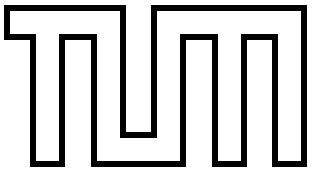


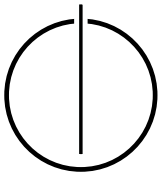
FAKULTÄT FÜR INFORMATIK

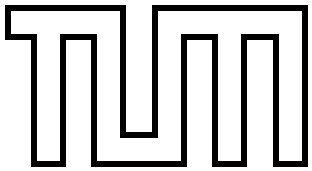
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Implementation and Evaluation of Verlet List-based Methods in AutoPas

Tina Vladimirova





FAKULTÄT FÜR INFORMATIK

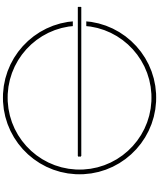
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Implementation and Evaluation of Verlet List-based Methods in AutoPas

Implementierung und Evaluierung von Verlet List-basierten Methoden in AutoPas

Author:	Tina Vladimirova
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisor:	Fabio Alexander Gratl, M.Sc.
Date:	15.02.2021



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

Munich, 15.02.2021 Tina Vladimirova

Acknowledgements

I would like to thank my advisor Fabio Gratl for providing valuable feedback and constant support.

Abstract

MD simulations can take days to compute, thus optimizing their performance is an important task. AutoPas is an open-source C++ library which tunes MD simulations and chooses the optimal algorithm for any given scenario. In the context of AutoPas, we implement the pairwise Verlet algorithm, which aims to improve on the existing Verlet lists neighbor identification algorithm. Other relevant components are added as well, such as a Structure of Arrays representation for the particle data. The new algorithm addresses the issue of scattered memory access and is designed to be well suited for shared memory parallelization. Our evaluation indicates that the pairwise Verlet algorithm performs well for a dense scenario or a larger interaction range between particles. However, other scenarios or other configurations such as the Structure of Arrays data layout appear to worsen its performance.

Zusammenfassung

Die Berechnung von Molekulardynamik Simulationen kann bis zu einigen Tagen dauern, somit ist die Optimierung von ihrer Performanz eine wichtige Aufgabe. AutoPas ist eine open-source C++ Bibliothek, die MD Simulationen tunet und den optimalen Algorithmus wählt für jedes gegebene Szenario. Im Kontext von AutoPas, wir implementieren den paarweisen Verlet-Listen Algorithmus, dessen Ziel ist, den existierenden Verlet-Listen Nachbarschaftssuchalgorithmus zu verbessern. Andere relevante Komponente werden auch hinzugefügt, wie eine Structure of Arrays Datenrepräsentierung für die Partikeldaten. Der neue Algorithmus zielt, gestreute Speicherzugriffe zu minimieren und passend für Shared-Memory-Parallelisierung zu sein. Unsere Evaluierung zeigt, dass der paarweise Verlet-Algorithmus gute Performanz aufweist für Szenarien mit hoher Dichte oder einen größeren Interaktionsbereich. Dagegen scheinen andere Szenarien oder Konfigurationen wie Structure of Arrays die Performanz zu verschlechtern.

Contents

Part I.

Introduction and Background

1. Introduction

The field of molecular dynamics aims to study the behaviour of a system of molecules over time. [HD18] [SSR07] This system could take the form of a gas, liquid, or solid, or include multiple bodies in different states. Example scenarios in practice include flow through a nanotube and protein folding. Simulating molecular dynamics phenomena has an advantage over physical experiments because it allows experts to observe scenarios that would otherwise be too expensive, dangerous or difficult to set up in a laboratory.[1] Thus, MD simulation is important for progress in the fields of molecular biology, drug discovery, materials science, etc. [DM11] [HD18] [SSR07]

Discussing the main characteristics of an MD simulation model will provide insight into the challenges the field presents. Interactions between particles are modeled through force fields [GST+19], which makes this problem a case of the more general N-body problem. Each body interacts with every other body, which leads to an unfavorable complexity of $O(N^2)$. MD simulations typically include millions to billions of molecules on a nanometer scale, and a scenario of a few real-time seconds needs to be segmented into timesteps as small as 10^{-15} seconds. Consequently, a typical MD simulation can take several days to compute.[1]
Modern computers are able to employ parallelization to reduce the runtime of a computationally intensive program, but this reveals a new category of problems such as race conditions and load balancing. Moreover, some staple MD algorithms were developed with older, single-core architectures in mind. [GST+19] [Gon12] Choosing the suitable algorithms, parallelization strategies and data representations can significantly improve the performance of a simulation.

This work is developed in the context of the AutoPas library. AutoPas is an open-source C++ library which aims to tune and optimize the performance of an N-body simulation. It works as a black box and selects the best combination of neighbor interaction algorithms, parallelization strategies, particle storage, etc. in order to achieve optimal performance for a given scenario. [GST+19]
In this project we expand the library with the pairwise Verlet algorithm [Gon12]. This algorithm is used for short-range interactions between particles and builds on the existing Verlet list algorithm. Other components are also added or adapted, such as suitable parallelization strategies and data representations. The performance of the new components is measured and compared to existing ones for a variety of scenarios.
The goal of implementing these components is to improve cache efficiency and parallelization and consequently the simulation runtime. In comparison to the classic Verlet list, the pairwise Verlet algorithm was developed with multi-core architectures in mind, which are nowadays the norm in high-performance computing. [Gon12]

[1] `https://www5.in.tum.de/lehre/praktika/pse/ws17/Vorbesprechung.pdf`

2. Theoretical background

This chapter provides relevant background knowledge about the physical aspect of molecular dynamics and describes some main components of the AutoPas library.

2.1. Molecular dynamics

2.1.1. Short-range interactions

The main focus of this work are the short-range interactions between particles. The interaction between two particles i and j is typically modeled by the Lennard-Jones potential:

$$U(r_{ij}) = 4\epsilon \left(\left(\frac{\sigma}{r_{ij}} \right)^{12} - \left(\frac{\sigma}{r_{ij}} \right)^{6} \right)$$

where r_{ij} is the distance between the two particles, ϵ is the energy parameter (strength of the potential), and σ is a size parameter (zero crossing). [Gra21]

Figure 2.1 shows the change in the potential in relation to r_{ij}. With the growth of r_{ij} the particles repel each other and then attract each other before the potential converges to zero. We define a cutoff radius r_c as a distance between the particles where U_{ij} is very near to zero. AutoPas utilizes this by discarding short-range interactions between particles that are at a larger distance than r_c in order to avoid interactions with a negligible effect on the system. Therefore, the force calculation is optimized from a complexity of $O(N^2)$ to $O(N)$. [GST+19]

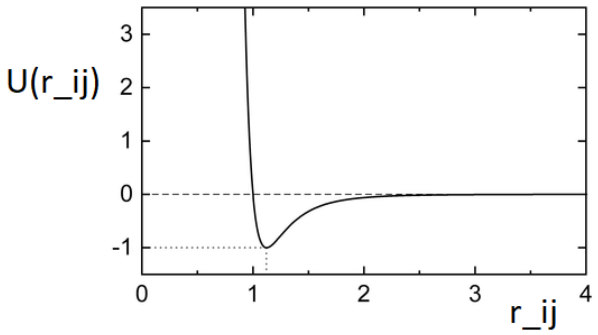


Figure 2.1.: Relationship between r_{ij} (the distance between particles i and j) and the Lennard-Jones potential $U(r_{ij})$[a]

[a] `https://en.wikipedia.org/wiki/Lennard-Jones_potential`; edited

2.1.2. Newton's 3rd law

Newton's 3rd law of motion can be applied to most N-body problems. In the case of particle interactions, the force from the interaction of particle i with particle j F_{ij} has the same magnitude as the force of interaction of particle j with particle i, but they are oriented in opposite directions. In short:

$$F_{ij} = -F_{ji}$$

This enables us to implement another optimization: the force between two particles may be calculated only once and applied to both particles in the respective directions. Thus, the amount of force calculations can be reduced by half. [GST+19]

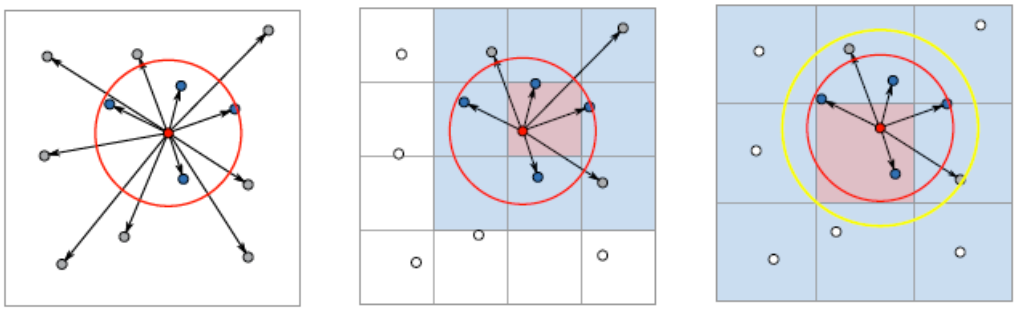
2.2. AutoPas

2.2.1. Containers

Containers in AutoPas deal with storing the particles and more importantly, determining which pairwise interactions between particles will be taken into account.

Algorithms

There are three main algorithms for selecting the interaction pairs which are relevant to this work: direct sum, linked cells and Verlet lists.



(a) Direct Sum (b) Linked Cells (c) Verlet Lists

Figure 2.2.: Three main containers in AutoPas. Direct Sum: the force is computed between the red particle and the blue particles in its red circle (with a radius of r_c). Linked Cells: The red particle from the red base cell interacts with the blue particles, which are located in the blue neighboring cells and lie within the cutoff radius r_c. Verlet Lists: The red particle saves all particles within the yellow circle in its neighbor list. It interacts only with the ones in the red circle defined by r_c. [GST+19]

Direct sum is the most intuitive algorithm, it simply calculates the distance between every pair of particles and computes their interactions if their distance is smaller than the cutoff radius r_c (Subsection 2.1.1). For every single particle, the algorithm sums up all the forces from its interactions with the other particles, hence the name direct sum.

This method has a complexity of $O(N^2)$, but it is also the easiest to implement and has no memory overhead because only the particle data needs to be stored. A visual depiction can be seen in Figure 2.2a. [GST+19]

Linked cells divides the domain into cells and stores the particles in that manner. The length of the cell is typically constrained to be larger or equal to r_c. In that case, for a given base particle, the algorithm checks the distance to all other particles located in the current cell and all of the neighboring cells, depicted in red and blue respectively in Figure 2.2b. Similarly to direct sum, if the distance is smaller than r_c, the force is computed and applied. Otherwise, if the cell length is smaller than r_c, the algorithm considers more cells outside of the direct neighbors. It is also possible to use asymmetrical cells to divide the domain, but this work is focused primarily on symmetric cells. [GST+19]
The main advantage of the linked cells algorithm is that the particles of a cell are stored contiguously in memory, which enables more effective distance calculations since the cutoff checks are done for all particles of a cell. Moreover, in combination with the SoA data layout (Subsection 2.2.2) the algorithm is especially well suited for vectorization, as the corresponding attribute of each particle is also stored contiguously. Finally, the complexity of this algorithm can be reduced to $O(N)$ if the number of cells is selected proportionally to the number of particles. [GST+19]
While the linked cells algorithm performs significantly less distance calculations than direct sum, 84% of them are nevertheless avoidable. We can compare the search volume to the volume of the sphere defined by r_c, also called interaction sphere.

$$\frac{r_c\, sphere}{search\, volume} = \frac{\frac{4}{3}\pi r_c^3}{(3r_c)^3} \approx 0.155$$

The volume of the interaction sphere makes up less than 16% of the volume of the total search space, thus there is about 84% probability that a distance calculation is unnecessary and the potential partner is not in range of r_c. [GST+19]

Verlet lists attempts to fix this problem by introducing a skin which goes around the sphere defined by r_c, seen in yellow on Figure 2.2c. The size of the skin is represented by a factor $s > 0$. The distance between a particle and all other particles within range $r_c \cdot s$ are computed and their data is stored in neighbor lists. Each particle has a neighbor list where its potential interaction partners are stored. In each iteration the distance between the pre-computed pairs is calculated and compared to r_c, in the same manner as the previous two algorithms. Since the particles are moving and the potential partners might leave the desired range, the neighbor lists need to be rebuilt periodically. [GST+19]
The amount of unnecessary distance calculations for this algorithm depends on the skin factor:

$$\frac{r_c\, sphere}{search\, volume} = \frac{\frac{4}{3}\pi r_c^3}{\frac{4}{3}\pi (r_c \cdot s)^3} = \frac{1}{s^3}$$

For example, for $s = 1.2$ the ratio is $\approx 58\%$ and for $s = 1.1$ it is $\approx 75\%$. Therefore, the proportion of unneeded checks is 42% and 25% respectively, in any case an improvement to the linked cells algorithm. In the standard Verlet lists algorithm all pairs of particles

need to be considered in the building of the neighbor lists, which brings the complexity to $O(N^2)$. An improved method uses the linked cells algorithm with a cell size $\geq r_c \cdot s$ to construct the Verlet lists. The cell size guarantees that the search sphere will include only the base cell and its neighboring cells. Therefore, it is enough to use linked cells to find potential partners instead of searching through the whole domain. [GST+19]

Implementation in AutoPas

We will briefly discuss the container hierarchy in AutoPas and its methods most relevant to this work. A simplified visual representation of the hierarchy can be seen in Figure 2.3.

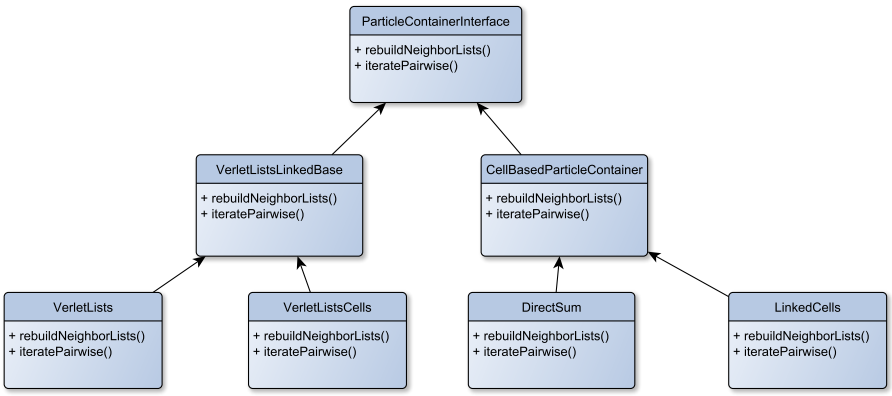


Figure 2.3.: Simplified container hierarchy in AutoPas. `ParticleContainerInterface` declares the methods `rebuildNeighborLists()` and `iteratePairwise()`. `VerletListsLinkedBase` is a base class for `VerletLists` and `VerletListsCells`, another Verlet lists-based container. `DirectSum` and `LinkedCells` inherit from the `CellBasedParticleContainer` class.

At the top of the hierarchy is `ParticleContainerInterface`, the base class for all containers. It includes methods for adding and removing particles, setters and getters for r_c and skin, etc. Most notably, this interface declares the methods `rebuildNeighborLists()` and `iteratePairwise()`.
The `rebuildNeighborLists()` method, as the name suggests, is responsible for the periodic rebuilding of the neighbor lists for containers that use such data structures. More specifically, the method is implemented by the Verlet lists container and other containers which are based on the Verlet lists neighbor selection algorithm. The `iteratePairwise()` method iterates over each pair of particles using a traversal algorithm, passed to it as an argument, and applies a functor to those pairs. Traversals and functors are discussed in more detail in Subsection 2.2.3. These methods are left empty in higher-level classes or containers which do no make use of them, such as `rebuildNeighborLists()` in the linked cells container. Since components such as traversals and neighbor lists data structures are specific to each container, these methods are implemented on the lowest levels of the hierarchy.

At the next level of the hierarchy, directly inheriting from `ParticleContainerInterface` is

the class `CellBasedParticleContainer`. It serves as a base class for containers that divide the domain in cells. The linked cells container is derived from this class, as well as the direct sum container, which stores particles in a single cell.

The `VerletListsLinkedBase` class also inherits directly from `ParticleContainerInterface`. It uses a `LinkedCells` object to store particles in cells, while also functioning as a base class for Verlet lists-based containers. Thus, the Verlet lists container is implemented on top of this class in its improved version explained in Section 2.2.1. Another subclass is the `VerletListsCells` container, which implements a Verlet list-based approach while making use of linked cells. The details of this container will be discussed in Chapter 4.

2.2.2. Data layout

The structures used to store the particle data are an important aspect of the simulation setup. AutoPas has two data layouts that we are interested in for the purposes of this work: Array of Structures (AoS) and Structure of Arrays (SoA).

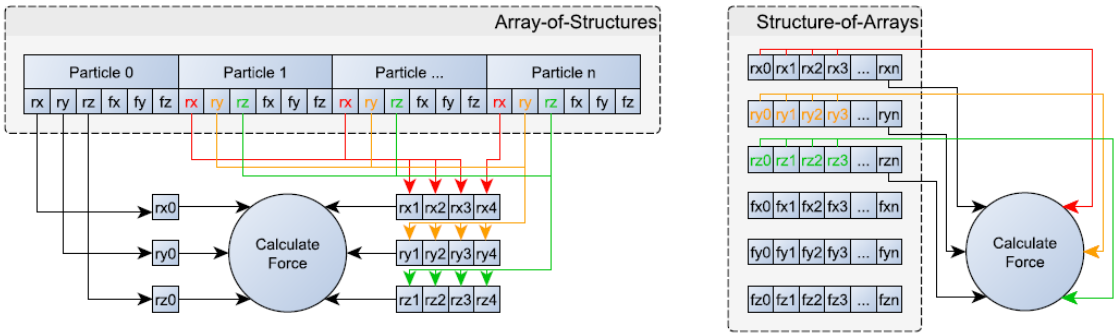


Figure 2.4.: A visual representation of the AoS and SoA data layouts. On the left is shown an array of particle objects which store position and force along the three axes. On the right is a structure of arrays, where each array is dedicated to a single particle attribute. [GST+19]

AoS is the default data layout in AutoPas. It stores an array of particle objects, which contain attributes such as position and force along the x, y and z axes. The array of structures is depicted on the left side on Figure 2.4. This data representation is easy to work with, but it is the less efficient option in the context of cache efficiency and vectorization. SIMD (Single Instruction, Multiple Data) vectorization is a feature of modern processors: an operation can be carried out on a whole vector of data as a single instruction. In the case of AoS, when the same property is accessed among multiple particles and loaded into a vector register, the program has to make jumps in memory to reach the relevant part of each particle's data. This gather operation requires unnecessary slow memory accesses. The same applies when the program is loading information from the memory into the cache, also known as cache prefetching. The cache contains the full data of a few particles, while storing only the relevant information for more particles would be more cache efficient. Trying to access the necessary data for a given particle if that portion is not in the cache re-

sults in a cache miss and provokes more slow memory accesses to retrieve that data. [GST+19]

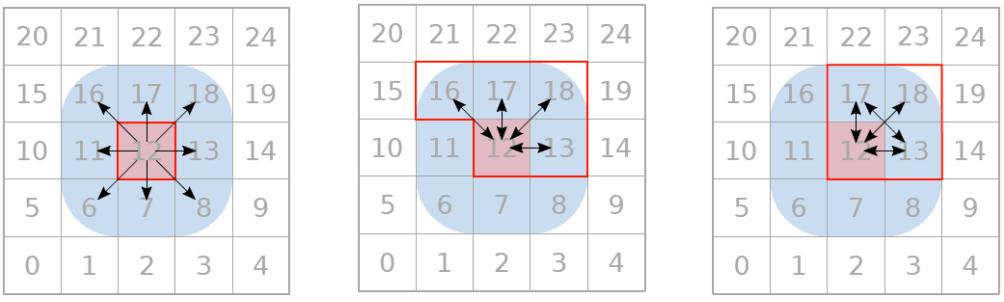
The SoA data layout solves this issue by storing arrays for each particle attribute and wrapping these arrays into a structure, as depicted on the right side of Figure 2.4. Thus, the data layout loads the same information for multiple particles contiguously into memory. This order makes cache prefetching more effective compared to AoS, as the cache loads the same amount of data, but now that includes only one property for more particles. A main disadvantage of SoA is that it is less intuitive than AoS. Additionally, AutoPas stores the data in an AoS layout and using SoA requires converting from and to AoS in every iteration. AoS is also at an advantage if we need to access single particles, because the whole data for the particle is contiguous. Nevertheless, SoA is generally considered the more efficient data layout, because force calculations tend to benefit more from storing the same information for multiple particles consecutively. [GST+19]

2.2.3. Traversals

Parallelization is a key component in MD simulations since it has the potential to improve their runtimes significantly. It is important to distribute the workload evenly among the threads, but also to maximally utilize the available computational resources. In AutoPas the traversals control in what order the particles are iterated over and how the force calculation is parallelized.

Algorithms

For this work we are interested in the traversals based on domain coloring, where each cell is assigned a color and all cells of the same color are processed in parallel. The coloring eliminates scenarios where a thread processing a cell would access data which is being modified by another thread, thus ensuring protection against race conditions. We will discuss the base steps for three possible colorings: c01, c18 and c08.



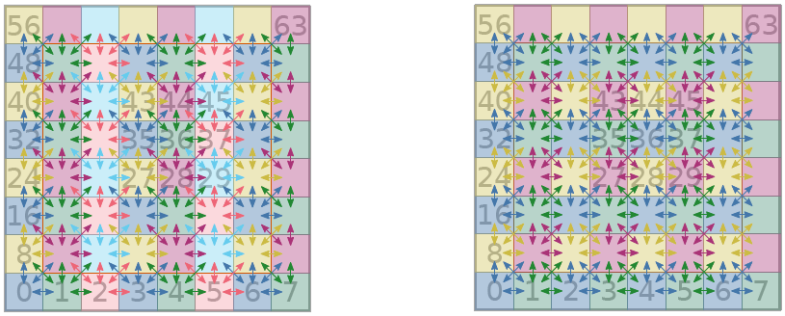
(a) c01 base step (b) c18 base step (c) c08 base step

Figure 2.5.: Base steps in AutoPas traversals. The c01 base step includes an interaction with all neighbors. The c18 base step includes only interactions with neighbors with a bigger cell index. The c08 base step is a variation of the c18 base step which allows less cells to be locked in a base step. [Gra21]

c01 The c01 base step depicted in Figure 2.5a is the most intuitive, as it simply interacts a base cell with all of its neighbors. The newton3 optimization is not applicable in this case, hence the computed forces are only applied to the base cell and its partners are not modified. This allows a high degree of parallelization: only one color is used for the whole domain and all cells can be processed at the same time with a high enough number of threads. [Gra21]

c18 This base step is a logical consequence of the newton3 optimization. A base cell interacts only with its neighbors with a higher index, as seen in Figure 2.5b. However, all of the necessary interactions are still taking place because the rest of the neighbors (with a lower index) interact with the current base cell in their own base steps. Although it cannot be parallelized to the degree of c01, it has the advantage of computing only half as many interactions in total. However, in contrast to the c01 base step, the forces are applied to both cells, which means more cells need to be locked to prevent race conditions. The 2D variant uses six colors Figure 2.6a, while in 3D 18 colors are necessary to enable the parallelized interaction along the z axis. Adding a third dimension creates the need for nine instead of six colors in each slice along the z axis to account for interactions with cells from the next slice. Furthermore, the color scheme must be alternated over each pair of slices to prevent base cells from neighboring slices from being processed at the same time. Consequently, a base step requires synchronized access to a block of size 3 x 3 x 2. [Gra21]

c08 The c08 traversal is similar to c18, but it manages to use less locks by constraining the interaction block to a size of 2 x 2 x 2. This works because the c08 base step includes interactions which do not involve the base cell. In comparison to the c18 base step, some interactions have been "moved" to another cell's base step to make the block more compact. As illustrated on Figure 2.5c, the interaction $13 \leftrightarrow 17$ would have been a part of cell 13's base step, but with c08's modification it can be processed in cell 12's base step. Similarly to c18, the 2D coloring uses four colors Figure 2.6b, while the 3D coloring needs to alternate the scheme over the z axis resulting in eight colors overall. In general, this coloring requires 2^d colors when d is the number of dimensions. [Gra21]



(a) c18 coloring in 2D (b) c08 coloring in 2D

Figure 2.6.: 2D domain coloring for traversals in AutoPas [Gra21]

These base step schemes can be applied to a container according to the respective data structure and thus a traversal is formed. Every container has a list of applicable traversals since not every base step can be applied to any container and the implementations are container-specific.
AutoPas implements another types of traversals as well, namely sliced traversals, which are based on slicing the domain along a dimension in a certain way. This type of traversal is not relevant to this work so we will not go into detail. [Gra21] [GST+19]

Implementation in AutoPas

We will shortly describe the traversal hierarchy and the general pipeline for the force calculation.
The traversal hierarchy starts with a `TraversalInterface` which declares the main methods for the pipeline: `initTraversal()`, `traverseParticlePairs()`, and `endTraversal()`. While `initTraversal()` and `endTraversal()` deal with setting up the traversal, for instance triggering a conversion from AoS to SoA and back, `traverseParticlePairs()` contains the bulk of the traversal implementation. Thus, it is usually implemented in the lowest levels of the traversal hierarchy.
Since we are interested in domain colorings, we will explain that part of the hierarchy in greater detail. The `CBasedTraversal` class is the base class for color-based traversals, derived from the `CellPairTraversal` class. It defines a parallelized loop which iterates over the domain in strides and applies a given loop body. Subclasses are derived from this class for the different colorings (c01, c18, c08, etc.). They specify the strides for that parallelized loop so that the particular coloring pattern will be applied over the domain.
The traversal implementations for a particular container are at the lowest positions in the hierarchy. They typically inherit from an interface defining the coloring, as well as a traversal interface for the container. The implementation of `traverseParticlePairs()` calls the main traversal method from the superclass for the respective coloring and implements a loop body to pass to it.
The traversals receive multiple template arguments, one of them being a pairwise functor. The functor can be a force calculation functor (Lennard-Jones or Lennard-Jones AVX) or another type of operation involving pairs of particles. The functor itself typically inherits from the `Functor` class and has an implementation for AoS and SoA data layouts. The traversal implementation iterates over pairs of particles or cells and calls on the functor's respective AoS or SoA implementation.

3. Related work

This work is motivated by P. Gonnet's research on neighbor selection algorithms and his exploration of the pairwise Verlet algorithm. [Gon12] Testing and performance measurements of the pairwise Verlet and other algorithms in Gonnet's work has been done with the mdcore package [1]. The package contains the core code for simple MD simulations. The library has not been changed since 2017 and the documentation is currently inaccessible.

Additionally, papers describing AutoPas have been important in understanding the library and the context of this work. [GST+19] [Gra21]

Some MD simulation packages implement neighbor lists in the sense of standard global Verlet. For example the HOOMD package[2] and the LAMMPS package[3] implement the traditional Verlet neighbor lists. They support different particle types with different cutoffs. Thus, it is possible to use multiple neighbor lists for the different pairs' cutoffs. However, these projects do not implement anything related to the pairwise Verlet algorithm discussed in this work.

[1] `https://github.com/AndySomogyi/mdcore`
[2] `https://hoomd-blue.readthedocs.io/en/stable/nlist.html`
[3] `https://lammps.sandia.gov/doc/neighbor.html`

Part II.

Implementation

4. Neighbor lists for `VerletListsCells`

In this chapter we discuss the refactoring of the VerletListsCells (VLC) container. This process facilitates the implementation of the pairwise Verlet algorithm, which is the main motivation for this work, in Chapter 5. Section 4.1 explains the main characteristics of the VLC container in terms of data structure and specific implementation, and Section 4.2 delves into the refactoring of the container.

4.1. VLC container

4.1.1. Overview of the data structure

The VLC container builds on top of the Verlet lists idea (Subsection 2.2.1) by once again only searching through the sphere defined by $r_c \cdot s$ around a particle for its potential partners. However, it uses the principle of the linked cells algorithm to store particles and their list of partners, also called neighbor lists. More specifically, the container saves particles in their respective cells, but it attaches a global neighbor list to each particle. The global neighbor list contains all potential partners of the given particle, regardless of which cell they belong to, in the same manner as the Verlet lists algorithm. Figure 4.1 illustrates the structure of a base cell's neighbor lists for a specific scenario.

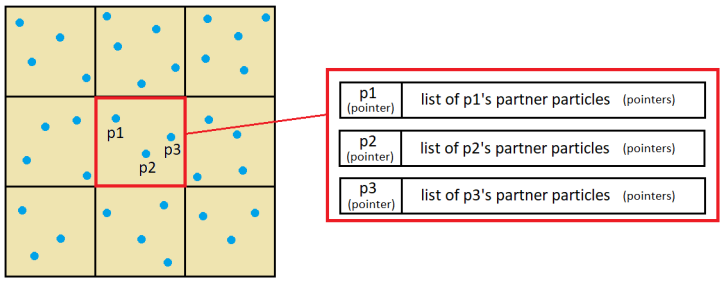


Figure 4.1.: Visual representation of the `VerletListsCells` container and its neighbor lists

In summary, a given particle and its attached neighbor lists are stored according to the cell that particle belongs to, however the partner particles are stored collectively in the particle's neighbor list, regardless of their location.

4.1.2. Implementation before the refactoring

In this subsection we will discuss the implementation of the VLC container before the refactoring was done.

The VerletListsCells class inherits from the VerletListsLinkedBase class (Subsection 2.2.1). This means the Verlet lists neighbor search algorithm is implemented in its improved version. Furthermore, while the actual particle data is stored in an internal linked cells data structure, the neighbor lists discussed in this and further sections store particle pointers. In the case of the VLC container, the neighbor lists are represented by a 2D vector of pairs, where a pair consists of a particle pointer and its neighbor list.

As already mentioned, a Verlet lists-based container has two main functionalities, namely the periodic rebuilding of the neighbor lists and the pairwise iteration for force calculations, represented respectively by the methods rebuildNeighborLists() and iteratePairwise(). In the case of this container, the rebuilding method iterates over the neighbor lists data

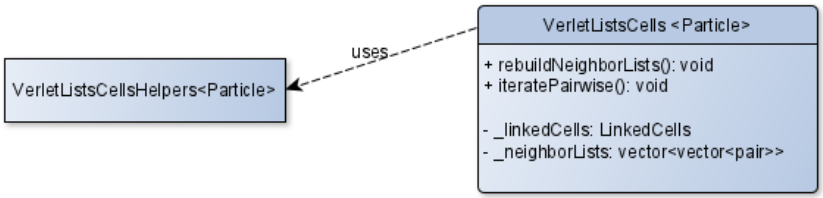


Figure 4.2.: Class diagram of VerletListsCells before refactoring

structure and reserves memory for each particle and its neighbor list. Afterwards, a generator functor is initialized and passed to a build traversal, which is responsible for iterating over the domain and generating the neighbor lists. The traversal employs the generator functor the same way it would use a force calculation functor, as discussed in Subsection 2.2.3. However, instead of calculating the force between given particles $p1$ and $p2$, the interaction in the AoS functor consists of particle $p2$ being inserted into particle $p1$'s neighbor list. The implementation of an SoA functor for list generation is discussed in Section 6.2, as it had not been implemented yet at that point.
The iteratePairwise() method takes a traversal pointer as an argument and applies it if it is suitable for the current container. The method attempts to cast the traversal argument to a VLCTraversalInterface type. If the cast is successful, then the given traversal is compatible with the VLC container. Otherwise, an exception is thrown. The pairwise iteration is delegated to the body of the traversal and that is where the force calculation is carried out.

4.2. Refactoring

The first step in this project was refactoring the existing VLC container. Our goal was to abstract the implementation of the neighbor lists structure and allow the container to work with other neighbor list types as well.
Decoupling the neighbor lists from the container is desirable, as it allows greater code reuse and general simplicity when new functionality is added in the future. The pairwise Verlet algorithm has significant similarities to the VLC container, which would allow us to add that algorithm as a second neighbor list type to this container. The algorithm and how it compares to VLC will be discussed in Chapter 5. Furthermore, this opens the possibility

for future improvements. Varying the neighbor list would have an advantage over adding a separate container because switching the neighbor list would be simpler and more efficient than instantiating a new container altogether.

4.2.1. Neighbor lists

As discussed in Subsection 2.2.1, AutoPas containers have a template argument for the type of particle. Therefore, we started by adding a second template argument to indicate the type of neighbor list. The functionality related to the existing neighbor lists was extracted into a new class `VLCAllCellsNeighborList`. Similarly, the generator functor was moved from an existing helper class for VLC to a new `VLCAllCellsGeneratorFunctor` class. The names of the classes indicate that a particle's partners are stored in a global neighbor list, as discussed in Section 4.1.

An interface class for the VLC neighbor list was created to define a clear structure and facilitate the addition of other neighbor lists. The general architecture is illustrated on Figure 4.3 and includes a default destructor and methods to build the neighbor lists. The

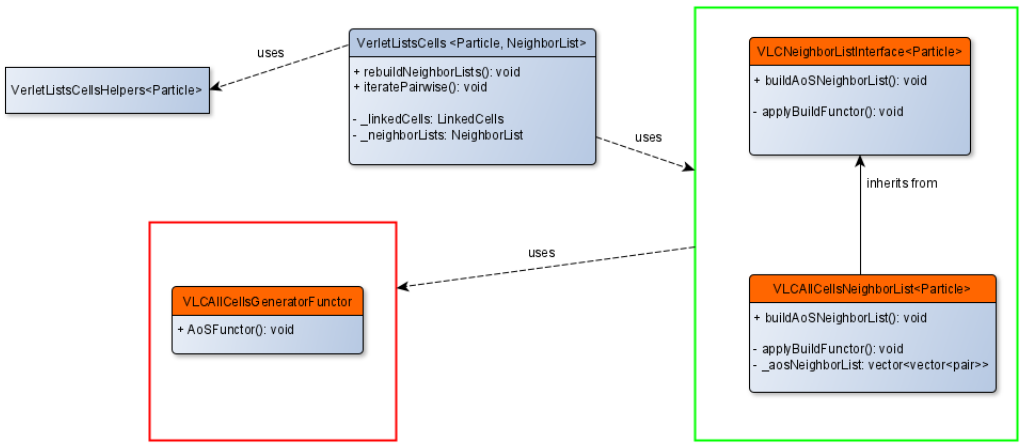


Figure 4.3.: Class diagram for the `VerletListsCells` container. The red group signifies the generator functors and the green group shows the neighbor list hierarchy. The orange components are the ones added in this iteration.

2D vector for the neighbor lists is moved from the VLC container to the new neighbor list as a private attribute and a getter method is added to allow access from the outside. These components will be added to each neighbor list separately instead of being inherited from `VLCNeighborListInterface`, since the type of the neighbor list data structure depends on the neighbor list class and cannot be defined in advance.
The `buildAoSNeighborList()` method substitutes the first part of VLC's implementation of `rebuildNeighborLists()`, namely initializing and reserving memory in the neighbor lists. This method makes an internal call to the private method `applyBuildFunctor()` which deals with creating and applying the generator functor as the second component of the building of the lists.

4.2.2. Main container

The VLC container preserves its structure inherited from `VerletListsLinkedBase`. We add an object of type `NeighborList` to use the functionality of the neighbor lists. Although the only neighbor list type possible at that point is `VLCAllCellsNeighborList`, this allows us to add a second neighbor lists type in the next section and use it with the same container. VLC implements `rebuildNeighborLists()` in a facade-like manner by delegating the call to the respective neighbor list's `buildAoSNeighborList()` method. Since the compiler generates code for all possible types that match the `NeighborList` template argument, the correct implementation of that method will be found through the template argument type deduction. The `iteratePairwise()` method remains unchanged for now as the main functionality is located in the respective traversal classes.

5. Pairwise Verlet neighbor list

5.1. Motivation

The main idea of the pairwise Verlet algorithm is to improve on the global Verlet lists (VL) algorithm (Subsection 2.2.1). VL does not work well for shared-memory parallel simulations, because memory is accessed in a scattered way and that causes excessive cache misses. For the VL algorithm particles are stored in one global list regardless of their location in the domain, and the same goes for each particle's partners. Therefore, iterating through a given particle's neighbor list requires nonuniform jumps in memory, which could be avoided if the particles are stored cell-wise. Moreover, in a shared-memory parallel program, as the name suggests, threads share a memory bus and higher-level caches. Thus, transferring information between cores is costly and that amplifies VL's memory access issue. [Gon12]
In the context of AutoPas, the pairwise Verlet algorithm improves on other neighbor selection algorithms as well. For instance, the `VerletListsCells` container, discussed in Chapter 4, stores particles in a cell-wise manner, but the neighbor lists contain partners from multiple neighbor cells similarly to the global VL algorithm. Since the partner particles are stored regardless of their location, algorithms which require knowing which cell a partner particle belongs to cannot be implemented. For example, the c08 traversal base step (Subsection 2.2.3) relies on interactions between specific cells which do not involve the base cell. That would require calculating the force only for the partners located in a particular cell and updating the rest of the neighbor list in other base steps. The pairwise Verlet lists algorithm gives us that additional information. We have already discussed the advantages of VL over linked cells in terms of search volume, so this container retains the advantages of the VL technique while additionally improving data locality.
Last but not least, this algorithm has not been explored a lot by previous works and it is curious for us to analyze its advantages and drawbacks for various scenarios.

5.2. Algorithm

The essence of this algorithm is that it creates local Verlet lists which describe only the interaction between a given pair of cells. The lists are constrained to particles within this pair of cells instead of particles scattered across multiple neighboring cells. The pseudocode in Algorithm 1 describes the procedure of building a pairwise neighbor list for a pair of cells $cellA$ and $cellB$.

If a base cell $cellA$ is interacting with $cellB$, each particle from $cellA$ is paired with a neighbor list filled with its potential interaction partners from $cellB$. The individual neighbor lists for each particle of $cellA$ form the local Verlet list for this pair of cells. [Gon12]
In the building phase a suitable traversal is chosen to iterate over the domain and create such neighbor lists for each pair of neighboring cells. If newton3 optimization is turned off,

Algorithm 1: Building pairwise neighbor lists - pseudocode

Input: base cell A and partner cell B
Output: local Verlet list for the pair of cells A and B

```
1 Function buildPairwiseVerletLists(cellA, cellB):
2     for particle i ∈ cellA do
3         for particle j ∈ cellB do
              // Checks if distance between two particles is within r_c · s
4             if dist(i, j) < r_c · s then
5                 neighborList[i].store(j);
6     return neighborList
```

an interaction needs to be carried out in both directions. In that case the interaction in Algorithm 1 needs to be carried out once more, this time with *cellB* as the base cell and *cellA* as its neighboring cell. If the optimization is turned on, then the traversal defines for each interaction between two cells, which one is the base cell, whose particles own the neighbor lists, and which cell is the neighbor, whose particles are inserted into the base cell's neighbor lists.

5.3. Implementation

The implementation of this algorithm was realized in the `VLCCellPairNeighborList` class, which inherits from `VLCNeighborListInterface`. The neighbor list class should be a second possible match to the `NeighborList` template argument of the VLC container, an alternative to `VLCAllCellsNeighborList`. We will follow the structure of the interface explained in Chapter 4 to describe the implementation.

5.3.1. Rebuilding

The data structure to store the neighbor lists is a three-dimensional vector of pairs. The first two dimensions define the base cell and its current partner cell. For each cell pair, the third dimension defines a vector of pairs. The pairs consist of a particle pointer and its neighbor list, where the main particle is from the base cell and the neighbor list contains particles pointers from the partner cell. A graphical illustration of the data structure can be seen on Figure 5.1.

Rebuilding of the neighbor lists

Using the same concept as Section 4.2, the `buildAoSNeighborList()` method deals with initializing the data structure and reserving memory in advance in order to make the filling of the list more efficient.
We begin by defining the size of the neighbor list data structure in each dimension. The first dimension has the same size as the number of cells of the domain. The second dimension requires an estimation of how many neighbors each cell has, including itself. In the standard

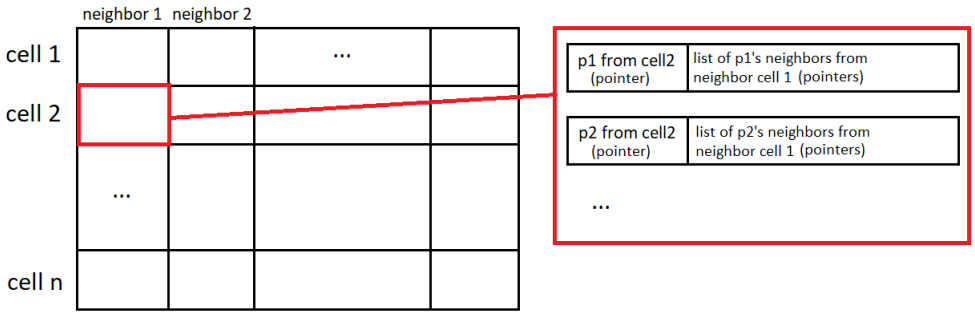


Figure 5.1.: Sketch of pairwise Verlet's data structure

case a cell has 26 neighbors (apart from itself), a 3 x 3 x 3 block without the base cell in the center. However, AutoPas allows a cell's side length to be less than $r_c \cdot s$, which would change the number of cells a base cell interacts with. Thus, we count how many cells fit in the sphere defined by $r_c \cdot s$. The result shows how many cells should be reserved in the second dimension, for every base cell. If the newton3 optimization is on, the result can be halved and the interaction of the base cell with itself also has to be accounted for. Finally, for each cell pair we create a vector of the same length as the number of particles in the base cell. In the vector we insert a pair for each particle of the base cell, where the first element of the pair is a pointer to that particle and the second element is an empty vector, later to be filled with neighbor pointers. Figure 5.1 provides an overview of the 3D vector and its size in each dimension.

Since each base cell has its own vector of neighboring cells, the local indices for each of these vectors (0 to number of neighboring cells) need to be mapped to the actual domain cells. In other words, the program has the global indices of the base cell and its partner cell and it needs to know where to find the corresponding neighbor list, or what the local index of the partner cell is in the base cell's vector. For this we use a vector of maps which saves the dependency between a local neighbor index and the corresponding global cell index for each base cell. Additionally, we have a map from particle to cell which is necessary in the neighbor lists generation.

Neighbor lists generation

After the reservation phase, the initialized vector is sent to the new generator functor class `VLCCellPairGeneratorFunctor`. The generator functor works similarly to the `VLCAllCellsGeneratorFunctor`. The AoS functor includes an interaction between two particles i and j, where the maps allow us to find the cells these particles belong to and consequently the cell pair's neighbor lists. Particle j is added to particle i's neighbor list, attached to the corresponding cell pair. An SoA functor for neighbor list generation is discussed in Section 6.2.
As already mentioned in Section 5.2, the way the lists are built is influenced by the build traversal used and the newton3 optimization. In our implementation we have chosen lc-c18

as a build traversal because of its simplicity and because of the guarantee that the base cell will always have an index smaller than or equal to the index of its partnering cell. Consequently, for a pair of interacting cells, the smaller index will store the neighbor lists, while the partner particles will come from the neighboring cell with the larger index. If the newton3 optimization is off, the traversal will call the AoS functor again with the particles switched, thus both will be contained in each other's neighbor lists.

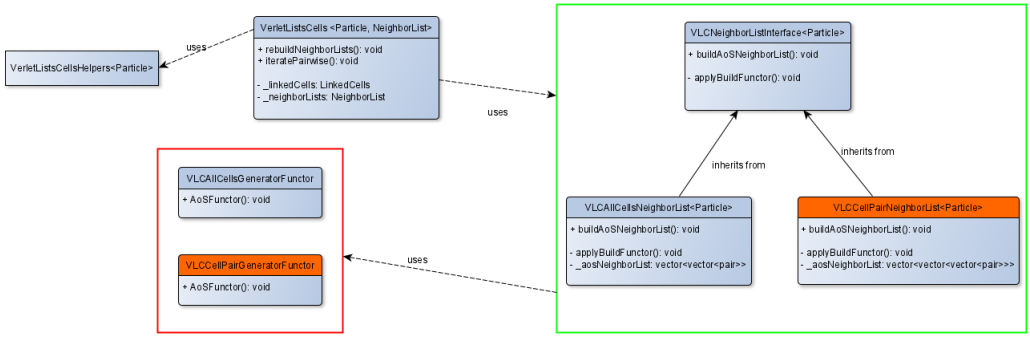


Figure 5.2.: Class diagram for the new pairwise Verlet container. The red group signifies the generator functors and the green group shows the neighbor list hierarchy. The orange components are the ones added in this iteration.

5.3.2. Force calculation

The force calculation is carried out in `VLCTraversalInterface`. Similarly to the VLC container, the loop body implemented in the respective VLC traversal subclass is responsible for the force calculation of a given base cell. For the pairwise Verlet neighbor lists the base step consists of iterating over all of a given base cell's neighbors and through the respective particle-neighbor list pairs. Each particle pair is then sent to the AoS functor of the respective force calculation functor where the force between them is computed. Since all VLC traversals so far were based either on the c18 (newton3 on) or the c01 (newton3 off) base step, no further iteration variants are required. Listing 5.1 is a code snippet which contains the base step implementation for a given cell index.

```
1  void processCellListsImpl(VLCCellPairNeighborList<Particle> &neighborList,
       unsigned long cellIndex, PairwiseFunctor *pairwiseFunctor){
2      auto &internalList = neighborList.getAoSNeighborList();
3        for (auto &cellPair : internalList[cellIndex]) {
4          for (auto &[particlePtr, neighbors] : cellPair) {
5            Particle &particle = *particlePtr;
6            for (auto neighborPtr : neighbors) {
7              Particle &neighbor = *neighborPtr;
8              pairwiseFunctor->AoSFunctor(particle, neighbor, useNewton3);
9            }
10         }
11       }
12     }
```

Listing 5.1: Force calculation iteration for one base cell

6. SoA for VLC container

We have discussed the advantages of the SoA data layout in Subsection 2.2.2. In summary, the SoA data layout is beneficial for cache efficiency in the force calculation phase and is more appropriate for vectorization than AoS. The VLC container and all additions and changes from chapters 4 and 5 only included the AoS data layout. In this chapter we discuss the implementation of the SoA data layout for both neighbor lists of the VLC container. Section 6.1 describes the representation of the neighbor lists in an SoA data layout and section 6.2 explains how the neighbor list generation can be implemented in an SoA functor which generates the neighbor lists in AoS from an internal linked cells SoA representation.

6.1. Neighbor list in SoA layout

6.1.1. Rebuilding

We have established that the particle data in AutoPas is stored in an AoS data layout by default. The data structure can be converted to SoA layout for the force calculation, and then transformed back to AoS.
Using the same principle, we add a method to the VLC neighbor list classes which generates SoA neighbor lists from the already filled AoS neighbor lists. The SoA neighbor lists thus have the same multi-dimensional vector structure as the AoS ones, albeit with a few differences. First, a particle is stored through its index instead of a particle pointer in order to be compatible with the force calculation functors for SoA. The force calculation will be described in more detail in Subsection 6.1.2. Additionally, an aligned allocator has to be added so that the particle data is alilgned with cache-line boundaries.

The `generateSoAFromAoS()` method iterates over all dimensions of the AoS neighbor lists and allocates the same amount of memory for the SoA neighbor lists. Inside the 2D vector for `AllCells` or 3D vector for `CellPair` respectively, the method adds the particle indices corresponding to the particle pointers from the same spot in the AoS neighbor lists. To make that step possible, we iterate over the particles in advance and initialize a map of particle pointers to global particle indices.
If the data layout is SoA, `generateSoAFromAoS()` is called from `rebuildNeighborLists()` in the VLC container after the call to `buildAoSNeighborList()`. This guarantees that the AoS neighbor lists will have been filled and the conversion to SoA will thus be successful.

6.1.2. Force calculation

In each traversal, the data gets loaded into an SoA object before the iteration and extracted afterwards. The SoA object stores attributes selected in advance for each particle. The particle data is stored in an SoA format in that SoA object, and the indices from the neighbor lists are used to fetch the data for the correct particles.

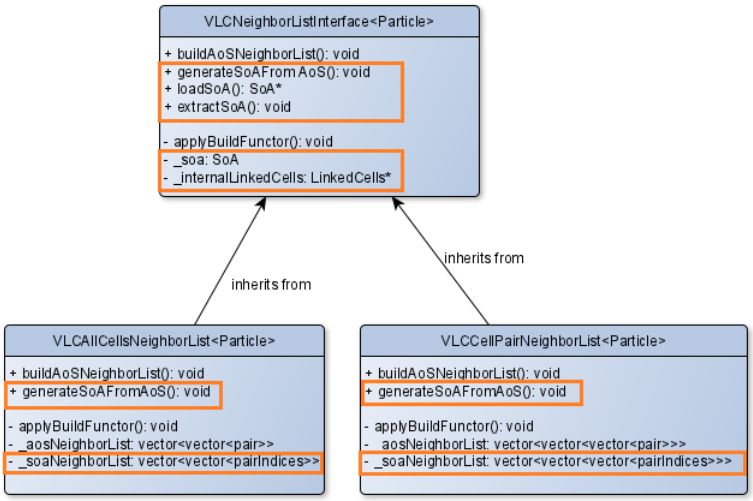


Figure 6.1.: Class diagram for the new SoA components. The newly added methods are colored in orange.

The force calculation can be done with the existing `SoAFunctorVerlet` which was created to compute the interaction between a particle and its neighbor list in the SoA format. It takes as arguments the loaded SoA object, the main particle's index, a neighbor list which is also filled with indices, and a newton3 option. The functor fetches the data of the neighbors from the SoA at the respective indices and pushes them into a buffer meant specifically for the neighbors. Afterwards, this buffer can be vectorized and the force is computed. [Gra21] The SoA functor makes the neighbor lists iteration simpler than the AoS version, as we have one less level of loop nesting and do not need to iterate over the particles of each neighbor list, instead we directly pass the whole neighbor list to the `SoAFunctorVerlet`.

6.2. List generation

Until this point we had generated the neighbor lists using the generator functor's `AoSFunctor`. It used particle information in AoS format to fill neighbor lists, also in AoS format. This section explains how to generate AoS neighbor lists from particle information in SoA format instead. Apart from the idea that using the SoA representation might be more efficient, we also want to maintain consistency among all functors and provide both an AoS and an SoA implementation.
In our generator functors for `AllCells` and `CellPair` we implement `SoAFunctorSingle()` and `SoAFunctorPair()`, which are inherited from the `Functor` class and take `SoAView` objects as arguments. An `SoAView` object provides a "window" view on the SoA between two given indices, which is used in our case to represent a cell. The `SoAFunctorSingle()` implements the interaction of a cell with itself and `SoAFunctorPair()` handles the interaction between two cells.
The pipeline goes as follows:

1. The particle data is loaded from the internal linked cells structure into SoA format.

The build traversal lc-c18 takes care of this, hence we do not need to implement a conversion. It gives the `SoAViews` to the generator functor for each pair of interacting cells according to the c18 base step and newton3 option.

2. The two cells in SoA format interact. Similarly to the AoS list generation, these SoA functors generate neighbor lists for a cell interacting with itself or for a pair of interacting cells. For each particle from the first cell, all particles from the second cell are checked for distance and inserted into the first particle's neighbor list if in range. The particle pointers for the neighbor lists are fetched from the SoA representation which includes 4 attributes, namely the pointer and the three positions to perform the distance check.
3. The build traversal handles the extraction of the SoA data, thus converting back to AoS.

This gives us two ways to generate AoS neighbor lists, either from an AoS or an SoA particle representation. This is also reflected in the VLC container, where we add a constructor argument to indicate which data layout will be used for the building phase.

7. c08 traversal

We have discussed the advantages of c08-based traversals over other domain colorings in Subsection 2.2.3. Furthermore, in Section 5.1 we explained why `VLCCellPairNeighborList` would allow the implementation of such a traversal, while the `AllCells` neighbor list would not. This section describes the implementation of the c08 traversal for the pairwise Verlet neighbor list.

The core idea of the c08 base step remains the same: interactions from the c18 base step are moved to another cell's base step to reduce the number of cells that need to be locked per base step. If we have the indices of all pairs of cells which should interact in a base step, we can implement this traversal by fetching the respective neighbor lists from the 3D vector and passing them through a force calculation functor as usual.
We have to define a separate iteration loop from the one in `VLCTraversalInterface`. We want to include the necessary interactions which take place outside of the base cell and exclude the pairs which are making the interaction block bigger than 2 x 2 x 2.

The `VerletListsLinkedBase` container uses an underlying linked cells data structure and the linked cells container has its own implementation of a c08 traversal. This allows us to reuse some of the lc-c08 implementation for this traversal. The lc-c08 implementation includes the main traversal class and an `LCC08CellHandler`. The cell handler computes which offsets should be added to given base cell's index to find all of the interacting pairs for a base step. These offsets are only computed once and can be applied to any base cell. Thus, we can reuse these offsets in our traversal, since the mechanism does not change. We create a traversal class `VLCCellPairC08Traversal` and our own cell handler which inherits from `LCC08CellHandler`. Figure 7.1 describes the class structure of the new c08 traversal and all related classes.

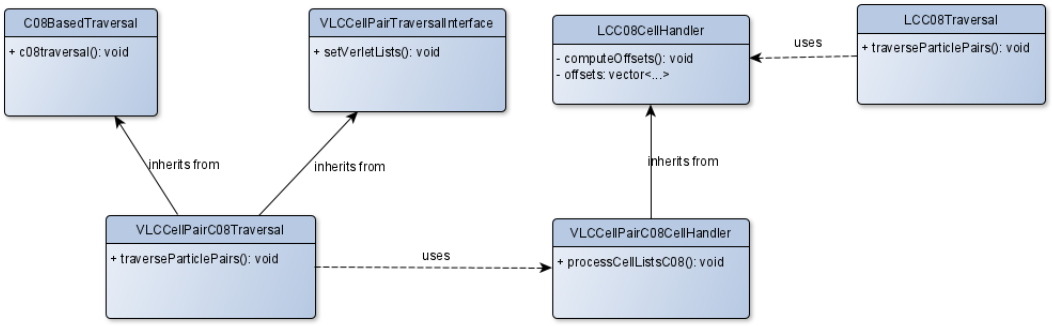


Figure 7.1.: Class diagram for the addition of the c08 traversal for `VLCCellPairNeighborList`

`VLCCellPairC08Traversal` inherits from `C08BasedTraversal`, explained in Subsection 2.2.3, and implements `traverseParticlePairs()` as follows:

1. We load the SoA object at the beginning of the iteration, if the data layout requires it.
2. A c08 traversal is triggered. We pass a loop body to it, which calls the cell handler's `processCellListsc08` for a given cell index.
3. The filled SoA object is emptied, if the data layout is SoA.

The cell handler's processing method reuses computed offsets from the `LCC08CellHandler` super class. For a given base cell, we iterate over the offset pairs and add them to our base cell index. Thus, the interacting cell pairs' indices are computed and we can fetch the local Verlet lists for this pair. Afterwards, as usual, we either pass each particle pair to the `AoSFunctor` or give the whole neighbor list and the main particle index to `SoAFunctorVerlet`.
If the newton3 optimization is off, we have to switch the cells and perform the force computation again for each pair of different cells.

Since this traversal is only applicable to `VLCCellPair` but not `VLCAllCells`, we made a new traversal interface to help us check for container compatibility. The new class we created is called `VLCCellPairTraversalInterface` and it will be used as a base class for traversals which are only compatible with the `VLCCellPair` neighbor list. Consequently, `VLCTraversalInterface` will be used for traversals which are compatible with both neighbor lists.
We change the implementation of `rebuildNeighborLists()` to accommodate for the different traversal interfaces and the difference between the neighbor lists. We add a method `setUpTraversal()` to each neighbor list class where the casts to the respective traversal interfaces are done. The `AllCells` neighbor list keeps the old implementation because nothing has changed. The `CellPairs` implementation first attempts to cast to the new `VLCCellPairTraversalInterface` interface - it is successful only if the traversal is c08 for `CellPair`. If it does not work, we try casting to `VLCTraversalInterface` and if that does not work either, then the traversal is incompatible we throw an exception.

Part III.

Results

8. Testing setup

We have tested the experimental scenarios on the CoolMUC-2 LRZ cluster. It has 812 nodes and each node supports up to 28 threads. Each CPU has 14 cores and the 28 threads are a result of hyperthreading, which allows 2 threads per core.
We have tested with one, four, eight, 16, and 24 threads. Independently of the number of cores, we decided to spread out the numbers of threads evenly and add a point for four threads to gain more insight into the initial speedup.

We have set two main scenarios, a grid and a gaussian random generator. The scenarios are tested with the $delta\_T$ parameter set to 0, which means the particles do not move and the measurements are more consistent among the iterations. The experiments last 100 iterations to avoid any random variances. The standard experiment has 125000 particles. For the grid scenario this translates to 50 particles per dimension. The standard particle spacing of 1.225 makes the domain be automatically generated with a side length of 56.12505. The average density for this scenario is about 71% (number of particles divided by volume). The gaussian scenario has the same amount of particles. We tuned the domain size to result in a similar single-threaded performance to the grid scenario in order for the two to be comparable. The result is a domain side length of 92 with a mean of 46 and a standard deviation of 18.5. The resulting average density is approximately 16%.
We have included a secondary scenario for some cases where the original scenario was considered too small. The new scenario has a million particles. For the grid this results in a domain side length of 112.25025. For the gaussian scenario the domain side length was increased to 185 with a mean of 92.5 and a standard deviation of 37 in order to keep a similar density.

The experiments were done for two values of r_c, namely 5 and 3.5. The bigger radius is on the high end of the range of usually used values of r_c. 3.5 is a representative of smaller to medium r_c. We used a skin factor of 1.2. The tested traversals are our familiar variations of domain coloring: c01 (without newton3 optimization) and c18 and c08 with a newton3 optimization. We mainly compare the performances of `VerletLists` and the VLC containers with its two neighbor lists, `VLCAllCells` and `VLCCellPair`. For `VerletLists` we test its only traversal vl-list-iteration as an equivalent to c01. The performance of the `LinkedCells` container was also measured, although it usually performs differently and incomparably to the Verlet lists-based containers. We will discuss results for the AoS and the SoA data layouts.
The runtime measurements described in the next sections refer to the force calculation time in non-tuning iterations. The neighbor lists were only built once in the beginning of the simulation. The time used to build the neighbor lists was recorded and subtracted from the timer to be able to truly compare the efficiency of the containers.
For most of the experiments the measurements were done with five samples and trimmed

mean of the middle three elements was taken to make sure outliers are not shifting the result too much. For some later experiments three samples were taken and their mean was recorded.

9. Performance in AoS data layout

The `LinkedCells` container is generally much slower than `VerletLists`, `VLCAllCells` and `VLCCellPair`. Figure 9.1 illustrates the scale of their difference.

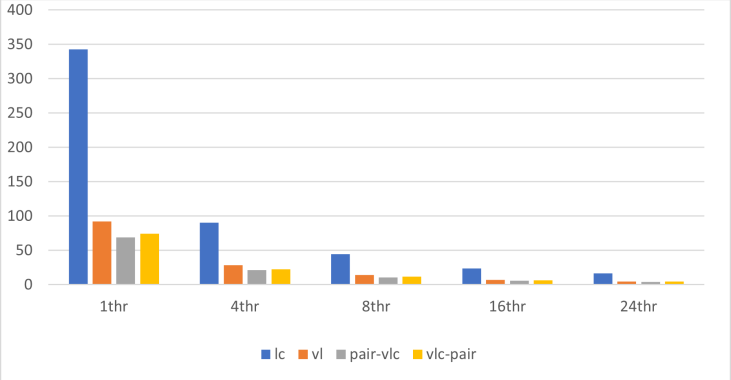


Figure 9.1.: Runtime comparison of containers including `LinkedCells` for a grid scenario with 125000 particles with c01-based traversal and $r_c = 5$

Thus, this chapter will analyze and compare the performance of the other three containers in more depth.

9.1. c01 traversal

9.1.1. Observations

Grid generator

We compare the runtime performance of the three chosen containers for a homogenous grid scenario. The bigger radius $r_c = 5$ yields `VLCCellPair` as a clear leader in runtime speed, as we can see on Figure 9.2. It has a speed advantage of 5 to 7% over `VLCAllCells`. For example, a large relative difference can be seen for 24 threads, where `VLCAllCells`'s computation lasts 4.47s compared to 4.19s for `VLCCellPair`.
`VLCCellPair` also achieves a speedup of more than 30% over `VerletLists` for a small number of threads. For eight threads, `VerletLists` needs 14.4s while `VLCCellPair` makes the computation in 10.95s. However, `VerletLists` has a greater speedup and it gets closer to the other containers' performance for a rising number of threads. For 24 threads, `VerletLists` performs the computation in 4.92s compared to `VLCCellPair`'s 4.19s, which makes the speedup achieved by `VLCCellPair` only 17%. The total speedup over 24 threads is 18.78 times for `VerletLists` and 16.55 and 16.6 times respectively for `VLCCellPair` and `VLCAllCells`.

For a smaller r_c of 3.5, `VLCCellPair` and `VLCAllCells` perform similarly and we cannot definitively determine which container is more efficient. The runtime performance graph on Figure 9.3 shows virtually no difference between the blocks in grey and yellow. For eight threads `VLCCellPair` computes for 5.71s and `VLCAllCells` is slightly faster with 5.65s, while for 16 threads the runtime is respectively 3.06s and 3.12s.

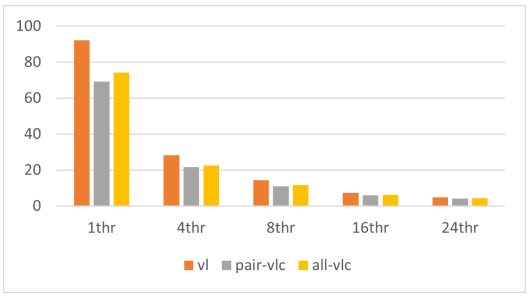


Figure 9.2.: Runtime comparison of containers for a grid scenario for 125000 particles with c01-based traversal and $r_c = 5$

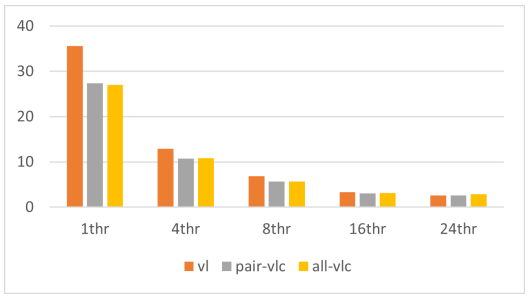


Figure 9.3.: Runtime comparison of containers for a grid scenario for 125000 particles with c01-based traversal and $r_c = 3.5$

`VerletLists` once again has the best total speedup of 13.8 times, compared to `VLCCellPair` and `VLCAllCells`'s 10.5 and 9.4 respectively. While the speedups are lower than the ones for $r_c = 5$, the differences between them are more pronounced. A visual representation of the speedups for both values of r_c can be seen on Figures Figure 9.4 and Figure 9.5. The lines on Figure 9.5 show a more drastic difference between the speedups above eight threads. `VLCCellPair` has a 30% speed advantage over `VerletLists` for one thread where the runtimes are 35.62s and 27.36 s respectively. However, the difference shrinks with a rising number of threads and `VerletLists` reaches `VLCCellPair`'s speed for 24 threads with 2.58s and 2.6s respectively.

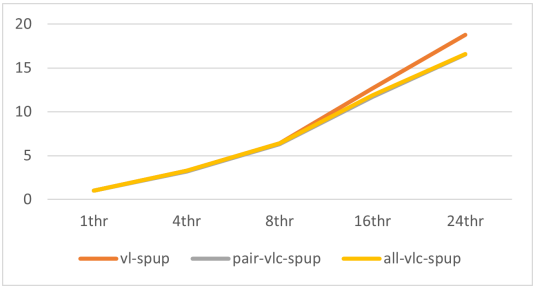


Figure 9.4.: Speedup comparison of containers for a grid scenario for 125000 particles with c01-based traversal and $r_c = 5$

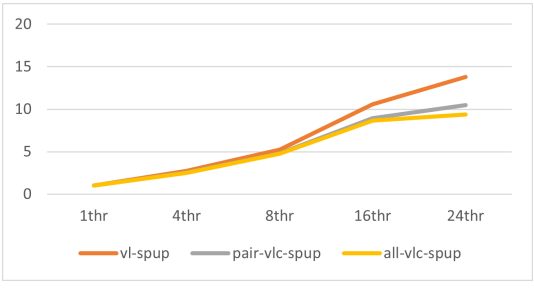


Figure 9.5.: Speedup comparison of containers for a grid scenario for 125000 particles with c01-based traversal and $r_c = 3.5$

Gaussian generator

Since the gaussian scenario is inhomogenous, similar characteristics can be seen albeit generally weaker than the grid scenario.

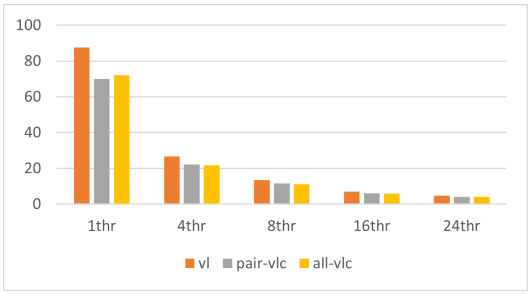


Figure 9.6.: Runtime comparison of containers for a gaussian generator scenario for 125000 particles with c01-based traversal and $r_c = 5$

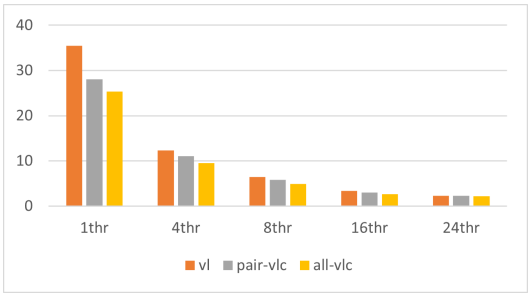


Figure 9.7.: Runtime comparison of containers for a gaussian generator scenario for 125000 particles with c01-based traversal and $r_c = 3.5$

For $r_c = 5$ `VLCAllCells` has the best performance with a very small advantage over `VLCCellPair` of under 5% for more than one thread. This can be seen on Figure 9.6, where the grey block is slightly lower than the yellow one for one thread and slightly higher for all other numbers of threads. Nevertheless, `VLCCellPair` maintains an improvement on `VerletLists` from 25% for one thread (runtime of 87.64s and 69.88s respectively) to about 10% for 24 threads (4.64s and 4.18s). This decrease is once again due to `VerletLists`'s superior parallelization speedup of 18.9 times.
A $r_c = 3.5$ makes this pattern even clearer with `VLCAllCells` consistently leading in performance, as can be seen on Figure 9.7. `VLCAllCells` performs the same computation for 85 to 90% of `VLCCellPair`'s runtime for up to 16 threads. However, for a higher number of threads `VLCCellPair` starts catching up. The runtime for `VLCCellPair` and `VLCAllCells` for 24 threads is 2.32s and 2.19s respectively.
Once again, `VerletLists`'s superior speedup allows it to catch up to `VLCCellPair` for a relatively high number of threads.

9.1.2. Analysis

Generally, `VLCCellPair` seems to be the most efficient choice when the neighbor lists contain a lot of particles, in this case for a higher r_c and a denser grid scenario. Increasing the skin factor or the particle density would have a similar effect. Consequently, a lower r_c and a lower density make `VLCAllCells` prevail over `VLCCellPair`. `VLCCellPair` is less beneficial in the case of shorter neighbor lists possibly because of the higher complexity of its data structure. The iteration requires an extra nested loop level in comparison to `VLCAllCells`, which could be inefficient for lists consisting of only a few particles. Additionally, large lists could be more suitable for `VLCCellPair` because in `VLCAllCells` the partner particles have to be fetched from different cells, which is not the case for `VLCCellPair`. However, this tradeoff could be lost when there are not a lot of particles in these neighbor lists to begin

with.
`VerletLists` is more efficiently parallelizable than both containers because its parallelization strategy is based on distributing single neighbor lists among the threads rather than cells containing multiple neighbor lists. However, `VLCCellPair` still performs better in most cases possibly due to its more efficient memory access discussed in Chapter 5.

9.2. c18 and c08 traversals

In this section we compare the performance of `VLCAllCells` and `VLCCellPair` for the c18 traversal and add `VLCCellPair`'s c08 traversal to the experiment. In comparison to the c01 section, we utilize the newton3 optimization. The `VerletLists` container does not support the use of the newton3 optimization, so including it in this section would be unnecessary.

9.2.1. Observations

Grid generator

The grid scenario once again shows good results for `VLCCellPair`.

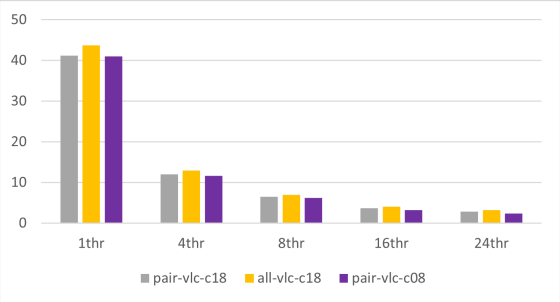


Figure 9.8.: Runtime comparison of containers for a grid scenario for 125000 particles with c18 and c08-based traversals and $r_c = 5$

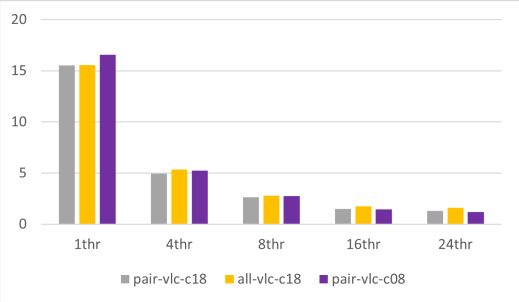


Figure 9.9.: Runtime comparison of containers for a grid scenario for 125000 particles with c18 and c08-based traversals and $r_c = 3.5$

For a r_c of 5 `VLCCellPair`'s c18 traversal is consistently performing better than the equivalent traversal for `VLCAllCells`. A visual representation can be seen on the grey and yellow blocks in Figure 9.8. The advantage of `VLCCellPair` rises from 6-7% for a low number of threads to 13% for 24 threads. The runtime for eight threads is 6.48s and 6.91s for `VLCCellPair` and `VLCAllCells` respectively. For a higher number of threads `VLCCellPair`'s speedup rises faster than `VLCAllCells`'s speedup, reaching 14.6 and 13.7 respectively. Consequently, for 24 threads the runtimes are 2.82s and 3.2s for `VLCAllCells`. To confirm the results of this experiment, we conducted a larger experiment with a million particles, as described in Chapter 8. The previous experiment included a large cutoff and skin and thus only about 1000 cells. Therefore, a domain coloring with 18 colors would not provide sufficient load for a high number of threads. This experiment shows similar results for the relationship between `VLCAllCells` and `VLCCellPair`'s performance with c18, which can be seen on

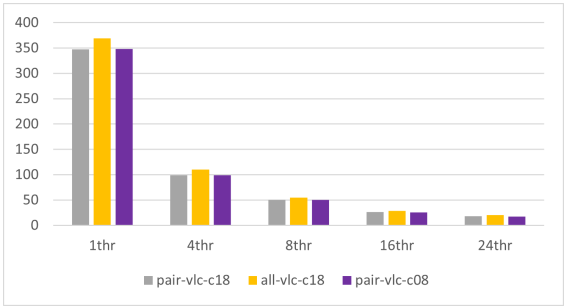


Figure 9.10.: Runtime comparison of containers for a grid scenario for a million particles with c18 and c08-based traversals and $r_c = 5$

Figure 9.10. The speedups once again increase rapidly between eight and 16 threads and end up at 19 times for `VLCCellPair` and 17.9 times for `VLCAllCells`. The runtime for eight threads is 50.39s for `VLCCellPair` and 54.34s for `VLCAllCells`, while the performance for 24 threads is respectively 18.27s and 20.58s, so the difference grows from almost 8% to over 12%.

In the next stage we add the c08 traversal for `VLCCellPair` to our experiment, which is represented as a purple block in Figure 9.8. It performs similarly to c18 for a low number of threads but brings up to 20% improvement on c18's runtime for 24 threads, where the results are 2.82s and 2.34s. Due to the relatively small number of cells in this scenario, these measurements might not be an accurate representation of the situation since c08 has more load per thread than c18.
The million particle experiment depicted on Figure 9.10 shows a slightly different pattern, where `VLCCellPair`'s c08 traversal mainly performs very similarly to c18 and has a very small advantage of under 5% for a high number of threads. The runtimes for eight threads are 50.39s and 49.97s respectively for c18 and c08, where the difference is negligible. For 24 threads we see a slightly clearer advantage of c08 with results 18.27s and 17.54s.

The usual smaller experiment with $r_c = 3.5$ is shown on Figure 9.9. For c18 the slight advantage of `VLCCellPair` is confirmed. `VLCCellPair` and `VLCAllCells` start virtually equal on a single thread with 15.51s and 15.55s. However, `VLCCellPair` scales better and the results for 24 threads are 1.28s and 1.6s, which shows a clear advantage for `VLCCellPair`. `VLCCellPair`'s c08 traversal starts out slower than c18 with 16.54s compared to 15.51s for a single thread. However, c08 catches up for a high number of threads and improves slightly on the runtime performance (less than 10%) with runtimes of 1.19s for c08 and 1.28s for c18 for 24 threads. This is a similar pattern to the million particle experiment with $r_c = 5$ which confirms the trend.

Gaussian generator

For $r_c = 5$ with the gaussian generator, the situation is similar to the grid scenario. However, the speedups are generally weaker here.

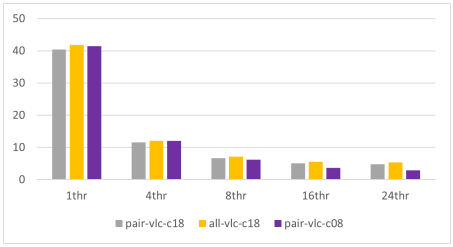


Figure 9.11.: Runtime comparison of containers for a gaussian scenario for 125000 particles with c18 and c08-based traversals and $r_c = 5$

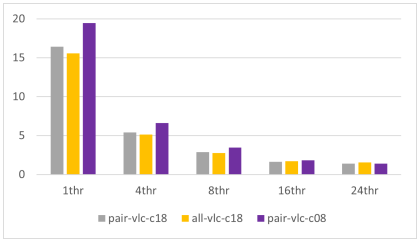


Figure 9.12.: Runtime comparison of containers for a gaussian scenario for 125000 particles with c18 and c08-based traversals and $r_c = 3.5$

`VLCCellPair` improves on `VLCAllCells` by more than 10% for a high number of threads and it speeds up slightly better as well. A visual representation of the data can be seen on fo the grey and the yellow blocks. At eight threads the runtime results are 6.65s and 7.09s for `VLCCellPair` and `VLCAllCells` respectively, which goes to 4.79s and 5.35s for 24 threads, showing the increasing advantage of `VLCCellPair`.
`VLCCellPair`'s c08 improves on c18 significantly for `VLCCellPair` and a high number of threads. This is depicted on where a noticeable difference can be seen between the grey and the purple block. The difference for 24 threads is as high as 60% with a runtime of 2.9s for c08 compared to c18's 4.79s for 24 threads. However, this might be an effect of the scenario, since the number of cells is relatively low and density is lower than the grid as well.
We compare the results to a million particle experiment with a gaussian generator, described briefly in Chapter 8. It shows a slightly weaker advantage of `VLCCellPair`'s c18 over `VLCAllCells` c18 reaching a maximum of 7% for 24 threads with runtimes 17.48s and 18.73s. However, c08's performance is slightly weaker than c18 until 24 threads where it catches up. For eight threads c08's runtime is 50s compared to c18's 47.47s with a 5% gap. For 24 threads c08 catches up with 17.26s to c18's 17.48s. A visual comparison is depicted on Figure 9.13 where the purple block is somewhat taller or equal to the grey block, in contrast to Figure 9.2.1. For $r_c = 3.5$ once again a lower density and smaller cells and neighbor lists allow `VLCAllCells` to take the lead. `VLCCellPair` catches up for a higher number of threads because of its superior speedup. Figure 9.12 illustrates the grey block starting out taller than the yellow one and eventually catching up and surpassing it. In terms of measurements, the performance for four threads is 5.4s for `VLCCellPair` and 5.14s for `VLCAllCells`, while the runtimes for 24 threads are 1.42s and 1.39s, demonstrating a small advantage or at least equivalent performance of `VLCCellPair`.
`VLCCellPair` with c08 is mostly slower than c18, as seen in Figure 9.12, and catches up with the rising number of threads. It starts out slower than c18 for a single thread with runtime of 19.47s as opposed to 16.43s. However, its high speedup allows it to catch up and become marginally better than c18 for 24 threads, namely 1.38s for c08 compared to 1.41s for c18.

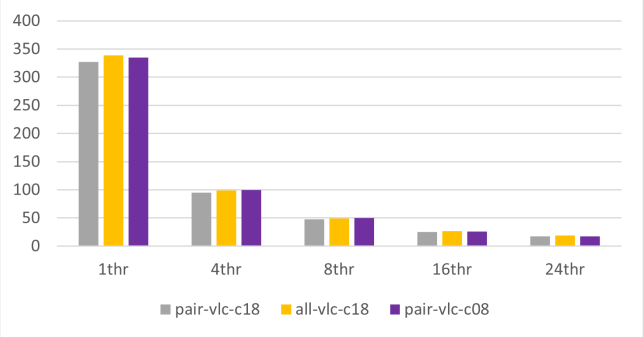


Figure 9.13.: Runtime comparison of containers for a gaussian scenario for a million particles with c18 and c08-based traversals and $r_c = 5$

9.2.2. Analysis

These results reinforce the idea that `VLCCellPair`'s advantage increases with longer neighbor lists. They also demonstrate that `VLCCellPair`'s c18 traversal parallelizes better than `VLCAllCells`'s c18 for a high number of threads. The speedup for `VLCCellPair` over 24 threads is larger even when its single-thread performance is better than `VLCAllCells`. In other words, the high speedup is not a result of a bad single-thread performance. It is worth noting that the newton3 optimization is used here, which means half as many neighbor lists are used for `VLCCellPair` in comparison to c01.
`VLCCellPair`'s c08 does not achieve a strong improvement over c18. c08 is beneficial for scenarios with a relatively small number of cells, where a c18 coloring does not give threads a high enough load.

10. Performance in SoA data layout

We test the same scenarios with the SoA data layout. It is worth noting that the list building time is significantly longer here possibly because of the conversion of the neighbor lists from SoA to AoS, which is not parallelized.

10.1. c01 traversal

10.1.1. Observations

`LinkedCells` clearly has the best runtime performance over all scenarios, as can be seen for example on Figure... . It will serve as a baseline for visual comparison of the other containers' performances.

Grid generator

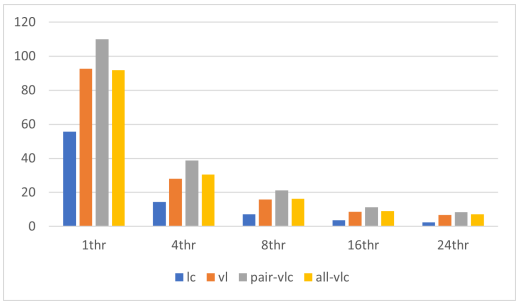


Figure 10.1.: Runtime comparison of containers for a grid scenario for 125000 particles with a c01-based traversal and $r_c = 5$ for SoA

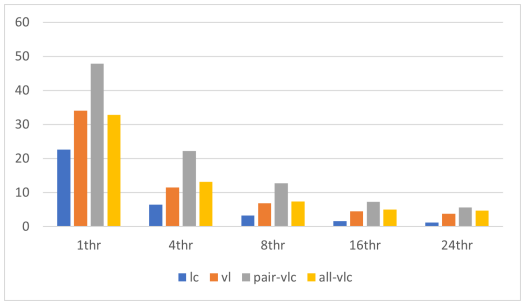


Figure 10.2.: Runtime comparison of containers for a grid scenario for 125000 particles with a c01-based traversal and $r_c = 3.5$ for SoA

For $r_c = 5$ in the grid scenario `VLCAllCells` and `VerletLists` are performing very similarly for small number of threads. `VerletLists` has a small advantage over `VLCAllCells` which fluctuates but stays under 10%. This relationship can be seen in Figure 10.3 in the yellow and orange blocks. For example, for 16 threads `VerletLists` runs for 8.72s and `VLCAllCells` runs for 8.96s, so the difference between them is inconsequential.
`VLCCellPair`, depicted as a gray block on Figure 10.3, is performing significantly slower than the other containers. `VLCAllCells` is performing the same work for 75 to 85% of `VLCCellPair`'s runtime. The largest difference in our data is for eight threads where `VLCAllCells`'s 16.2s makes up 76% of `VLCCellPair`'s 21.12s. It is worth noting that the

difference sinks slightly for 24 threads, because `VLCCellPair` has a better speedup, especially for a high number of threads. The higher speedup however might be a result of `VLCCellPair`'s worse single-thread performance of 110.07s compared to `VLCAllCells`'s 91.9s.

A r_c of 3.5 makes `VLCAllCells`'s advantage over `VLCCellPair` even larger, although it sinks again for 16 and 24 threads after reaching its highest point at eight threads. For eight threads `VLCCellPair` runs for 12.7s compared to `VLCAllCells`'s 7.39s. In contrast, `VLCCellPair`'s runtime for 24 threads is 5.64s and `VLCAllCells` runs for 4.7s.
`VerletLists` also maintains a small advantage over `VLCAllCells` which reaches its highest point for 24 threads, where `VLCAllCells`'s 4.7s are improved upon by `VerletLists`'s 3.83s.

Gaussian generator

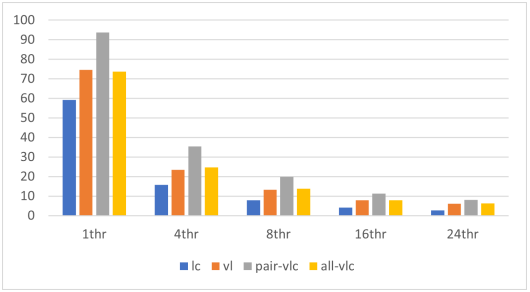


Figure 10.3.: Runtime comparison of containers for a gaussian scenario for 125000 particles with a c01-based traversal and $r_c = 5$ for SoA

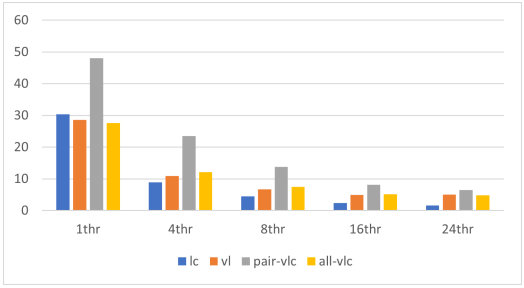


Figure 10.4.: Runtime comparison of containers for a gaussian scenario for 125000 particles with a c01-based traversal and $r_c = 3.5$ for SoA

We go on to the less dense gaussian scenario. For $r_c = 5$, it is worth noting that `LinkedCells`'s performance is slightly worse than the grid scenario in contrast to the other containers which have a slightly shorter runtime here. `VLCAllCells` is once again performing faster than `VLCCellPair` where it takes about 70% of `VLCCellPair`'s runtime and the difference decreases again for 16 and 24 threads. For eight threads `VLCCellPair` lags behind `VLCAllCells` with 19.79s and 13.75s respectively. In contrast, for 24 threads the runtimes are 6.26s and 6.1s.
`VLCAllCells` and `VerletLists` perform very similarly with no consistent winner. For example, the runtime for 16 threads is 7.92s for `VerletLists` and 7.87s `VLCAllCells`.
For $r_c = 3.5$ the pattern is similar. `VLCAllCells` has an even bigger lead on `VLCCellPair` which shrinks again for a higher number of threads. The performance of `VLCAllCells` for eight threads is 7.5s, while `VLCCellPair` is almost twice as slow with 13.73s. `VLCAllCells` and `VerletLists` are competing closely again with `VerletLists` having a small edge for four, eight and 16 threads.

10.1.2. Analysis

These results repeat the pattern from AoS c01 where the gap between `VLCAllCells` and `VLCCellPair` grows in favor of `VLCAllCells`'s performance for sparser scenarios or

shorter lists. The difference is that `VLCAllCells` already performs significantly better than `VLCCellPair` in the SoA data layout, likely once again due to `VLCCellPair`'s more complex data structure. The larger number of smaller neighbor lists could be hindering efficient vectorization, especially if they are smaller than the vector size. Additionally, iterating over `VLCCellPair`'s larger number of neighbor lists simply requires more instructions. More testing has to be done in regard to vectorization and the number of particles outside of the r_c which still get vectorized in `SoAFunctorVerlet`.
Similarly to the equivalent AoS scenario, `VerletLists` reaches and sometimes surpasses `VLCAllCells`, possibly due to its highly parallelizable traversal.

10.2. c18 and c08 traversals

10.2.1. Observations

Grid generator

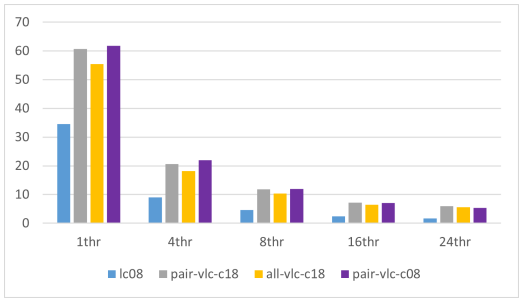


Figure 10.5.: Runtime comparison of containers for a grid scenario for 125000 particles with c18 and c08-based traversals and $r_c = 5$

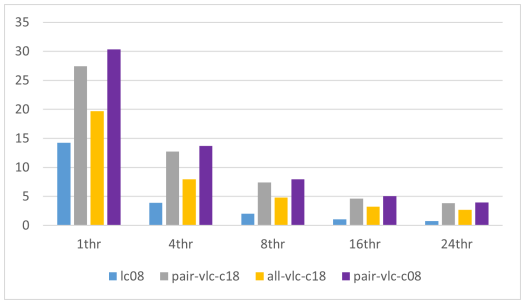


Figure 10.6.: Runtime comparison of containers for a grid scenario for 125000 particles with c18 and c08-based traversals and $r_c = 3.5$

For c18 grid $r_c = 5$ `VLCAllCells` maintains an advantage over `VLCCellPair` albeit weaker than c01. The results from this experiment are depicted in Figure 10.5. `VLCAllCells` completes the same computation as `VLCCellPair` for 85 to 95% of `VLCCellPair`'s runtime. For example, the runtime for eight threads is 11.81s for `VLCCellPair` and 10.3s for `VLCAllCells`, demonstrating that the difference between the containers is smaller than in the c01 scenario. `VLCCellPair`'s c08 traversal achieves a speedup over c18 only for a higher number of threads. Until eight threads it is actually slightly slower, for example 11.81s for c18 and 11.97s for c08. The runtime for 24 threads is respectively 5.96s for c18 and 5.39s for c08, showing a more significant difference. It is possible that once again the smaller amount of cells skews the results in favor of c08 because it has enough load per thread in contrast to c18.

For $r = 3.5$ the advantage of `VLCAllCells` over `VLCCellPair` grows slightly compared to the scenario with the bigger radius. `VLCAllCells`'s runtime makes up about 60 to 70% of `VLCCellPair`'s runtime, for example 4.66s for `VLCCellPair` and 3.23s for `VLCAllCells` for 16 threads.

The c08 traversal does not achieve a speedup for this scenario, as can be seen on Figure 10.6. `VLCCellPair`'s c18 is consistently faster, although the difference decreases again for a high number of threads. The performance for 24 threads is 3.87s and 3.95s, showing that c08 has almost caught up to c18.

Gaussian generator

The gaussian generator shows similar relationships in a slightly less pronounced manner.

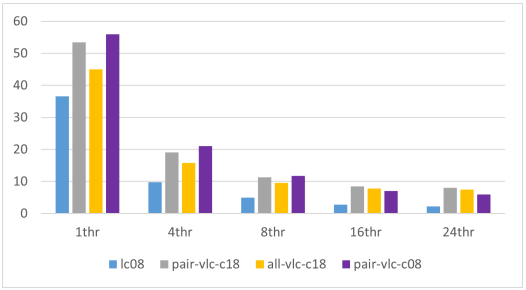


Figure 10.7.: Runtime comparison of containers for a gaussian scenario for 125000 particles with c18 and c08-based traversals and $r_c = 5$

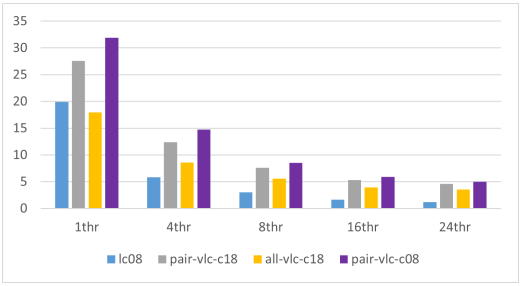


Figure 10.8.: Runtime comparison of containers for a gaussian scenario for 125000 particles with c18 and c08-based traversals and $r_c = 3.5$

For a r_c of 5 the advantage of `VLCAllCells` over `VLCCellPair` is now smaller, the results getting as close as 7.47s and 8s respectively for 24 threads. `VLCCellPair`'s c08 performs similarly to c18 up to eight threads, where the runtimes are 11.3s and 11.78s. Afterwards it speeds up faster and the runtimes for 24 threads are 8s and 5.97s where c08 has a noticeable advantage. A visual representation of the result from this scenario can be seen on Figure 10.7.

A smaller r_c of 3.5 has a similar effect to shrinking the radius in the grid scenario. It makes the gap between `VLCAllCells` and `VLCCellPair` increase again with their closest results for 24 threads being 3.57s and 4.6s. `VLCAllCells`'s runtime is about 78% of `VLCCellPair`'s runtime, showing a larger difference in contrast to the one in the grid scenario. This relationship depicted in Figure 10.8.
Here the c08 traversal does not add a speedup at all, instead starting at 31.9s for a single thread significantly slower than c18's 27.58s. With an increasing number of threads c08 starts catching up to c18 but does not manage to surpass it, their runtimes being 4.6s and 5s respectively.

10.2.2. Analysis

These results show the same main characteristics we have described for c01, although slightly less pronounced. `VLCAllCells`'s c18 is consistently performing better than `VLCCellPair`'s c18 and the difference between them grows with the decrease of r_c. However, it stays generally smaller than in the equivalent scenarios for c01. `VLCCellPair`'s c18 maintains its slower runtime due to its omplex structure and other factors, as discussed in Subsection 10.1.2.

The c08 traversal for `VLCCellPair` is scaling better than c18, but that is largely due to its slow performance on a single thread. It seems that it is speeds up more than c18 for a high number of threads. More tests should be done with a higher number of threads and various numbers of particles and cells to allow deeper analysis.

Part IV.

Conclusion

11. Conclusion and outlook

Throughout the course of this thesis, we have discussed the drawbacks of the standard Verlet lists algorithm and other Verlet lists-based neighbor identification algorithms. The pairwise Verlet algorithms aims to solve them by storing particles and their neighbors in local neighbor lists, attached to each pair of interacting cells. We have described the implementation of the algorithm in the context of the AutoPas library. The implementation also includes a refactoring of the existing `VerletListsCells` container to allow it to be used with different neighbor lists. We have extended both of `VerletListsCells`'s neighbor lists with an SoA data representation, since they were previously only available in an AoS data layout. Finally, we have added a c08 traversal implementation for the new `VLCCellPairNeighborList`, which was previously not possible for the `VerletListsCells` container.

The performance of the new components was evaluated through experiments on a homogenous and a non-homogenous scenario. The results for the AoS data layout imply that the newly implemented `VLCCellPair` has an advantage over other containers for scenarios with higher density or a larger interaction sphere containing more particles. However, the SoA representations of `VerletListsCells`'s neighbor lists perform worse than existing containers, possibly due to their increasingly complicated internal data structures. The c08 traversal shows a minimal if any improvement over existing traversals apart from a few specific scenarios.

Opportunities for future work involve further testing for a larger variety of numbers of particles and cells. It could also be useful to record the length of the neighbor lists and relate that data to the runtime performance measurements. Furthermore, the vectorization should be tested in more depth. More specifically, it should be investigated how many particles which are in the skin but not in range of r_c get vectorized anyway. Aditionally, a mechanism can be implemented for the `VerletListsCells` container to switch between directly between neighbor lists without treating them as two different containers. This would allow the program to attach a new neighbor list to the same container without instantiating a new container.

Part V.

Appendix

A. Measurements

	lc	vl	pair-vlc	all-vlc
1thr	343.001	92.27777	69.25184	74.23159
4thr	90.623	28.24345	21.59087	22.6331
8thr	44.80333	14.39547	10.94407	11.61568
16thr	23.67733	7.263862	5.90605	6.245913
24thr	16.38333	4.914659	4.185186	4.470103

Table A.1.: AoS grid c01 r5

	lc	vl	pair-vlc	all-vlc
1thr	132.2327	35.62301	27.36077	26.99466
4thr	37.01367	12.9084	10.75693	10.86769
8thr	18.39733	6.814098	5.708789	5.652336
16thr	9.515667	3.361244	3.062866	3.122032
24thr	6.814	2.582559	2.603976	2.874748

Table A.2.: AoS grid c01 r3.5

	lc	vl	pair-vlc	all-vlc
1thr	293.4523	87.63786	69.87526	72.11787
4thr	77.91633	26.62959	22.21482	21.79346
8thr	38.80567	13.44605	11.54507	11.2556
16thr	20.16667	7.062128	6.066742	5.958575
24thr	13.749	4.641154	4.175116	4.042463

Table A.3.: AoS gaussian c01 r5

	lc	vl	pair-vlc	all-vlc
1thr	112.8363	35.48398	28.08733	25.31884
4thr	32.282	12.33534	11.01785	9.525487
8thr	15.813	6.482511	5.780759	4.901681
16thr	8.362333	3.356375	3.05047	2.659253
24thr	5.828667	2.30728	2.320137	2.187162

Table A.4.: AoS gaussian c01 r3.5

	lc-c18	pair-vlc-c18	all-vlc-c18	lc-c08	pair-vlc-c08
1thr	184.6427	41.12838	43.70488	183.6307	40.98779
4thr	48.22967	11.98281	12.90938	46.534	11.59045
8thr	25.19767	6.484162	6.911332	23.55067	6.170517
16thr	14.21767	3.679921	4.036005	12.255	3.204321
24thr	10.973	2.824812	3.197444	8.506333	2.339864

Table A.5.: AoS grid c08 and c18 r5

	lc-c18	pair-vlc-c18	all-vlc-c18	lc-c08	pair-vlc-c08
1thr	70.94067	15.51134	15.55486	70.17333	16.54312
4thr	19.01633	4.949728	5.333517	18.39333	5.23006
8thr	9.690333	2.636085	2.799546	9.305667	2.752693
16thr	5.021	1.503901	1.747318	4.778	1.462444
24thr	3.538667	1.284146	1.598824	3.277667	1.19197

Table A.6.: AoS grid c08 and c18 r3.5

	pair-vlc-c18	all-vlc-c18	pair-vlc-c08		
1thr	347.3191	368.9896	347.7056		
4thr	98.71118	110.3123	98.96622		
8thr	50.38971	54.34377	49.9715		
16thr	26.10418	28.41782	25.50508		
24thr	18.27272	20.57875	17.5418		

Table A.7.: AoS grid c08 and c18 r5 for a million particles

	lc-c18	pair-vlc-c18	all-vlc-c18	lc-c08	pair-vlc-c08
1thr	157.838	40.41814	41.81303	157.6693	41.40479
4thr	40.911	11.58701	12.07228	39.99567	12.0161
8thr	23.36933	6.654232	7.094235	20.173	6.153588
16thr	17.89433	5.034938	5.551584	11.74833	3.606895
24thr	17.26067	4.791941	5.354592	9.891333	2.900221

Table A.8.: AoS gaussian c08 and c18 r5

	lc-c18	pair-vlc-c18	all-vlc-c18	lc-c08	pair-vlc-c08
1thr	61.332	16.42832	15.55358	60.893	19.47416
4thr	16.64933	5.404432	5.135573	15.979	6.606346
8thr	8.396333	2.867629	2.753634	8.031333	3.464581
16thr	4.539	1.619676	1.693575	4.077333	1.837433
24thr	3.522	1.417217	1.568516	2.808667	1.384068

Table A.9.: AoS gaussian c08 and c18 r3.5

	pair-vlc-c18	all-vlc-c18	pair-vlc-c08		
1thr	327.0787	339.0069	334.878		
4thr	95.12653	98.4587	99.65673		
8thr	47.4746	48.96924	50.00657		
16thr	24.74828	26.22618	25.60723		
24thr	17.48364	18.72616	17.25847		

Table A.10.: AoS gaussian c08 and c18 r5 for a million particles

	lc	vl	pair-vlc	all-vlc	
1thr	55.686	92.55919	110.0698	91.90398	
4thr	14.413	28.00099	38.69611	30.4222	
8thr	7.189667	15.85064	21.11923	16.20724	
16thr	3.679667	8.721519	11.39006	8.963648	
24thr	2.513667	6.684405	8.466011	7.134246	

Table A.11.: SoA grid c01 r5

	lc	vl	pair-vlc	all-vlc	
1thr	22.58667	34.04288	47.86673	32.85955	
4thr	6.452667	11.49797	22.24536	13.21039	
8thr	3.25	6.875309	12.70454	7.393243	
16thr	1.671667	4.521446	7.290292	5.0428	
24thr	1.169667	3.83015	5.639666	4.697018	

Table A.12.: SoA grid c01 r3.5

	lc	vl	pair-vlc	all-vlc	
1thr	59.159	74.64619	93.62867	73.68063	
4thr	15.749	23.46404	35.43848	24.68823	
8thr	7.933333	13.28738	19.78533	13.75439	
16thr	4.046333	7.917509	11.27627	7.874818	
24thr	2.763333	6.103584	8.120661	6.262046	

Table A.13.: SoA gaussian c01 r5

	lc	vl	pair-vlc	all-vlc	
1thr	30.365	28.60979	48.03111	27.59629	
4thr	8.915667	10.88278	23.48876	12.07942	
8thr	4.517	6.697428	13.72939	7.504649	
16thr	2.369333	4.926437	8.172642	5.131123	
24thr	1.637333	5.101712	6.500943	4.782827	

Table A.14.: SoA gaussian c01 r3.5

	lc-c18	pair-vlc-c18	all-vlc-c18	lc-c08	pair-vlc-c08
1thr	35.21167	60.71523	55.41704	34.603	61.74105
4thr	9.357667	20.64794	18.23532	8.980667	22.02496
8thr	4.897	11.8144	10.30863	4.569	11.96761
16thr	2.775667	7.214153	6.416066	2.408	7.118758
24thr	2.139667	5.959052	5.580674	1.699667	5.390563

Table A.15.: SoA grid c18 and c08 r5

	lc-c18	pair-vlc-c18	all-vlc-c18	lc-c08	pair-vlc-c08
1thr	36.92233	53.42051	45.04526	36.564	55.95169
4thr	9.981	19.11642	15.83673	9.796	21.07208
8thr	5.507333	11.30134	9.514928	4.983333	11.78433
16thr	3.981667	8.506459	7.805402	2.746	7.005136
24thr	3.613333	8.012931	7.472682	2.239333	5.972975

Table A.16.: SoA gaussian c18 and c08 r5

	lc-c18	pair-vlc-c18	all-vlc	lc-c08	pair-vlc-c08
1thr	14.40767	27.4465	19.6966	14.25033	30.31907
4thr	4.092	12.71996	7.940469	3.901333	13.68618
8thr	2.122667	7.436246	4.826174	2.034333	7.978481
16thr	1.143	4.661499	3.227044	1.068	5.052022
24thr	0.840333	3.866012	2.691053	0.749667	3.950691

Table A.17.: SoA grid c18 and c08 r3.5

	lc-c18	pair-vlc-c18	all-vlc-c18	lc-c08	pair-vlc-c08
1thr	20.1	27.58032	17.92252	19.91667	31.91475
4thr	6.056333	12.39603	8.589706	5.800667	14.73217
8thr	3.169333	7.603854	5.581374	3.001667	8.539324
16thr	1.723667	5.278422	3.941323	1.637	5.90581
24thr	1.274667	4.600653	3.570039	1.164	5.013003

Table A.18.: SoA gaussian c18 and c08 r3.5

List of Figures

List of Tables

Bibliography

[DM11] Jacob Durrant and J McCammon. Molecular dynamics simulations and drug discovery. *BMC biology*, 9:71, 10 2011.

[Gon12] 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.

[Gra21] Gratl, Fabio Alexander and Seckler, Steffen and Bungartz, Hans-Joachim and Neumann, Philipp. N Ways to Simulate Short-Range Particle Systems: Automated Algorithm Selection with the Node-Level Library AutoPas. *Computer Physics Communications*, 2021.

[GST+19] Fabio Gratl, Steffen Seckler, Nikola Tchipev, Hans-Joachim Bungartz, and Philipp Neumann. Autopas: Auto-tuning for particle simulations. pages 748–757, 05 2019.

[HD18] Scott A. Hollingsworth and Ron O. Dror. Molecular dynamics simulation for all. *Neuron*, 99(6):1129–1143, 2018.

[SSR07] Ralf Schneider, Amit Sharma, and A. Rai. *Introduction to Molecular Dynamics*, volume 739, pages 3–40. 10 2007.