. The solution is to only transfer the particles that migrate into and out of dirty cells. When a particle moves into another cell, it is referred to as an "inflow cell" while cells with leaving particles are called "outflow cells". The inflow cells also need to be dirty because the exchanged particles do not have any neighbor lists in the new cell yet. As their original cells' neighbor lists are cleared, the interaction between the new residing cell and the base cell is also updated, because of the dirty neighborship relation.

For outflow cells, a complete rebuild is not necessary. Their particles don't have too high movement as the cells would be dirty otherwise. The same holds for the neighbors of the outflow cells. Therefore, their lists would only be invalidated if a particle is deleted. Marking the leaving particles as dummies and not deleting them immediately allows to not consider them in the force calculation and leave the corresponding neighbor lists as they are. The necessity for the complete rebuilding of this cell is eliminated as it no longer contains invalid pointers. The neighborship relations between the particles' original cells and their targets are rebuilt because the target cells need to be dirty for the exchange to take place. With this help, the outflow cells don't need to be marked as completely dirty, and thereby some unnecessary overhead can be avoided.

In Figure 4.7, this improvement is depicted by the red crosses which mark the cell-wise interactions as not necessary for the orange outflow cell and its neighbors. The rebuild interactions between dirty cells in yellow and inflow cells in green still have to take place completely and are visualized with the gray arrows. The gray cells represent the neighbor cells which are considered during the update at least once.

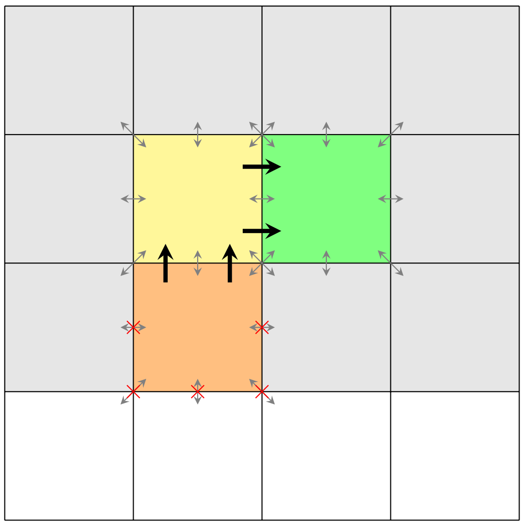


Figure 4.7.: Allowed particle transfer from orange outflow cells into yellow dirty cells and green inflow cells from dirty cells visualized by black arrows. The gray arrows represent the necessary rebuild interactions between cell pairs. The gray boxes represent the neighbors of the dirty cells which have to be considered at least once during rebuilding. As it is not necessary to rebuild neighbors of outflow cells the corresponding arrow is marked with a red cross.

To avoid the dirty state spreading over larger parts of the domain than intended, the implementation distinguishes between movement-dirty exchanging-dirty cells. This has the reason that exchanging dirty cells should not allow particle migration to take place. Algorithm 7 provides a more formal description of the exchange procedure.

Algorithm 7 Exchanging particles between cells - pseudocode

```
for $cell \in cells$ do
    for $particle \in cell.particles$ do
        if $getTargetCell(particle) \neq cell$ then
            if $cell.getDirty()$ OR $getTargetCell(particle).isDirty()$ then
                exchangeParticle(particle)
                setExchangeDirty(getTargetCell(particle))
            end if
        end if
    end for
end for
```

4.4. One-colored rebuild traversal

As discussed earlier in subsection 2.4.2, there exist several approaches to iterate over pairs of particles in the `LinkedCells` container in a parallel manner. In the following, a new traversal is discussed which aims for better parallelization of the neighbor list rebuilding.

4.4.1. Algorithmic Idea

When filling the neighbor lists, there is the necessity of traversing particle pairs. The applied operation is in this case adding a particle to another particle's neighbor list if their distance is within the interaction range. This is defined by the so-called `GeneratorFuntor`. Every Verlet-List-based container implements such a functor to define how the neighboring particles are stored in the data structures. For traversing the simulation domain during rebuilding, a chosen parallel traversal of the `LinkedCells` container is used. Currently, every Verlet List-based container uses a c08-based traversal for updating the lists. This results in a coloring scheme with eight colors and a distance of two cells in each direction [3]. In Figure 4.8, the cells which are processed in parallel by one color slice and their respective base steps are visualized in 2D.

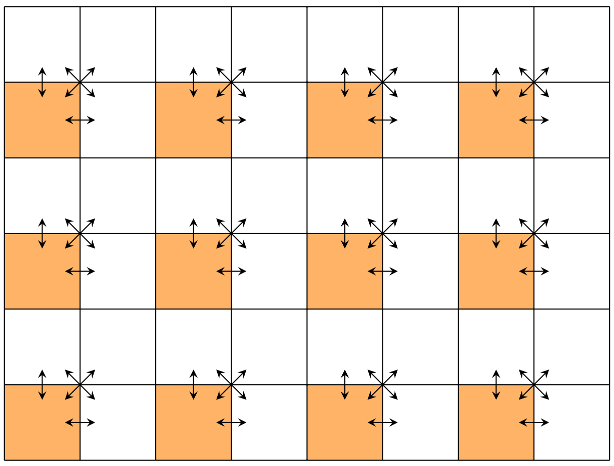


Figure 4.8.: Cells that are processed in parallel are highlighted in orange color. Their base step interactions are visualized by the black two-sided arrows

In every generator functor for filling the neighbor list of the Verlet-List-based containers, adding a particle to a particle's neighbor list is only performed once per particle pair. This does not depend on the usage of the `Newton3` optimization as this is only relevant for the force calculation. The only difference among the various types of Verlet-List-based containers in that regard is the data structure of the neighbor lists. For two particles *i* and *j*, either *i* is stored in the list of *j* or *j* is stored in the neighbor list of *i*. The procedure for two given cells A and B is described more formally by Algorithm 8. The neighbor lists are only accessed at the position of particle *i* in the respective neighbor list structure but the neighbor list of particle *j* remains untouched.

Algorithm 8 Pseudocode for the Verlet-List-based Generator functions, inspired by [17]

```
for $particle\ i \in cellA$ do
    for $particle\ j \in cellB$ do
        if $distance(i, j) < cutoff + skin$ then
            neighborLists[i].add(j)
        end if
    end for
end for
```

As a result, all cells can be traversed in parallel because concurrent memory access at the same location can not happen and it does not matter whether Newton's third law is used or not. The consequence is, that base steps which require more colors in other settings can also be applied in this special case for one color.

The generator functor for filling the neighbor lists requires only one cell to serve as a base cell at the same time. Because of the shifted diagonal interaction, the c08 base step can not be used. Therefore, the idea of the newly implemented traversal is that the base step is oriented on the c18-based traversals. The synchronization strategy is the same as for the one-color-based traversal. As can be seen in Figure 4.9 in 2D, there is only a one-sided interaction between the pairs of cells. One-sided interaction means, that only the neighbor lists of the current base cell are modified but not for the interaction partner. Therefore, as long as a base cell is only assigned to one thread, all interactions can be computed in parallel.

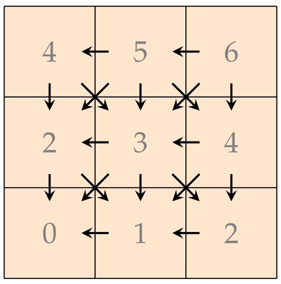


Figure 4.9.: Visualization of the one colored c18-base step. The arrows represent the one-sided rebuild interaction. The direction of the arrows declares the target neighbor lists. As a cell has only interactions which come from the base step neighboring cells, this interaction is safe from race conditions and therefore the whole domain can be painted in one color.

The new traversal aims to save time in the partial rebuilding routine, as it is able to process all cells in parallel. In other color-based traversals, when the dirty cells are unevenly distributed among the different colors, the workload will be unevenly distributed among the colors as well. This problem might manifest for a higher number of threads as some of them might be idle as there is too less workload in the current color slice.

4.4.2. Implementation

The described changes are integrated into AutoPas by declaring a new class `LCC01B18Traversal` which is derived from the `LCC01Traversal`. This overrides the methods `computeOffsets()` for identifying the neighbor interactions and `processBaseCell()` for applying the base step. The first function computes the offsets of the forward neighbors as given by the procedure of the c18 base step. The latter iterates over the computed offsets for a given base cell and computes the indices of the neighbor cells which should be processed. These indices can then be used to access the location of the cells in the global cell vector and apply the pairwise operation on the particles in the neighbor and base cell. As the `traverseParticlePairs()` is inherited from the base class, this new traversal only uses one color as intended. The resulting class hierarchy is visualized by Figure 4.10 in which the newly added class and methods are highlighted in orange.

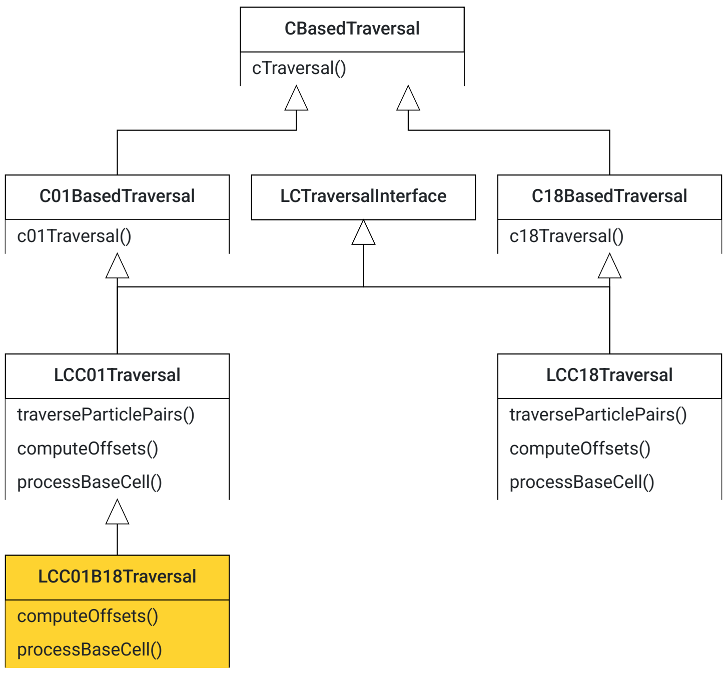


Figure 4.10.: Class diagram visualizing the hierarchy of the new traversal class. The added classes and methods are highlighted in orange.

5. Results

To evaluate the effect of the modifications which are considered in this thesis, computer experiments were conducted for various experimental scenarios. This chapter describes the testing setup and discusses the results.

5.1. Testing Setup

All experiments discussed in the following section are performed on the CooLMUC-2 cluster of the LRZ[1] to guarantee program executions without disturbances by background processes and to enable scenarios with a large number of particles. It provides 812 nodes with 28 cores per node with two hyper threads for each core. This results in 56 threads for shared memory parallelization, each running at a frequency of 2.6 GHz. The program is compiled with GCC[2] 11.2.0 and the performance measurements are performed with VTune[3] version 2021.7.1 (build 619561).

Each presented container improvement was executed on every scenario with a different number of threads and for various skin sizes. The tested program executions include all Verlet List-based containers and their corresponding dynamic implementations. For the `PairwiseVerletLists` container, the partial rebuilding implementation is compared to the dynamic algorithm. Additionally, the implemented one-colored-c18-base-step rebuild traversal was measured compared to the currently used rebuild traversal which is based on the c08-step.

5.2. Scenarios

To provide a better overview of the strengths and weaknesses of the presented modifications under various circumstances, different scenarios were simulated. Their runtimes and application profiles are reported to identify reasons for the respective execution times. The general setting of the chosen experiments is described in the following subsections.

5.2.1. Falling Drop

The falling drop scenario includes two objects, a small sphere of particles that can be interpreted as a drop, and a big cube of particles on the bottom of the domain, representing a basin. Overall, the simulation includes around 120,000 molecules. The boundaries on every border are reflective and a constant gravitational force is applied to all particles. This leads to the drop colliding into the basin after some iterations. After this collision, some particles get bounced away by the drop and accelerate to a higher velocity. Due to the changing particle movement

[1]`https://doku.lrz.de/display/PUBLIC/Linux+Cluster`
[2]`https://gcc.gnu.org/`
[3]`https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html`

over time, the measurements are performed for the whole duration of the simulation, namely 15,000 iterations. This experiment is chosen to evaluate how well the different algorithms perform on changing particle speed. The concrete input file definition is attached in Appendix A.11.

5.2.2. Bursting Cube

The bursting cube experiment includes one single object. A cube with around 100,000 particles is accelerated by a gravitational force in the same way as in the previous scenario. In contrast to the falling drop example, there is no basin of liquid at the bottom of the simulation domain. Instead, the cube collides against the floor and its particles spread over the ground. The simulation domain is reflective on all boundaries- Therefore, a thermostat controls the energy of the simulation domain and is preventing the from jumping into the air again. As the domain is bigger compared to the falling drop example, the distribution of the particles will change from a dense cube to a carpet-like structure at the bottom of the simulation. This experiment is chosen to measure the algorithm's ability to react to changing particle velocity and a different distribution of the molecules over time. The concrete input file definition can be found in Appendix A.12.

5.2.3. Constant Velocity Cube

The constant velocity cube scenario is expected to behave differently compared to both aforementioned scenarios. It includes once more one single cube holding around 50,000 particles. In contrast to the falling drop and bursting cube examples, there is no gravitational force acting on the particles. Instead, the particles' velocity is configured to point in one direction. The strength of the potential ϵ between the molecules is configured to be small so that the interaction between the particles is low. This leads to nearby constant velocity over the simulation time. In this scenario, the speedup of the dynamic algorithm over the static implementation is expected to be smaller because the optimal rebuild frequency is expected to stay constant. The input file is provided in Appendix A.13.

5.3. Performance Results

In the following section, the performance of the different implementations is discussed. For comparing the modifications with the existing AutoPas code, the speedup is computed. It is calculated by diving the runtime of the AutoPas reference code by the runtime of the modification. Therefore, values greater than one indicate a performance advantage of the improvement. In all speedup plots, this threshold is visualized by a dotted line. Additionally, potential reasons for the observations are discussed and, wherever possible, justified by metrics obtained with VTune.

5.3.1. Dynamic Algorithm vs. Static Container

The performance measurements for the dynamic rebuilding routine verify the initial assumption. In scenarios with a high velocity difference, namely the falling drop and bursting cube experiments, the dynamic algorithm scores better runtime results than the static reference implementation. This observation is consistent over all containers and skin sizes. However, there are differences regarding the speedup value when comparing different containers and skin sizes.

First, a larger skin size yields a lower total speedup for the dynamic rebuilding routine. This is due to the fact that for a growing skin size, the list updates become less frequent. In the case of the static container, the rebuild frequency can be lower for larger skin sizes. In the dynamic algorithm, the rebuild condition threshold increases, leading to fewer updates. Therefore, the simulation spends less time rebuilding the neighbor lists. Consequently, the potential time saving decreases with a growing skin factor. Figure 5.1 visualizes this behavior for the `VerletListCells` container in the falling drop example but is also representative for other containers and the bursting cube scenario. The speedup graphs for the other containers can be found in Appendix A.1 and A.2.

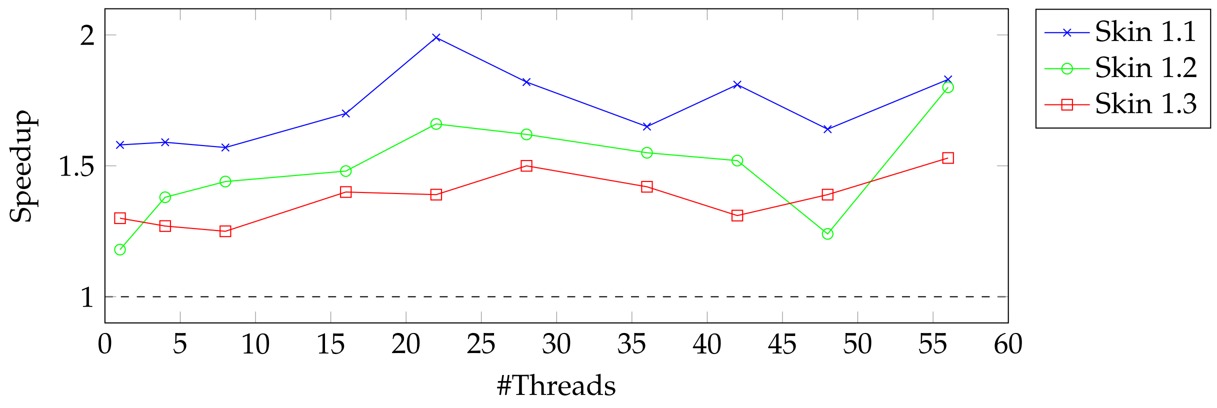


Figure 5.1.: Speedup of the dynamic algorithm compared to the static reference for the `VerletListCells` container in the falling drop example. The different lines represent the different skin sizes.

Second, the speedup differs between the various containers. The `VerletLists` container yields the lowest speedup. For the `PairwiseVerletLists` container, the highest runtime difference can be observed. The speedup of the `VerletListCells` container is located slightly above the VL container but below the PVL container. An explanation for this behavior is the concrete implementation of the neighbor lists. As the `VerletLists` container uses a 1D map from the particle pointers to vectors of pointers. The `PairwiseVerletLists` algorithm needs more complicated structures for mapping the neighbor lists to cell pairs, as discussed in subsubsection 4.3.2. Therefore, rebuilding the lists becomes more time-consuming. The dynamic implementation leads to fewer rebuilds and, therefore, higher time-saving potential for more expensive rebuilds. Figure 5.2 visualizes the proportion of the execution times spent in the neighbor list rebuilds for every measured container in the bursting cube example for a skin size of 1.1 and one thread. The measured proportions support the assumption of the more expensive rebuilds for the VLC and PVL containers compared to the basic Verlet Lists algorithm.

Figure 5.3 provides the various containers' runtimes in the bursting cube example for a skin size of 1.1. The static implementations are represented by the dotted line and their respective dynamic improvements are depicted by the same color with a solid curve. The dotted curves are always above their respective solid counterparts which indicates a clear performance

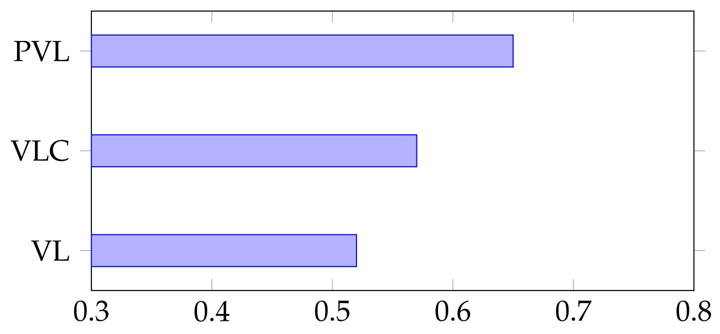


Figure 5.2.: Proportion of the execution time spent in list rebuilds for the static Verlet Lists-based containers in the bursting cube example for a skin size of 1.1 and 1 thread.

advantage of the dynamic implementation in this experiment. As seen in the plot, the PVL implementation needs the most time by default, making it easier to gain a higher speedup. The VL and VLC container perform comparably, and the difference in the runtime plot is not as clearly visible. Similar behavior can be observed for the other skin sizes. For the concrete runtime graphs, have a look at Figure A.3 and Figure A.4.

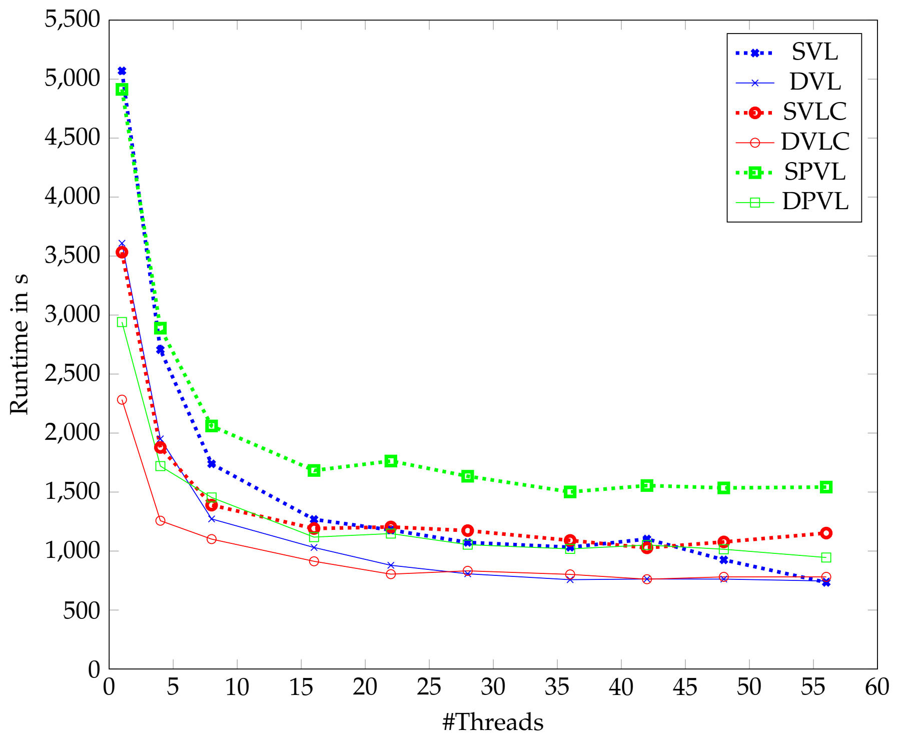


Figure 5.3.: Runtime in seconds for the bursting cube example for different containers with skin size 1.1.

However, when having a look at the speedup plot in Figure 5.4, the curves differ for both containers. This matches the results of the different efforts of rebuilding the various containers. As the observation does not differ much for other skin sizes, their plots are attached in Appendix A.5 and A.6.

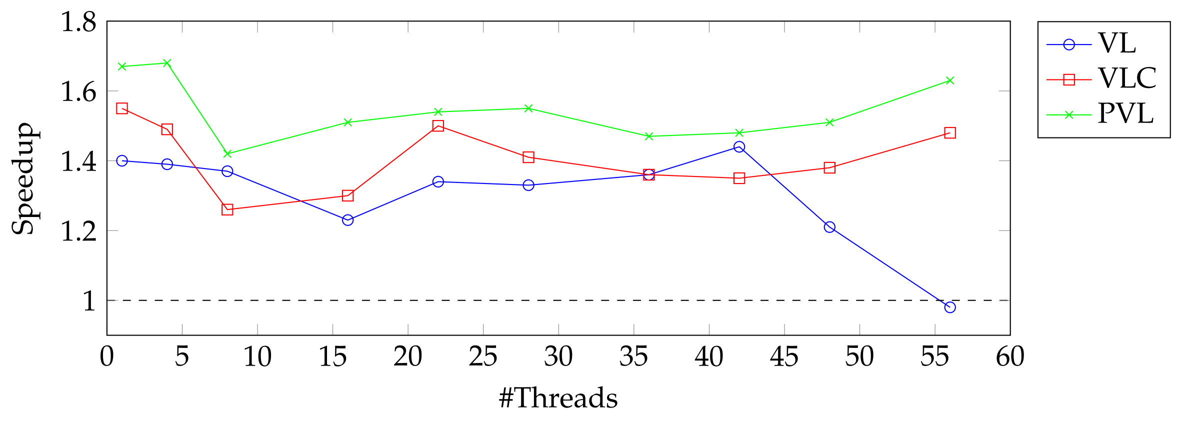


Figure 5.4.: Speedup of the dynamic algorithm compared to the static reference for all containers in the bursting cube scenario for a skin factor of 1.1.

Compared to the bursting cube and falling drop experiments, the speedup of the dynamic algorithms is much lower in the constant velocity cube scenario. The difference in the speedup among the various containers and skin factors is no longer significant. Additionally, there seems to be no correlation between the strength of the speedup and the skin factor. In most cases, the dynamic algorithm still has an advantage but, except for some outliers, the speedup is very close to one. One explanation for such behavior may be, that there is no difference in the velocities over time. Therefore, it is not required to be able to adapt to a changing environment as the rebuilds are necessary in a fixed interval anyway. The reason for the speedup still favoring the dynamic algorithm might be that the guess about the particle movement for the static algorithm was not perfect. As a result, the dynamic criterion triggers the list updates slightly less often. The speedup for a skin factor of 1.3 for the various containers is visualized in Figure 5.5. This graph is representative for the other skin sizes which can be found in Appendix A.7 and A.8.

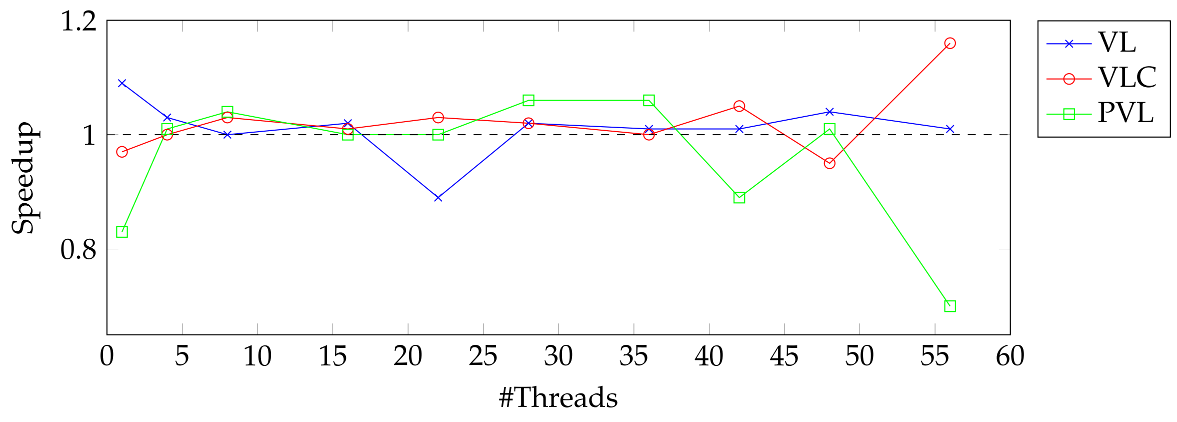


Figure 5.5.: Speedup of the dynamic algorithm compared to the static reference for the various containers for a skin factor of 1.3 in the constant velocity cube example. The different lines represent the different containers.

5.3.2. Partial vs. Dynamic Rebuilding

In the bursting cube example, the partial rebuilding beats the dynamic implementation for skin factors 1.1 and 1.2 when running the application with more threads. For a skin factor of 1.3, the dynamic implementation performs worse than the dynamic reference implementation for every number of threads. Figure 5.6 depicts this by the different curves representing the various skin sizes.

One observation is the positive relationship between the number of threads and the speedup. A potential reason could be that the partial rebuilding procedure introduces overhead for identifying the dirty cells and determining the lists that need to be updated. Especially for larger domains, like in the bursting cube scenario, this overhead significantly influences the total execution time. In both cases, the overhead can be reduced by assigning more threads. Because no concurrent memory access at the same location occurs, the corresponding `for`-loops allow for easy parallelization with OpenMP. For a higher number of threads, the achieved time saving seems to be able to exceed the overhead.

Additionally, the speedup of the partial routine compared to the dynamic algorithm is higher for lower skin factors. This observation is due to the amount of rebuilt cells for the various skin sizes. A lower skin factor leads to fewer list updates for the dynamic algorithm because the threshold of the rebuild criterion is higher. The increase of the skin factor from 1.1 to 1.2 reduces the number of rebuilt cells for the dynamic container by 58%. For the partial rebuilding implementation, the proportional decrease is lower with a value of 49%. The reason is, that the skin size only influences the number of cells with high particle movement. The decrease for the moving cells is with a proportion of 59% according to the decrease of the dynamic algorithm. For the total amount of rebuilt cells, the inflow cells must be considered as well. The decrease of the inflow cells is with a proportional value of 40% below the value of the moving cells. This leads to a lower reduction of the total amount of rebuilt cells for the partial rebuilding compared to the dynamic container and, as a result, to a lower speedup. The actual numbers for the rebuild statistic are provided in Table 5.1.

Container (Skin Factor)	Rebuilt Cells	Moving Cells	Inflow Cells	Avg List Rebuilds / Iteration
Dynamic (1.1)	226,880,568	-	-	NA
Partial (1.1)	13,519,949	10,155,247	5,808,883	NA
Dynamic (1.2)	95,195,700	-		176,058
Partial (1.2)	6,838,622	4,119,677	3,477,077	157,513
Dynamic (1.3)	52,783,919	-	-	122,730
Partial (1.3)	4,425,775	2,259,656	2,362,545	121,512

Table 5.1.: This table includes various metrics for the neighbor list rebuilds for the partial and dynamic PVL container in the bursting cube scenario. Entries with NA indicate that the respective could not be obtained because of technical limitations such as integer overflows. As the dynamic algorithm does not distinguish between moving and inflow cells but rebuilds every cell in each list update, the respective entries do not contain a value.

A reason for the partial algorithm being slower than the dynamic implementation for higher skin sizes, as it is shown in Figure 5.6, is provided by the number of rebuilt neighbor lists. From Table 5.1, it can be concluded that for a skin size of 1.3, the number of neighbor lists being updated is nearly equal for the dynamic and partial algorithms. The advantage of the dynamic implementation results from the overhead of the partial rebuilding routine. While the dynamic algorithm can clear every list and rebuild them again, the partial procedure has to identify which lists to clear and update. For an equal number of rebuilt lists, the construction costs like reserving memory and traversing the particle pairs are expected not to be different. The runtime difference is a consequence of the discussed overhead of the partial rebuilding container.

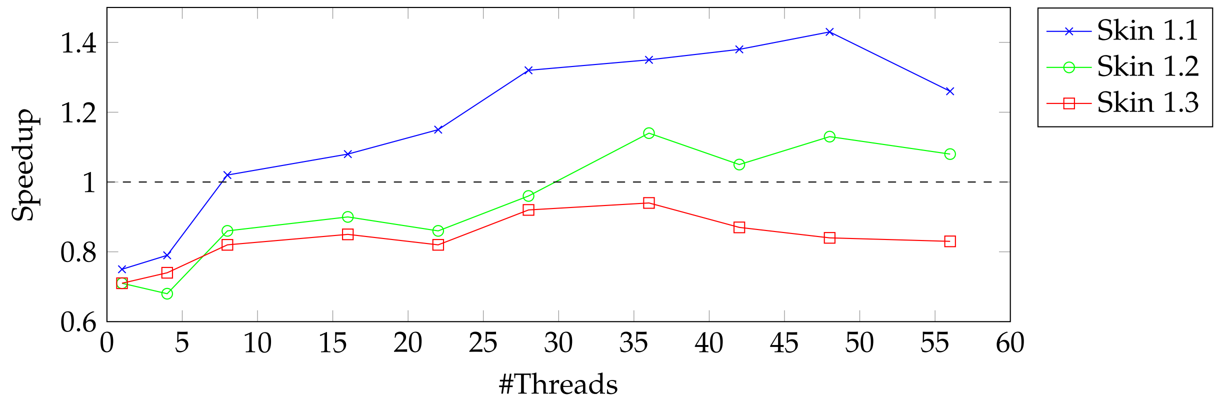


Figure 5.6.: Speedup of the partial rebuilding algorithm compared to the dynamic PVL container in the bursting cube example. The different lines represent the different skin sizes.

In the results for the falling drop scenario, the same correlation between the speedup and the skin size can be observed. Opposed to this, no correlation between the speedup and the number of threads is present as visualized in Figure 5.7. The reason might be the smaller domain. As the simulation area is more limited compared to the bursting cube scenario, it contains fewer cells. Adding more threads for iterating over all cells while identifying the dirty neighbor lists might add more overhead for synchronization because the workload is too low. However, for a small skin factor of 1.1, the partial rebuilding routine performs better than the dynamic algorithm for every number of threads.

In the constant velocity cube example, the partial rebuilding has the highest runtime compared to the static and dynamic implementations. Figure 5.8 visualizes this by highlighting the speedup for the dynamic and partial algorithms with the static container as the reference implementation. The curves for the partial algorithm are constantly below the lines of the dynamic routine and the threshold of one for every skin factor. Therefore, the partial implementation is the worst-performing algorithm in this scenario. This observation is as expected as the partial rebuilding procedure is designed for scenarios with few high-moving particles. In the constant velocity cube example, all particles start with the same velocity. Because there is no gravitational force acting on them, the movement is expected to stay constant. Therefore, using the partial rebuilding algorithm does not yield any runtime benefit but the overhead of the routine remains.

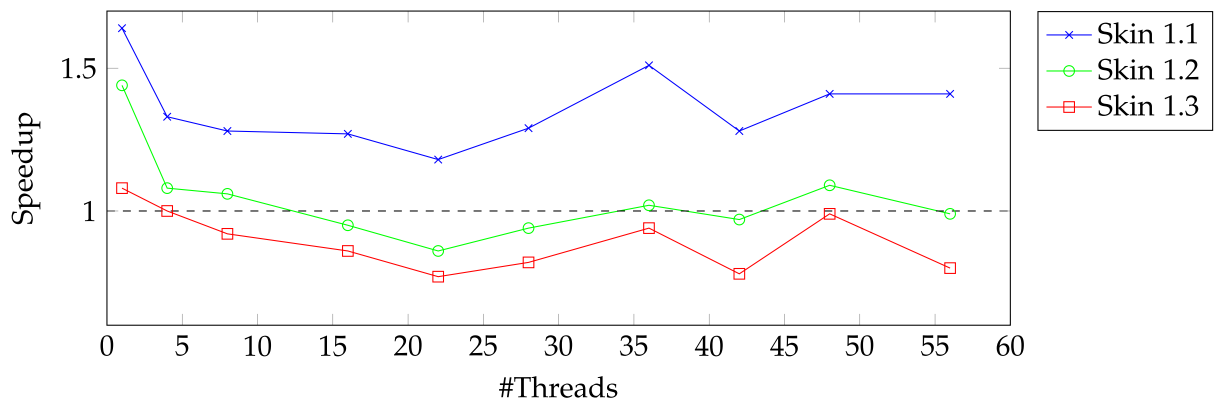


Figure 5.7.: Speedup of the partial rebuilding algorithm compared to the dynamic PVL container in the falling drop scenario. The different lines represent the different skin sizes.

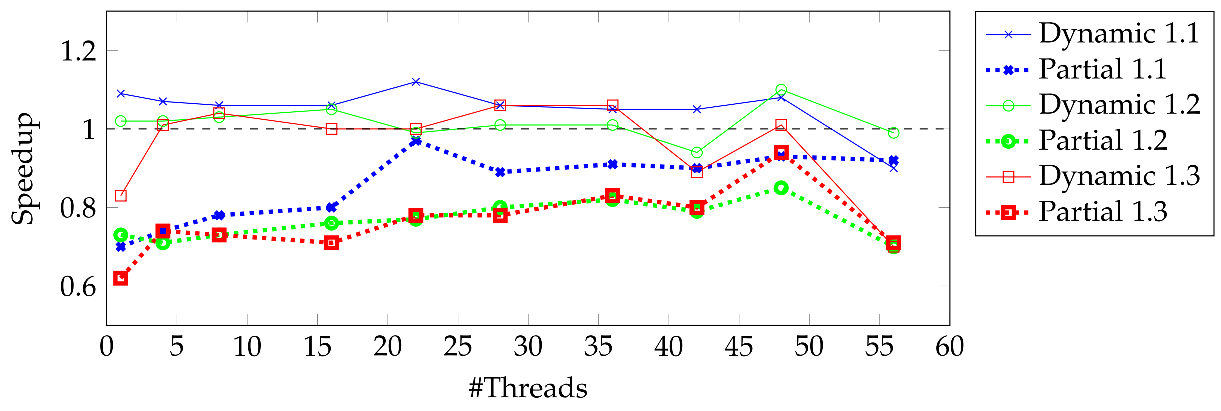


Figure 5.8.: Speedup of the partial rebuilding algorithm compared to the dynamic PVL container in the constant velocity cube scenario. The reference implementation to calculate the speedup is the static PVL container.

5.3.3. C01_b18 vs. C08 Rebuild Traversal

The measurements for the one-colored c18-base-step traversals indicate an advantage over the previously used c08-based traversal. It was applied to the VLC, PVL and `PartialVerletLists` container. Except for some outliers, the speedup was above the threshold of one for every container, configured number of threads, and skin factor in every scenario. Figure 5.8 visualizes the speedup of the c01_b18 traversal for the partial rebuilding algorithm in the constant velocity cube experiment.

A reason for the speedup being in favor of the new traversal is provided by the CPU's spin time. The metrics obtained with VTune indicate a higher spin time for the c08 traversal compared to the c01_b18 traversal for the same number of threads. Table 5.2 provides an overview of the total CPU time, the effective computation time, and the spin time. VTune states that a potential reason for the spin and wait time being high can be load imbalance. Because the new rebuild traversal is designed to achieve a better workload for every thread, the measurements prove the success of this extension.

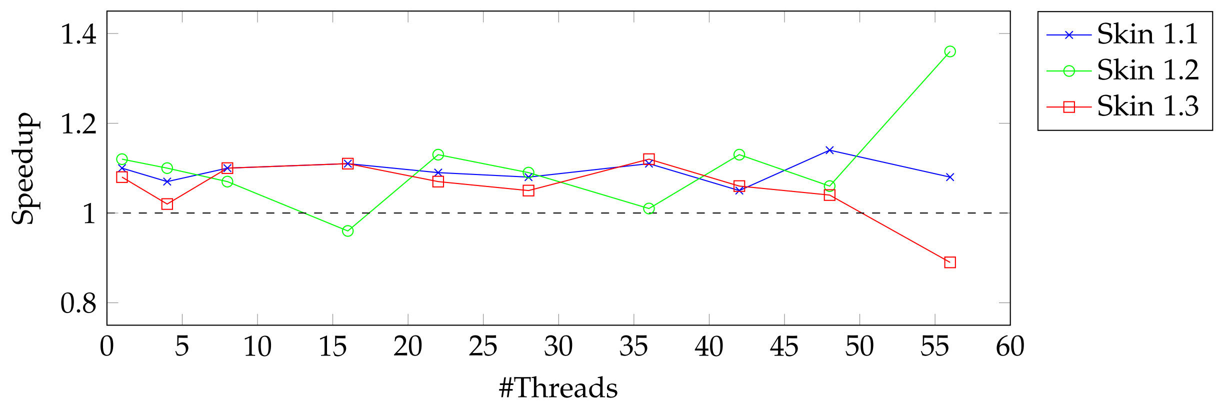


Figure 5.9.: Speedup of the `PartialVerletLists` container with c01_b18 rebuild traversal compared to the c08 traversal in the constant velocity cube example. The different lines represent the different skin sizes.

Traversal (Skin Factor)	CPU Time	Effective Time	Spin Time
c08 (1.1)	3,322	1,878	1,442
c01_b18 (1.1)	3,177	1,956	1,218
c08 (1.2)	3,159	1,647	1,510
c01_b18 (1.2)	2,749	1,589	1,150
c08 (1.3)	3,218	1,646	1,571
c01_b18 (1.3)	2,805	1,538	1,267

Table 5.2.: Various CPU-usage related metrics for the `PartialVerletLists` container in the constant cube example when running the application with 42 threads.

Similar observations regarding the speedup are visible for the other containers and experiments as well. Figure 5.10 depicts the speedup for the `PairwiseVerletLists` implementation in the bursting cube scenario.

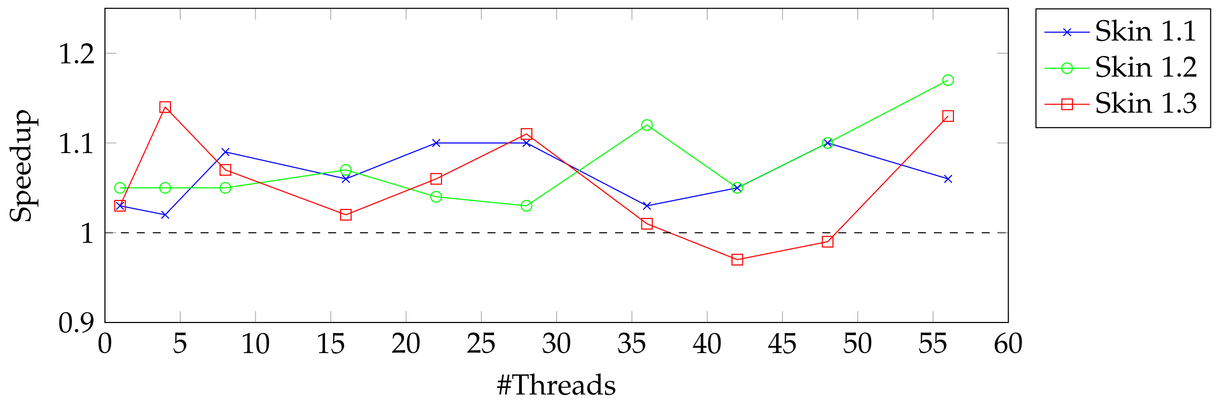
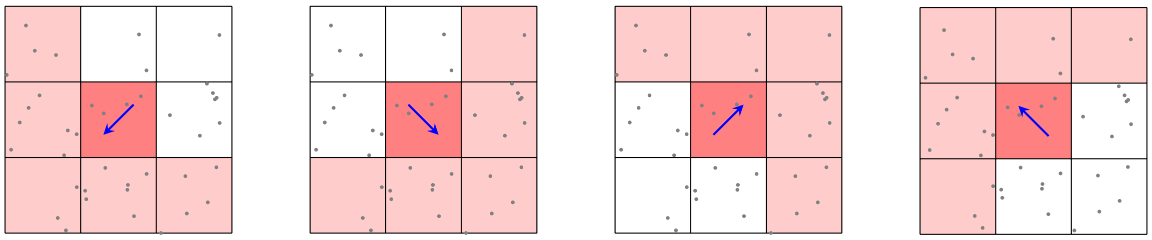


Figure 5.10.: Speedup of the `PairwiseVerletLists` container with c01_b18 rebuild traversal compared to the c08 traversal in the bursting cube example. The different lines represent the different skin sizes.

6. Future Work

The improvements presented in Chapter 4 are only a subset of possibilities to enhance the AutoPas library. In the following, some more algorithmic improvements regarding the rebuilding of neighbor lists are presented.

Given a cell with high-moving particles, the current partial rebuilding routine updates all neighbor lists for every adjacent cell. In some scenarios, the number of these cell-wise updates could be reduced by looking at the direction of the displacement of the high-moving particles. When identifying these particles, not only the dirty state of a cell should be stored, but also the displacement direction. The list updates then only need to be carried out on the neighboring cells located in the direction of the displacement. Figure 6.1 visualizes the different displacement patterns in 2D. When multiple particles have different displacements, the union of the light red cells must be considered.



(a) Particle displacement toward the left lower corner results in the lower and left neighbor cells being rebuilt.

(b) Particle displacement toward the right lower corner results in the lower and right neighbor cells being rebuilt.

(c) Particle displacement toward the right upper corner results in the upper and right neighbor cells being rebuilt.

(d) Particle displacement toward the left upper corner results in the upper and left neighbor cells being rebuilt.

Figure 6.1.: Different particle displacement directions visualized by the blue arrow and the resulting dirty cells in red and their necessary rebuild neighbors in light red color.

As multiple different directions can lead to every neighbor being updated, this routine should only be used in rare scenarios. Applications for this procedure might be experiments where many molecules have a common movement direction for a longer time. An example is provided by the falling drop experiment, where a larger amount of particles move only toward the bottom for many timesteps. This routine will introduce overhead, but there might be scenarios where the time saving exceeds the effort. Therefore, more research on this idea may be beneficial.

Another approach to reducing the neighbor list rebuilds is given by enhancing the rebuild criterion. As already mentioned, Yao et al. [10] suggest checking the sum of the largest two particle displacements within a cell and marking it as dirty when the sum exceeds the skin size. Because this condition is more restrictive, the number of rebuild iterations can be decreased further. It is therefore worth considering integrating this approach in AutoPas as well and evaluating the speedup of this improvement.

Apart from rebuilding Verlet List, Verlet List Cells, and Pairwise Verlet List, the dynamic algorithm discussed in subsection 4.2.1 is also applicable to other containers. For example, the `LinkedCells` AutoPas container does not exchange its particles between the cells in every iteration. So far, the updates of the cell memberships are also controlled by the rebuild frequency entered by the user of AutoPas. The problem of under-approximating the optimal frequency arises in this context as well, especially in simulations with changing circumstances. Therefore, it might be beneficial to integrate a dynamic approach for the Linked Cells particle exchange as well.

7. Conclusion

Within this thesis, different approaches were evaluated for reducing the time spent on neighbor list rebuilds for the Verlet List algorithm. Implementing a dynamic rebuilding algorithm addressed the initial problem of unnecessarily many updates when using a fixed rebuild frequency. This aims to provide more flexibility in changing scenarios and relieves the user by not having to think of an optimal configuration for the rebuild frequency. Furthermore, a procedure was constructed for reducing the number of neighbor list updates per rebuild iteration by integrating a partial rebuilding approach from the literature into AutoPas. Additionally, a new traversal for the Linked Cells container was provided with the goal of better parallelization of the rebuild routine. The improvements were described regarding their algorithmic idea and the concrete integration into the AutoPas library.

Different simulations were conducted with various configurations to compare the considered improvements with the existing AutoPas algorithms. The results for scenarios with a high velocity difference of the particles over time show an advantage of the dynamic algorithm. The benefit is consistently present and higher for smaller skin sizes. This implies that the initial goal of achieving more flexibility in a changing environment was reached. The time saving takes values of up to 50% in experiments with changing circumstances over time. Therefore, this algorithm is a valuable extension of the AutoPas library.

The partial rebuilding procedure is advantageous over the dynamic implementation for small skin sizes and large domains. For configurations with a larger skin size, the partial approach has a higher execution time. The best-case scenario achieved a time saving of up to 20% compared to the dynamic algorithm. Finally, because there exist configurations for which the partial algorithm is the better choice, it was worth extending AutoPas with this feature.

The new rebuild traversal also succeeded in reducing the computational effort of the neighbor list updates. The execution time for traversing the particle pairs in the rebuilds was decreased compared to the previously used traversal. The execution time was reduced by up to 10%, proving a useful enhancement for AutoPas.

For further improvements of the neighbor list rebuilding, the suggested possibilities for future work can be taken into consideration.

A. Appendix

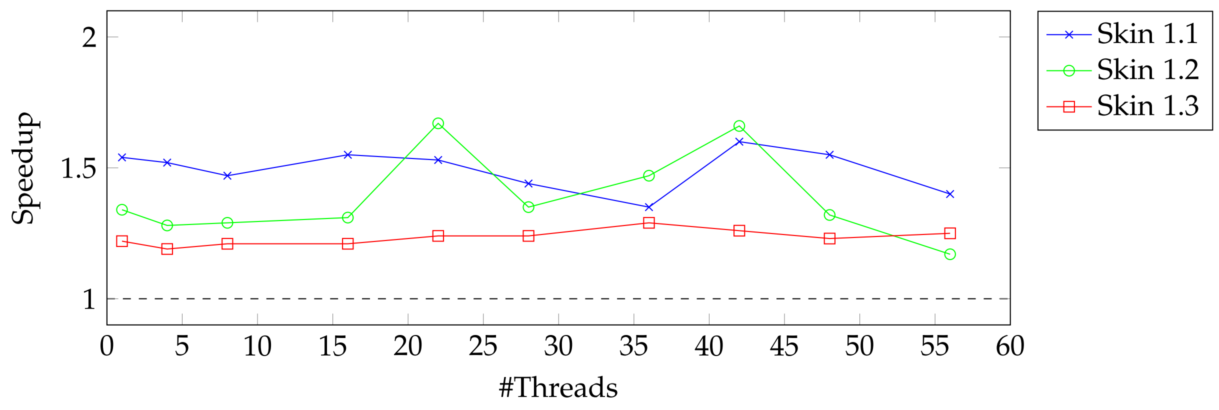


Figure A.1.: Speedup of the dynamic algorithm compared to the static reference for the `VerletLists` container in the falling drop example. The different slopes represent the different skin sizes.

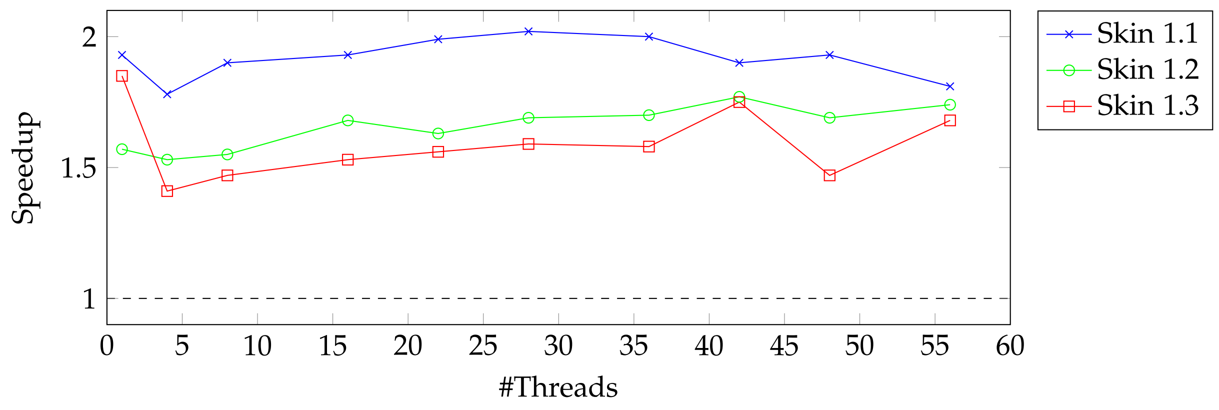


Figure A.2.: Speedup of the dynamic algorithm compared to the static reference for the `PairwiseVerletLists` container in the falling drop example. The different slopes represent the different skin sizes.

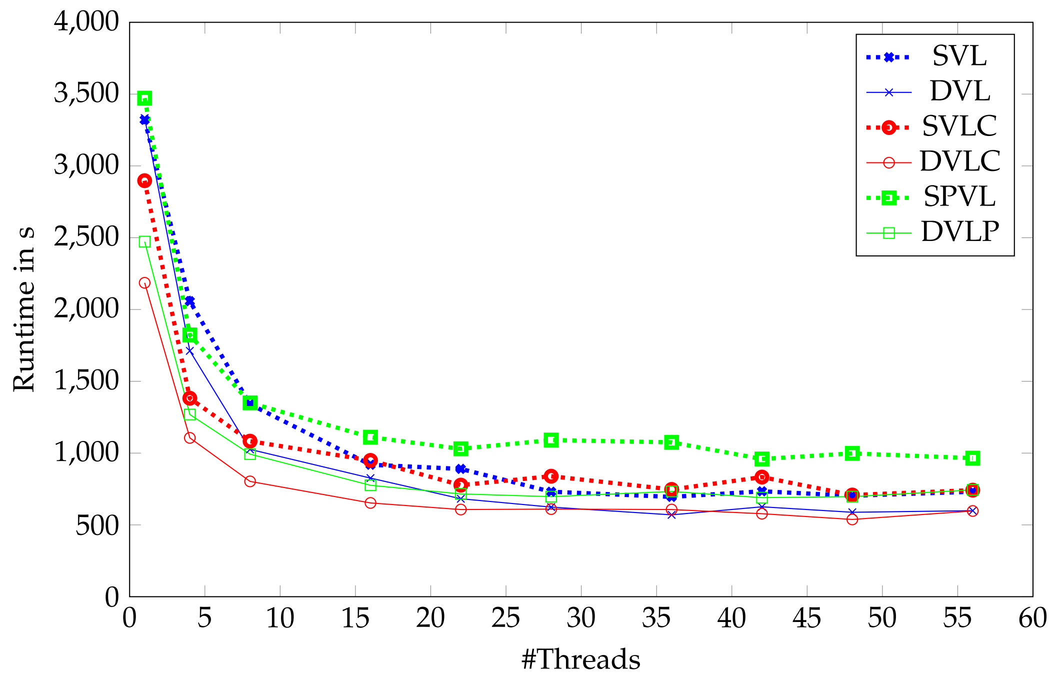


Figure A.3.: Runtime in seconds for the bursting cube example for all containers with skin size 1.2.

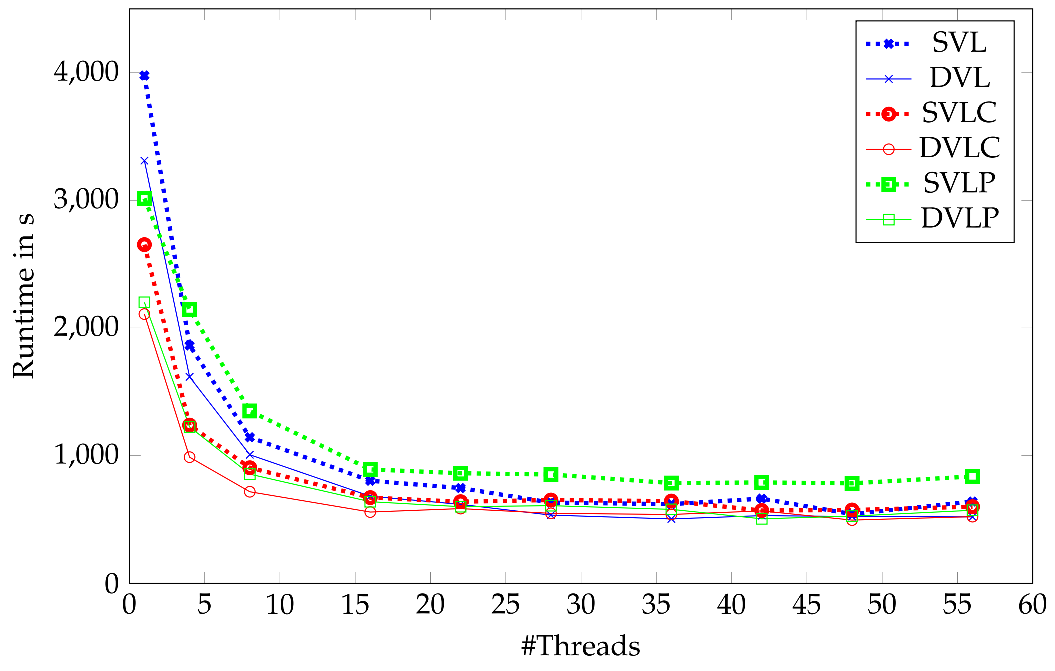


Figure A.4.: Runtime in seconds for the falling bursting cube example for all containers with skin size 1.3.

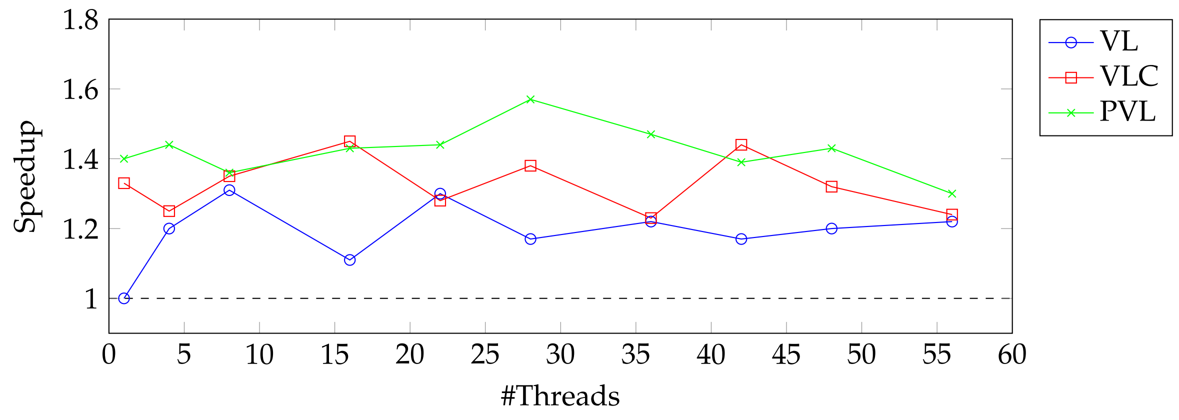


Figure A.5.: Speedup of the dynamic algorithm compared to the static reference for all containers in the bursting cube scenario for a skin factor of 1.2.

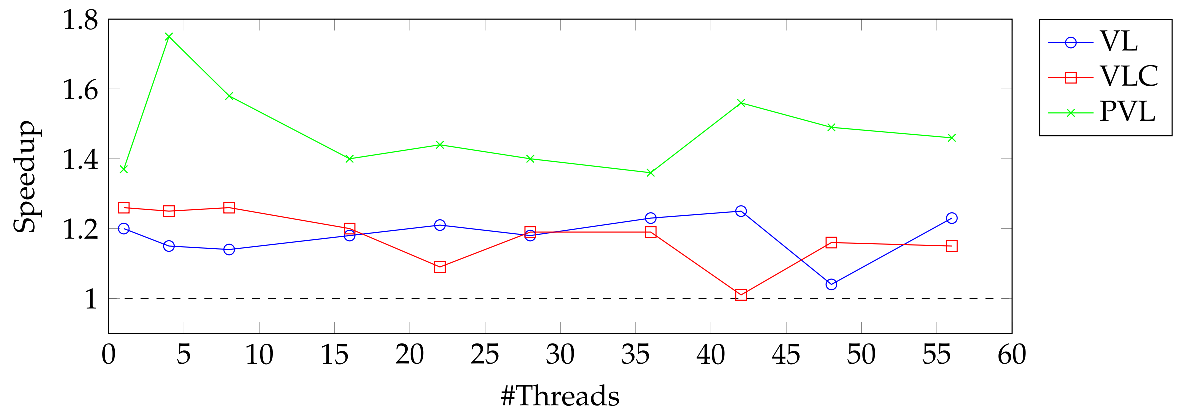


Figure A.6.: Speedup of the dynamic algorithm compared to the static reference for all containers in the bursting cube scenario for a skin factor of 1.3.

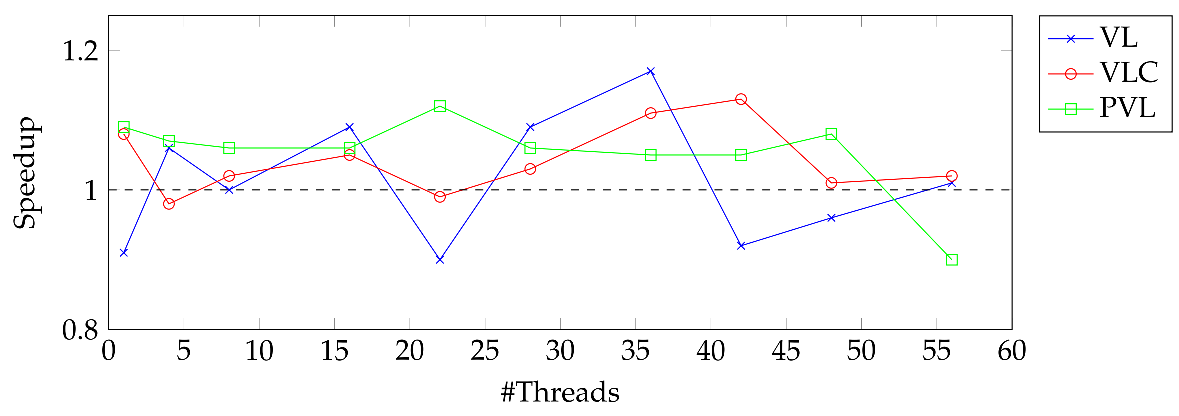


Figure A.7.: Speedup of the dynamic algorithm compared to the static reference for the various containers for a skin factor of 1.1 in the constant velocity cube example. The different slopes represent the different containers.

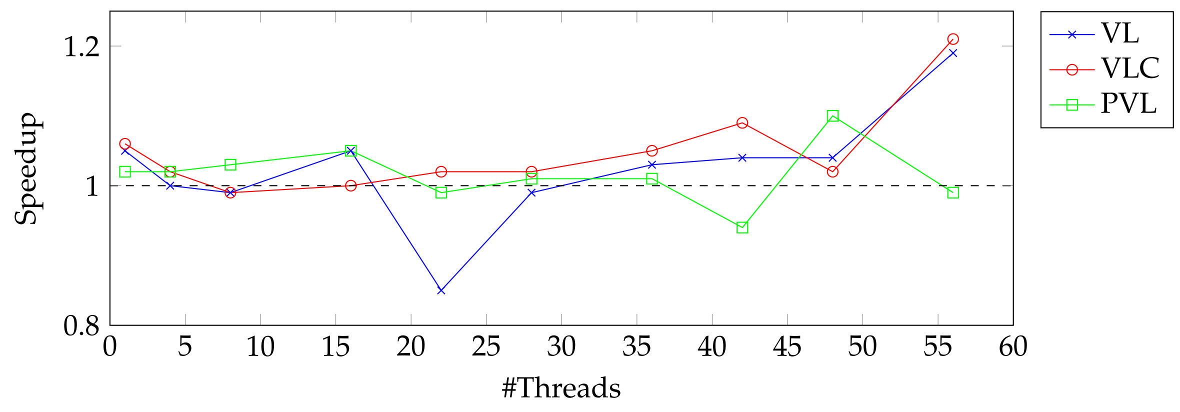


Figure A.8.: Speedup of the dynamic algorithm compared to the static reference for the various containers for a skin factor of 1.2 in the constant velocity cube example. The different slopes represent the different containers.

Container (Skin Factor)	Rebuilt Cells	Moving Cells	Inflow Cells	Avg List Rebuilds / Iteration
Dynamic (1.1)	41,814,288	-	-	NA
Partial (1.1)	6,200,867	4,486,950	3,551,484	NA
Dynamic (1.2)	16,865,530	-	-	NA
Partial (1.2)	2,937,414	1,643,612	2,022,981	136,769
Dynamic (1.3)	9,002,880	-	-	115,583
Partial (1.3)	1,805,359	847,596	1,291,508	104,950

Table A.1.: Table for various metrics for the neighbor list rebuilds for the partial and dynamic PVL container in the falling drop scenario. Entries with NA indicate that the respective value was to high to be measured. As the dynamic algorithm does not distinguish between moving and inflow cells but rebuilds every cell in each list update, the respective entries do not contain a value.

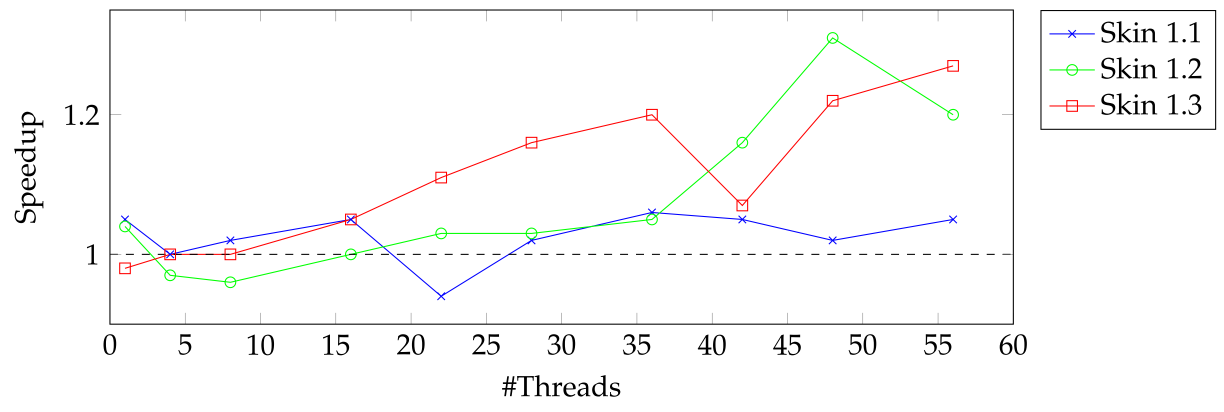


Figure A.9.: Speedup of the `PartialVerletLists` container with c01_b18 rebuild traversal compared to the c08 traversal in the bursting cube example. The different slopes represent the different skin sizes.

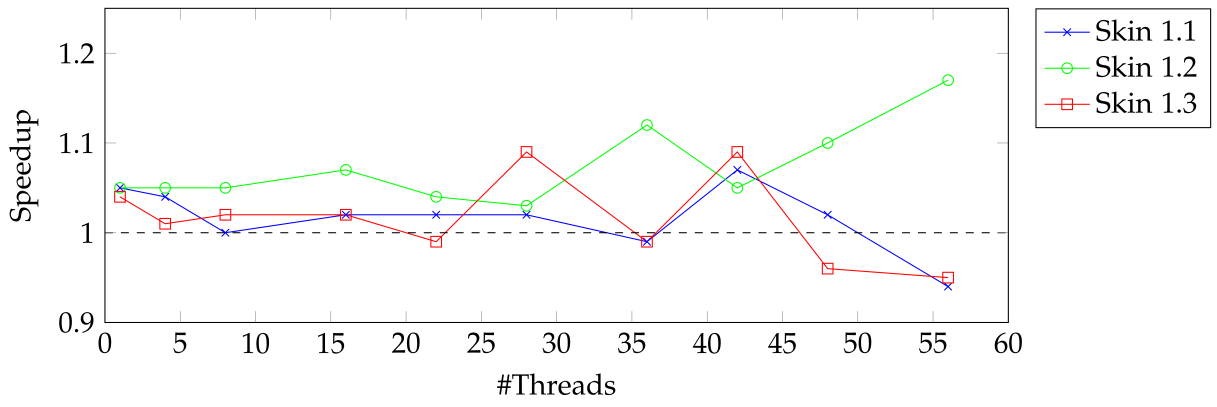


Figure A.10.: Speedup of the `VerletListCells` container with c01_b18 rebuild traversal compared to the c08 traversal in the bursting cube example. The different slopes represent the different skin sizes.

```
container                          :  [DynamicVerletLists]
verlet-rebuild-frequency           :  1000
verlet-skin-radius-per-timestep    :  0.0003
selector-strategy                  :  Fastest-Absolute-Value
data-layout                        :  [AoS]
traversal                          :  [ vl_list_iteration ]
tuning-strategy                    :  full-Search
tuning-interval                    :  2500
tuning-samples                     :  3
tuning-max-evidence                :  10
functor                            :  Lennard-Jones avx
newton3                            :  [disabled]
cutoff                             :  3
cell-size                          :  [1]
deltaT                             :  0.0005
iterations                         :  15000
boundary-type                      :  [reflective,reflective,reflective]
globalForce                        :  [0,0,-12]
Objects:
  # "basin"
  CubeClosestPacked:
    0:
      particle-spacing             :  1.122462048
      bottomLeftCorner             :  [1, 1, 1]
      box-length                   :  [96, 56, 20]
      velocity                     :  [0, 0, 0]
      particle-type                :  0
      particle-epsilon             :  1
      particle-sigma               :  1
      particle-mass                :  1
  # "drop"
  Sphere:
    0:
      center                       :  [36, 30, 60]
      radius                       :  12
      particle-spacing             :  1.122462048
      velocity                     :  [0, 0, 0]
      particle-type                :  1
      particle-epsilon             :  1
      particle-sigma               :  1
      particle-mass                :  1
fast-particle-warn                 :  false
```

Figure A.11.: Input YAML definition for the falling drop example for the dynamic `VerletLists` container and a skin factor of 1.1.

```
container                          :  [PairwiseVerletLists]
verlet-rebuild-frequency           :  4
verlet-skin-radius-per-timestep    :  0.15
selector-strategy                  :  Fastest-Absolute-Value
data-layout                        :  [AoS]
traversal                          :  [ vlp_c01, vlp_c18, vlp_sliced,
                                      vlp_sliced_balanced, vlp_sliced_c02 ]
tuning-strategy                    :  full-Search
tuning-interval                    :  2500
tuning-samples                     :  3
tuning-max-evidence                :  10
functor                            :  Lennard-Jones avx
newton3                            :  [enabled]
cutoff                             :  3
box-min                            :  [0, 0, 0]
box-max                            :  [200, 200, 85]
cell-size                          :  [1]
deltaT                             :  0.0005
iterations                         :  10000
boundary-type                      :  [reflective,reflective,reflective]
globalForce                        :  [0,0,-12]
Objects:
  CubeClosestPacked:
    0:
      particle-spacing             : 1
      bottomLeftCorner             : [ 75, 75, 10 ]
      box-length                   : [ 50, 50, 30 ]
      velocity                     : [ 0, 0, 0 ]
      particle-type                : 0
      particle-epsilon             : 1
      particle-sigma               : 1
      particle-mass                : 1
thermostat:
  initialTemperature               :  50
  targetTemperature                :  5
  deltaTemperature                 :  0.25
  thermostatInterval               :  10
  addBrownianMotion                :  true
fast-particle-warn                 :  true
```

Figure A.12.: Input YAML definition for the bursting cube example for the static PairwiseVerletLists container and a skin factor of 1.2.

```
container                         :  [PartialPairwiseVerletLists]
verlet-rebuild-frequency          :  1000
verlet-skin-radius-per-timestep   :  0.0009
selector-strategy                 :  Fastest-Absolute-Value
data-layout                       :  [AoS]
traversal                         :  [ vlp_c01, vlp_c18, vlp_sliced,
                                     vlp_sliced_balanced, vlp_sliced_c02 ]
tuning-strategy                   :  full-Search
tuning-interval                   :  2500
tuning-samples                    :  3
tuning-max-evidence               :  10
functor                           :  Lennard-Jones avx
newton3                           :  [enabled]
cutoff                            :  4
box-min                           :  [0, 0, 0]
box-max                           :  [100, 100, 85]
cell-size                         :  [1]
deltaT                            :  0.0005
iterations                        :  5000
boundary-type                     :  [reflective,reflective,reflective]
globalForce                       :  [0,0,0]
Objects:
  # "water"
  CubeClosestPacked:
    0:
      particle-spacing            : 1.5
      bottomLeftCorner            : [ 15, 15, 30 ]
      box-length                  : [ 50, 50, 50 ]
      velocity                    : [ 0, 0, -5 ]
      particle-type               : 0
      particle-epsilon            : 0.5
      particle-sigma              : 1
      particle-mass               : 1
thermostat:
  initialTemperature              :  20
  targetTemperature               :  20
  deltaTemperature                :  2
  thermostatInterval              :  20
  addBrownianMotion               :  false
fast-particle-warn                :  false
```

Figure A.13.: Input YAML definition for the constant velocity cube example for the `PartialVerletLists` container and a skin factor of 1.3.

List of Figures

List of Tables

Bibliography

[1] M. Griebel, S. Knapek, and G. Zumbusch. *Numerical simulation in molecular dynamics, vol. 5 of Texts in Computational Science and Engineering*. 2007.

[2] Q. Li, Z. Dong, S. Zhou, F. Han, C. Li, H. Chang, and Z. Zhang. "Investigation of Grain Boundary Content on Crack Propagation Behavior of Nanocrystalline Al by Molecular Dynamics Simulation". In: *physica status solidi (b)* 259.7 (2022), p. 2100570. DOI: `https://doi.org/10.1002/pssb.202100570`. eprint: `https://onlinelibrary.wiley.com/doi/pdf/10.1002/pssb.202100570`. URL: `https://onlinelibrary.wiley.com/doi/abs/10.1002/pssb.202100570`.

[3] F. A. Gratl, S. Seckler, H.-J. Bungartz, and P. Neumann. "N ways to simulate short-range particle systems: Automated algorithm selection with the node-level library AutoPas". In: *Computer Physics Communications* 273 (2022), p. 108262. ISSN: 0010-4655. DOI: `https://doi.org/10.1016/j.cpc.2021.108262`. URL: `https://www.sciencedirect.com/science/article/pii/S001046552100374X`.

[4] A. Satoh. "1 - Outline of Molecular Simulation and Microsimulation Methods". In: *Introduction to Practice of Molecular Simulation*. Ed. by A. Satoh. London: Elsevier, 2011, pp. 1–27. ISBN: 978-0-12-385148-2. DOI: `https://doi.org/10.1016/B978-0-12-385148-2.00001-X`. URL: `https://www.sciencedirect.com/science/article/pii/B978012385148200001X`.

[5] A. Straßer. *Entwicklung von Potentialparametern der intermolekularen Wechselwirkung in Wasser zur Berechnung thermodynamischer und struktureller Eigenschaften reinen flüssigen Wassers und wässriger Lösungen - Integralgleichungsmethoden und deren Kopplung mit quantenchemischen Verfahren*. Apr. 2003. URL: `https://epub.uni-regensburg.de/10084/`.

[6] J. F. Boudreau and E. S. Swanson. *Applied Computational Physics*. Oxford University Press, Dec. 2017. ISBN: 9780198708636. DOI: `10.1093/oso/9780198708636.001.0001`. URL: `https://doi.org/10.1093/oso/9780198708636.001.0001`.

[7] Wikipedia contributors. *Lennard-Jones potential — Wikipedia, The Free Encyclopedia*. [Online; accessed 18-July-2023]. 2023. URL: `https://en.wikipedia.org/w/index.php?title=Lennard-Jones_potential&oldid=1165331912`.

[8] F. A. Gratl, S. Seckler, N. Tchipev, H.-J. Bungartz, and P. Neumann. "AutoPas: Auto-Tuning for Particle Simulations". In: *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. May 2019, pp. 748–757. DOI: `10.1109/IPDPSW.2019.00125`.

[9] L. Verlet. "Computer "Experiments" on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules". In: *Phys. Rev.* 159 (1 July 1967), pp. 98–103. DOI: `10.1103/PhysRev.159.98`. URL: `https://link.aps.org/doi/10.1103/PhysRev.159.98`.

[10] Z. Yao, J.-S. Wang, G.-R. Liu, and M. Cheng. "Improved neighbor list algorithm in molecular simulations using cell decomposition and data sorting method". In: *Computer Physics Communications* 161.1 (2004), pp. 27–35. ISSN: 0010-4655. DOI: `https://doi.org/10.1016/j.cpc.2004.04.004`. URL: `https://www.sciencedirect.com/science/article/pii/S0010465504002097`.

[11] F. Gratl and S. Newcome. *PSE Molecular Dynamics Intro Worksheet 3*.

[12] S. J. Newcome. "AutoPas: Optimising Multi-site Molecular Dynamics Simulations with Auto-tuning and Kokkos". en. In: *European Seminar on Computing*. June 2022.

[13] D. Asch. "Auto-Tuning Verlet List Skin Lengths in AutoPas". en. Bachelor's Thesis. Technical University of Munich, Mar. 2023.

[14] P. Gonnet. "Pairwise verlet lists: Combining cell lists and verlet lists to improve memory locality and parallelism". In: *Journal of Computational Chemistry* 33.1 (2012), pp. 76–81. DOI: `https://doi.org/10.1002/jcc.21945`. eprint: `https://onlinelibrary.wiley.com/doi/pdf/10.1002/jcc.21945`. URL: `https://onlinelibrary.wiley.com/doi/abs/10.1002/jcc.21945`.

[15] S. Páll, A. Zhmurov, P. Bauer, M. Abraham, M. Lundborg, A. Gray, B. Hess, and E. Lindahl. "Heterogeneous parallelization and acceleration of molecular dynamics simulations in GROMACS". In: *The Journal of Chemical Physics* 153.13 (Oct. 2020), p. 134110. ISSN: 0021-9606. DOI: `10.1063/5.0018516`. eprint: `https://pubs.aip.org/aip/jcp/article-pdf/doi/10.1063/5.0018516/16736127/134110\_1\_online.pdf`. URL: `https://doi.org/10.1063/5.0018516`.

[16] A. P. Thompson, H. M. Aktulga, R. Berger, D. S. Bolintineanu, W. M. Brown, P. S. Crozier, P. J. in 't Veld, A. Kohlmeyer, S. G. Moore, T. D. Nguyen, R. Shan, M. J. Stevens, J. Tranchida, C. Trott, and S. J. Plimpton. "LAMMPS - a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales". In: *Computer Physics Communications* 271 (2022), p. 108171. ISSN: 0010-4655. DOI: `https://doi.org/10.1016/j.cpc.2021.108171`. URL: `https://www.sciencedirect.com/science/article/pii/S0010465521002836`.

[17] T. Vladimirova. "Implementation and Evaluation of Verlet List-based Methods in AutoPas". en. Bachelor's Thesis. Technical University of Munich, Feb. 2021.