SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY - INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Implementation of Verlet Lists for 3-Body Interactions in AutoPas

Alexander Haberl

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY - INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Implementation of Verlet Lists for 3-Body Interactions in AutoPas

Implementierung von Verlet Listen für 3-Körper Wechselwirkungen in AutoPas

Author:	Alexander Haberl
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisor:	Markus Mühlhäußer, M.Sc.; Fabio Gratl, M.Sc
Date:	15.04.2024

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

Munich, 15.04.2024 Alexander Haberl

Acknowledgments

I want to thank Professor Hans-Joachim Bungartz and the Chair of Scientific Computing for providing this thesis topic.
My gratitude also extends to my advisors Markus and Fabio for their support throughout this bachelor thesis, as they have helped me with both practical and theoretical questions. Our weekly meetings also helped me to keep thinking more critically about the findings made during the writing of this thesis.
I give thanks to the Leibniz Rechen Zentrum for access to the CoolMUC-2 cluster which was used for the simulations that are the basis of the evaluation in this thesis. Additional thanks to Fabio for running some simulations on the HSUper cluster.
My family has been a great help throughout this thesis, so I want to thank them for their general support during this time.

Abstract

2-Body molecular dynamics simulations do not always capture the correct qualitative changes to a system. In such cases, we need higher accuracy, which can be achieved by also considering 3-body interactions. To this end, we extend the molecular simulations library AutoPas with an implementation of Verlet lists that can handle 3-body interactions.

We consider three different approaches to iterate over all triplets in the force calculation step. The first uses 2-body neighbor lists and iterates over all pairs in them. The second approach intersects the lists of neighboring particles to obtain all mutual neighbors. Lastly, we save 3-body neighbor lists which contain all pairs of particles close enough for force calculation, which we can iterate over to find all triplets.

We find that the third approach is the fastest in medium to high-density simulations, while the first approach outperforms it in low-density scenarios. We also find that the 3-body neighbor lists take up much more memory than the 2-body neighbor lists, with the average neighbor list having a length that is proportional to the square of the average 2-body list length.

We keep both implementations for AutoTuning purposes and to provide a less memory-intensive solution.

Zusammenfassung

Molekulardynamische Simulationen, die nur Zwei-Körper-Wechselwirkungen betrachten, erfassen nicht immer die korrekten qualitativen Veränderungen eines Systems. In solchen Fällen ist eine höhere Genauigkeit notwendig, die erreicht werden kann, indem zusätzlich Drei-Körper-Wechselwirkungen berücksichtigt werden. Zu diesem Zweck erweitern wir die Bibliothek für molekular Simulationen AutoPas um eine Implementierung von Verlet-Listen, die Drei-Körper-Wechselwirkungen simulieren kann.

Wir stellen drei verschiedene Ansätze auf, um alle Tripel für die Kraftberechnung abzuarbeiten. Der erste benutzt Zwei-Körper-Nachbarlisten und iteriert über alle Paare darin. Der zweite Ansatz bildet die Schnittmenge von Listen benachbarter Partikel, um alle gemeinsamen Nachbarn zu finden. Für den letzten speichern wir Drei-Körper-Nachbarlisten, welche alle Paare von Partikeln enthalten, die nah genug für die Kraftberechnung sind. Dann können wir einmal über diese iterieren, um alle Tripel zu erhalten.

Wir finden, dass der dritte Ansatz in Simulationen mit mittlerer bis hoher Dichte am schnellsten ist, während sich der erste Ansatz besser für Szenarien mit geringer Dichte eignet. Wir stellen auch fest, dass die Drei-Körper-Nachbarlisten viel mehr Speicher verbrauchen als die Zwei-Körper-Nachbarlisten, wobei die durchschnittliche Länge der Drei-Körper-Nachbarlisten proportional zum Quadrat der durchschnittlichen Länge der Zwei-Körper-Listen ist.

Wir behalten beide Implementierungen für Auto-Tuning-Zwecke bei und um eine weniger speicherintensive Lösung bereitzustellen.

Contents

Part I.

Introduction and Background

1. Introduction

Molecular dynamics simulations are a powerful tool in many different sciences, such as chemistry [1], biology [2], and physics [3]. They can be used to validate experimental findings, as well as extend these into scenarios for which physical experiments may be unsuited, due to for example very high pressure, temperature, or tiny substance volumes. The ability to visualize these simulations at a particle level may also grant new insights into the physical or chemical processes happening at an atomic or molecular level.

Molecular dynamics simulations became more widely used with the ever-increasing computational power offered by new CPUs and GPUs. But with higher computational power come higher requirements, such as bigger particle simulations or more accurate and thus computationally expensive simulations. To meet these challenges and demands there exist many different simulators which implement different algorithms and approaches to molecular simulations. However, there does not exist one optimal algorithm for all simulations [4], meaning that we can waste a lot of time and energy if we choose badly.

This is where AutoPas wants to simplify things for domain scientists, or others, new to molecular dynamics simulations by choosing the algorithm based on how well it performs in the current simulation scenario. This automatic algorithm selection is a special case of auto-tuning and it removes the need to have in-depth knowledge of the many simulation algorithms with their advantages and disadvantages.

At the moment AutoPas already has a wide selection of algorithms for 2-body simulations, which are introduced in [5]. Since the restriction to 2-body interactions is sometimes not accurate enough for the topic of the simulation [6], we are starting to adapt these algorithms to support 3-body interactions as well. This leads to the goal of this thesis, adapting the Verlet list algorithm for 3-body interactions.

In this thesis, we give an introduction to the theory behind N-Body simulations, as well as some algorithmic optimizations to it, and introduce the basics of AutoPas in Chapter 2. We give an overview of other established molecular dynamics simulators in Chapter 3. Then we introduce our three approaches and implementations of the 3-body neighbor identification algorithms in Chapter 4 and evaluate them based on runtime, parallel efficiency, and memory consumption in Chapter 5.

2. Background

2.1. N-Body Simulations

2.1.1. Theoretical Basis

N-Body simulations deal with a system of many particles that all interact with one another. We employ these simulations to get an understanding of the changes that occur in the system over time. To this end the simulation advances in discrete time steps of length δt for which every body's position $\vec{x}$ and velocity v are updated. The interactions between all N bodies can be described by the equations of motion [7]:

$$F_i = m_i a_i, \text{ or } F_i = \frac{\partial U}{\partial x} \tag{2.1}$$

$$a_i = \frac{\partial v_i}{\partial t} \tag{2.2}$$

F is the force acting on a body, m is its mass, a is its acceleration, v is its velocity, and U is the potential of the system. If we know the potential of the system, we can derive the acceleration and from there the velocity of bodies with these equations. A popular algorithm used for simulation is the velocity Verlet algorithm in Equation 2.3. This version was adapted from [7]:

$$v_i(t + \frac{1}{2}\delta t) = v_i(t) + \frac{1}{2}\delta t F_i(t)/m_i \tag{2.3a}$$

$$x_i(t + \delta t) = x_i + \delta t v_i(t + \frac{1}{2}\delta t) \tag{2.3b}$$

$$v_i(t + \delta t) = v_i(t + \frac{1}{2}\delta t) + \frac{1}{2}\delta t F_i(t + \delta t)/m_i \tag{2.3c}$$

Here we make a force calculation for every time step and update the velocity and position of all particles accordingly. The only thing we are missing now is how to compute the total potential of the system U. It can be written as a sum of potentials between all possible constellations of bodies [7].

$$U = \sum_i \sum_{j>i} u_2(\vec{x}_i, \vec{x}_j) + \sum_i \sum_{j>i} \sum_{k>j} u_3(\vec{x}_i, \vec{x}_j, \vec{x}_k) + \dots \tag{2.4}$$

The u_m are m-body potentials that act between sets of m bodies. From this, we can get the total force acting on a specific body i by derivation, which results in the sum of the forces derived from the potentials between i and all constellations of other bodies, which is shown in Equation 2.5. The forces $F_{i,j,k}$ are the forces that the pair of bodies j and k enact on body i. Computing the potential exactly is very computationally expensive so we approximate it instead by stopping the calculation after some term and do not consider higher-order potentials. We can categorize simulations by the highest degree of potential they consider in the force calculation. Most commonly used are 2-body interactions which only consider forces between pairs of bodies. This thesis is focused on interactions of degree three where we also consider interactions between triplets of bodies.

$$F_i = \sum_{j \neq i} F_{i,j} + \sum_{j \neq i} \sum_{k > j, k \neq i} F_{i,j,k} + \dots \tag{2.5}$$

2.1.2. Algorithmic Optimizations

Even with this simplification, we are still left with the computationally intensive task of computing these potentials for all pairs or triplets of particles. This approach would require $\mathcal{O}(N^2)$ or $\mathcal{O}(N^3)$ potential computations. So we introduce another tool to reduce the number of needed computations further.

There exist so-called short-range potentials which quickly converge to zero the further apart particles are. If we use these potentials we can introduce a cutoff radius r_{cut}. We then simply consider the forces induced by bodies outside this radius to be zero. For our purposes, all three bodies of the triplet have to be in each other's cutoff sphere for the triplet to be considered. This is visualized in Figure 2.1 b). This allows us to significantly reduce the number of potential calculations necessary since we only have to consider bodies inside the cutoff sphere.

We can express the average number of bodies in the cutoff sphere as $M = \rho * \frac{4}{3}\pi r_{cut}^3$ where ρ is the density of bodies. Thus the total number of potential calculations is in $\mathcal{O}(N * M)$ for the 2-body case and $\mathcal{O}(N * M^2)$ for the 3-body case. Since M depends on the density it in turn depends on the total number of particles N, which is why we still keep this factor in the big $\mathcal{O}$ notation, but under normal circumstances M is much smaller than N. In the 2-body case, we get all particle pairs by considering a base particle and all particles in its cutoff sphere. In the 3-body case, we then have to consider all pairs in a base particle's sphere.

Another optimization that is frequently employed is making use of Newton's third law of motion. This law states that any force has an equally strong opposing force [8]. Equation 2.6 shows how this law is used for 2- and 3-body interactions to compute one force out of the others. This saves us one potential calculation for every pair or triplet of bodies, but it also makes parallelization more difficult as we are updating the forces for groups of particles instead of single particles. We will refer to this optimization as Newton3 in this thesis.

$$\text{2-Body interactions: } F_{i,j} = -F_{j,i} \tag{2.6}$$

$$\text{3-Body interactions: } F_{i,j,k} = -(F_{k,i,j} + F_{j,i,k}) \tag{2.7}$$

This leaves us with the main challenge of N-body simulations, which is to find close neighboring bodies in an efficient and parallelizable way. In this thesis, we will be focusing on single atoms as the bodies in the simulation, which we will refer to as particles from here on out.

2.2. Axilrod Teller Potential

The Axilrod Teller potential is a short-range 3-body potential that approximates a part of the van der Waals attraction between atoms and is used in conjunction with 2-body potentials to boost the overall accuracy of the simulation. The van der Waals attraction is approximated through a quantum mechanical theory called the perturbation theory [9]. The resulting 3-body potential is given in Equation 2.8.

$$U(\vec{x}_i, \vec{x}_j, \vec{x}_k) = \mu \frac{3 \cos\gamma_i \cos\gamma_j \cos\gamma_k + 1}{r_{ij}^3 r_{jk}^3 r_{ki}^3} \qquad [9] \tag{2.8}$$

Here r_{ij} denotes the length of the vector from particle i to j which is given by $|\vec{r_{ij}}| = |\vec{x_j} - \vec{x_i}|$. Using the equality of Equation 2.9 where ϕ is the angle between vectors $\vec{a}$ and $\vec{b}$ we can rewrite the Axilrod Teller potential to match Equation 2.10, avoiding the expensive computation of the cosine function:

$$\cos\phi = \frac{\vec{a} \cdot \vec{b}}{|\vec{a}||\vec{b}|} \qquad [10], p.6 \tag{2.9}$$

$$U(\vec{x}_i, \vec{x}_j, \vec{x}_k) = \mu \frac{3(\vec{r}_{ij} \cdot \vec{r}_{ik})(\vec{r}_{ji} \cdot \vec{r}_{jk})(\vec{r}_{ki} \cdot \vec{r}_{kj}) + r_{ij}^2 r_{jk}^2 r_{ki}^2}{r_{ij}^5 r_{jk}^5 r_{ki}^5} \tag{2.10}$$

2.3. 3-Body vs 2-Body Interactions

3-Body interactions are a lot more computationally expensive as compared to 2-body interactions. The reason for this is twofold. Firstly, 3-body potentials are more computationally expensive, as their evaluation requires more floating point operations than 2-body potentials. Secondly, as we consider all triplets of particles, we have more potentials to compute. This number grows linearly in the density of particles and the volume of the cutoff sphere in the 2-body case but quadratically in the 3-body case.

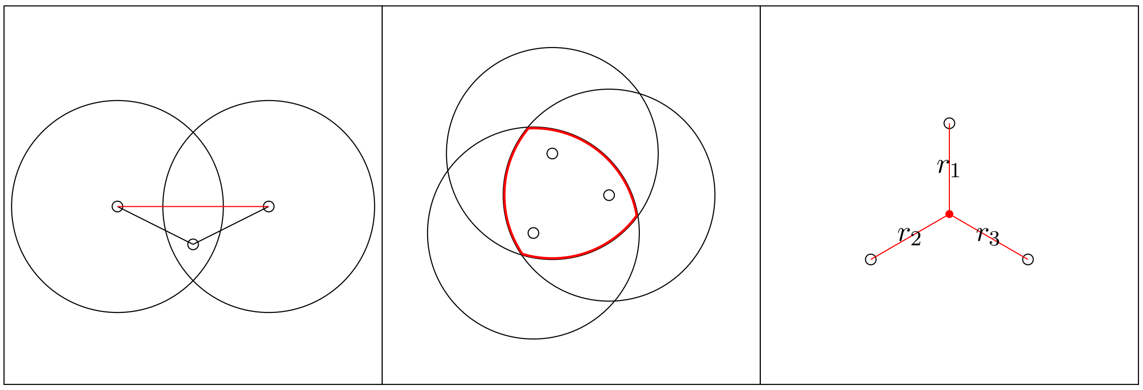
Many applications of N-body simulations work fine with 2-body interactions, but others need the higher accuracy that 3-body interactions provide. This is the case when the qualitative outcome of the simulation changes between 2-body and 3-body interactions. Ströker discusses that it is necessary to consider 3-body and even higher-order interactions to describe the behavior of materials over large temperature and pressure ranges with high accuracy [6].

Another difference between 3-body and 2-body interactions can be observed in the choice of cutoff criterion. In particular, there is no choice in the 2-body case, as there only exists one distance that could be used. For 3-body interactions on the other hand multiple possibilities have been considered and used:

1. at least two distances have to be less than r_{cut} for a triplet to contribute to the overall potential [11]. As shown in Figure 2.1 a), the two black distances are less than the cutoff, while the red distance is larger than the cutoff.

2. all three distances have to be less than r_{cut} for the triplet to be considered [12]. As shown in Figure 2.1 b), the third particle has to be inside the intersection of the cutoff spheres of the other two particles. The red border marks the overlap of all three cutoff spheres in which all particles must lie to form an eligible triplet.
3. the sum of distances of the particles to their center of mass has to be less than r_{cut} [13]. As shown in Figure 2.1 c), the sum of r_1, r_2, and r_3 has to be less than the cutoff. The center of mass is shown by the small red dot.

These cutoff criteria get more restrictive in the number of triplets they consider. The current standard in AutoPas is the second criterion, which we will therefore be using.



a) Pair cutoff b) Triplet cutoff c) Center of Mass cutoff

Figure 2.1.: Visualizing the three cutoff criteria for 3-body interactions.

2.4. AutoPas

AutoPas is a library for N-body simulations that has the goal of finding the best algorithm for the current situation in a simulation through the use of auto-tuning. This should remove the need for domain experts to also know the simulation side of molecular dynamics to be able to select optimal algorithms for the simulation [5]. Additionally, this can make simulations of rapidly changing systems faster, since there does not exist any one best-performing algorithm for all possible situations. So having the ability to adapt the algorithm mid-simulation can be profitable [14], [15].

AutoPas requires users to implement the particles they want to simulate and the functor that can be used to calculate forces between pairs or triplets of particles. Given these two, AutoPas handles the force calculation and auto-tuning itself. For this, it has different containers which act as storage for the particles. These containers are built in a Verlet-like manner, meaning they keep a valid state for several iterations before they have to be rebuilt in some way. The different containers make the particles accessible for the different traversals in a way that is practical for that traversal. The 3-body traversals implement one function `iterateTriwise()` which enumerates all particle triplets eligible for force calculation and calls the user's force functor to compute all forces in a single time step. So the traversals constitute the neighbor identification algorithms used in the simulation. The `LogicHandler` is responsible for the validity of containers, issuing a rebuild if necessary. Lastly, AutoPas has an `AutoTuner` which takes care of the container and traversal selection.

AutoPas is designed in a way that allows the user to split the entire simulation domain into different subdomains that can be taken care of on different compute nodes. For this we have to duplicate particles near the border and have them available to both subdomains, otherwise, we would lose cross-domain interactions. These duplicated particles are called Halo particles in their non-native subdomain. Halo particles only serve as interaction partners, but never get force updates themselves [5]. This avoids doing the force calculations on them twice.

A single time step of the simulation can be divided into two phases. In the first, we update particle positions or movement from the forces computed and exchange Halo particles between multiple subdomains if applicable. This is left up to the user of AutoPas to do in every iteration. We also check the validity of the underlying container, triggering a rebuild if necessary. This is done by AutoPas and only has to be triggered by the user. The second phase is the force calculation in which the `iterateTriwise()` method is called.

We now introduce the two main containers that form the basis of this Thesis, and on which the neighbor identification algorithms work.

2.4.1. Linked Cells

The Linked Cells [16] approach divides the entire simulation domain into a grid of cubes with side lengths r_{cut}. This way the cutoff sphere of a particle in a particular cell will be fully contained in the cube made up of the neighboring cells. We majorly cut down the number of distance calculations necessary to find all triplets akin to the potential calculations as in Subsection 2.1.2. We again reduce the number of distance calculations from $\mathcal{O}(N^3)$ to $\mathcal{O}(N * M^2)$ in the 3-body case, where $M = \rho * r_{cut}^3$.

In AutoPas every grid cell has a list of all the particles inside it. This means that we can design parallel algorithms working on entire cells, which take advantage of the regular grid structure to avoid data races when accessing cells. This will then also guarantee that there exist no data races on the level of single particles during the traversal. A disadvantage of this basic implementation of Linked Cells is that the approximation of the cutoff sphere via a cube is not great, so we end up doing many superfluous distance calculations. This is visualized in Figure 2.4 on the left as a two-dimensional example. To judge how many superfluous distance calculations are made we can compare the hit rate of different approaches which is the ratio of actual force calculations over the total amount of distance calculations made in one iteration. This way a higher hit rate means less unnecessary distance computations.

There is already one 3-body Linked Cells traversal implemented in AutoPas. It is the lcc01 traversal. This traversal has a base step in which it takes a cell as its base cell for which it does force updates. The base step consists of three steps to find all triplets of close particles for the force calculation, which are visualized in Figure 2.2. First, it finds all triplets in the base cell itself, marked in blue. Then it has to consider triplets that span two cells, so it pairs the base cell and every surrounding cell up to look for triplets, one such pair is given by the blue base cell and the red cell. Lastly, it also has to consider triplets that span three cells, so it looks for triplets in all triplets of cells, where one is the blue base cell and the other two are neighboring cells from the surrounding cells, marked orange.

Since every base step only updates the forces of particles in the base cell, we can run multiple base steps in parallel without having to worry about data races on force updates.

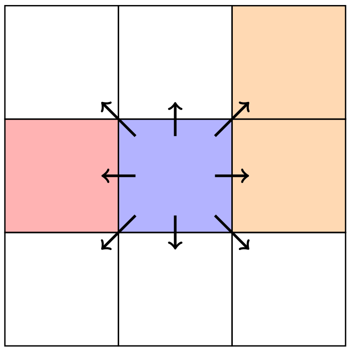


Figure 2.2.: Base step of the lcc01 traversal, blue=Base Cell, red=single Cell in triplet search, orange=pair of touching cells for triplet search, inspired by [5].

2.4.2. Verlet Lists

Verlet lists, first mentioned in [17], are a data structure that stores close pairs of particles, for which a force calculation has to be computed. Specifically, it stores for every particle a list of particles nearby, often called neighbors, which are in a cutoff sphere around the particle. This way we only have to traverse these lists to get all interacting particle pairs in an iteration. The building of such lists is quite expensive. Naively it takes $\mathcal{O}(N^2)$ distance calculations which can be reduced to $\mathcal{O}(N * M)$ with the help of Linked Cells which were discussed in Subsection 2.4.1. To compensate for this costly rebuilding step Verlet lists do not use the force cutoff radius for the cutoff sphere but add a small additional radius to it which is called the Verlet skin r_{skin} [17]. This allows us to rebuild the neighbor lists every couple of iterations when there is a chance for a particle to have crossed the entirety of the Verlet skin region.

Typically the skin radius is set between $0.05 * r_{cut}$ and $0.2 * r_{cut}$ [18]. There is a clear tradeoff for bigger skin radii. The larger r_{skin} becomes the less often we have to rebuild the Verlet lists. On the other hand, we increase the amount of superfluous distance calculations during a single iteration, as all particles in the skin still do not contribute towards the forces. For 2-body interactions and these skin radii the number of unnecessary calculations lies between 13.6% and 42.1% [18].

Figure 2.3 shows how Verlet lists are implemented in AutoPas. On the left is a global hash table that stores the neighbor list of every particle. The neighbor lists themselves then contain pointers to the neighboring particles, avoiding needless duplication of particles, since particles appear in multiple neighbor lists. Verlet lists are built on top of Linked Cells in

AutoPas, so the actual particles are stored in the Linked Cells container and the Verlet list container only works on pointers to these particles. The neighbor lists themselves are built with the help of Linked Cells traversals.

With this, we can already see what advantages and disadvantages Verlet lists have. They possess a higher hit rate than the previously discussed Linked Cells. The right half of Figure 2.4 shows the area, in which particles contribute to the force calculation, and the area, where particles are considered for the neighbor identification algorithm, for Verlet lists. The ratio of these areas is much higher for Verlet lists than for Linked Cells. As a downside, Verlet lists require significantly more memory to store all of the neighbor lists and they introduce a layer of indirection when accessing particles for force calculations. At the same time, we do not preserve any spatial information of where which particle lies in our simulation domain since we are using a hash map to access the particles and their neighbor lists. This can make the design of parallel algorithms much harder.

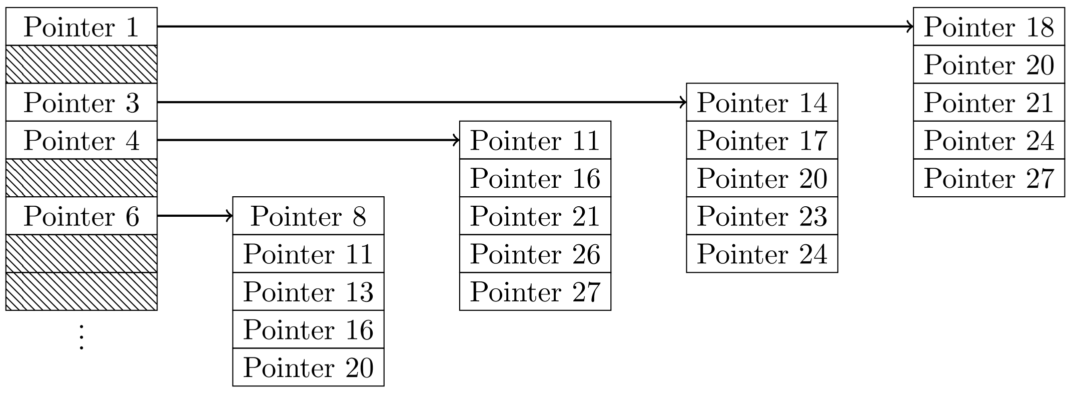


Figure 2.3.: Global Verlet list, Hashmap with particle pointers as keys, storing lists of particle pointers as values.

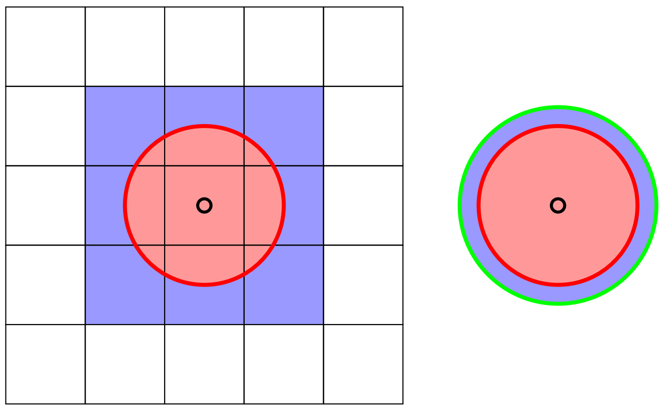


Figure 2.4.: Visualizing the area of superfluous distance calculations (blue) and area of force calculations (red) for Linked Cells (left) and Verlet lists (right), shows that Verlet lists do less superfluous distance calculations, resulting in a higher hit rate.

3. Related Work

3.1. LAMMPS

LAMMPS[1] is a popular molecular dynamics simulator designed to run on large parallel systems with many compute nodes through the use of domain decomposition [19]. It has a variety of both CPU and GPU accelerators for high-performance simulations. LAMMPS also provides a wide variety of other features including many different potentials and model options to choose from. It also supports 3-body interactions[2] and implements the Axilrod Teller potential as well as other 3-body potentials. The neighbor identification algorithms of LAMMPS exclusively use Verlet lists. The approach used to compute all triplet interactions in the 3-body case iterates over all particles and for each it iterates over the neighbor list in a double for loop to enumerate all pairs of neighbors of the particle. The ListIteration approach we implemented works the same way and is introduced in Section 4.2.

3.2. DL_POLY

DL_POLY[3] is a molecular dynamics simulator designed for high-performance computers that is based on Linked Cells [20]. To provide an efficient implementation it again implements domain decomposition and uses MPI for communication between CPUs. In contrast to AutoPas DL_POLY does not implement shared memory parallelization. It is written in Fortran and implements some 3-body potentials, such as the Tersoff potential [21] and valence angle potentials[4]. It even implements 4-body interactions for the valence angle potentials. The many-body potentials are also computed via Linked Cells.

3.3. HOOMD-blue

HOOMD-blue[5] is specialized in doing nano- and colloidal-scale molecular dynamics simulations and implements a Python interface [22]. A primary focus of HOOMD-blue is GPU acceleration with the use of CUDA. It uses Verlet lists for nano-scale simulations and also implements some 3-body potentials[6], such as the Rev-Cross potential described by Ciarella and Ellenbroek in [23]. To iterate all particle triplets they use the same approach as LAMMPS. It also implements domain decomposition to run simulations on multiple CPUs or GPUs at once.

[1] https://www.lammps.org/index.html
[2] LAMMPS doc for 3-body potential commands: https://docs.lammps.org/pairs.html
[3] https://www.scd.stfc.ac.uk/Pages/DL_POLY.aspx
[4] DL_POLY user manual: https://www.ehu.eus/sgi/ARCHIVOS/dlpoly_man.pdf
[5] https://glotzerlab.engin.umich.edu/hoomd-blue/
[6] HOOMD-blue many-body potentials:
https://hoomd-blue.readthedocs.io/en/latest/module-md-many_body.html

4. Implementation

4.1. Overview

This thesis contributes multiple 3-body traversals for the `Verlet List` container, which can be seen in Figure 4.1 and will be explained in the following sections. All of these traversals are derived from the TriwiseTraversalInterface and implement the central function `traverseParticleTriplets()`. This is the function that is called in `iterateTriwise()` to compute all particle interactions. We can split the new traversals into two groups. The first comprises the ListIteration3B, ListIntersectionSorted3B, and ListIntersectionHashing3B traversals. These use 2-body neighbor lists to find all 3-body interaction triplets. The other group only contains the PairListIteration3B which uses 3-body neighbor lists that store all 3-body interaction partners.

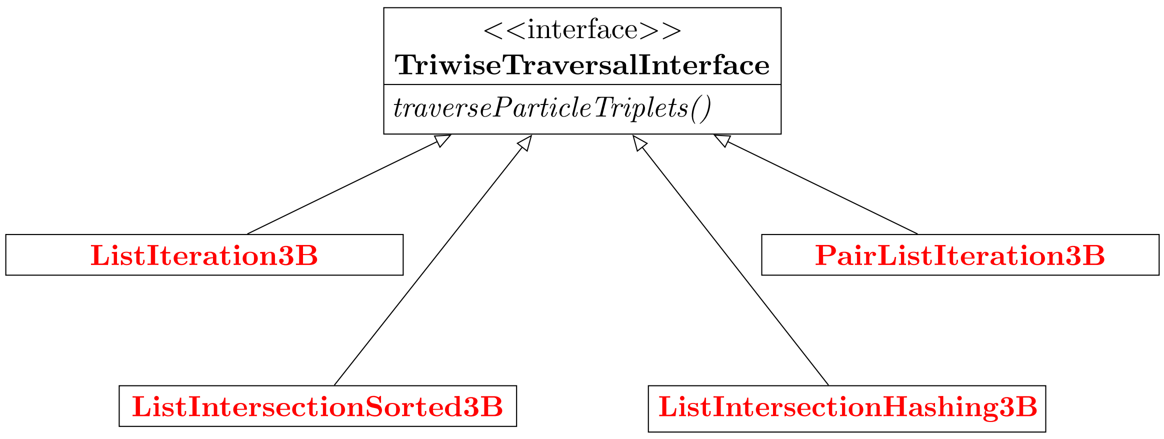
Our traversals build on the already implemented container in AutoPas, which takes care of particle management. The main changes we had to make to the container were to add a `rebuildNeighborLists(TriwiseTraversal)` function that works with triwise traversals instead of pairwise ones.

All of the traversals have been implemented to work without Newton3 and only the ListIteration3B traversal works with Newton3 enabled. All non-Newton3 traversals have been parallelized using OpenMP, while the Newton3 version is not parallelized. The parallelization of the Newton3 approach requires extensive use of particle locks to ensure thread safety, for which there was no time in this project's scope.

In Figure 4.2 we see the two other necessary contributions we made. For one we created a new VerletListGeneratorFunctor which can be used with any 3-body traversal to build 3-body neighbor lists. Lastly, we contributed a 2-body Linked Cells traversal which is used to build neighbor lists for the 3-body traversals working with 2-body neighbor lists.

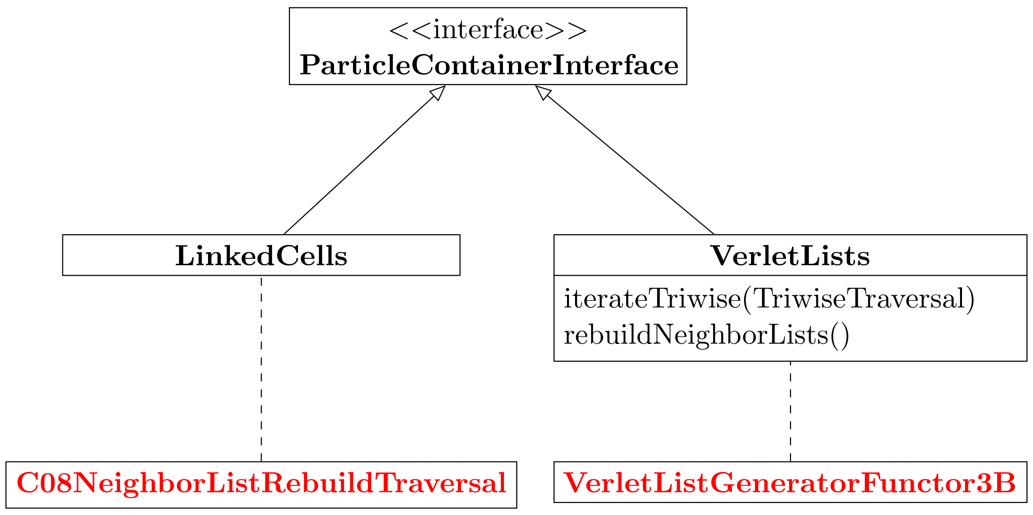
4.2. Iteration Approach

Our first approach is an adaptation of the 2-body traversal currently used in AutoPas. This traversal is called the ListIteration traversal in the 2-body case, which iterates once over every neighbor list to find all pairs of particles. The 3-body version is called ListIteration3B, but we will focus on the 3-body traversal and call it ListIteration from here on out. For the 3-body traversal, we use the same neighbor lists as in the 2-body case. We iterate over all particles and for each we iterate over its neighbor list in a double for-loop instead of a single for-loop to find all pairs of neighbor particles in the neighbor list. We can then call the force functor with the particle and both neighbors to execute a force calculation for the



<<interface>>
TriwiseTraversalInterface
traverseParticleTriplets()
ListIteration3B
PairListIteration3B
ListIntersectionSorted3B
ListIntersectionHashing3B

Figure 4.1.: Overview of contributed traversals, new traversals in red.



<<interface>>
ParticleContainerInterface
LinkedCells
VerletLists
iterateTriwise(TriwiseTraversal)
rebuildNeighborLists()
C08NeighborListRebuildTraversal
VerletListGeneratorFunctor3B

Figure 4.2.: Structural overview of AutoPas containers with our contributions highlighted in red.

Algorithm 1: ListIteration

```
1 Function iterateTriwise():
2     for $i \in Particles$ do
3         for $j \in i.neighborList$ do
4             for $k > j,\ k \in i.neighborList$ do
5                 AxilrodTeller($i$, $j$, $k$)
```

Figure 4.3.: Pseudocode of `iterateTriwise()` for the ListIteration approach.

particle. The pseudocode is given in Algorithm 1. This algorithm is very simple but also has a lower hit rate, as many of the pairs in a neighbor list are of particles on different sides of the cutoff sphere and as such are too far apart to contribute to the force calculation.

For the Newton3 case special care has to be taken when constructing the 2-body neighbor lists. The functor for building neighbor lists only adds the neighbor relation to one list. Without the Newton3 optimization the functor is called twice for each particle pair and the neighbor relation is saved in both lists. With the optimization the functor is called only once per particle pair, meaning the neighbor relation is saved in only one list. This makes sure that the neighbor relation is asymmetric in the Newton3 case. So we can simply iterate over the resulting lists without having to worry if force calculations have already been done for a pair of neighbors.

The Linked Cells traversal used for the building process is called the lcc08 traversal. It is comprised of a base step, shown in Figure 4.4, which computes all possible interactions between the cells connected by the arrows. We then iterate over the cells in the order shown, treating every cell as a base cell once. This way we compute all possible interactions for every cell. In three dimensions we can color the domain in eight different colors, such that every color is a set of cells on which the base step can be performed simultaneously, without leading to race conditions. The traversal is optimized to avoid unnecessary force calculations, so it stops after the last owned cell, cell 11 in Figure 4.4. This avoids force calculations solely between Halo cells full of Halo particles. But this can be a problem for the neighbor lists in the Newton3 case of our 3-body traversal.

Figure 4.4 specifically shows an edge case where this becomes a problem. In the last row of owned cells, the neighbor relation of particles in the owned Cell 10 to particles in Halo Cell 13 may only be saved in the neighbor lists of particles in Halo Cell 13, because of the functor for neighbor list building. As the lcc08 traversal stops after this row, the neighbor relations between Halo particles in Cells 13 and 14 are never computed and saved. Particles of Cell 13 only have neighbor entries for particles of Cell 10, Cell 10 has neighbor entries for particles of Cell 14 and Cell 14 has no neighbor entries for either Cell 10 or 13. So when iterating over the neighbor list of any particle we can never get a triplet that contains particles of all three Cells.

A possible fix would be to introduce a policy of saving neighbor relations predominantly in owned Cells. However, since we use an existing 2-body traversal to construct the neighbor lists this may have unintended performance implications for the traversal. Instead, we implemented a Linked Cells traversal for building neighbor lists which also computes neighbor relations between all Halo Cells. This traversal will be further explained in Subsection 4.3.1.

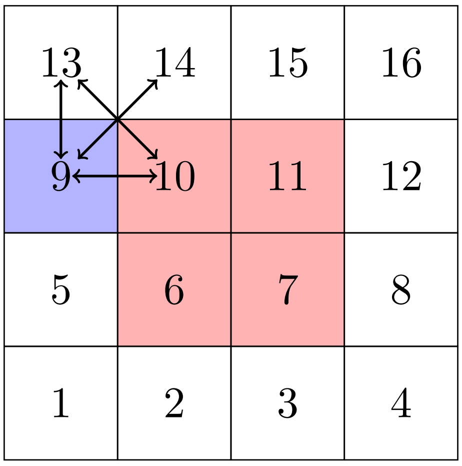


Figure 4.4.: Base step of the lcc08 traversal, white=Halo Cell, red=Owned Cell, blue=Base Halo Cell, inspired by [5].

This iteration approach faces the same complications for parallelization in the Newton3 case as all Verlet traversals. Since the global map of neighbor lists does not store any spacial information about the corresponding particles parallelization would have to rely on locking the entire particle during the addition of forces to avoid race conditions when updating the force on a particle. This should render these parallelization schemes quite inefficient. From Figure 4.5 it becomes clear that the Newton3 optimization is more than twice as fast in the single-threaded case, but without parallelization, it is slower than the non-Newton3 variant when we use more than three threads. So while a solution for the Newton3 case of the iteration approach has been implemented it will not be taken into consideration for the rest of this thesis.

4.3. Intersection Approach

Our second approach is to find triplets of particles by intersecting the 2-body neighbor lists of two neighboring particles to get their common neighbors. We do this in every iteration step. The simplified algorithm of this approach is shown in Algorithm 2. We have to be careful to not call the force functor for a particle triplet twice, so for the triplets (i, j, k) and (i, k, j). To avoid this we do not intersect the entire neighbor list of i with other neighbor lists, but only the remainder of the list, after the current particle j.

This algorithm has a higher iteration overhead than the ListIteration traversal, but it also has a higher hit rate, as particles in the intersection are in both lists and therefore very close. The only way that such a triplet will not contribute to the force calculation is if one particle falls into the skin zone of the Verlet cutoff sphere.

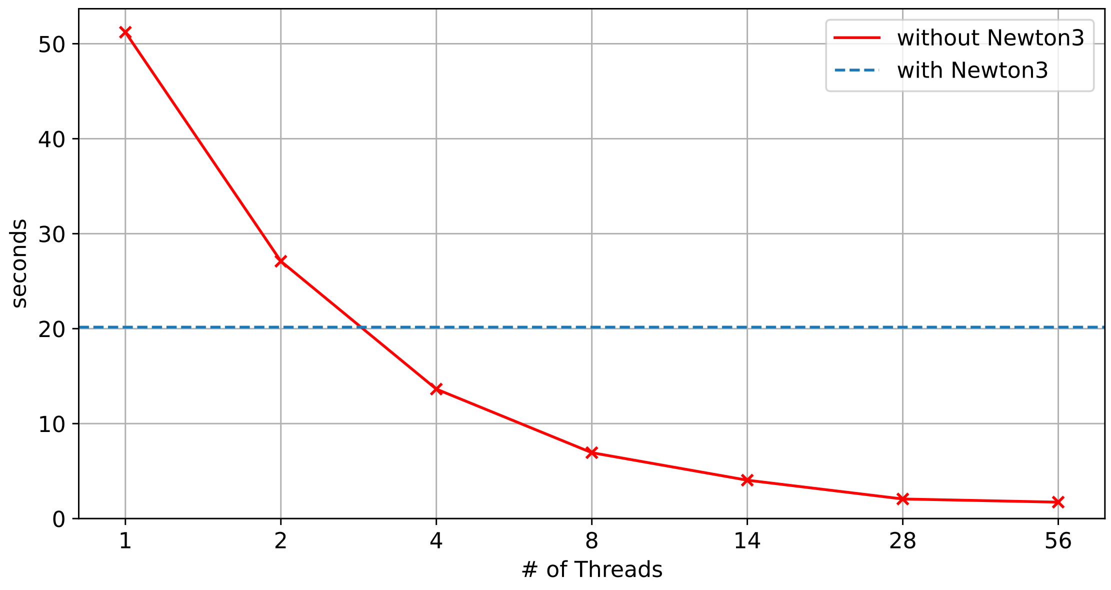


Figure 4.5.: Comparison of the median iteration time of the ListIteration approach with and without Newton3, shows that the parallel version without Newton3 vastly outperforms the single-threaded implementation with Newton3.

Algorithm 2: ListIntersection

```
1 Function iterateTriwise():
2     for $i \in Particles$ do
3         for $j \in i.neighborList$ do
4             intersection ← intersect($i.neighborList$, $j.neighborList$)
5             for $k \in intersection$ do
6                 AxilrodTeller($i$, $j$, $k$)
```

Figure 4.6.: Simplified version of `iterateTriwise()` for the ListIntersection approach.

4.3.1. Problems

This approach faces similar problems to the Newton3 case of the list iteration approach in Section 4.2. Since the Linked Cells traversal avoids force calculations between Halo particles, the neighbor relation between Halo particles is not saved in their neighbor lists. This means during the traversal when intersecting the neighbor list of an owned particle with one of a Halo particle other Halo particles will not be found as common neighbors. This results in the traversal missing 3-body interactions of the type (Owned, Halo, Halo).

As a solution to this, we created a new 2-body Linked Cells traversal solely for neighbor list building that does not avoid any force calculations between Halo particles. For this reason, we decided to make this an extra traversal since it can not apply the same Halo-avoiding optimizations.
The c08 traversal had to be adjusted in two ways. Firstly the iteration had to be extended to also execute the base step on the last row of cells 13 to 16. In turn we had to add a new bounds check, so that the base step on the last row of cells does not lead to out-of-bounds access. Secondly, a check avoiding force calculations between Halo Cells had to be removed from the cell functor.

4.3.2. Different Approaches for Neighbor List Intersection

Our first approach to the list intersection is the ListIntersectionSorted traversal which uses std::set_intersection()[1], to obtain the intersection of two neighbor lists. A prerequisite for this function is for the given ranges to be sorted by some criterion. We decided to sort the pointers in the neighbor lists by their address. We sort all neighbor lists once after every rebuild. After obtaining the intersection of two neighbor lists, we simply iterate over it and can call the force functor for the two particles and the one from their intersection.

After the first implementation, we went on to optimize the buffer used to store the intersection, by reusing the same thread-local buffer for all intersections, instead of allocating a new buffer for every intersection. With this approach, there exists the downside of having to sort the neighbor lists which may make this traversal slower.

We also tried to avoid the overhead from sorting the neighbor lists. So in a second approach, we created the ListIntersectionHashing traversal which uses a hash intersect.
We first create a hash set from a neighbor list. Then we can check if any particles of a neighboring neighbor list are in the hash set and call the force functor for this triplet.

The sorting approach comes with the overhead from sorting the lists, but the hashing approach leads to less cache efficiency, as it introduces a new degree of indirection through the hash set. In Figure 4.7 we compare the median iteration time of both approaches and we conclude that the hash intersection is not worthwhile, as it is about two times slower than the sorting approach. This means the additional layer of memory indirection is more impactful than the overhead from sorting.

[1] std::set_intersection() documentation: https://en.cppreference.com/w/cpp/algorithm/set_intersection

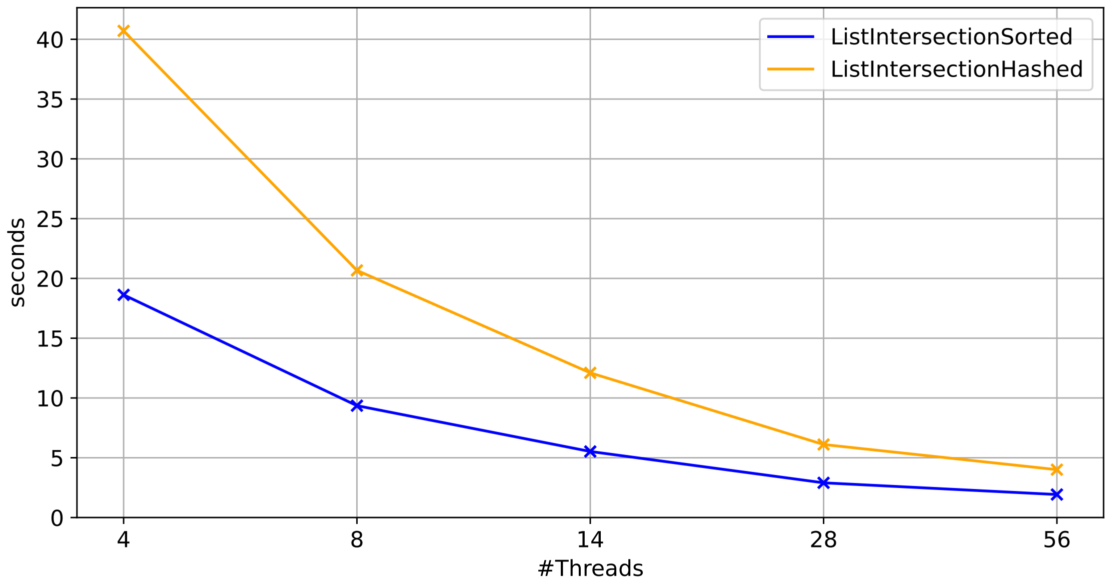


Figure 4.7.: Median `iterateTriwise()` time of ListIntersection sorting and hashing approach, shows that the hashing approach is about twice as slow as the sorted approach.

4.4. 3-Body Neighbor Lists

4.4.1. Algorithm and Implementation

As a third approach, we created the PairListIteration traversal which employs 3-body neighbor lists which already store the pairs of neighbor particles that are eligible for force calculations. This approach has the same high hit rate as the ListIntersection approach, with the only superfluous distance calculations being done for pairs where a particle is in the Verlet skin region.

For this, we created a new functor that creates these lists. This way we can use it in conjunction with any 3-body traversal to build these pairwise neighbor lists. Afterward, we only need to iterate over these lists once and call the force functor for the particle and all pairs in its neighbor list. The pseudocode of this traversal is shown in Algorithm 3.

At the start, we used the existing 3-body Linked Cells traversal lcc01 for neighbor list building. After profiling it became clear that the rebuilding of neighbor lists is a serious bottleneck for this traversal. So we switched to a 3-body Verlet list traversal to try and speed up the neighbor list building. For this, we choose the ListIteration traversal.

One clear downside of this traversal is that it needs more memory than all other solutions as we store all pairs of potential interaction partners instead of the single particles. When using a Verlet list traversal instead of a Linked Cells traversal we add the memory cost of normal Verlet lists on top of that.

Algorithm 3: PairListIteration

```
1 Function iterateTriwise():
2     for $i \in$ Particles do
3         for $(j, k) \in$ i.pairwiseNeighborList do
4             AxilrodTeller($i$, $j$, $k$)
```

Figure 4.8.: Pseudocode of `iterateTriwise()` for the PairListIteration approach.

4.4.2. 3-Body Neighbor List Model

We can model the expected length of the 3-body lists given the length of their respective 2-body lists. We can say a pair of particles is in the 3-body list if and only if both particles are in the respective 2-body list and they are less than $r_{cut} + r_{skin}$ far apart. For simplicity, we will assume that the particles are uniformly randomly distributed in the cutoff sphere of the list. We can then use the probability distribution for lengths between two uniformly random points in a sphere, derived in multiple ways in [24] and [25], and shown in Equation 4.1. $P(s)$ is the probability that two uniformly random points in a sphere of radius r are exactly s far apart for $0 \leq s \leq 2r$.

$$P(s) = 3\frac{s^2}{r^3} - \frac{9s^3}{4r^4} + \frac{3s^5}{16r^6} \tag{4.1}$$

Using this probability distribution we can find the probability that two random points are less than the radius of the sphere far apart.

$$\begin{aligned} P(s \leq r) &= \int_0^r P(s)\,ds && (4.2)\\ &= \int_0^r 3\frac{s^2}{r^3} - \frac{9s^3}{4r^4} + \frac{3s^5}{16r^6}\,ds && (4.3)\\ &= \left[\frac{s^3}{r^3} - \frac{9s^4}{16r^4} + \frac{s^6}{32r^6}\right]_0^r && (4.4)\\ &= \frac{15}{32} \approx 47\% && (4.5)\end{aligned}$$

With n particles in the 2-body neighbor list, we have $n(n-1)/2$ unique particle pairs, of which every 15 out of 32 should be inside the 3-body neighbor list. This means the 3-body neighbor lists contain about 47% of all particle pairs of the 2-body neighbor list. So they are quadratic in size compared to their 2-body counterparts, as one might expect. This means both the average 3-body list length and memory requirement are bigger than their 2-body counterpart by a factor that is proportional to the average 2-body list length. This means that 3-body lists are significantly more expensive than 2-body lists the denser the simulated particles are because the 2-body lists get longer.

Comparing the model to the actual lists, we found that this theoretical model seems to slightly underestimate the size of lists and thereby the total memory consumption by about ten percent, as can be seen in Figure 4.9. We think that one factor contributing to this is that we underestimate the length of longer lists more than we overestimate the length of shorter lists, because of the quadratic scaling with list length. We also see that the use of vectors for the neighbor lists means that we have allocated but unused memory, which adds another factor of around 1.5 to the memory requirement.

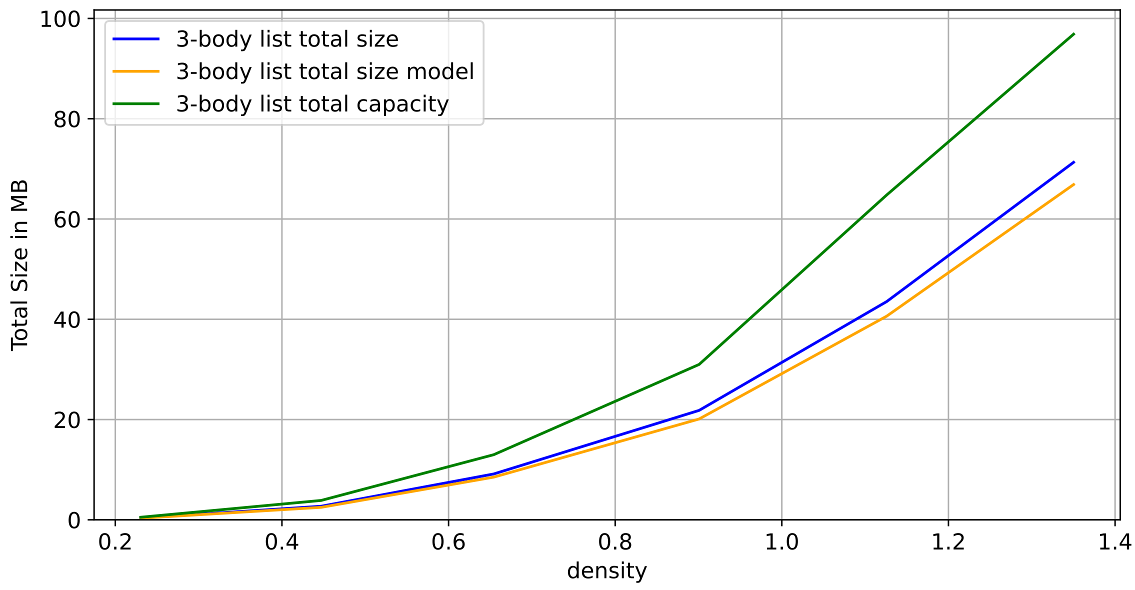


Figure 4.9.: Memory requirement of 3-body neighbor lists, used memory in blue, allocated memory in green, and the predictions of the model (orange). This shows that our model underestimates the memory requirement slightly and that the total capacity is about 1.5 times as much as the used memory.

4.5. Theoretical Running Time Comparison

In this section, we want to compare the theoretical running times of the iteration steps and neighbor list rebuilds of the different traversals.

4.5.1. Running Time of IterateTriwise

For simplicity, we write the running times $T(N)$ in terms of the mean 2-body neighbor list length n and assume that every list is of exactly that size. We will denote the number of particles in the simulation by N. n in some ways depends on N, but akin to M in Subsection 2.1.2 n is generally much smaller than N.

The ListIteration traversal iterates over all pairs in every list exactly once, therefore its running time is $T(N) = N * \frac{n*(n-1)}{2}$. For the PairListIteration traversal, we iterate once over the 3-body neighbor lists making its runtime $T(N) = N * z$, if z is the average 3-body list length. We can use our model from Subsection 4.4.2 to approximate this length as $\frac{15}{32} * \frac{n*(n-1)}{2}$ resulting in a runtime of $T(N) \approx N * \frac{n*(n-1)}{4}$.

The ListIntersection approaches work a bit differently. They make an intersection and then iterate over the intersection for every entry in every particle's neighbor list. This gives us a runtime of $T(N) = N * n * (T(intersect) + i)$ where $T(intersect)$ is the time the intersection takes and i is the size of the intersection. In the worst case, i can be as big as n and the runtime of the intersection depends on the approach. For the sorting approach, the worst-case running time of the intersect is given by the length of both lists, so it is $2n$. For the hashing approach, the running time is given by the length of the second list, so it is n, but we have to build the hash set for every particle once, which takes time n per particle. The final running times are $T(N) = N * n * (2n + n) = N * 3n^2$ for the sorted version and $T(N) = N * [n * (n + n) + n] = N * (2n^2 + n)$ for the hashing version.

We see in Table 4.1 that all traversals are in the same complexity class, taking $\mathcal{O}(N * n^2)$ time. The fastest traversal is the PairListIteration. The slowest are the ListIntersection approaches, but they have a higher hit rate than the ListIteration traversal at least. We also made a worst-case analysis for them for both the size of the intersection, as well as the runtime of the intersection procedure, so the average case should perform better than this analysis shows. We also already showed that the hashing approach performs worse than the sorting approach in Figure 4.7, even though it should have a better runtime, with the reason most likely being the additional layer of memory indirection incurred by the hashing approach.

Traversal	Running time `iterateTriwise()`	Running time `rebuildNeighborLists()`
ListIteration	$N * \frac{n*(n-1)}{2}$	$N * M$
ListIntersectionSorted	$N * 3n^2$	$N * M + N * (n * log(n))$
ListIntersectionHashing	$N * (2n^2 + n)$	$N * M$
PairListIteration	$N * \frac{n*(n-1)}{4}$	$N * M + N * \frac{n*(n-1)}{2}$

Table 4.1.: Theoretical running times of `iterateTriwise()` and `rebuildNeighborLists()` for the different traversals.

4.5.2. Running Time of RebuildNeighborLists

The running times of the rebuild of the ListIteration traversal and the hashing approach for the ListIntersection approach are the same, given by the runtime of the lcc08 traversal. Its runtime depends on the average amount of particles per cell $M = \rho * r_{cut}^3$, so something like $T(N) = N * M$. For the sorted ListIntersection approach we have to sort all lists, which results in a runtime of $T(N) = N * M + N * (n * log(n))$.

The list rebuild time for the PairListIteration traversal depends on the 3-body traversal used to build those lists. In our case, we use the ListIteration traversal, for which we also have to rebuild the 2-body neighbor lists. This results in a running time of $T(N) = N * M + N * \frac{n*(n-1)}{2}$.

From Table 4.1 we can see that the 3-body neighbor list rebuild takes substantially more time than the other rebuilds, but since it also has the theoretically fastest iteration step this difference may be made up over the iterations without a rebuild. If we compare it to the ListIteration traversal over three iterations, with one list rebuild we get running times of $T(N) = N * M + \frac{5}{4} * N * n * (n-1)$ for the PairListIteration and of $T(N) = N * M + \frac{3}{2} * N * n * (n-1)$ for the ListIteration traversal. So the PairListIteration traversal would come out ahead when looking at multiple iterations.

5. Evaluation

5.1. Test Environment

We ran most of our experiments with the different traversals under different conditions on the CoolMUC-2[1] of the Leibniz-Rechenzentrum[2] (LRZ). We used a single Haswell-based node with two Intel Xeon E5-2697 v3 cores, giving us 28 physical cores capable of two-way hyperthreading to run our simulation scenarios. The cores operate at a frequency of 2.6 GHz and have 64 GB of RAM.
We also ran one experiment on a node of the HSUper cluster which provides 2 Intel Xeon Platinum 8360Y processors, which gives a total of 72 cores that are split over the two processors and 4 NUMA nodes. Their nominal frequency is 2.4GHz and they also provide two-way hyperthreading. The node has 256GB of memory available.

We used the exemplary particle simulator, called md-flexible, that comes with AutoPas. In all the scenarios we run, the force calculation cutoff radius is set to 2.5 with an additional skin radius of 0.2. We always rebuild the neighbor lists every 10 iterations and simulate 100 iterations in total. The main scenario that we ran for data collection for running times and parallel efficiency uses the cube closest packing generator of md-flexible. This generator fills the simulation domain with particles as in the hexagonal closest packing (HCP) of spheres in a cube. The spacing between particles, which is equivalent to 2r in the HCP, is set to 1.2, which results in a particle density of about 0.83. We ran this scenario, which we call CubePack at two scales by varying the size of the simulation domain. Their particle numbers and particle densities are shown in Table 5.1.

In the following sections, we are comparing the raw running times of the different traversals as well as their parallel efficiency. In Section 5.5 we analyze the memory consumption of the neighbor lists used by different traversals and in Section 5.6 we look at a few different scenarios to see if there are significant differences in the performance of a traversal depending on the simulation conditions. For this, we ran a scenario Uniform, which distributes particles uniformly randomly in the simulation domain. We fix the number of particles and vary the domain volume to get scenarios with different particle densities. We also take a look at the Gauss scenario, which uses a Gaussian distribution to generate particles in the simulation domain. Here we fix the simulation domain and vary the standard deviation which gives us differently spread out particles throughout the domain. The particle numbers and particle densities for these distributions are given in Table 5.1 as ranges, with the exact number depending on the parameters.

[1]https://doku.lrz.de/coolmuc-2-11484376.html
[2]https://www.lrz.de

Scenario name	Owned Particles	Halo Particles	Domain Volume	Owned Particle Density
CubePack_M	$\sim$ 1 100 000	$\sim$ 150 000	1 331 000	$\sim$ 0.826
CubePack_T	$\sim$ 106 000	$\sim$ 35 000	125 000	$\sim$ 0.848
Uniform	512,000	[$\sim$ 44 300 to $\sim$ 82 500]	[681 472 to 4 096 000]	[$\sim$ 0.125 to $\sim$ 0.75]
Gauss	[317 811 to 606 949]	[38 909 to 51 287]	1 000 000	varies throughout domain

Table 5.1.: Statistics of the number of particles, domain volume, and owned particle density for different scenarios.

5.2. Comparison of Running Time

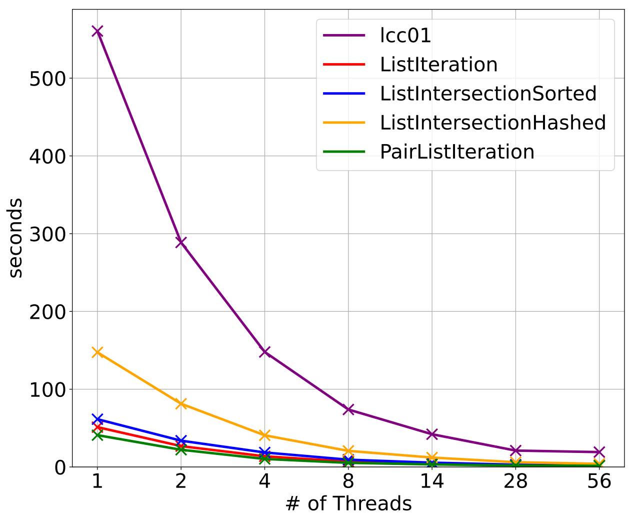
In this section, we will analyze the differences in running time between the different Verlet list traversals and the previously implemented Linked Cells traversal lcc01.

5.2.1. Total Running Time

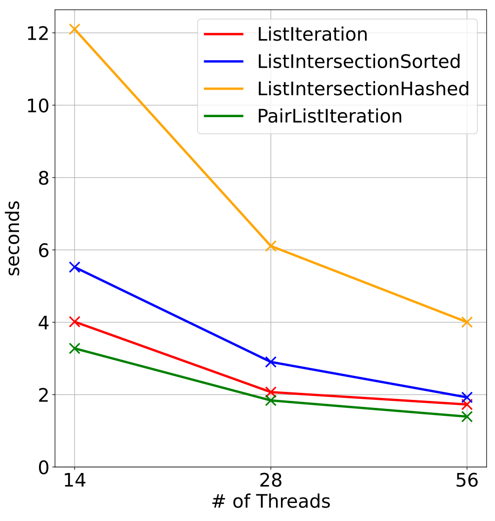
We first look at the total time one iteration of the simulation takes. Here we have to factor in that the rebuild iterations for Verlet lists will take significantly longer, as we have to rebuild the lists. Additionally, all prior cache entries will be unusable after a rebuild, as all neighbor lists have changed, so the iteration over all particle triplets will also take longer. Therefore we split our measurements into two sets, one made up of the rebuild iterations and the other of all other iterations. We can then compute the median times of both sets and get a weighted average of the times, depending on the relative frequency of rebuild iterations. We call this value the weighted median iteration time and will be using it to compare the different traversals.

In Figure 5.1a we can see that all Verlet traversals outperform the lcc01 traversal. They are at least 3.5 times faster, with the fastest being about 9 times faster than the Linked Cells traversal. This big difference in iteration time comes from the difference in hit rates between the 3-body Verlet list traversals and the 3-body Linked Cells traversal. In this scenario, the lcc01 traversal only has a hit rate of 1.5%, while the ListIteration traversal has 30.5% and the other traversals all have a 66.9% hit rate. The hit rate h is given by $h = \frac{f}{f+d}$ where f is the number of force calculations and d is the number of unnecessary distance calculations. Since the number of force calculations is constant for all traversals, we can express the number of unnecessary distance calculations as $d = f\frac{1-h}{h}$. With this, we can calculate that the Linked Cells traversal makes about 29 times as many unnecessary distance calculations as the ListIteration traversal and about 133 times as many as the other Verlet traversals. But since one distance calculation takes less time than one full force calculation and the Linked Cells traversal has fewer memory indirections when accessing particles it is only 9 times slower.

In Figure 5.1b we take a closer look at all of the Verlet traversals and compare them against each other. We can see that the PairListIteration traversal is the fastest with the ListIteration traversal being a bit slower. Both ListIntersection traversals are even slower with the hashing approach being much worse.



(a) Weighted median iteration time of all Verlet traversals and the lcc01 traversal in the CubePack_M scenario.



(b) Zoomed-in version of (a) showing weighted median iteration time of all Verelt traversals for 14, 28, and 56 threads.

Figure 5.1.: Weighted median of iteration time of Verlet traversals and the lcc01 traversal, shows that all Verlet traversals outperform the lcc01 traversal, from the Verlet traversals the PairListIteration traversal is the fastest closely followed by the ListIteration traversal.

5.2.2. List Rebuilding

We now want to focus specifically on the list rebuilding time, as the different traversals require slightly different rebuilding steps.

In Figure 5.2 we can observe that the list rebuilding scales very badly with a growing number of threads. We believe that this is caused by the amount of memory allocation required during the rebuilding process, as the lists are currently implemented as vectors that are extended via the `push_back()` operation. This means during the rebuilding process, there will be multiple memory allocations per list to extend them.

We can also see that the PairListIteration traversal loses out, as it has to build much bigger lists than the other traversals. The lists have about 14 times the length of the 2-body lists in this scenario, but the rebuild operation is only about 3 to 4 times slower than in the 2-body list case. This discrepancy is probably because the system calls that allocate more memory take most of the time for the short list lengths we are dealing with. With the 14 times increase in length, we require 4 more system calls, as the vectors we use as neighbor lists grow by doubling their lengths when they are full. The rebuild of the 3-body neighbor lists also takes longer than the theoretical running time discussed in Subsection 4.5.2, which was the sum of the running times of the rebuild of the ListIteration traversal and one iteration of it. We attribute this to the fact that the memory allocations most likely take more time than the force calculations in a single iteration of the ListIteration traversal.

Of the traversals using 2-body lists, the sorted variant of the ListIntersection traversal is a bit slower than all others, as this includes the sorting time needed after rebuilding, but the overall time invested into sorting is less than we initially anticipated. This makes the rebuild about 1.1 to 1.2 times slower than the rebuilds of the hashing variant and the ListIteration approach, which both use the same rebuilding method.

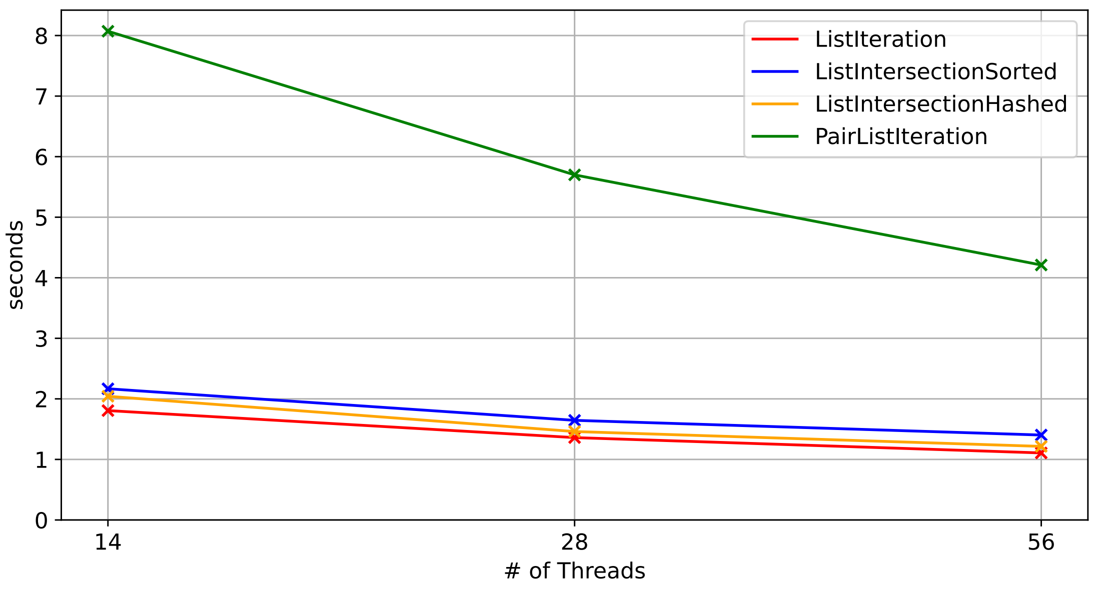


Figure 5.2.: Median list rebuilding times of Verlet traversals for 14, 28, and 56 threads in the CubePack_M scenario, shows that the 3-body list building for the PairListIteration takes about 3 to 4 times as long as the other rebuilds and that the sorting overhead for the ListIntersection approach makes it only about 1.1 to 1.2 times slower.

5.2.3. IterateTriwise

This section focuses on the time taken up by traversing all particle triplets. Figure 5.3 shows that the PairListIteration traversal has a faster iteration time than any other traversal. This comes from its high hit rate and the fact that the length of the 3-body neighbor lists is about half the amount of all pairs of particles in the 2-body neighbor lists, all of which are traversed by the ListIteration traversal. This matches our theoretical findings in Subsection 4.5.1, although the PairListIteration traversal is not twice as fast but only about 1.5 times as fast as the ListIteration traversal. This is because only doing the distance calculation takes less time than the full force calculation. So while the ListIteration traversal makes twice the functor calls, half of those only do the faster distance calculation.

Otherwise, the ListIteration traversal is in second place, which mostly comes from the fact that the ListIntersection traversals have to compute the intersection every iteration step, which has a higher runtime, as discussed in Subsection 4.5.1. This deficit is not made up for by the higher hit rate the intersection traversals have. It is also clear that the hash version of the ListIteration traversal is much worse in this aspect too, leading us to not consider it a viable option from here on out.

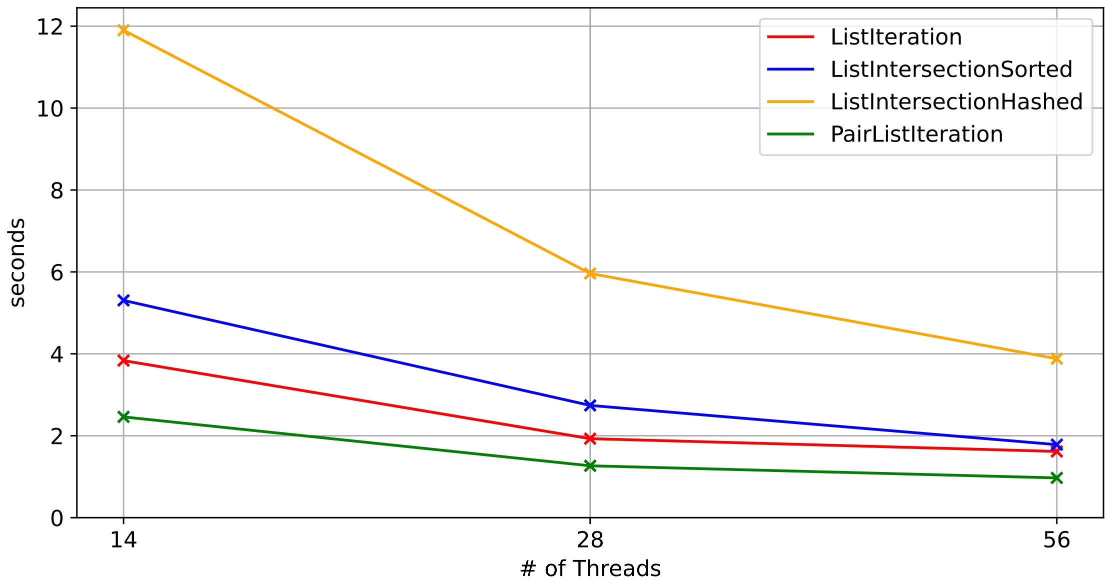
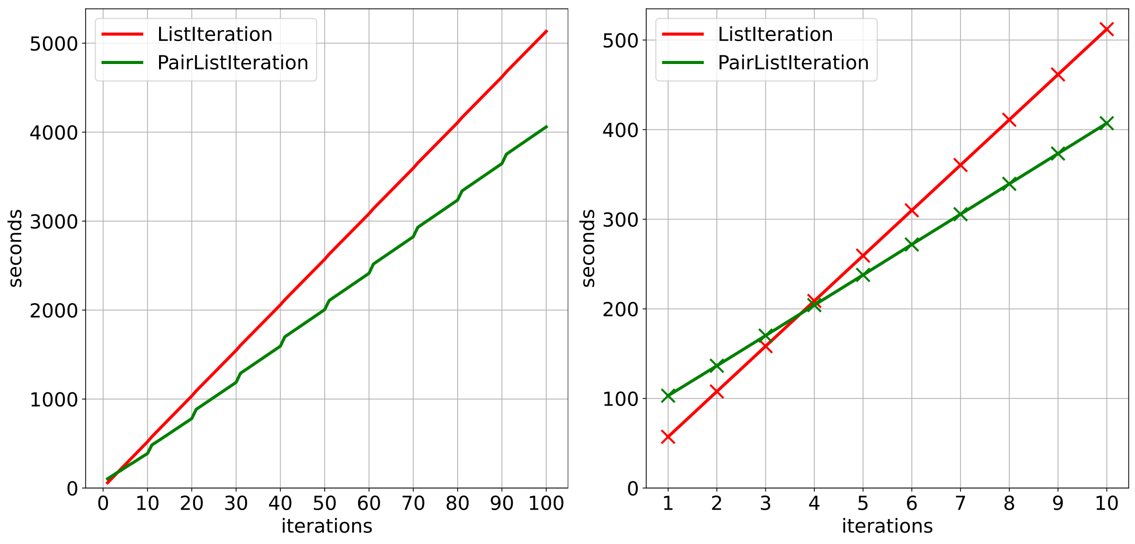


Figure 5.3.: Weighted median `iterateTriwise()` time of Verlet traversals for 14, 28, and 56 threads in the CubePack_M scenario, shows that the PairListIteration traversal is the fastest, being about 1.5 times faster than the ListIteration traversal. ListIntersection traversals are slower, with the hashing variant being about twice as slow as every other traversal.

The lower `iterateTriwise()` run time of the PairListIteration traversal is enough to offset its longer list rebuild time over the non-rebuilding time steps. In our experiments, we find that at around four iterations the PairListIteration traversal takes as much time as the ListIteration traversal. Any iteration afterward without a rebuild increases the advantage the PairListIteration traversal has over the ListIteration traversal. This can be seen in Figure 5.4b which plots the cumulative time of 10 iterations, where the first is a rebuild and all others are non-rebuild iterations, for single-threaded execution. Figure 5.4a shows the cumulative time over 100 iterations for single-threaded execution where we can see how the costly rebuild operation of the PairListIteration traversal every ten iterations is compensated for in the faster `iterateTriwise()` time.



(a) Cumulative time for one thread over all 100 iterations in the CubePack_M scenario for the ListIteration and PairListIteration traversals, shows that PairListIteration is faster when looking at many iterations.

(b) Zoomed-in version of (a) looking at 10 iterations, where the first is a rebuild iteration, shows that the PairListIteration is slower until iteration four, but faster afterward.

Figure 5.4.: Cumulative time of consecutive iterations in the CubePack_M scenario for the ListIteration and PairListIteration traversals, shows how the longer rebuild time of the PairListIteration gets compensated by the faster `iterateTriwise()` times over multiple iterations.

5.3. Traversal Improvements

In this section, we evaluate the changes we made to the traversals in an attempt to improve their running times. We use the improved versions of these traversals for all other measurements.

5.3.1. PairListIteration Traversal Improvements

For the PairListIteration we changed the 3-body traversal used for building the neighbor lists from a Linked Cells traversal to a Verlet list traversal. In Figure 5.5 we show the different rebuilding times when using different traversals. We can observe that the ListIteration traversal is the fastest in building the 3-body neighbor lists. With a lower amount of threads up to about 14 it achieves a speedup of 2 compared to the lcc01 traversal. The speedup remains above 1.75 for a higher number of threads. This reduction in rebuilding time makes the PairListIteration traversal overall faster than the ListIteration traversal. Since the ListIntersection approach is worse than the ListIteration approach we now use the

ListIteration traversal to build the 3-body neighbor lists. The bad parallel efficiency of the list rebuilding is still a problem with the Verlet traversal.

This new rebuilding approach means that we also save 2-body neighbor lists. This memory requirement will be analyzed in Section 5.5.

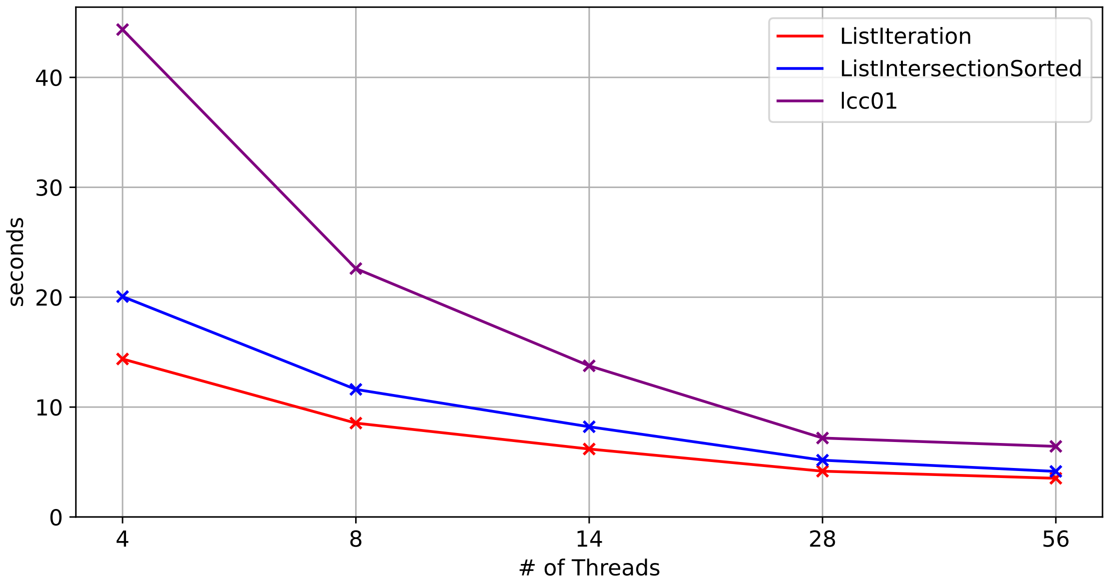


Figure 5.5.: Median 3-body neighbor list rebuild time when using ListIteration, sorted ListIntersection approaches and the lcc01 traversal for the CubePack_M scenario, shows that the ListIteration traversal is the fastest option, followed by the ListIntersection and then the lcc01 traversal.

5.3.2. ListIntersection Traversal Improvements

In the ListIntersection approach, we optimized the intersection buffer's lifetime to cut down on the number of memory allocations needed. Figure 5.6 shows the weighted median iteration time of the first and improved versions, but we can not observe any substantial speedup for the improved version. We still kept the improved version, as it utilizes the buffer better, but it does not seem to be a critical performance bottleneck of this traversal.

We showed in Subsection 4.5.1 that the intersection in every iteration has more overhead than the for loops used in the ListIteration and PairListIteration traversals. For the intersection of two lists we also have to first load both from memory, which results in more non-uniform memory accesses than in the iteration approaches. When comparing the last level cache misses of the ListIntersectionSorted traversal with those of the ListIteration traversal we find that the intersection approach has about 4 times as many. We believe that these are the more critical factors holding back this approach.

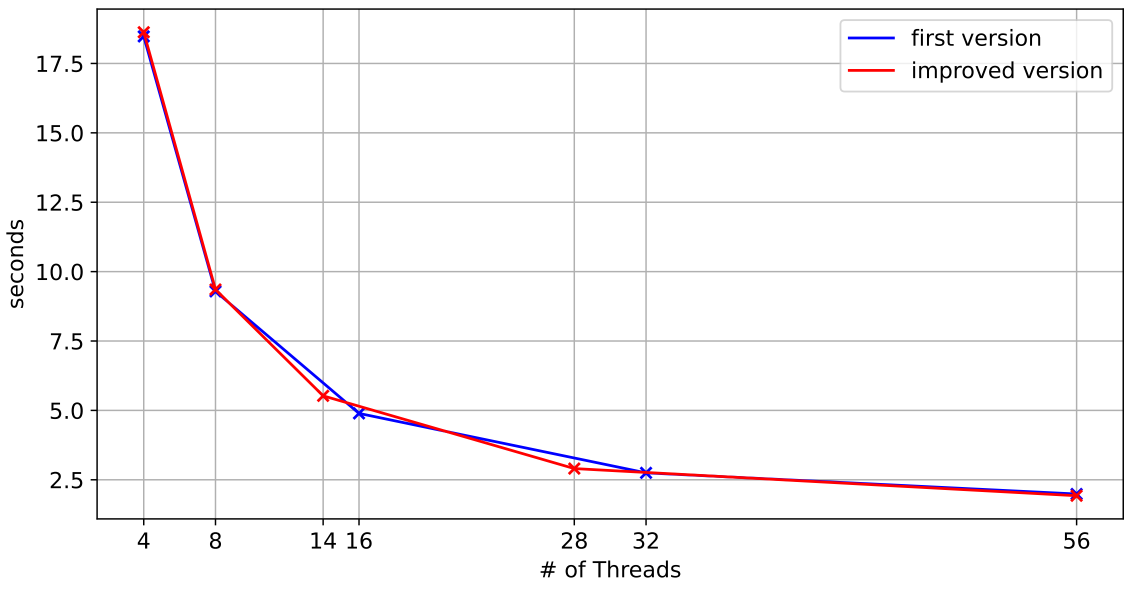


Figure 5.6.: Weighted median iteration time of the first and improved version of the ListIntersectionSorted traversal in the CubePack_M scenario, shows that the buffer optimization did not result in noticeable performance gains.

5.4. Parallel Speedup and Efficiency

Now we want to compare the parallel scaling that these traversals have. Since we want to run particle simulations at a large parallel scale, high parallel efficiency is necessary for the traversals to excel in these scenarios.

All traversals scale quite badly with hyper-threading which shows that one thread already uses most of the core's shareable resources, so hyper-threading is quite poor. It gives the highest boost to the intersection approach, which we believe is caused by the higher memory bound this traversal has, because of its more diverse memory accesses.

Figure 5.7 shows the parallel efficiency of all traversals in the smaller CubePack_T scenario of about 110 000 particles. We observed that the ListIteration traversal achieves the overall highest parallel efficiency of all Verlet traversals at 87.5% with 28 threads. The PairListIteration traversal starts very strong with a parallel efficiency of more than 85% but slowly drops off and only achieves a parallel efficiency of around 70% for the full 28 threads. This drop-off has to do with the differing parallel efficiency of `iterateTriwise()` and the list building. The parallel efficiency of the particle iteration stays above 82%, while the list rebuilding efficiency drops to 55% for 14 threads and to 42% for 28 threads. This means that the list rebuilding becomes more pronounced for higher thread numbers. We can observe this when comparing the time that list rebuilding and triplet iteration take up of one simulation step. List rebuilding takes up two-thirds of a rebuilding iteration for one thread but already takes up 80% of the time for 28 threads. The ListInersectionSorted traversal has an efficiency of around 80% overall, dropping to 75% for 28 threads.

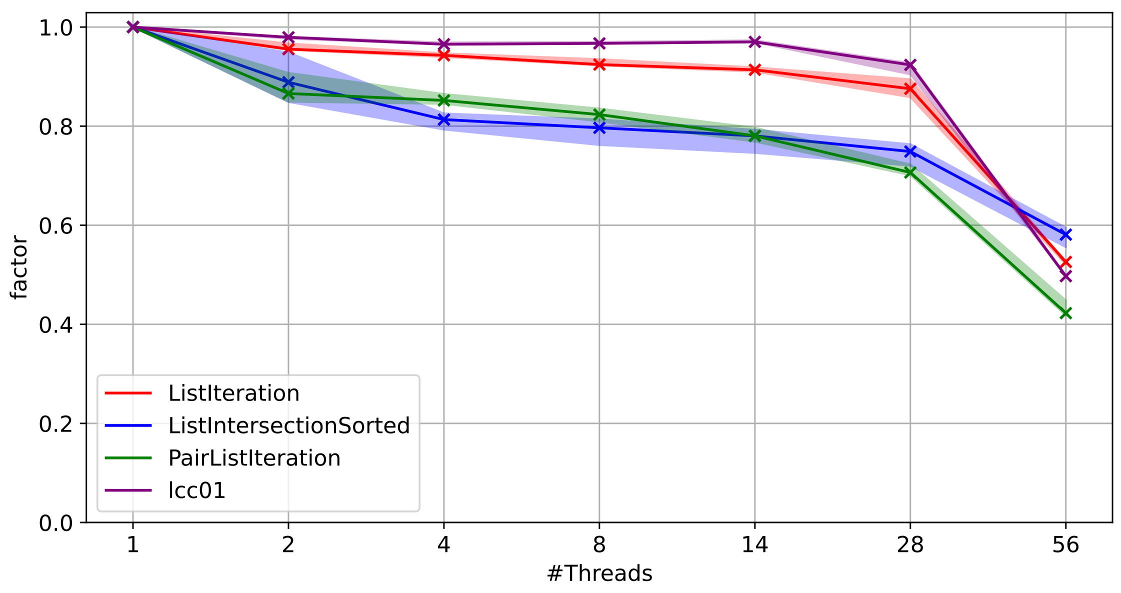


Figure 5.7.: Parallel efficiency with error ranges of traversals from 1 to 56 threads in the CubePack_T scenario, shows all traversals have bad parallel efficiency with hyperthreading at 56 threads. The lcc01 and ListIteration traversals have over 90% parallel efficiency, with the ListIteration dropping to 87.5% for 28 threads, while the other Verlet traversals range between 75 and 85% parallel efficiency.

In Figure 5.8 we also take a look at the parallel efficiency in the bigger CubePack_M scenario of about 1.1 million particles. Here we observe a higher parallel efficiency of the PairListIteration traversal. It is generally about 10% higher than in the smaller scenario. We believe that this effect comes from better load balancing with higher particle count. Every thread gets an equally sized subset of all particles to process. Since we completely skip the computation of forces for Halo particles the number of Halo particles in the subset of a thread can make a big difference in the work it has to do. This means for bigger scenarios, where the fraction of Halo particles becomes less when compared to owned particles, the impact of these Halo particles is reduced. The scaling of the ListIteration and ListIntersectionSorted traversals is similar to how it was in the smaller scenario.

An effect we do not know the cause of is the dip in parallel efficiency of the PairListIteration for two threads compared to higher thread numbers, which is especially noticeable in the bigger scenario.

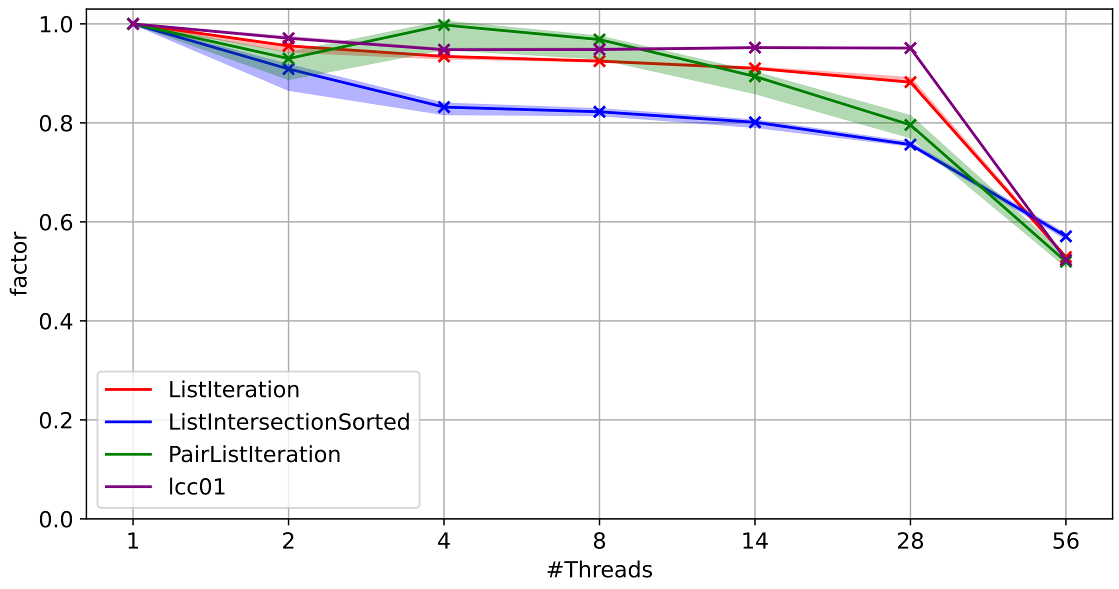


Figure 5.8.: Parallel efficiency with error ranges of traversals from 1 to 56 threads in the CubePack_M scenario, shows the lcc01, ListIteration, and PairListIteration traversal have parallel efficiencies of more than 90% with the list intersection approach only achieving an 80 to 90% efficiency.

From these observations, we can conclude that for bigger scenarios the PairListIteration performs best of the Verlet traversals for up to 28 threads. But this trend may not continue forever, as the parallel efficiency seems to drop off slowly, as the lower parallel efficiency of the list rebuilding becomes more pronounced. So we ran a scenario on the HSUper cluster with up to 72 physical cores per node to see if there is a point where the higher parallel efficiency of the ListIteration traversal will make it a faster traversal for a higher number of cores. The scenario contains 4 million uniformly randomly distributed particles at a particle density of 0.5.

In Figure 5.9 we compare the weighted median iteration time of the PairListIteration traversal against the ListIteration traversal from the HSUper run. Here we can see that the ListIteration traversal outperforms the PairListIteration traversal for 72 and 144 threads. There is a noticeable drop in parallel efficiency from 18 to 36 threads, as we now use two NUMA nodes instead of one. Another drop in parallel efficiency is from 36 to 72 threads because we are using 2 CPUs for the same domain. This data suggests that the low parallel efficiency of the neighbor list building does become a problem for the PairListIteration traversal at high thread numbers.

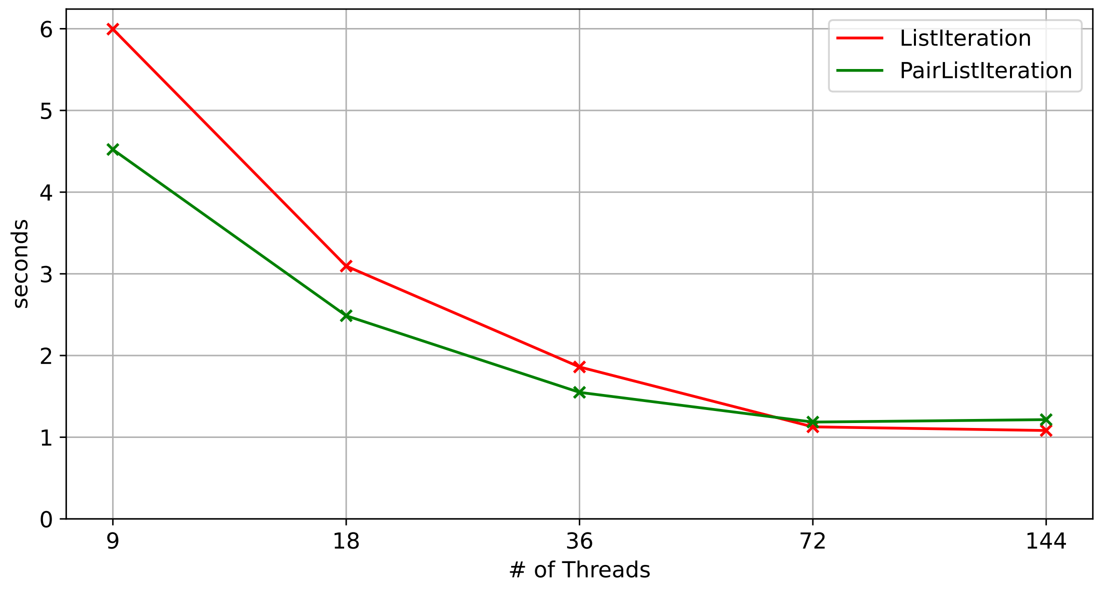


Figure 5.9.: Weighted median iteration time for ListIteration and PairListIteration for a scenario of $\sim 4\,000\,000$ particles at a density of 0.5 with up to 144 threads, shows that the ListIteration traversal does perform better than the PairListIteration traversal for 72 and more threads because of its higher parallel efficiency.

5.5. Memory Usage

In this section, we want to take a look at the memory consumption of Verlet lists. For this, we use a small scenario in which we use uniformly randomly distributed particles. We vary the density of particles and inspect the influence on memory consumption. We measured both the used memory and all allocated memory for the neighbor list. Since we are using vectors as lists the allocated memory will be less than two times the used memory for neighbor lists.

We already validated our model for the 3-body neighbor lists in Subsection 4.4.2, which tells us that the 3-body neighbor lists grow quadratically, while the 2-body neighbor lists grow linearly in the mean length of the 2-body lists. Since the average list length grows proportional to the particle density the memory consumption scales quadratically with particle density. In the case of having a high owned particle density of 1.5, we get an average memory footprint of 47.5 kB per 3-body list, while the average 2-body list only takes up 876 Byte. This means that adding the memory consumption of 2-body neighbor lists on top for our improvement increases the total memory consumption by 1.8% to 10%, as can be seen in Figure 5.10. Because of the quadratic scaling of the 3-body lists, the addition of 2-body lists is more noticeable for shorter list lengths. However, saving 2-body neighbor lists in addition to the 3-body lists, for the improved neighbor list building of the PairListIteration traversal, does not add much to the overall memory footprint of the Verlet lists.

With the generally high memory consumption of 3-body neighbor lists the PairListIteration traversal may need too much memory at higher densities. At a density of 1.5 the average allocated memory needed for the 3-body neighbor lists is at around 64 kB. With this size, we will already need 64 GB to save these lists for a million particles. This memory consumption becomes even more of a problem with even higher densities or larger cutoffs and skin radii.

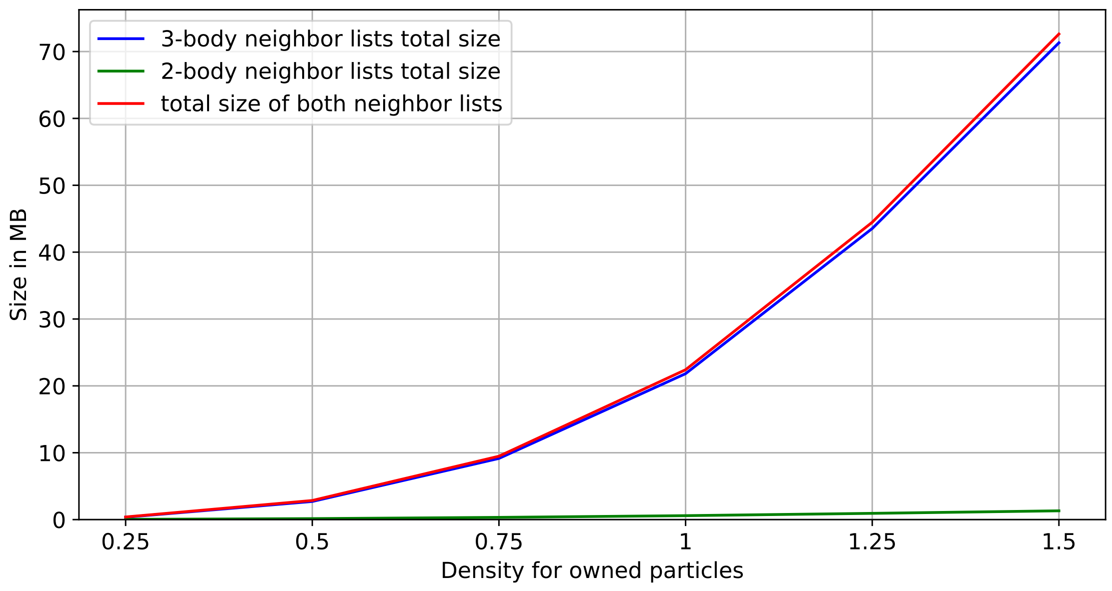


Figure 5.10.: Memory consumption of 3-body and 2-body neighbor lists, as well as their combined memory consumption for different particle densities, showing that additionally saving 2-body neighbor lists does not increase the memory footprint of 3-body neighbor lists by much.

5.6. Comparison of Traversals for Different Scenarios

In this section, we compare the two best-performing traversals ListIteration and PairListIteration on different scenarios to see how they affect their runtime and efficiency. We in one case want to test the impact that particle density has on the traversals, and we also want to investigate how the particle distribution affects their running times.

5.6.1. Comparison for Various Densities

We ran simulations of the Uniform scenario which has 512 000 uniformly randomly distributed particles for different domain volumes, giving us different particle densities to compare. In Figure 5.11 we see that the ListIteration traversal performs better than the PairListIteration for low particle densities, below 0.5. We can see that the graphs get a slight upward trend from 28 to 56 threads, which has to do with the higher hyperthread efficiency that the PairListIteration traversal has. This leads to a performance gain with hyperthreading when compared to the ListIteration traversal.

The PairListIteration seems to have a much worse parallel efficiency for lower particle densities, as we can see for densities of 0.5 and 0.375 that the PairListIteration is still faster for 4 Threads, but loses this advantage with higher thread numbers of 8 and 14. The curve for a particle density of 0.5 is quite different when we compare it to the run on the HSUper cluster in Figure 5.9 where we also had the same density. There, the ListIteration traversal only became faster than the PariListIteration at 72 Threads, compared to the 14 in this scenario. We attribute this to the bigger scenario we ran on the HSUper cluster. There we used around four million particles instead of the 512 000 on the CoolMUC-2. The PairListIteration has a higher parallel efficiency in bigger scenarios, as we discussed in Section 5.4. Thus, the difference in parallel efficiency between the ListIteration and PairListIteration traversal is bigger in the smaller CoolMUC-2 scenario, leading to the ListIteration outperfoming the PairListIteration earlier.

The general trend with the ListIteration traversal being faster for lower particle densities could be caused by the short list lengths that arise there. As this means that the lower `iterateTriwise()` runtime of the PairListIteration traversal is less noticeable, as the overall runtime of the triplet iteration per neighbor list becomes lower. This means that the PairListIteration can not make up the longer list rebuilding time through the faster iteration time, leading to it being overtaken by the ListIteration traversal.

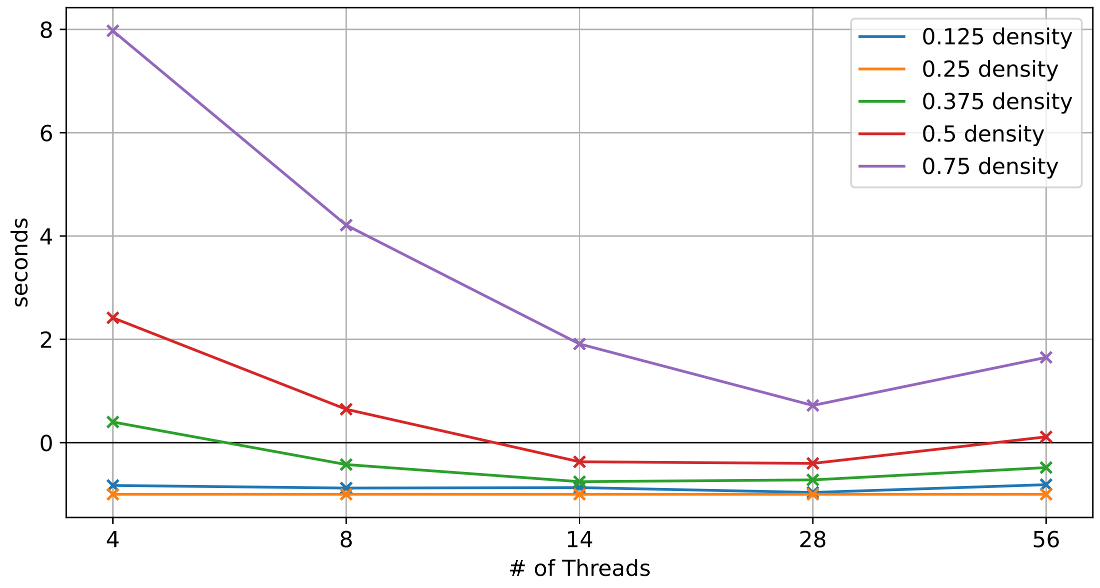


Figure 5.11.: Difference of weighted median iteration time between ListIteration and PairListIteration for different densities in the Uniform scenario relative to the values of 0.25 density, positive = PairListIteration is faster, negative = ListIteration is faster, shows that the ListIteration traversal is faster for some thread numbers for particle densities of 0.5 or less.

5.6.2. Comparison for Gaussian Distribution

We simulated the Gauss scenario, which has a different distribution of particles, to find out if the ListIteration traversal would be better suited for different scenarios. In Figure 5.12 we compare the weighted median iteration times of the ListIteration and PairListIteration traversals again with varying standard deviations. This also leads to varying numbers of particles for the different simulations. We can see that both traversals achieve very similar speeds for higher standard deviations. This should be caused by both the smaller simulation size, as well as the overall lower density of particles.

The Gaussian distribution also leads to a very dense region around the mean of the distribution. This leads to high memory consumption for the 3-body neighbor lists, this became so much of a problem that we ran out of memory on the CoolMUC-2, when using a standard deviation of 30. This shows that the higher memory requirement can become a problem in a simulation with local regions of highly dense particles.

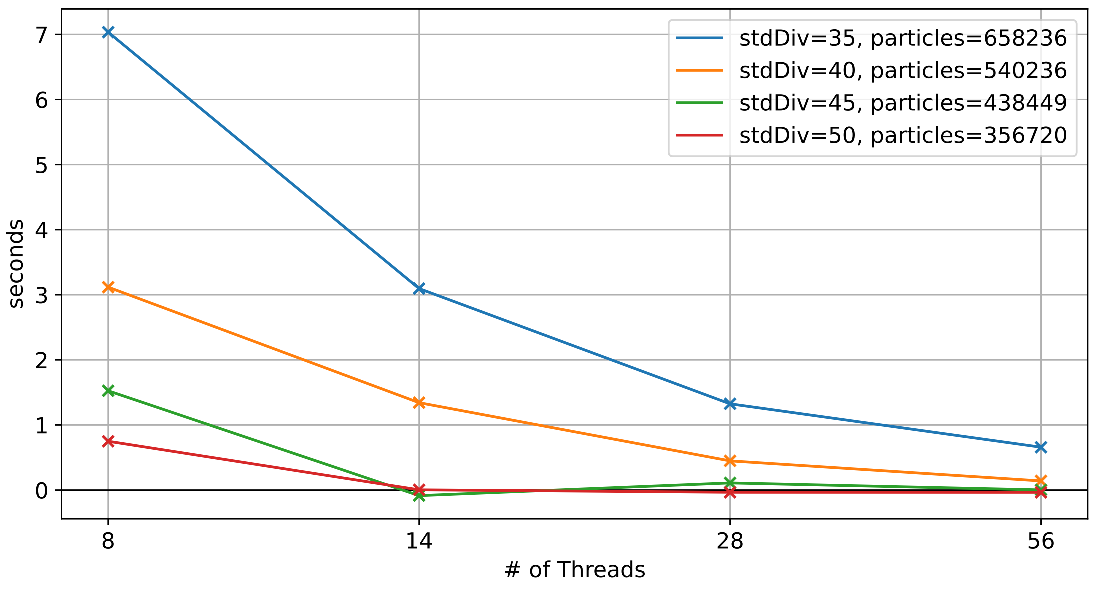


Figure 5.12.: Difference of weighted median iteration time between ListIteration and PairListIteration for the Gauss scenario with different standard deviations, positive = PairListIteration is faster, negative = ListIteration is faster, showing that both the ListIteration and PairListIteration traversal are similarly fast for more than 14 threads and standard deviations of 45 or more, otherwise the PairListIteration outperforms the ListIteration traversal.

6. Conclusion

In this thesis, we introduced and implemented multiple approaches for a 3-body neighbor identification algorithm based on Verlet lists as traversals in AutoPas. We then compared their performance in different scenarios as well as their memory requirements.

This has shown that the PairListIteration traversal performed the best out of all approaches in medium to high-density scenarios. The ListIteration traversal is better suited than it for very low-density simulations, with a particle density of less than 0.5. We find that both traversals have an overall high parallel efficiency with the PairListIteration approach having better parallel efficiency for bigger scenarios. Both traversals have a low parallel efficiency for the list rebuilding process. This dampens the overall efficiency of the PairListIteration for high numbers of threads, where the ListIteration can perform better. The intersection approach did not perform as well as the other approaches, because the intersection of all lists takes longer than simple iteration and the less predictable memory accesses made by it.

On the aspect of memory, we created a model for the lengths and memory requirement of 3-body neighbor lists based on the lengths of the 2-body lists and validated this model with measurements from our simulations. We concluded that the model slightly underestimates the memory consumption by roughly 10%. This model shows that the PairListIteration approach needs much more memory than the ListIteration approach, which is one of its disadvantages. Because of our findings, we propose to keep both the ListIteration and PairListIteration traversal for AutoTuning purposes in AutoPas.

For future work, we think an investigation into parallelized implementations of Newton3 versions of the traversals could be interesting, as in the case of the ListIteration traversal the Newton3 implementation was much more efficient than the single-threaded version without Newton3. Another area of future improvement could be made to the structure of the Verlet lists in AutoPas since their building incurs many memory allocations when growing the vectors. To circumvent this we can use a buffer of neighbor list vectors to reuse over multiple rebuilds, we could also use the model of 3-body neighbor list lengths to allocate the likely neighbor list length from the start, which should avoid a lot of the allocations for vector resizing. It would also be interesting to see how our implementation of the ListIteration traversal compares to the approaches of LAMMPS and HOOMD-blue.

List of Figures

List of Tables

Bibliography

[1] Hoi Ling Luk, Johannes Feist, J. Jussi Toppari, and Gerrit Groenhof. Multiscale molecular dynamics simulations of polaritonic chemistry. *Journal of Chemical Theory and Computation*, 13(9):4324–4335, August 2017.

[2] Arieh Warshel. Molecular dynamics simulations of biological reactions. *Accounts of Chemical Research*, 35(6):385–395, April 2002.

[3] José Alejandre, Dominic J. Tildesley, and Gustavo A. Chapela. Molecular dynamics simulation of the orthobaric densities and surface tension of water. *The Journal of Chemical Physics*, 102(11):4574–4583, March 1995.

[4] Nikola Plamenov Tchipev. *Algorithmic and implementational optimizations of molecular dynamics simulations for process engineering*. PhD thesis, Technische Universität München, 2020.

[5] Fabio Alexander Gratl, Steffen Seckler, Hans-Joachim Bungartz, and Philipp Neumann. N ways to simulate short-range particle systems: Automated algorithm selection with the node-level library autopas. *Computer Physics Communications*, 273:108262, April 2022.

[6] Philipp Ströker. Bestimmung thermodynamischer eigenschaften von fluiden mit einer weiterentwickelten molekularen simulationsmethodik und hochgenauen ab initio-potentialen. 2023.

[7] Michael P Allen et al. Introduction to molecular dynamics simulation. *Computational soft matter: from synthetic polymers to proteins*, 23(1):1–28, 2004.

[8] Isaac Newton. *Philosophiae naturalis principia mathematica*, volume 1. G. Brookman, 1833.

[9] B. M. Axilrod and E. Teller. Interaction of the van der Waals Type Between Three Atoms. *The Journal of Chemical Physics*, 11(6):299–300, 06 1943.

[10] Seymour Lipschutz. *Linear Algebra 4th ed.* McGraw-Hill, 2009.

[11] C.F. Cornwell and L.T. Wille. Parallel molecular dynamics simulations for short-ranged many-body potentials. *Computer Physics Communications*, 128(1–2):477–491, June 2000.

[12] Penporn Koanantakool and Katherine Yelick. A computation- and communication-optimal parallel direct 3-body algorithm. In *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, November 2014.

[13] Liam D O'Suilleabhain. Three body approximation to the condensed phase of water. *Master's thesis, University of California, Berkeley, Berkeley, CA*, 2013.

[14] Steffen Seckler, Fabio Gratl, Matthias Heinen, Jadran Vrabec, Hans-Joachim Bungartz, and Philipp Neumann. Autopas in ls1 mardyn: Massively parallel particle simulations with node-level auto-tuning. *Journal of Computational Science*, 50:101296, March 2021.

[15] Fabio Alexander Gratl, Steffen Seckler, Nikola Tchipev, Hans-Joachim Bungartz, and Philipp Neumann. Autopas: Auto-tuning for particle simulations. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, May 2019.

[16] Dennis C Rapaport. *The art of molecular dynamics simulation*. Cambridge university press, 2004.

[17] Loup Verlet. Computer "experiments" on classical fluids. i. thermodynamical properties of lennard-jones molecules. *Physical Review*, 159(1):98–103, July 1967.

[18] Pedro Gonnet. Pairwise verlet lists: Combining cell lists and verlet lists to improve memory locality and parallelism. *Journal of Computational Chemistry*, 33(1):76–81, 2012.

[19] Aidan P. Thompson, H. Metin Aktulga, Richard Berger, Dan S. Bolintineanu, W. Michael Brown, Paul S. Crozier, Pieter J. in 't Veld, Axel Kohlmeyer, Stan G. Moore, Trung Dac Nguyen, Ray Shan, Mark J. Stevens, Julien Tranchida, Christian Trott, and Steven J. Plimpton. Lammps - a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales. *Computer Physics Communications*, 271:108171, February 2022.

[20] Ilian T. Todorov, William Smith, Kostya Trachenko, and Martin T. Dove. Dl_poly_3: new dimensions in molecular dynamics simulations via massive parallelism. *Journal of Materials Chemistry*, 16(20):1911, 2006.

[21] J. Tersoff. Modeling solid-state chemistry: Interatomic potentials for multicomponent systems. *Physical Review B*, 39(8):5566–5568, March 1989.

[22] Joshua A. Anderson, Jens Glaser, and Sharon C. Glotzer. Hoomd-blue: A python package for high-performance molecular dynamics and hard particle monte carlo simulations. *Computational Materials Science*, 173:109363, February 2020.

[23] Simone Ciarella and Wouter G. Ellenbroek. Associative bond swaps in molecular dynamics. 2019.

[24] Shu-Ju Tu and Ephraim Fischbach. A new geometric probability technique for an n-dimensional sphere and its applications to physics, 2000.

[25] Maurice George Kendall and Patrick Alfred Pierce Moran. Geometrical probability. 1963.