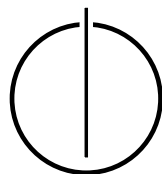


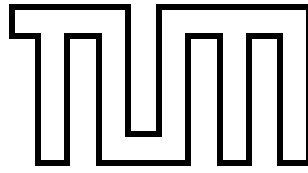
FAKULTÄT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

**Development and Evaluation of
Shared-Memory Parallelizations for
Verlet-Lists in AutoPas**

Tobias Alexander Humig





FAKULTÄT FÜR INFORMATIK

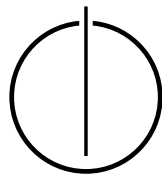
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

**Development and Evaluation of Shared-Memory
Parallelizations for Verlet-Lists in AutoPas**

**Entwicklung und Evaluierung von
Shared-Memory-Parallelisierungen für
Verlet-Listen in AutoPas**

Author: Tobias Alexander Humig
Supervisor: Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisor: Fabio Alexander Gratl, M.Sc.
Date: 16.09.2019



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

Munich, 16.09.2019

Tobias Alexander Humig

Acknowledgements

At this point, I want to thank all people that supported me while writing this thesis. Thank you, Fabio Gratl, for providing a perfect environment to get to know the problems and solutions of molecular dynamics simulations during the lab course last semester, and for our weekly meetings during the last four months. This close support really motivated me to try out own ideas, since I could always ask for immediate feedback. Thanks also to my friends and family who proofread this thesis, and especially to my girlfriend Ajvi for being always available for feedback and support.

Abstract

Molecular dynamics simulations are very compute-intensive tasks. In order to speed them up, massive parallelization for the force calculation is necessary. Verlet Lists are a data structure capable of reducing the naively seen quadratic computational complexity to a linear one. A library that solves the force calculation of these simulations is the C++ template library *AutoPas* [GST⁺19]. It includes many different approaches for this problem and automatically chooses the fastest one of them for the current state of the system at run time.

This thesis extends *AutoPas* by adding several new approaches. All of them are built on the idea of Verlet Lists and are parallelized using *OpenMP 4.5*. Small parts of the library are refactored to prepare it for these new algorithms.

The new approaches are tested on their Time to Solution and Strong Scaling behavior for different simulation settings on two different platforms. They are compared to an existing solution, and it is shown that they outperform it on many occasions.

Zusammenfassung

Molekulardynamiksimulationen sind sehr rechenintensive Programme. Um sie zu beschleunigen ist eine starke Parallelisierung der Kraftberechnung erforderlich. Verlet Listen sind eine Datenstruktur die fähig ist, den naiv gesehen quadratischen Rechenaufwand auf einen linearen zu reduzieren. Eine Bibliothek, die die Kraftberechnung für diese Simulationen löst, ist die C++ template Bibliothek *AutoPas*. Sie umfasst viele unterschiedliche Lösungsansätze für dieses Problem und wählt zur Laufzeit automatisch den Schnellsten für den aktuellen Zustand des Systems aus.

Diese Arbeit erweitert *AutoPas* um einige neue Lösungsansätze. All diese bauen auf der Idee von Verlet Listen auf und werden mit *OpenMP 4.5* parallelisiert. Kleine Teile der Bibliothek werden umstrukturiert um sie auf diese neuen Algorithmen vorzubereiten.

Die neuen Lösungsansätze werden auf ihre Schnelligkeit und starke Skalierbarkeit in verschiedenen Simulationsumgebungen auf zwei verschiedenen Plattformen getestet. Sie werden mit einer existierenden Lösung verglichen und es wird gezeigt, dass sie diese in vielen Fällen übertreffen.

Contents

Acknowledgements	vii
Abstract	ix
Zusammenfassung	xi
I. Introduction and Background	1
1. Introduction	2
2. Theoretical Background	3
2.1. Intermolecular Potentials	3
2.2. Algorithm Types for Calculating the Forces	4
2.2.1. Direct Sum	4
2.2.2. Linked Cells	5
2.2.3. Verlet Lists	8
2.2.4. Verlet Cluster Lists	10
3. Introduction to AutoPas	11
3.1. Non-Functional Goals of the Library	11
3.2. Software Design	11
3.2.1. Library Interface	11
3.2.2. Containers and Traversals	12
II. Towards Parallelization of Verlet Lists	15
4. Refactoring of Existing Code	17
4.1. Removal of Template Parameters from <code>iteratePairwise()</code>	17
4.2. Centralizing the Traversal Instantiation	18
4.3. Introduction of SoA Views	18
4.4. Neighbor Lists SoAFunctor Refactoring	19
4.5. Automated Common Unit Tests for all Containers and Traversals	20
5. Containers and their Parallelizations	21
5.1. <code>VerletClusterLists</code> Container	21
5.1.1. Creating Clusters	21
5.1.2. Class Structure	21
5.1.3. Building the Container and Neighbor Lists	22

5.1.4.	Traversal Interface	24
5.1.5.	<i>verlet-clusters</i> Traversal	24
5.1.6.	<i>verlet-clusters-static</i> Traversal	25
5.1.7.	<i>verlet-clusters-coloring</i> Traversal	26
5.2.	VarVerletListAsBuild Container	28
5.2.1.	The VarVerletLists template	29
5.2.2.	Container Creation using the VarVerletLists template	30
5.2.3.	Future Improvements	30
5.2.4.	<i>var-verlet-lists-as-build</i> Traversal	30
6.	Experiments and Analysis	32
6.1.	Computation Platforms	32
6.2.	Automated Performance Measurement	32
6.3.	Verlet Cluster Lists Refactoring	33
6.4.	Time to Solution	35
6.4.1.	<i>verlet-clusters</i> Traversal	35
6.4.2.	<i>verlet-clusters-static</i> Traversal	38
6.4.3.	<i>verlet-clusters-coloring</i> Traversal	41
6.4.4.	Overall Interpretation for the VerletClusterLists Container	44
6.4.5.	<i>var-verlet-lists-as-build</i> Traversal	46
6.5.	Strong Scaling	50
6.5.1.	<i>verlet-clusters</i> Traversal	50
6.5.2.	<i>verlet-clusters-static</i> Traversal	53
6.5.3.	<i>verlet-clusters-coloring</i> Traversal	56
6.5.4.	<i>var-verlet-lists-as-build</i> Traversal	60
III.	Conclusion	64
7.	Summary	65
8.	Future Work	67
	Bibliography	71

Part I.

Introduction and Background

1. Introduction

In today's science, molecular dynamics simulations help to tackle many problems of great importance. They assist in medicine, for example in drug research [PGL⁺18], in battery research [MCPR18], or in studying the behavior of nanotubes [PC17] that are themselves used in many applications as fighting water scarcity [IAAA⁺16]. They provide closed, reproducible environments and complete observability over its processes and can, therefore, aid in multiple steps of the scientific method.

A molecular dynamics simulation calculates the movement of particles inside a domain over time, according to certain basic rules. Since in theory every pairwise particle-interaction needs to be considered, they are an instance of the general N-body problem. Direct methods for solving these have a computational complexity of $\mathbf{O}(N^2)$. Verlet Lists are capable of reducing this to a linear computational complexity in certain conditions. Since the particles in these simulations are free to move in the whole domain, simulating their movement is a problem of highly dynamic nature. With particle numbers in the millions, countless different situations occur and having one simulation code that performs efficiently in all these situations gets unlikely.

For that reason, the C++ node-level template library *AutoPas* is developed. It is applicable to the general N-body problem. To tackle the highly dynamic nature of the problem, the library employs many different approaches to solve it. Depending on the current state of the simulation, it automatically chooses the fastest one.

In this work, several new approaches are added to *AutoPas*. They are all based on the idea of Verlet Lists and are parallelized using OpenMP 4.5 in order to utilize all available hardware resources fully. Up until now, there have not been many efficient Verlet Lists parallelizations in the library. Having them available enriches its pool of distinct algorithms and makes it more viable in simulation settings where Verlet Lists are the most promising approach. In the library, small parts are refactored to prepare it for the new approaches.

In the end, the performance of all new approaches is measured in different simulation environments and on different platforms. The results are analyzed for strengths and weaknesses and compared to an existing solution.

2. Theoretical Background

2.1. Intermolecular Potentials

The interaction between particles is described by a pairwise potential. This potential defines the force that particles exert on each other, depending on their relative position. In this work, only short-ranged potentials are considered. We call a potential short-ranged, if the force decays faster than $1/r^3$, with r being the distance between two particles. For this subclass of potentials, a certain *cutoff* radius for each potential can be introduced. Since the force decays so fast, all forces between particles further apart than this cutoff radius can be neglected [GKZ10]. This is a significant advantage, as these particle interactions do not have to be considered for force calculation.

A common potential for Molecular Dynamics Simulations is the Lennard-Jones potential. For two particles with distance r_{ij} and parameters ϵ and σ , it is:

$$U(r_{ij}) = 4 \cdot \epsilon \cdot \left(\left(\frac{\sigma}{r_{ij}} \right)^{12} - \left(\frac{\sigma}{r_{ij}} \right)^6 \right) \quad (2.1)$$

Since its decay is of order 6, it is a short-ranged potential. The cutoff for it is typically chosen to be $2.5 \cdot \sigma$. Figure 2.1 visualizes the potential for $\epsilon = \sigma = 1$. It can be seen that the potential for $r \geq 2.5$ is very close to zero.

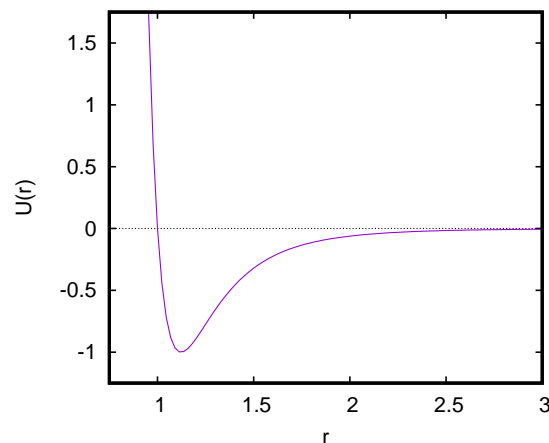


Figure 2.1.: Lennard-Jones Potential with $\epsilon = \sigma = 1$. Source: [GKZ10]

When calculating the forces between particles, an optimization to divide the number of force calculations by two can be made. This is due to Newton's third law [New87]:

”To every action there is always opposed an equal reaction: or the mutual actions of two bodies upon each other are always equal, and directed to contrary parts.”

This states that if a particle p_1 exerts a force on a particle p_2 , p_2 always exerts the same force with opposite direction on particle p_1 . For this reason, it is enough to calculate the force from p_1 to p_2 , and use that value negated as force from p_2 to p_1 .

2.2. Algorithm Types for Calculating the Forces

There are three classical archetypes for algorithms that calculate forces of short-range pairwise interactions. Ordered by implementation complexity from low to high, they are *Direct Sum*, *Linked Cells*, and *Verlet Lists*. The two more complex algorithm types can be built on top of the simpler ones. All of them are explained in this section. Additionally, an enhancement of Verlet Lists is discussed, the *Verlet Cluster Lists*.

Algorithm 1: Force Calculation Between Two Particles

Input: p1, p2, potential, cutoff

Output: force particle p2 exerts on particle p1 using the given potential and cutoff

```
1 Function forceFromToCut(p1, p2, potential, cutoff):
2   if distance (p1, p2) ≤ cutoff then
3     return potential.forceFromTo (p1.position, p2.position)
4   else
5     return 0
```

All algorithms explained now will use Algorithm 1 to calculate the force that one particle acts on another. It implements the cutoff for the force between two particles. If the particles are closer together than the cutoff, it returns the force that the potential defines. Otherwise, it returns zero.

2.2.1. Direct Sum

The first algorithm type is the trivial approach for iterating over the pairwise interactions. Algorithm 2 shows an example implementation in pseudo-code. The algorithm gets the

Algorithm 2: Direct Sum Algorithm

Input: particles, potential, cutoff

Output: particles with added forces

```
1 Function calculateForces(particles, potential, cutoff):
2   for i ← 0 to particles.size() - 1 do
3     for j ← 0 to particles.size() - 1 do
4       if i ≠ j then
5         force = forceFromToCut (particles[i], particles[j], potential, cutoff)
         particles[i].force += force
```

array of particles and a potential for the force calculation as input. It then iterates over all particle pairs using a double `for` loop. In this case, both particle pairs (p_1, p_2) and (p_2, p_1) are visited, so we do not take advantage of Newton's third law. All pairs (p_1, p_1) are excluded by the `if` statement, as particles do not interact with themselves. For each of these pairs, the force between the first and the second particle is added to the force accumulator of the first particle. Since in each iteration of the inner `for` loop, the only variable that is written to is the force accumulator of the first particle, all iterations of the outer loop are independent of each other, which makes the code embarrassingly parallel.

Algorithm 3: Direct Sum Algorithm With Newton 3

Input: particles, potential, cutoff
Output: particles with added forces

```

1 Function calculateForcesDirectSum(particles, potential):
2   for  $i \leftarrow 0$  to particles.size() - 1 do
3     for  $j \leftarrow i + 1$  to particles.size() - 1 do
4       force = forceFromToCut (particles[ $i$ ], particles[ $j$ ], potential, cutoff)
5       particles[ $i$ ].force += force
6       particles[ $j$ ].force -= force

```

Algorithm 3 shows an example implementation that utilizes Newton's third law. Since Newton's third law states that for all particle p_i and p_j , $f(p_i, p_j) = -f(p_j, p_i)$, the double `for` loop is now written to only iterate over particle pairs (p_i, p_j) with $j > i$ and to use this symmetry. This essentially halves the number of iterations of the inner `for` loop, but introduces dependencies between iterations of the outer `for` loop, since each iteration accesses both the force accumulator of the first and second particle of each pair. Unfortunately, this makes exploiting Newton's third law more difficult.

Overall, while this approach is easy and fast to implement, it scales $\mathbf{O}(n^2)$ with n being the number of particles, because of the double `for` loop with an outer and inner iteration count of $\mathbf{O}(n)$. This, of course, can not be used for particle numbers in the millions and higher, so more sophisticated approaches are needed.

2.2.2. Linked Cells

When observing the behavior of the Direct Sum algorithm, one quickly notices that it iterates over all particle pairs existing, no matter how far away the particles are, although the force between those further away from each other than the given cutoff radius is always zero. The second algorithm type, called Linked Cells [AT87], heavily takes advantage of this property of the potential. Assuming that, in all areas of the simulation domain containing n particles, each particle does not have more than a fixed number of neighbors c within the cutoff radius, it effectively reduces the run time from $\mathbf{O}(n^2)$ to $\mathbf{O}(c \cdot n)$.

In order to achieve this, the Linked Cells algorithm divides the simulation domain into cells with a minimum side length of the cutoff radius. Then, the particles are sorted into their cell, according to their position. Now, for each particle, only the particles in the same cell and in all neighbor cells can exert a force greater than zero on the particle, so the

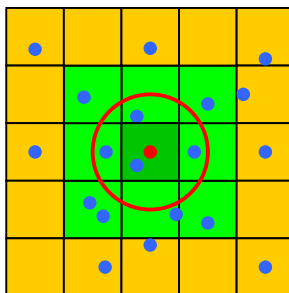


Figure 2.2.: Visualization of the Linked Cells algorithm. The red particle interacts with all particles in the red cutoff circle. Only particles in the green cells are considered as potential neighbors by the Linked Cells algorithm.

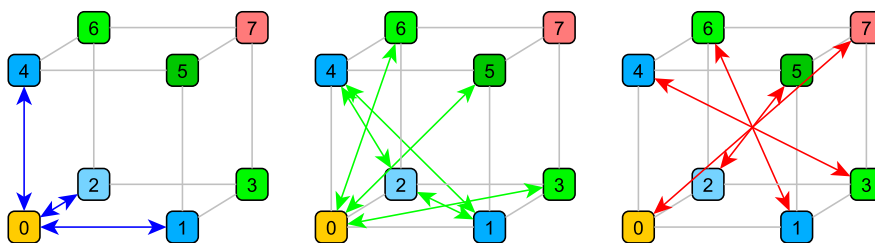


Figure 2.3.: C08 Traversal: Base Step Interactions for cell 0 in 3D. Colors visualize interactions along one (blue), two (green), and three (red) dimensions. The numbers indicate the color of each cell. Source: [GST⁺19]

algorithm only iterates over those. This is illustrated in Figure 2.2. Algorithm 4 shows an example implementation. First, all particles are sorted into their cells. Then, for each cell, first the pairwise interactions within, and then those with its neighbors are calculated. These steps are both extracted into other methods.

Utilizing Newton’s third law while calculating the forces in parallel requires a bit more work here, than for Direct Sum. One possible way is the *c08* traversal. This approach is now explained shortly since it is used in some implementations and as a comparison in the benchmarks in later chapters.

The *c08* traversal works the following way: To ensure that no data races occur while updating the force of the particles using Newton’s third law in parallel, an eight-way cell-coloring of the domain is introduced. To avoid race conditions, two neighboring cells are not allowed to have the same color. Then, each color is worked on separately with OpenMP barriers between them. During the processing of each color, the cells of that color are distributed to the different threads using dynamic scheduling. For each cell, a thread executes the base step shown in Figure 2.3. Doing this for all cells in all colors yields the forces between all particles [GST⁺19].

Overall, the Linked Cells algorithm reduces the number of particle neighbor candidates massively, compared to the Direct Sum algorithm. However, while this is already significant,

Algorithm 4: Linked Cells implementation

Input: particles, potential, cutoff**Output:** particles with added forces

```
1 Function handleCell(cell, potential, cutoff):
2   foreach particle1 ∈ cell do
3     foreach particle2 ∈ cell do
4       if particle1 ≠ particle2 then
5         force = forceFromToCut (particle1, particle2, potential, cutoff)
6         particle1.force += force

7 Function handleCellPair(cell1, cell2, potential, cutoff):
8   foreach particle1 ∈ cell1 do
9     foreach particle2 ∈ cell2 do
10      force = forceFromToCut (particle1, particle2, potential, cutoff)
11      particle1.force += force

12 Function calculateForcesLinkedCells(particles, potential, cutoff):
13   cells = sortParticlesIntoCells (particles, cutoff)
14   foreach cell ∈ cells do
15     handleCell (cell, potential, cutoff)
16     foreach neighborCell ∈ cell.neighbors do
17       handleCellPair (cell, neighborCell, potential, cutoff)
```

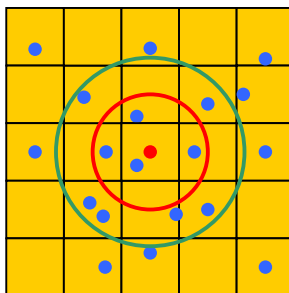


Figure 2.4.: Visualization of the VerletLists algorithm. The red particle interacts with all particles in the red cutoff circle. All particles in the green skin circle are saved in the neighbor list of the red particle.

a simple calculation shows that even here, with uniformly distributed particles in the domain, only around 16% of the potential neighbors are within the cutoff radius.

$$\text{Volume of cells } C = 27 \cdot \text{cutoff}^3 \quad (2.2)$$

$$\text{Volume of sphere with radius cutoff } S = \frac{4}{3}\pi \text{cutoff}^3 \quad (2.3)$$

$$\frac{S}{C} = \frac{\frac{4}{3}\pi \text{cutoff}^3}{27 \cdot \text{cutoff}^3} = \frac{\frac{4}{3}\pi}{27} \simeq 15.5\% \quad (2.4)$$

Decreasing the number of unnecessary neighbor calculations and thereby increasing this hit rate of neighbors for a particle is the idea of the final algorithm archetype.

2.2.3. Verlet Lists

The third algorithm archetype for pairwise interactions is called Verlet Lists [Ver67], named after Loup Verlet. Its idea is to save references to all particles in the neighborhood of a particle in the particle's neighbor list. These neighbor lists are then used for the next few force calculation iterations and are renewed once the particles have moved so far that their neighbor lists are not valid anymore. Assuming a fixed maximum number of neighbors c similar to Linked Cells, this also leads to a performance of $\mathbf{O}(c \cdot n)$ for each iteration of the force calculation.

So how far are the particles allowed to move? This depends on how the neighborhood of a particle is defined. Of course, the neighborhood has to contain all particles that are within the cutoff radius. To allow some movement of the particles without invalidating the neighbor lists, a *skin radius* is added to the cutoff radius, so that the neighborhood for a particle is defined as all particles that are within $\text{interaction_length} = \text{cutoff} + \text{skin}$ distance. Figure 2.4 shows that. The neighbor lists are then valid as long as all particles have not moved more than $\text{skin}/2$ units. This ensures that even if two particles both move away from each other this maximum distance, the neighbor list is still valid.

Additionally, to calculating the forces, the neighbor lists have to be built. This is often done using the Linked Cells algorithm with a cell side length of *interaction_length*, which also has run time $\mathbf{O}(c \cdot n)$, making that the final run time of the Verlet Lists algorithm.

Algorithm 5: Verlet Lists Implementation

Input: particles, potential, cutoff, skin
Output: particles with added forces

```

1 Function generateNeighborLists(particles, interactionLength):
2   cells = sortParticlesIntoCells (particles, interactionLength)
3   foreach cell ∈ cells do
4     foreach otherCell ∈ cell.neighbors ∪ cell do
5       foreach particle ∈ cell do
6         foreach possibleNeighbor in otherCell do
7           if particle ≠ possibleNeighbor then
8             if distance (particle, possibleNeighbor) ≤ interactionLength
9               then
10                particle.neighbors.append(possibleNeighbor)
11
12 Function calculateForcesVerletLists(particles, potential, cutoff, skin,
13   shouldRebuild):
14   if shouldRebuild then
15     generateNeighborLists (particles, cutoff + skin)
16   foreach particle ∈ particles do
17     foreach neighbor ∈ particle.neighbors do
18       force = forceFromToCut (particle, neighbor, potential, cutoff)
19       particle.force += force

```

Algorithm 5 shows an example implementation of VerletLists. It gives the caller an option if the neighbor lists should be rebuilt or not. At the first time, this has to be **true**. The generation of new neighbor lists is extracted into another method. There, similar to the Linked Cells algorithm, the particles are sorted into cells. For each particle, its outer cell and all neighboring cells are searched for neighbor particles that are closer than the interaction length. These are then saved in the particle’s neighbor list. The actual force calculation then only iterates over all neighbors for each particle and adds up the forces from these pairwise interactions. In contrast to the Linked Cells algorithm, no unnecessary neighbor distance calculations happen during the force calculation.

Attentive readers will probably already have noticed how significant the additional memory usage is compared to Linked Cells or Direct Sum. Whether this is worth it compared to Linked Cells depends on the hardware used and on the implementations, but also on the simulation. Furthermore, using SIMD instructions and being cache efficient in implementations of Verlet Lists is a huge challenge [AMS⁺15]. Since this is very important for achieving excellent performance, modifications of Verlet Lists have been invented.

2.2.4. Verlet Cluster Lists

One modification of the Verlet Lists algorithm is the Verlet Cluster Lists algorithm. The introduction of particle clusters for Verlet Lists was first done by GROMACS [AMS⁺15]. This algorithm allows easy vectorization by design while reducing the memory footprint compared to traditional Verlet Lists. The key idea is that particles close together have similar neighbors. These particles then form a cluster. So, instead of constructing neighbor lists for each particle and calculating the interactions in pairs of particles, the neighbor lists are constructed for clusters of particles, and the interactions are calculated between two clusters and within each cluster. Two clusters are neighbors if a particle in one cluster contains at least one neighbor in the other cluster.

Since in each cluster interaction, the interactions between a fixed number of multiple particle pairs are calculated, these interactions can be vectorized. For this, the cluster size is chosen to fit the characteristics of the SIMD hardware used. Furthermore, as the number of clusters is considerably smaller than the number of particles, the total number of neighbors to save decreases, too. This leads to a significant improvement in memory usage.

3. Introduction to AutoPas

AutoPas [GST⁺19] is a C++ template library that aims to achieve optimal node-level performance for calculating the pairwise-forces in N-body simulations. It meets this claim by implementing multiple algorithms to solve the force calculation and choosing the fastest one of them for the current state of the simulation dynamically at run time. This is called auto-tuning. The user only has to provide the pairwise force calculation kernel, called *Functor*, and the particles. In order to fully utilize a node, massive parallelization is needed. Every algorithm implements its own parallelization strategy with OpenMP.

3.1. Non-Functional Goals of the Library

The library has mainly four non-functional goals:

1. Ease of Use
2. Modularity
3. Platform Independence
4. Good Performance

Since this work extends the library, it also takes them into account. At first, the library should be easy to use. Users should be able to use the library without first understanding the whole implementation and without any knowledge of the strengths of certain algorithms. The library automatically chooses the best algorithm available. Secondly, the library-internal code should be organized very modularly. Each part of the library should be interchangeable to allow flexible adjustments to the needs of the current simulation state. Third, the library should be platform-independent to be able to run on many different architectures and be compiled with all major compilers. As the last point, the library should yield good performance in as many cases as possible, independent of the hardware, the number of threads, the simulation size, or particle distribution.

3.2. Software Design

3.2.1. Library Interface

To be easy to use, the library uses the Facade Pattern. Its entire interface is represented by the `AutoPas` class. It has a template parameter to determine what type of particles it should work on. To this class, the user provides instances of a custom particle class and of a custom functor class that represents the potential to use. For this, the `AutoPas` class provides methods to add particles, iterate over them, and traverse the particle pairs with a

given functor. It further gives many options to configure more details about the domain, cutoff, skin, the tuning strategy, and more. Figure 3.1 visualizes that.

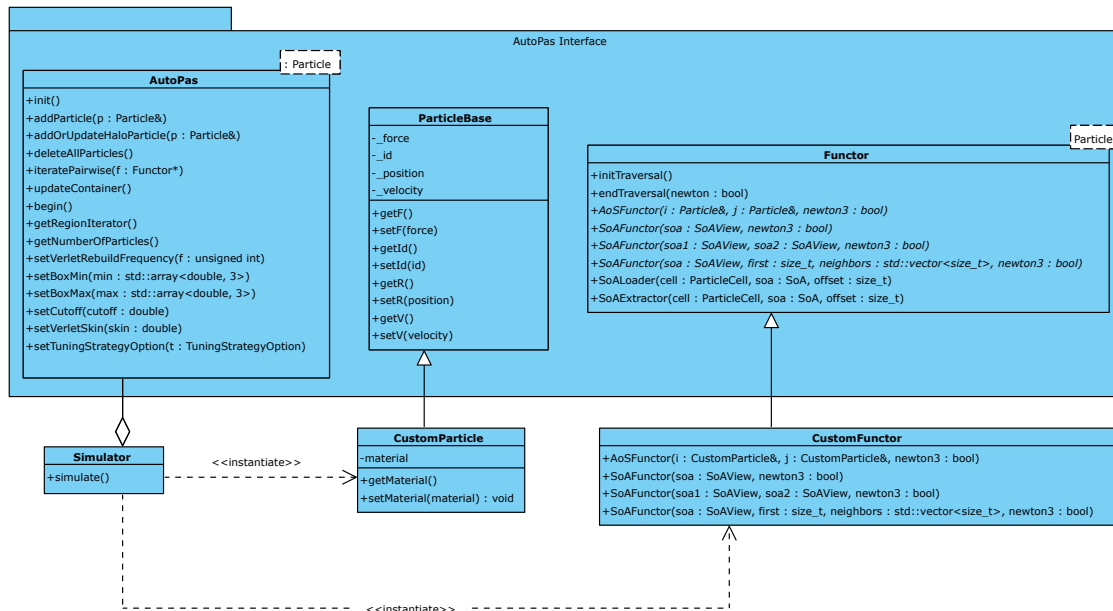


Figure 3.1.: UML Class Diagram of the interface of *AutoPas* and User Code that utilizes the library for own simulations.

3.2.2. Containers and Traversals

Internally, each algorithm to drive the force calculation consists of a container type and a traversal type. This combination is also called a configuration. The container is responsible for saving the particles in a certain manner. Examples include the **DirectSum** container, that saves all particles inside one cell, the **LinkedCells** container that saves all particles in cells with a certain cell length, and the **VerletLists** container that saves its particles in an underlying **LinkedCells** container, but additionally saves neighbor lists for its particles. These containers are implementations of the algorithm types outlined in Section 2.2.

The traversal is responsible for the order the particle pairs in a container are processed in. Its main challenge is parallelizing the force calculation efficiently. One container type can have multiple traversal types, but a traversal type always belongs to only one container type. Examples for traversals are the *C08* traversal, shortly in explained in Subsection 2.2.2 or the *Sliced* traversal [GST⁺19] for the **LinkedCells** container.

Figure 3.2 shows the base interfaces for all containers and traversals. The container mostly has methods to specify the domain it should represent, and to add, remove, or iterate over particles. Its most important methods for this thesis are `rebuildNeighborLists()` and `iteratePairwise()`. In the first method, the container has the opportunity to prepare for upcoming force calculations by building the neighbor lists. It receives the traversal it builds the neighbor lists for to decide, for example, if the neighbor lists should be built for a traversal that uses Newton’s third law or not. This method is called periodically and always before

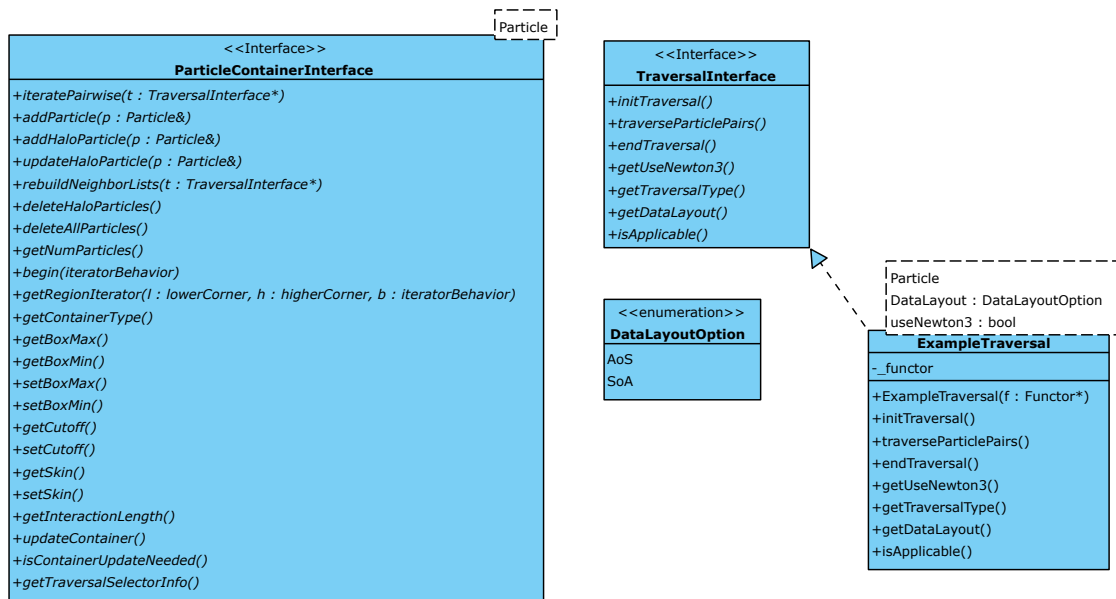


Figure 3.2.: UML Class Diagram of the interfaces for all containers and traversals.

the first call to the second method, `iteratePairwise()`. There, the container calls the `initTraversal()`, `traverseParticlePairs()`, and `endTraversal()` methods on the given traversal. These are the most important methods for a traversal. While the first and the last one prepare and finalize the traversal, the second method, `traverseParticlePairs()`, contains the whole logic of how the particle pairs in the container are iterated over. Internally, the traversal contains the functor to use for the pair interactions, as it can be seen in Figure 3.2. The main focus of this work is to add new containers and traversals as new configuration options for the auto-tuning.

Each traversal also has template parameters that specify what data layout it should work on and if it is supposed to use Newton’s third law. In the `isApplicable()` method, the traversal can check whether it can be applied with the current template arguments. In this thesis, two different data layout options are used. The Array-of-Structures data layout (AoS), and the Structure-of-Arrays data layout (SoA). These data layouts specify how the particles are saved in memory. The following codes explain the difference between both data layouts:

```

1 class ParticleContainerAoS {
2     static constexpr size_t N = 10;
3     class Particle {
4         array<double, 3> r;
5         array<double, 3> f;
6         array<double, 3> v;
7         size_t id;
8     }
9
10    array<Particle, N> particles;
11 }
  
```

```

1 class ParticleContainerSoA {
2     static constexpr size_t N = 10;
3
4     array<array<double, 3>, N> r;
5     array<array<double, 3>, N> f;
6     array<array<double, 3>, N> v;
7     array<size_t, N> id;
8 }
  
```

In the AoS case, the data of one particle lies continuously in memory. In the SoA case, this particle is split up to have the member variables of all elements of the array lie continuously in memory. The data layout of SoAs matches the data layout in the vector registers for the force calculation. Therefore, amongst other things, the SoA data layout helps greatly in getting the best performance out of vectorization¹.

Since in C++, arrays of objects normally use the AoS data layout, this is the default layout that all containers use. If the SoA data layout is selected for a traversal, the traversal copies the data needed for the force calculation from the AoS into a SoA. A SoA is represented by the class `SoA`.

¹<https://software.intel.com/en-us/articles/memory-layout-transformations>

Part II.

Towards Parallelization of Verlet Lists

After the introduction to the theoretical backgrounds, the outcomes of this work are now described in this chapter. For most containers and parallelizations, some refactoring of the existing code had to be done in order to implement them properly. These changes are attended first. Then, all new containers and their parallelizations are covered. Finally, the performance of each new configuration is analyzed.

4. Refactoring of Existing Code

Let us start with the refactorings. All following changes were done to increase the maintainability and extensibility of existing code. Most of it was *preparatory refactoring* for implementing the new containers and traversals, following the suggestion of Kent Beck¹:

”for each desired change, make the change easy (warning: this may be hard),
then make the easy change”

4.1. Removal of Template Parameters from `iteratePairwise()`

Previously, the `iteratePairwise(TraversalInterface* traversal)` method that every container has to have had legacy template parameters and legacy parameters. Its original signature was:

```
1 template <class ParticleFunctor , class Traversal>  
2 void iteratePairwise(ParticleFunctor *f, Traversal *traversal , bool newton3)
```

Especially the template parameters prevented the method from being declared pure virtual in the `ParticleContainerInterface`. Therefore, all containers had to implement it themselves and calling it required the code shown in Listing 4.1. The code uses the container type returned by `getContainerType()` to dynamically cast the given `ParticleContainer` pointer to its actual class. The resulting object is then passed to a generic lambda that calls the `iteratePairwise()` method on it. This is possible since the lambda deduces the actual container class from the dynamic cast.

```
1 template <typename Particle , typename ParticleCell , typename FunctionType>  
2 void withStaticContainerType(std::shared_ptr<ParticleContainer<Particle ,  
   ParticleCell>> &container , FunctionType &&func) {  
3     auto container_ptr = container.get();  
4     switch ( container->getContainerType() ) {  
5         case ContainerOption::directSum :  
6             func(dynamic_cast<DirectSum<Particle , ParticleCell> *>(container_ptr));  
7             return ;  
8             // ... all container options ...  
9         case ContainerOption::linkedCells :  
10            func(dynamic_cast<LinkedCells<Particle , ParticleCell> *>(container_ptr));  
11            return ;  
12        }  
13    }  
14  
15    // Calling statement  
16    withStaticContainerType(container , [](auto cont){cont.iteratePairwise(...)});
```

Listing 4.1: Code originally necessary for calling the `iteratePairwise()` method.

¹<https://twitter.com/kentbeck/status/250733358307500032>

In the body of the `iteratePairwise()` method, actually, only the traversal itself is necessary as a parameter, as long as the traversal saves its data layout and Newton 3 option and the underlying functor. Additionally, it has to provide methods for initializing the traversal, traversing the particle pairs, and finalizing the traversal. For this, the `TraversalInterface` was extended, as it can be seen in Figure 4.1.

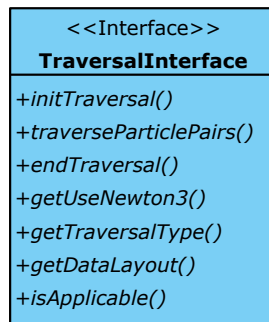


Figure 4.1.: UML Class Diagram of the new `TraversalInterface`.

Now, the new signature of the method is `iteratePairwise(TraversalInterface* traversal)`, and it is added as a pure virtual to the `ParticleContainerInterface`. This makes calling it as easy as any other method.

4.2. Centralizing the Traversal Instantiation

The previous way of instantiating traversals was through a `TraversalSelector` object. Every container provided a method `generateTraversalSelector()` that would return such an object properly initialized with the dimensions, the cutoff, and the cell length used in the container. This `TraversalSelector` object then provided a method `generateTraversal()` that instantiated a traversal given a functor and a `TraversalOption`.

Since this method could instantiate all traversals from all containers, the `TraversalSelector` class introduced dependencies from every container to every traversal. Utilizing traversals in a container implementation, therefore, produced circular dependencies.

This was resolved by replacing the `TraversalSelector` object in the container with a `TraversalSelectorInfo` object that only holds the container-specific info of the dimension, the cutoff, and the cell length used. The `generateTraversal()` method from before is now a static method that additionally gets one of these info objects. Thereby, the containers have no dependency on any traversal, but only on the info object, and the dependency problems are resolved.

4.3. Introduction of SoA Views

In the previous implementation, the `Functor` class provided methods that calculated the pairwise interactions for all particles in one SoA and between two SoAs. The signatures were

`SoAFunctor(SoA &soa, bool useNewton3)` and `SoAFunctor(SoA& soa1, SoA& soa2, bool useNewton3)`.

While this was sufficient for most Linked Cells implementations, some new configurations need finer-grained control. They need the possibility only to pass parts of SoAs to these functions. The easiest solution for this problem would be copying these parts into a new SoA, but this costs valuable performance. Instead, we introduced the class `SoAView` to replace the `SoA` parameters of the `Functor`. The new signatures are now `SoAFunctor(SoAView soa, bool useNewton3)` and `SoAFunctor(SoAView soa1, SoAView soa2, bool useNewton3)`.

A `SoAView` is a lightweight view on a certain part of a `SoA`. It provides the same methods as the `SoA` for accessing the particles within (`begin()` and `getNumParticles()`), so the implementation of the functors did not need to be changed. Internally, it just holds a pointer to the beginning of the actual `SoA` to view, a start index and an end index. Furthermore, the `SoAView` provides an implicit conversion from a `SoA`. That way, no calls to the methods that now have a different signature had to be changed. Figure 4.2 shows the new class.

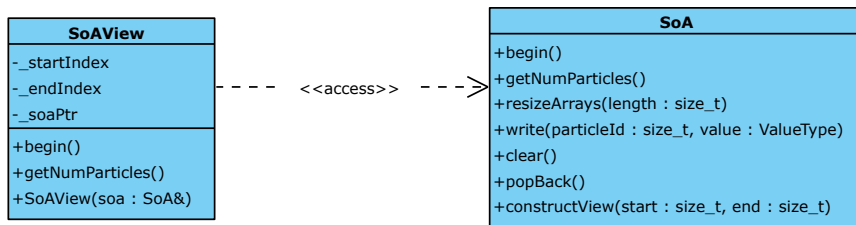


Figure 4.2.: UML Class Diagram of `SoAView` and `SoA`

4.4. Neighbor Lists SoAFunctor Refactoring

Originally, the signature of the `SoAFunctor()` method intended for Verlet Lists traversals of the `Functor` class was:

```

1 virtual void SoAFunctor(SoA soa, const std::vector<std::vector<size_t>> &
  neighborList, size_t iFrom, size_t iTo, bool newton3)
  
```

This type for the neighbor list is exactly the type which the `VerletLists` container, the first Verlet Lists container of the library, uses as the type for the `SoA` neighbor list. It works in the following way: For each particle of the given `SoA`, the neighbor list contains a `std::vector<size_t>>` with the same index. This vector holds all the indices in the `SoA` from all neighbors of the particle. `iFrom` and `iTo` specify a range of these indices. Inside this range, the method should calculate all pair interactions of the particles with its neighbors. The `VerletLists` container was the only container that used this method.

The problem with this signature comes, when a container uses multiple smaller neighbor lists instead of one big neighbor list. This, for example, will happen in the `VarVerletListsAsBuild` container (see Section 5.2). To be able to use this method with a neighbor list that only contains the neighbors for a part of all particles, an empty vector has to be inserted into the neighbor list for every particle the neighbor list does not contain neighbors for. As an example, the `VarVerletListsAsBuild` container will have $8 \cdot num_threads$ neighbor lists.

This makes the memory usage scale linearly with the number of threads used, just because of empty vectors.

Our solution for this problem is to change to signature of the `SoAFunctor()` to:

```
1 virtual void SoAFunctor(SoAView soa, const size_t indexFirst, const std::  
    vector<size_t> &neighborList, bool newton3)
```

Here, the functor only gets the index of a particle in the SoA and a vector of the indices of all its neighbors, instead of the neighbor list for all particles. In the implementation, this changes only the location of the loop over the particles from inside the functor to the caller. That way, no empty vectors have to be generated to fill up the neighbor list, and the signature of the method is better understandable, too.

4.5. Automated Common Unit Tests for all Containers and Traversals

For testing its components, *AutoPas* uses *GoogleTest*. The most important unit tests for the traversals are the tests where their result of one simulation step is compared to the result of a reference traversal. Also, for each container, there are some tests that check adding particles, deleting them, and iterating over them. These tests have been written manually for many containers and traversals, but most of the code is copy-and-paste, and the reference traversal is calculated many times.

Changing these tests is very error-prone because of the copy-and-paste, since it is easy to forget one of the tests. Also, the tests run quite slow.

For this reason, we added two parameterized test suites. The first automatically calculates the same simulation step for all traversals and compares the results to one reference traversal. The second one automatically executes some standard tests for all containers in the library. When new traversals and containers are added to the library, the test suites automatically generate the tests for those. In the future, many more tests can be pulled out from special container tests into these common test suites.

5. Containers and their Parallelizations

In the following sections, all new containers and their new traversals are described. Two containers are covered. The first container, called `VerletClusterLists`, was there before, but was rewritten, and three traversals were added. The other container, the `VarVerletListsAsBuild` is new and has one traversal.

5.1. VerletClusterLists Container

The `VerletClusterLists` container is an implementation of the Verlet Cluster Lists algorithm outlined in Subsection 2.2.4. As explained there, it divides the particles into clusters of a fixed size (a fixed number of particles per cluster) and saves neighbor clusters, instead of neighbor particles. Compared to other Verlet containers, it is expected that this container performs better with the SoA data layout and benefits more from vectorization units the hardware provides since the Verlet Cluster Lists algorithm was designed for that [AMS⁺15]. Before this work, there has already been an unfinished implementation of this container that only supported the AoS data layout and was written monolithic with high coupling and low cohesion. Extending it with support for SoA and different traversals resulted in poor performance. Therefore, the container was mostly rewritten to achieve better extensibility and maintainability, as well as better performance. The performance improvement can be seen in Section 6.3. Some essential parts like handling halo particles and deleting particles are still missing, but no features have been broken.

5.1.1. Creating Clusters

Before the structure and implementation is described, our algorithm to create clusters of particles is explained. This container, as in the previous implementation [Ngu18], solves that with the following approach: First, the domain is divided into towers along the x- and y-axis. The side length of those towers depends on the number of particles in the domain. Inside each tower, the particles are sorted into clusters of a fixed size along the z-axis. If the last cluster (the topmost) does not contain enough particles, dummy particles are added to ensure that each cluster has the same size. Figure 5.1 illustrates this in 2D with a cluster size of four.

5.1.2. Class Structure

Now, the class structure of the `VerletClusterLists` container is outlined. The class diagram in Figure 5.2 shows some important elements. The container divides its domain into many `ClusterTowers`, using the scheme from above. Each tower is responsible for all particles within it. A tower provides methods for adding particles, building and accessing its clusters. Internally, it saves all particles in a `FullParticleCell`. The clusters are generated

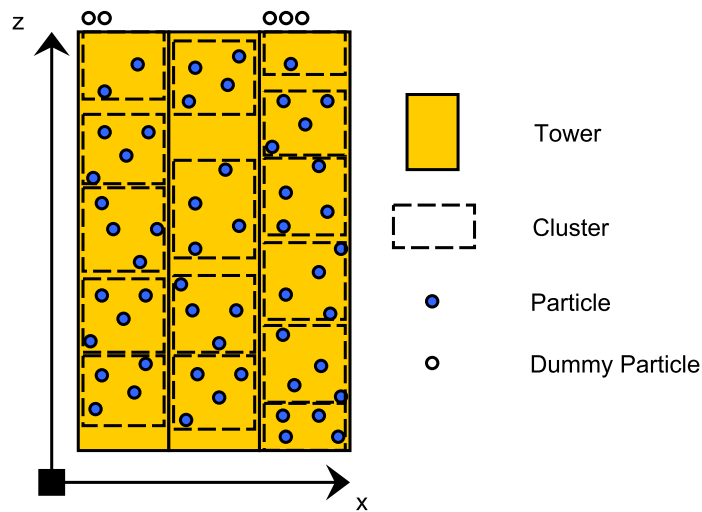


Figure 5.1.: Clustering of particles in 2D

as an index structure over the `FullParticleCell`. The tower is also responsible for filling in dummy particles if the last cluster is not full. A `Cluster` provides access to the particles it contains and to its neighbors. If the data layout is AoS, the particles can be accessed through the `getParticle()` method. If the data layout is SoA, the cluster additionally supplies a `SoAView` on its particles. The `VerletClusterLists` container provides a method for the traversals to generate a SoA for all particles and to initialize these `SoAViews` for each cluster, as well as to extract this SoA again.

5.1.3. Building the Container and Neighbor Lists

At this point, we know the overall class structure of the container. We know where particles and neighbors are saved, and how the container supports using the SoA data layout. It provides methods for loading and extracting the particles into a SoA and provides access to it through `SoAViews` for each cluster. Since traversals are responsible for the iteration over the particle pairs, the only unknown part left is how the container and its neighbor lists are built when `rebuildNeighborLists()` is called on the container. As this step involves several non-trivial sub tasks, the *Replace Method with Method Object* refactoring was used while rewriting the container, to ensure low coupling and high cohesion. Now, a special class `VerletClusterListsRebuilder` is responsible for rebuilding the container and its neighbor lists. The steps are:

1. Collect all particles from old towers and delete the towers.
2. Add new particles handed to `addParticle()`.
3. (*Delete pending particles marked by `ParticleIterator::deleteCurrentParticle()`.*)

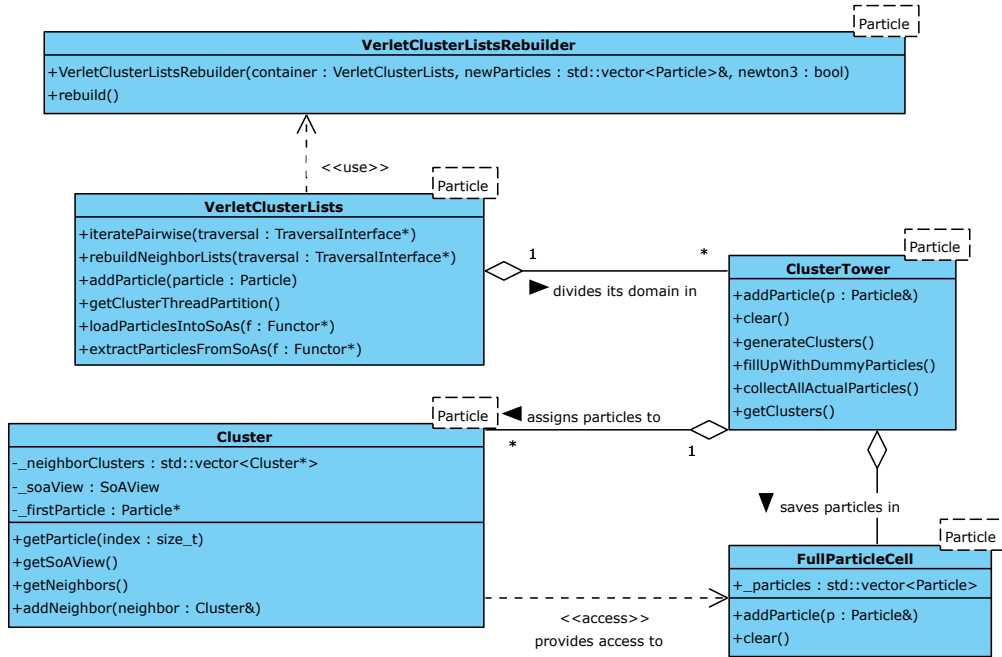


Figure 5.2.: UML Class Diagram of the VerletClusterLists container.

4. Estimate new optimal tower side length using the particle collection.
5. Generate new towers and sort particles into them.
6. Make towers generate the cluster index structures.
7. Calculate the neighbors for each cluster.
8. Make towers add dummy particles to fill up last cluster.

In detail, the following happens: At first, all particles of the container are collected from the old towers so that the towers can be deleted. This is done using move semantics to avoid copying all particles. To this collection, new particles that were passed to `addParticle()` are added. At this point, the particle deletion of the container should happen, but this is not implemented at the moment. Then, the new optimal tower side length is estimated. The heuristic for this has been taken from the previous implementation. It is:

$$new_tower_side_length = \sqrt[3]{\frac{cluster_size \cdot domain_volume}{num_particles}} \quad (5.1)$$

Using this, the new towers are instantiated, and the particles are sorted into them. This is done in parallel with a lock for each tower. Now, the clusters are generated as an index structure for each tower, also in parallel. The seventh step, the neighbor list generation, is the most complicated. It is also done in parallel per tower. The neighbor lists can be generated to contain symmetric pairs of clusters or not, depending on whether Newton's third law is used by the traversal they are generated for. A detailed description of that can

be found in Subsection 5.1.7. For each tower, all towers that hold potential neighbor clusters are calculated first. Since the tower side length has no lower bound, they can be arbitrary many in each direction. For each cluster of the current tower, the distance to the clusters of all neighbor towers is now calculated. If the potential neighbor cluster is in range, it is added as a neighbor. Two clusters are in range if there is a particle in one cluster that is within the interaction length of one particle of the other cluster. The last step is to fill up the last cluster of each tower with dummy particles, to ensure that each cluster has the same size. These dummy particles are placed far away from the domain and each other to ensure that no additional pairwise interactions are calculated. Each tower remembers how many of the last particles are dummy particles. That way, they can be differentiated from real particles. Compared to the total amount of particles, there are negligible many dummy particles with this approach, but having them allows to not check the size of a cluster before working with it every time.

From all these steps, the seventh, the neighbor list generation, takes the longest by far. Optimizing the build time should, therefore, aim to optimize the neighbor list generation. For example, Newton's third law could also be used for that.

5.1.4. Traversal Interface

As a short preparation for the description of the traversals for this container, the abstract class they have to inherit from are covered. Traversals for this container have to inherit from the `VerletClustersTraversalInterface`. This abstract base class provides access to the `VerletClusterLists` object to traverse through a protected member. Furthermore, it allows traversals to opt-in to calculate a static cluster-thread-partitioning when the neighbor lists are rebuilt. This will be further explained in Subsection 5.1.6. Additionally, as required by the library, all traversals have to implement the `TraversalInterface`.

5.1.5. `verlet-clusters` Traversal

The *verlet-clusters* traversal exists as the first easy idea of how a traversal over this container could be parallelized. Its AoS variant was used by the previous container implementation in the `iteratePairwise()` method. In this implementation, it supports both the AoS and the SoA data layout, but can not exploit Newton's third law in general. In the SoA case, the traversal uses the methods provided by the container for loading and extracting the particles in a SoA during `initTraversal()` and `endTraversal()`.

Parallelizing the force calculation is done in the following way by this traversal: Since Newton's third law is not utilized, there is no dependency between different towers. These towers can, therefore, be distributed to all threads using OpenMP dynamic scheduling. This ensures proper good load-balancing, as the number of clusters contained can greatly vary between towers. Each thread calculates the forces for all clusters (which is equal to all particles) in its current tower. For that, it iterates over all clusters. For each cluster, it first calculates the interactions of the particles within itself. This is done using Newton's third law since this is the only thread that writes to these particles. Then, the thread iterates over all neighbor clusters of this cluster and calculates the forces between the particles of both of them. Here, since, the neighbor clusters may be part of a different tower, Newton's third law must not be used to rule out potential race conditions.

A future improvement could also use Newton’s third law for cluster pairs that lie both inside the same tower since the same thread calculates the forces for both of them. In the current implementation, the necessary information is not saved in the neighbor lists or clusters. Therefore it is not possible without the performance loss of calculating the tower for the clusters.

5.1.6. verlet-clusters-static Traversal

The *verlet-clusters-static* traversal works very similar to the *verlet-clusters* traversal. It therefore also supports AoS and SoA, but not using Newton’s third law. The only difference lies in the distribution of clusters to the threads. It aims to calculate a static distribution of clusters to threads after the container builds the neighbor lists to get rid of the scheduling overhead of the OpenMP dynamic scheduling.

This static distribution of clusters to threads tries to partition the amount of work that needs to be done during the force calculation as equally as possible. For this, the work is divided into many small chunks with a certain cost. In the end, all chunks should be distributed to the threads, and each thread should have around the same amount of cost. Here, one chunk will exactly be the calculation of all forces acting on one cluster. That includes the pair interactions within the cluster and with all its neighbors without Newton’s third law. There are several possible ways to determine the cost of one such chunk. The easiest would be to assign them all the same cost because it is always one cluster. This heuristic for the run time cost is wrong in most cases since the number of pair interactions vastly depends on how many neighbors the cluster has. Therefore, we chose the number of neighbor clusters as a heuristic for the cost.

Having this, we still need to calculate the distribution. This is done by the container of traversals opt-in. At first, the total cost for all chunks of work is calculated by iterating over all clusters and summing up the number of their neighbors. The optimal amount of cost per thread is then calculated by:

$$total_cost = \sum_{c \in Clusters} \mathbf{num_neighbors_of_cluster}(c) \quad (5.2)$$

$$optimal_cost_per_thread = \frac{total_cost}{num_threads} \quad (5.3)$$

Knowing this, the clusters can now be distributed to the threads by iterating over them once more. For each thread, the starting cluster and the number of clusters to work on are saved. Each thread gets the minimum number of clusters assigned so that the accumulated cost of those clusters is greater than the *optimal_cost_per_thread*. Unfortunately, this will result in a slight imbalance, since the threads do not get exactly the same amount of work and the last thread will get considerably less than the others. However, assuming that the chunks of work are small and there are very many, this load imbalance is acceptable. Figure 5.3 illustrates this distribution.

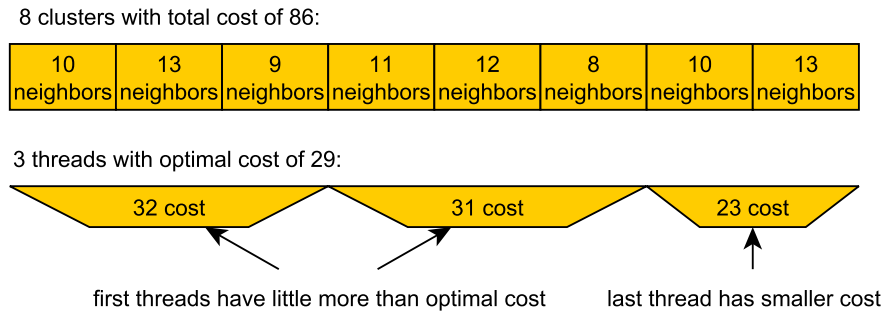


Figure 5.3.: Small Example of the Static Distribution of Clusters to Threads

As a final note, sometimes, the traversal chooses the number of threads to be strictly less than the maximum number of threads. This is done when there is only a small amount of chunks and spawning more threads would probably cause more overhead than it speeds up the calculation. Currently, this estimation is done very trivially. Each thread will at least get one thousand clusters assigned.

The future improvement from the *verlet-clusters* traversal also applies here. Additionally, it might be worth it to allow some threads with a little less than the optimal cost to align the workload of the last thread with the others.

5.1.7. *verlet-clusters-coloring* Traversal

The third traversal of the `VerletClusterLists` container is the *verlet-clusters-coloring* traversal. Its goal is to utilize Newton's third law for the `VerletClusterLists` container. It supports both the AoS and the SoA data layout and works with and without Newton's third law. Similar to the other traversals for the `VerletClusterLists` container, this traversal uses the methods provided by the container to load and extract a SoA and accesses it only through the `SoAViews` provided by the `Cluster` objects.

To avoid data races while using Newton's third law, this traversal uses a 6-way coloring of the towers. For this coloring, the notion of a color cell has to be introduced. The general idea of domain coloring is to mark elements with the same color that can be processed in parallel without data races. Here, these elements are the color cells. The color cells cover the XY-plane of the domain. A color cell has at least a side length greater or equal to the interaction length, but it must also be a multiple of the tower side length. Now, each tower belongs to a color cell, according to its position. Because of the way we chose the side length of a color cell, all clusters inside the towers of the cells that are not direct neighbors (including diagonal neighborhood) are never neighbors of each other. The color cells at the border may contain fewer towers than other color cells due to the arbitrary tower side length. Figure 5.4 visualizes this layout.

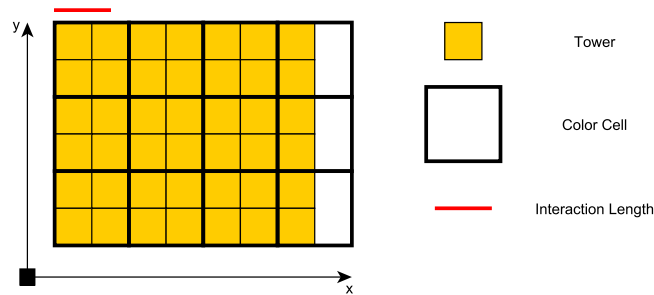


Figure 5.4.: Color Cells covering Towers

Since we want to use Newton's third law, we have to make sure that towers that can be traversed in parallel do not share the same neighbors. This is guaranteed using the coloring of these color cells and the base step for the interaction of each color cell shown in Figure 5.5 and Figure 5.6.

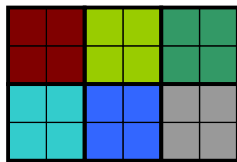


Figure 5.5.: 6-way Coloring of the Color Cells

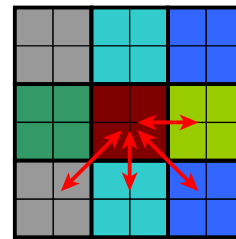


Figure 5.6.: Base Step of the Coloring

Within each color, we can now distribute the color cells of that color to all threads using OpenMP dynamic scheduling. To calculate all forces, we iterate over each color one by one and introduce a barrier between them.

How do we guarantee that each thread only calculates the pair interactions that it is supposed to while it is working on one color cell? Our solution is to generate the neighbor lists with Newton's third law enabled appropriately. The container saves them in a way so that a thread working on a color cell only has to calculate the pair interactions between all clusters of all towers within that cell and their neighbors with Newton's third law. The neighbor lists with Newton's third law are built in a way to contain the base step of the coloring already.

To build the neighbor lists in this way, the `VerletClusterListsRebuilder` introduces an order of the towers inside the domain. In one step during the neighbor list generation, for each tower, all towers that contain potential neighbors for the clusters within it are calculated. In this step, all towers with a lower index in that order are excluded. Furthermore, if the neighborhood of two clusters inside the same tower is considered, only the cluster that is lower in z-direction saves the other cluster as its neighbor. The order of the towers is defined as follows: At first, all towers within a color cell that is more to the left or to the top of another color cell are lower in the order than the towers of the other color cell. Secondly, within a color cell, a tower that is more to the left or to the top is lower in the order. An

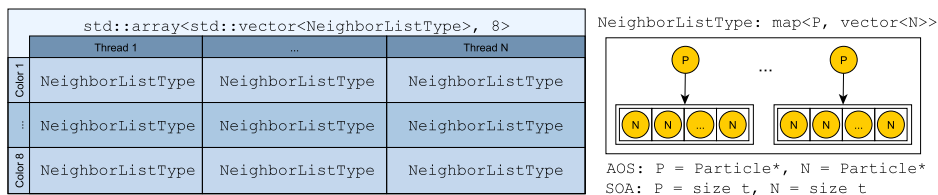


Figure 5.8.: VarVerletListsAsBuild neighbor lists

example is given in Figure 5.7. For the clusters in tower 19, the towers 20, 23, 25, 26, 27, 28, 29, 30, 31, 32, 33, and 35 contain potential neighbor clusters.



Figure 5.7.: Order of towers for neighbor lists with Newton's third law

Comparing Figure 5.7 with Figure 5.6, the reader will notice that if the towers of the red cell contain only towers with a higher index in their neighborhood, the resulting neighbor lists represent exactly the base step of the coloring.

5.2. VarVerletListAsBuild Container

The primary goal of this container is to be able to use Newton's third law for classical Verlet Lists. This is difficult to achieve since neighbor lists alone do not contain enough information to judge if two particles can be traversed in parallel by different threads without data races.

The `VarVerletListsAsBuild` container therefore saves additional information within its neighbor lists. It stores the particles in an underlying Linked Cells container and uses the `C08` traversal for building the neighbor lists with Newton's third law. The functor provided to the traversal generates a distinct neighbor list per thread and per color. Each of these neighbor lists then maps traversed particles to a vector of their neighbors.

This is illustrated by Figure 5.8. Depending on the data layout, the data type stored by this data structure is different. For the Array-of-Structures data layout, the reference to a particle is saved using a `Particle*`. For the Structure-of-Arrays data layout, it is saved using the index of the particle in a global SoA containing all particles. In the neighbor list build phase, the neighbor lists are built using the AoS data type. Only if a traversal is configured to use SoA, the SoA neighbor lists are additionally generated from the AoS neighbor lists.

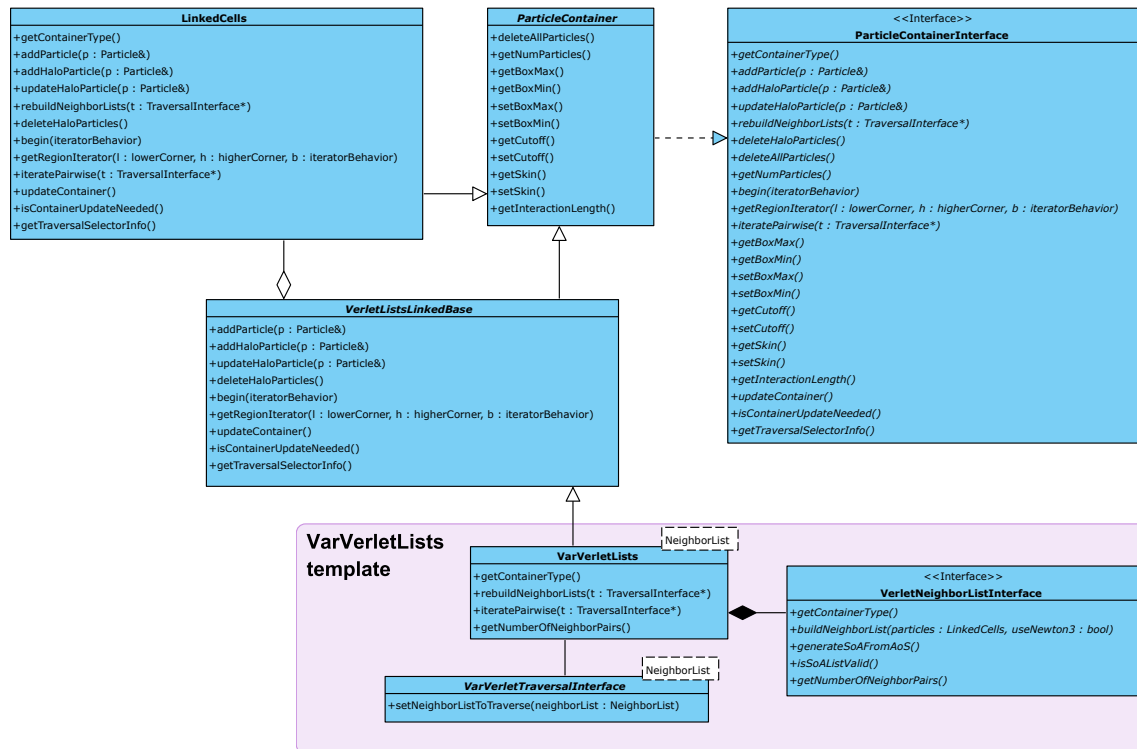


Figure 5.9.: VarVerletLists template

This generation of the SoA neighbor list consists of copying the neighbor lists and replacing the `Particle*` with the corresponding index. Since the traversal is responsible for loading the particles in a SoA, the mapping between particles and their indices has to be defined commonly for the container and its traversals. As this mapping is essentially just an ordering, the order of the particle iterator of the underlying Linked Cells container is used.

This underlying container, as well as the implementation of the `ParticleContainerInterface`, is provided by the `VarVerletLists` template.

5.2.1. The VarVerletLists template

The `VarVerletLists` template, as seen in Figure 5.9, was added to ensure code reuse between different Linked Cells based Verlet Lists containers without increasing the inheritance hierarchies even more. It detaches the implementation of new neighbor list types from the complex container hierarchy. These neighbor lists may then store additional information or use other data types in their internal implementation. More precisely, it introduces a new `VerletNeighborListInterface` that encompasses all important functionality of neighbor lists and separates it from the actual container. Internally, this container uses Linked Cells as underlying storage of the particles and provides this data structure to the `VerletNeighborListInterface` during the building process.

Traversals for the container using such a neighbor list `NeighborList` have to inherit from the `VarVerletTraversalInterface<NeighborList>`, next to the common `TraversalInterface`. This parent class provides the traversal with access to the neighbor list of the container.

5.2.2. Container Creation using the VarVerletLists template

To create the `VarVerletListsAsBuild` container, one new class `VerletNeighborListAsBuild` which implements the `VerletNeighborListInterface` has to be added. This class contains and builds the neighbor list described above, so the final container is now available using the type `VarVerletLists<VerletNeighborListAsBuild>`. Traversals for this container are subclasses of the `VarVerletTraversalInterface<VerletNeighborListAsBuild>` and the common `TraversalInterface`. Figure 5.10 shows both the new neighbor list class and one traversal.

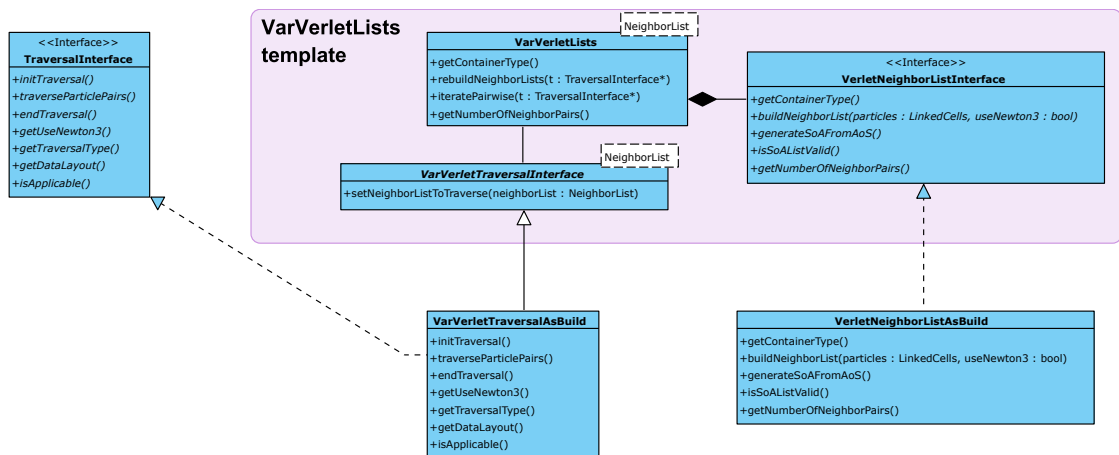


Figure 5.10.: VarVerletLists: Adding a new neighbor list

5.2.3. Future Improvements

In the future, the container could be rewritten to support all Linked Cells traversals instead of just the *C08* traversal. That would make this container valuable in many more situations, since *C08* is not optimal in all simulations. On the other hand, it would make auto-tuning considerably more complicated since many configurations would be added, which increases the search space significantly. This might be countered by studying the relationship between the performance of a Linked Cells traversal and the `VarVerletListsAsBuild` container that uses it. If there was a known relationship, it could be included in the auto-tuning.

5.2.4. var-verlet-lists-as-build Traversal

The *var-verlet-lists-as-build* traversal utilizes the additional information provided by the neighbor list to statically give each thread the same neighbor lists to traverse as it had built before. Since the *C08* traversal is used for building, Newton's third law is supported. This comes at the cost of having thread barriers between the neighbor lists of each of the eight colors.

Why do we expect this static load-balancing to achieve good performance? The heuristic is based on the following assumption, where $NeighborList_c^t$ is the neighbor list of thread t

during color c :

$$factor_c^t = \frac{\mathbf{time_to_build}(NeighborList_c^t)}{\mathbf{time_to_traverse}(NeighborList_c^t)} \quad (5.4)$$

$$Factors = \{factor_c^t : t \in Threads, c \in Colors\} \quad (5.5)$$

$$proportion_ratio = \frac{\mathbf{max}(Factors)}{\mathbf{min}(Factors)} \simeq 1 \quad (5.6)$$

Equation 5.6 states that the time to build a neighbor list is proportional to the time needed to traverse it. *proportion_ratio* quantifies how much this relationship fits. It is assumed that there is only a small error, thus it is expected to be almost 1.

Further, let us look at the load-balancing during build (times are without scheduling overhead):

$$balancing_ratio = \frac{time_to_traverse}{time_to_traverse_with_perfect_load_balancing} \quad (5.7)$$

The *time_to_traverse_with_perfect_load_balancing* is the time it takes with optimal load balancing that distributes all work to the threads such that all threads need the same time for their part. Considering that our implementation of the *C08* traversal uses dynamic scheduling with a chunk size of one cell, we can expect its load balancing to be near to the theoretical optimum. Thus, this error is probably nearly 1, too.

If each thread now statically gets the same neighbor lists to traverse as to build, the error in load balancing for the traversal is proportional to *proportion_ratio* · *balancing_ratio*.

$$traversal_balancing_error \propto proportion_ratio \cdot balancing_ratio \quad (5.8)$$

According to this, and assuming that indeed *proportion_ratio* \simeq 1 holds, we can use a time consuming dynamic scheduling for the load balancing during the build and perform equally good during the traversal with static load balancing.

Furthermore and next to AoS, this traversal also supports the SoA data layout. For this, during *initTraversal()*, it loads all particles of the underlying Linked Cells container into a global SoA, in the same order as the neighbor list expects, that is the order of the iterator provided by the Linked Cells container. After the force calculation, during *endTraversal()*, it extracts those particles into the Linked Cells container again.

6. Experiments and Analysis

6.1. Computation Platforms

All traversals were tested on two machines, the two partitions CoolMUC2 and CoolMUC3 of the Linux cluster at Leibnitz Supercomputing Centre. The CoolMUC2 uses Intel Xeon E5-2697 v3 CPUs with Haswell Architecture, 28 cores, and 64GB RAM per node. For the CoolMUC3, it is an Intel Xeon Phi CPU with Knights Landing architecture, 16GB High Bandwidth Memory, 96GB RAM, 64 cores and four-way hyper-threading per node.

6.2. Automated Performance Measurement

For measuring the performance of all traversals, a script was written that automatically tests given traversals in certain situations. In this script, *md-flexible* was used, a simulation tool contained in the *AutoPas* project to test configurations in different simulation environments easy and fast.

There are two modes in which the new configurations were tested, *Time to Solution* and *Strong Scaling*. In the first mode, *Time To Solution*, the time needed for completing a certain number of force calculation iterations with a given number of particles is measured. The number of threads to use is fixed at the number of hardware threads available on the machine. Multiple iterations, depending on their expected duration, are done to compensate measurement errors and Verlet Lists building times. In each test of this mode, the combinations shown in Table 6.1 are measured.

Particles	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}
Iterations	10000	5000	2000	1000	500	400	100	50	25	15	15

Table 6.1.: Combinations of Particles and Iterations for *Time To Solution* Measurements.

In the second mode, *Strong Scaling*, a problem with the fixed size of 2^{18} particles and 30 iterations is calculated with different numbers of threads. These are listed in Table 6.2. For the CoolMUC3, it is noteworthy that 128 and 256 threads only work with hyper-threading. This has to be kept in mind when looking at the measurement results.

CoolMUC2	1	2	4	8	16	28					
CoolMUC3	1	2	4	8	16	32	64	128	256		

Table 6.2.: Number of Threads used for each Machine in the *Strong Scaling* Measurements.

In all diagrams, the measured time is displayed as *Million Force Updates Per Second (MFUPS)*. One force update is the calculation of all forces acting on one particle in one

iteration. The MFUPS of a simulation run are therefore calculated as:

$$MFUPS = \frac{total_number_of_particles \cdot number_of_iterations}{simulation_duration_in_seconds \cdot 10^6} \quad (6.1)$$

The denser the particles in the domain are, the more time one force update needs, as a particle then has more neighbors within the cutoff radius. Because of that, the MFUPS are expected to decrease with static domain size, but increasing numbers of particles as in the *Time To Solution* measurement mode.

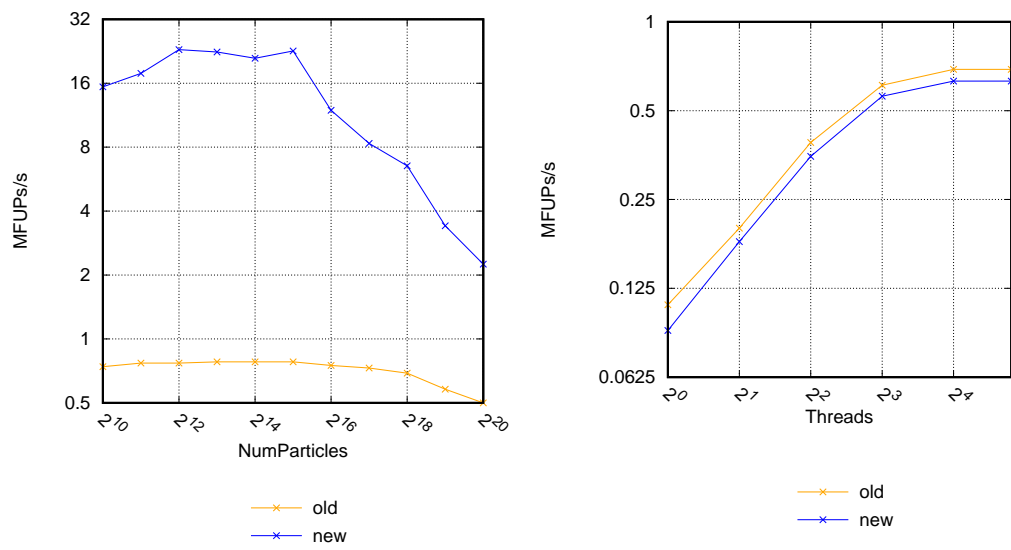
For all test runs, the following simulation parameters are used: The domain size is fixed at $25 \cdot 25 \cdot 25$, the cutoff is 1.0, and the skin is 0.2. For a Linked Cells container with cell length 1.2, this means that the average cell contains between ~ 0.11 and ~ 115.96 particles for 2^{10} to 2^{20} particles in the domain.

These particles are distributed in two different ways. The first is a uniform distribution. This represents a simulation environment where the particles are distributed very evenly across the whole domain, for example the simulation of a gas. The second is a gaussian distribution with a mean of 10.5 and a standard deviation of 5.5. Here, the particles are heavily centered around (10.5, 10.5, 10.5), and there are fewer particles at the border. This roughly forms a droplet. It challenges the traversals with a very unbalanced simulation domain.

In the upcoming sections, diagrams of measurements for all four new traversals in *Time To Solution* and *Strong Scaling* with all allowed data layouts and Newton 3 options, with both particle distributions, and on both machines are shown. Each configuration of a traversal will first be compared to the reference *c08* traversal of the `LinkedCells` container of the library. Then, all those configurations are compared against each other for both particle distributions.

6.3. Verlet Cluster Lists Refactoring

Before showing the performance measurements for all traversals in their current implementation, the performance improvement gained through rewriting the `VerletClusterLists` container is shortly covered. As mentioned in Section 5.1, the reason for rewriting the container came from the poor performance that the SoA implementation for the old container yielded. Figure 6.1a shows the maximum performance gain from all configurations after the refactoring. Here, the new implementation is, depending on the number of particles, around an order of magnitude faster. To illustrate that this performance increase is not gained at the cost of other configurations for this container, its worst change is shown in Figure 6.1b. It can be seen that the overall performance there is slightly worse in this strong scaling, but the overall behavior has stayed the same.

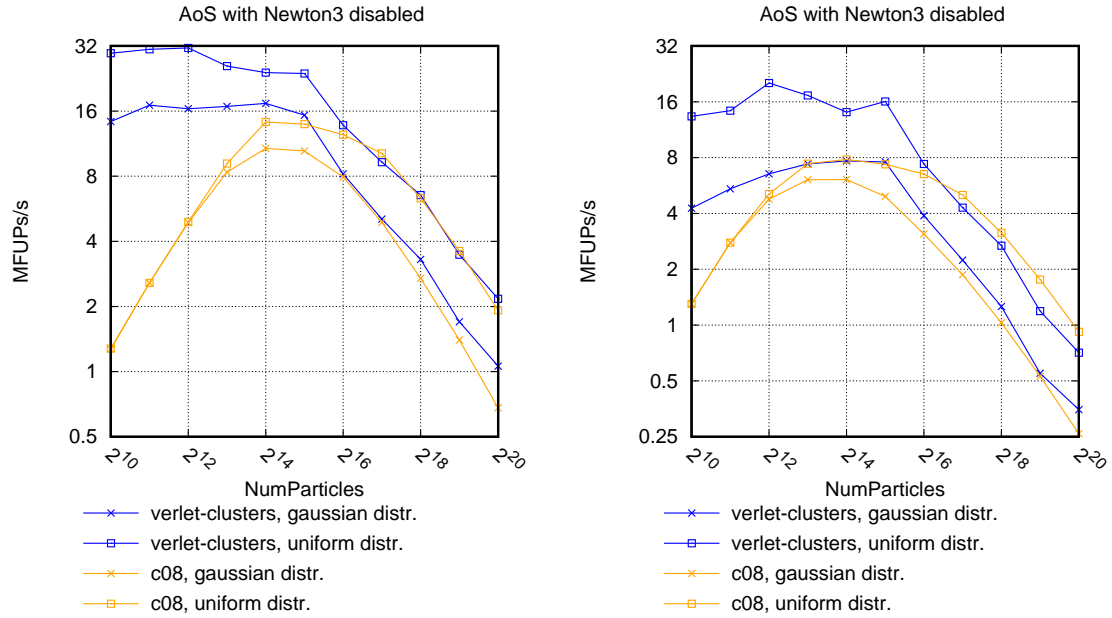


(a) Best Case Change: *verlet-clusters* traversal, SoA data layout, without Newton's third law, uniform distribution
 (b) Worst Case Change: *verlet-clusters-coloring* traversal, AoS data layout, without Newton's third law, gaussian distribution

Figure 6.1.: Best and Worst Case Performance Change through the VerletClusterLists Refactoring.

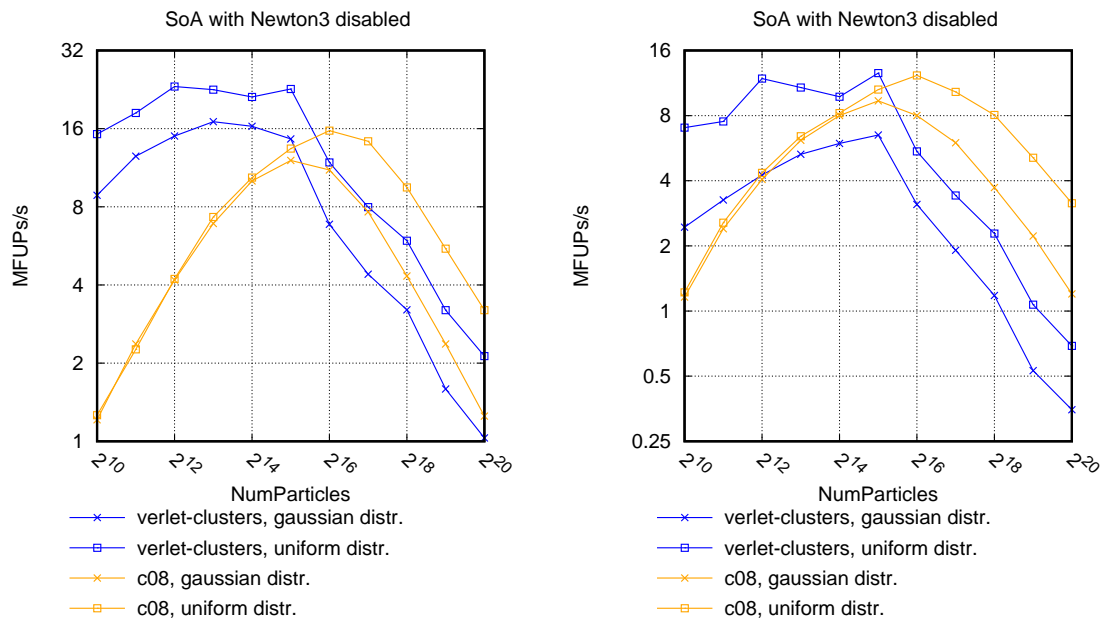
6.4. Time to Solution

6.4.1. verlet-clusters Traversal



(a) CoolMUC2: AoS with Newton3 disabled

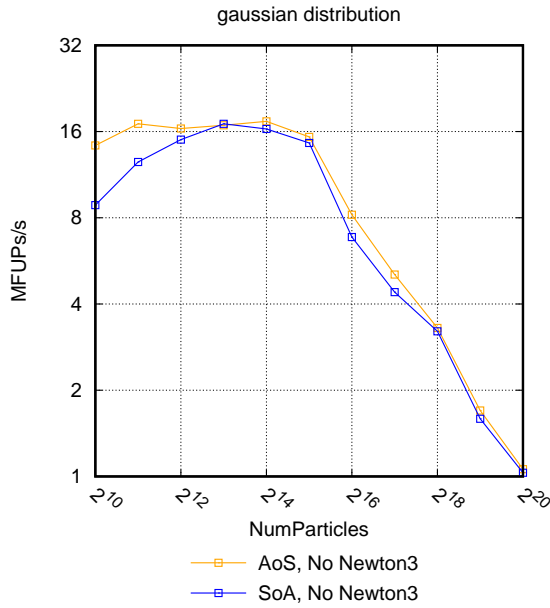
(b) CoolMUC3: AoS with Newton3 disabled



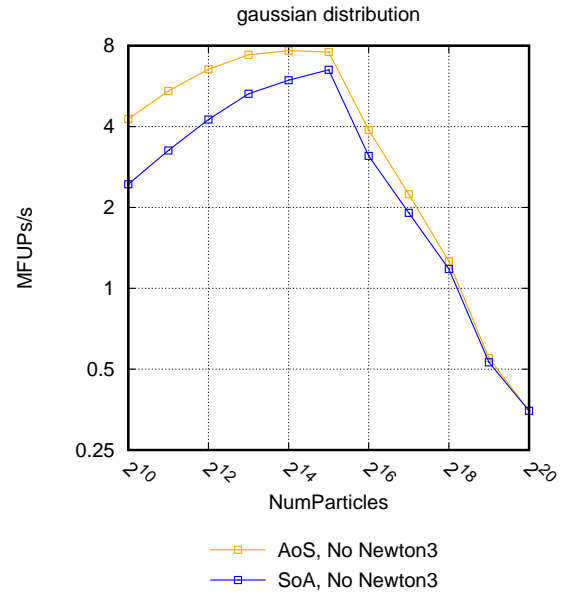
(c) CoolMUC2: SoA with Newton3 disabled

(d) CoolMUC3: SoA with Newton3 disabled

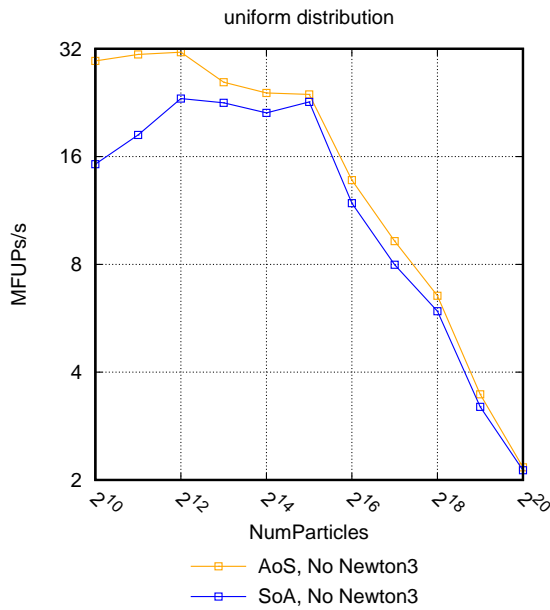
Figure 6.2.: Comparison of *Time To Solution* between *verlet-clusters* and *c08*.



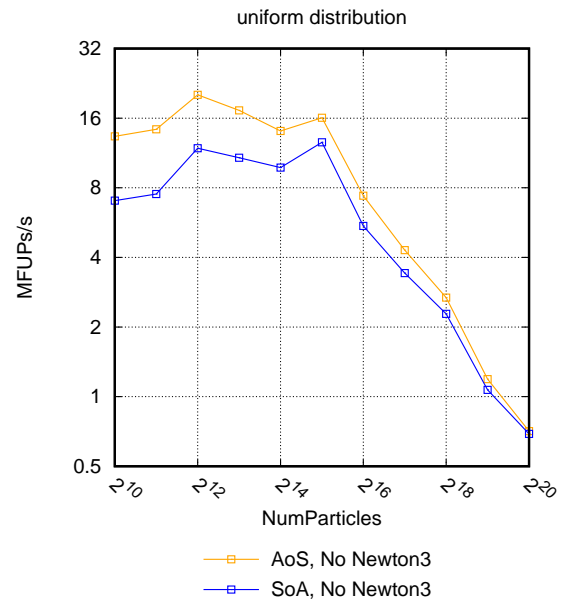
(a) CooLMUC2: gaussian distribution



(b) CooLMUC3: gaussian distribution



(c) CooLMUC2: uniform distribution



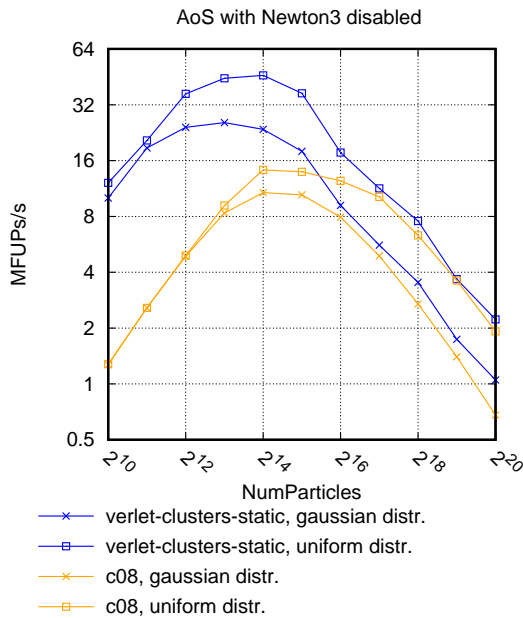
(d) CooLMUC3: uniform distribution

Figure 6.3.: Comparison of *Time To Solution* between all data layout and Newton 3 options of *verlet-clusters*.

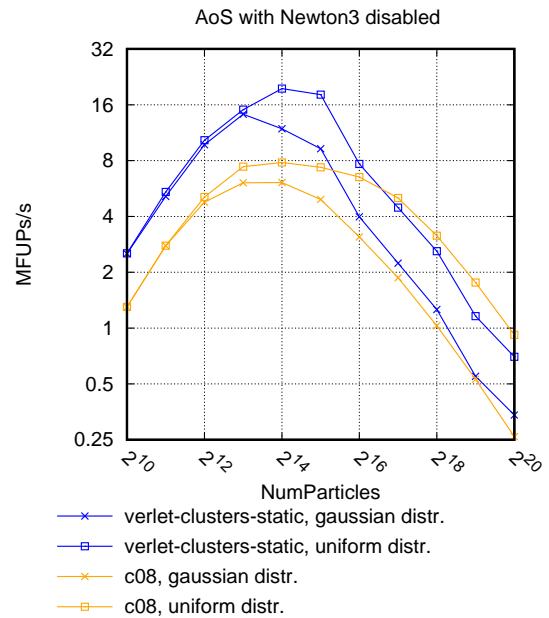
Overall, the performance of the *verlet-clusters* traversal here is always better than the *c08* traversal for a low number of particles, as it can be seen in Figure 6.2. Especially with very few particles, it sometimes is up to 20 times faster. With increasing numbers of

particles, only the AoS configuration is still competitive, as Figure 6.2a and Figure 6.2b show. The configuration using the SoA data layout gets considerably worse compared to the *c08* traversal, starting from around 2^{15} particles, as pictured in Figure 6.2a and Figure 6.2b. Figure 6.2 also shows that the performance difference between the two particle distributions is always around factor two, often smaller. This is quite different than for the *c08* traversal, where there is almost no difference with few particles, but a massive one of factors $2.5 - 4$ for the maximum number of 2^{20} particles. This suggests that the load-balancing of the *verlet-clusters* traversal works better than that of the *c08* traversal. Since both traversals use dynamic scheduling, the improvement probably comes from the smaller chunk size. Comparing the two machines, it seems that the traversal utilizes the CoolMUC2 a little bit better, since the performance of the *c08* traversal is a lot better than the *verlet-clusters* traversal on the CoolMUC3, especially with the SoA data layout. This can be seen in Figure 6.2d. Since this is more noticeable in the SoA data layout, the reason for this might be the overhead generated by this layout through loading the particles in the SoA, generating the `SoAViews`, and extracting them out again. This code is not enough optimized for many slow cores. Overall, the additional benefit of vectorization does not seem to make the SoA data layout better than the AoS data layout. It is just enough to compensate for the additional overhead in most cases, but sometimes not even that. The more particles the simulation contains, the better the SoA data layout gets. This is illustrated by Figure 6.3. Since both graphs in Figure 6.3a and in Figure 6.3c, as well as in Figure 6.3b and in Figure 6.3d, take a similar course, the particle distribution does not seem to influence the performance of the data layouts for this traversal.

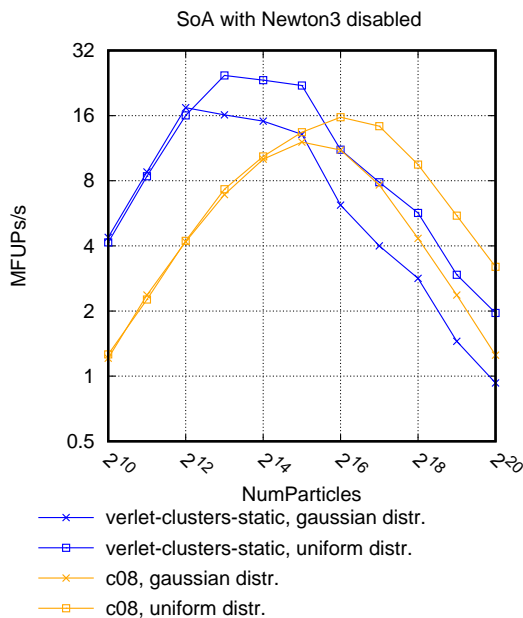
6.4.2. verlet-clusters-static Traversal



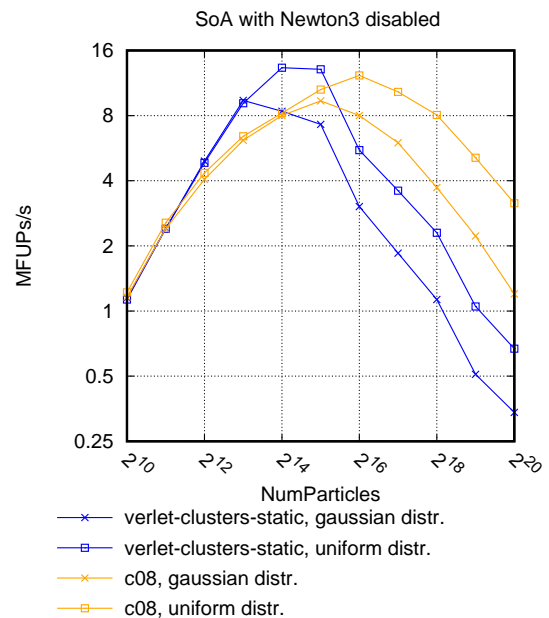
(a) CoolLMUC2: AoS with Newton3 disabled



(b) CoolLMUC3: AoS with Newton3 disabled



(c) CoolLMUC2: SoA with Newton3 disabled



(d) CoolLMUC3: SoA with Newton3 disabled

Figure 6.4.: Comparison of *Time To Solution* between *verlet-clusters-static* and *c08*.

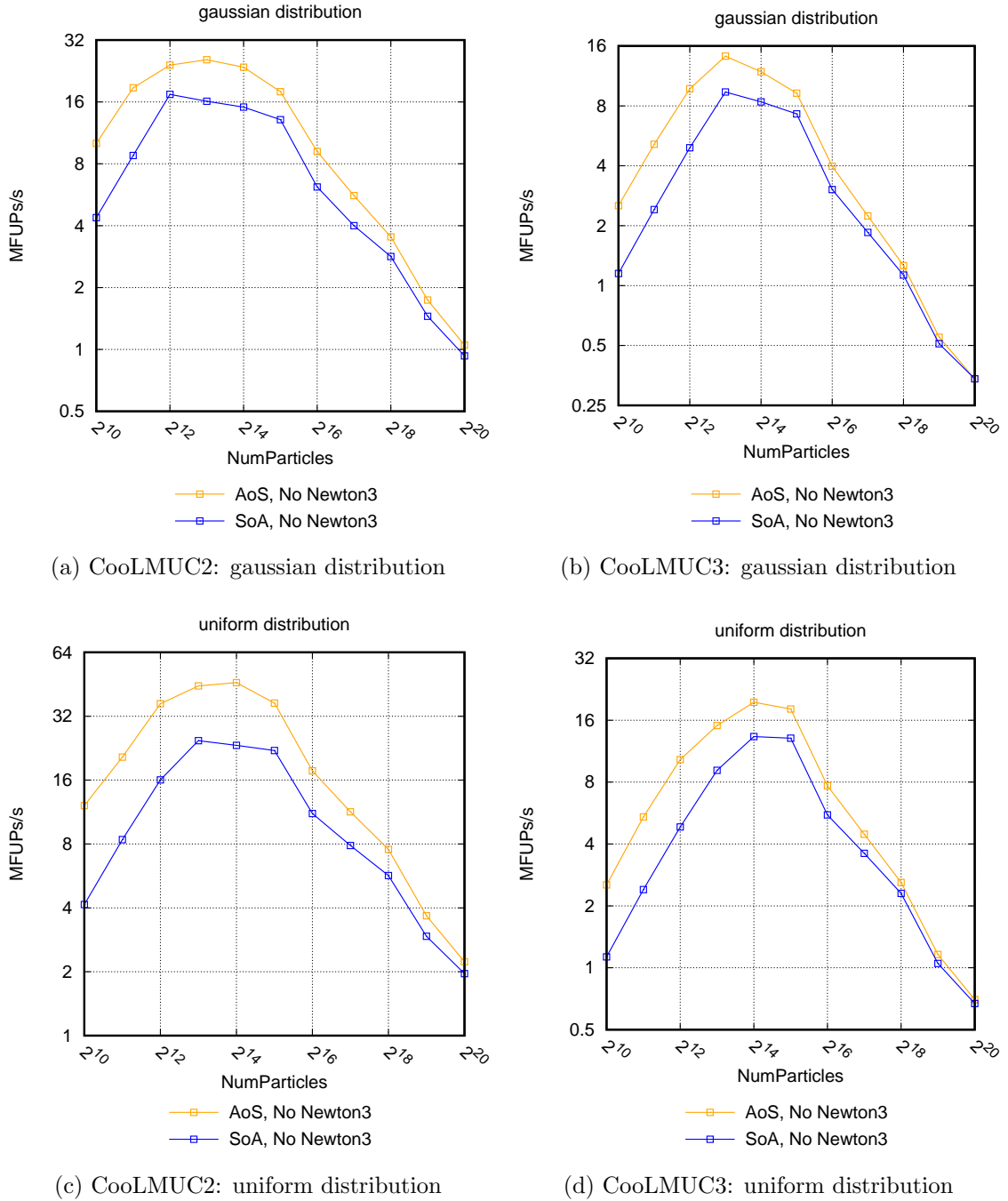
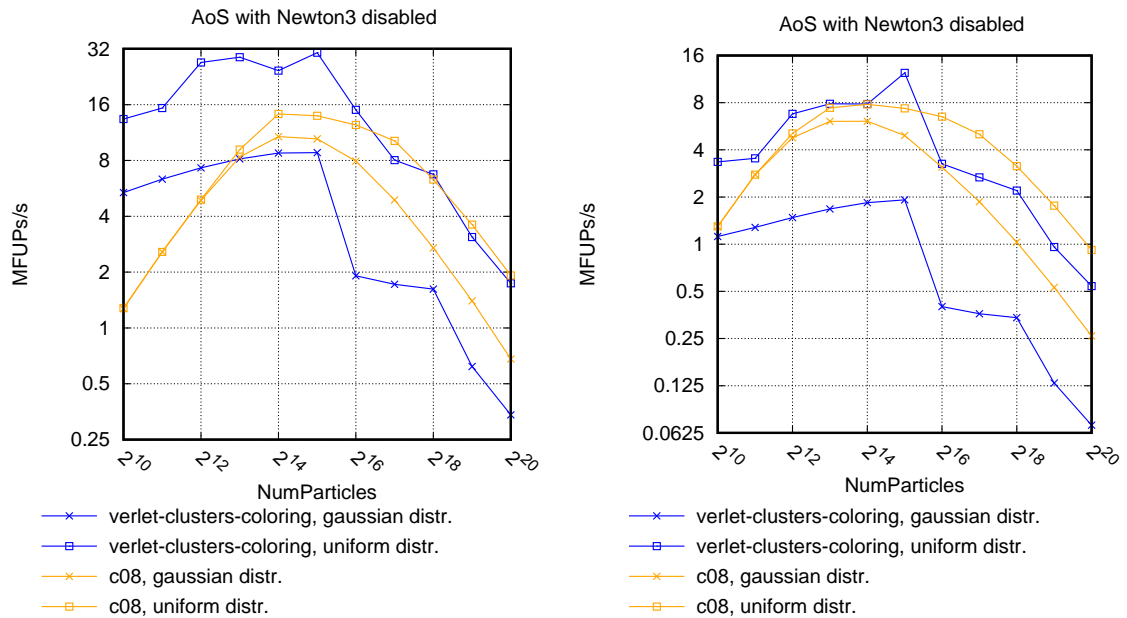


Figure 6.5.: Comparison of *Time To Solution* between all data layout and Newton 3 options of *verlet-clusters-static*.

The performance of the *verlet-clusters-static* traversal is expected to be similar to the performance of the *verlet-clusters* traversal. This is mostly true. It is also significantly better with a low to medium number of particles on most configurations and at least similar

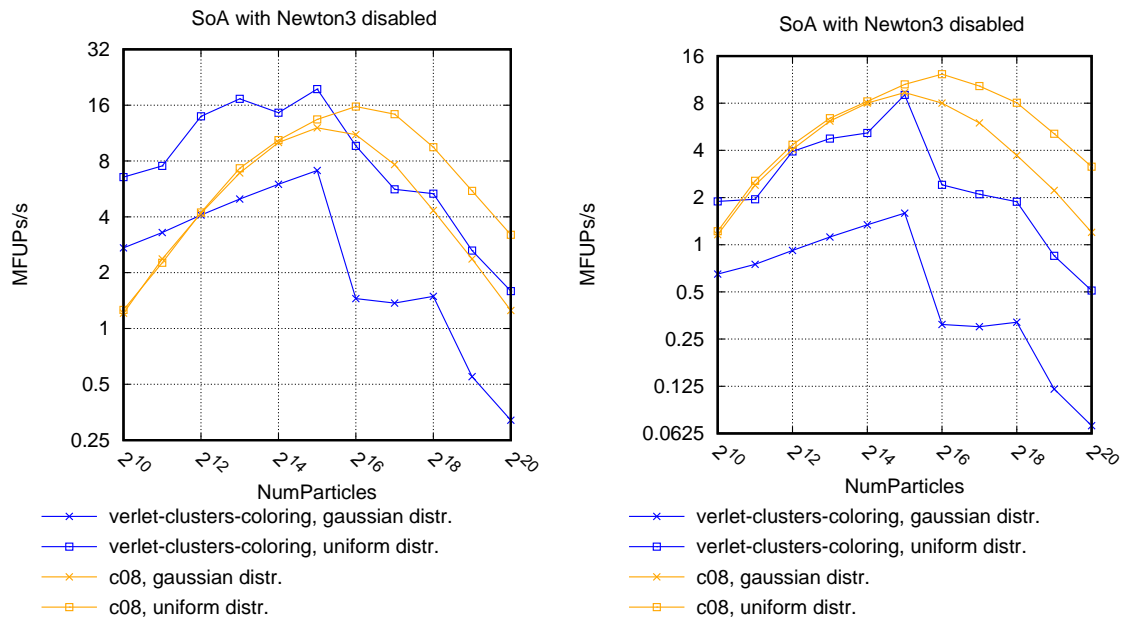
with the SoA data layout on the CooLMUC3, compared with the *c08* traversal. This is pictured in Figure 6.4. In the AoS data layout, it even outperforms the *verlet-clusters* traversal. While the *verlet-clusters* traversal always stays below 32 MFUPs in Figure 6.2a, the *verlet-clusters-static* traversal exceeds it for $2^{12} - 2^{15}$ particles in Figure 6.4a. However, it shares the same problems as its predecessor of the SoA data layout and is worse on CooLMUC3, as it can be seen in Figure 6.4c and Figure 6.4d. One distinction is that the difference between both particle distributions is more similar to the *c08* traversal than to the *verlet-clusters* traversal for only a few particles. This holds for all machines and data layouts as illustrated by Figure 6.4. The reason for this might be the absence of the scheduling overhead. Comparing the data layouts against each other with both particle distributions shows, similar to the *verlet-clusters* traversal, that the SoA data layout performs worse overall, but gets more competitive with a higher number of particles, as pictured in Figure 6.5. Looking at these figures, one also notices that changing the particle distribution does not change how the data layouts perform relative to each other. Figure 6.5a and Figure 6.5c look very similar, and so do Figure 6.5b and Figure 6.5d. Overall, all this suggests that the static load balancing for this traversal works very well and the performance gained through less scheduling overhead is greater than the loss through unbalanced threads.

6.4.3. verlet-clusters-coloring Traversal



(a) CoolMUC2: AoS with Newton3 disabled

(b) CoolMUC3: AoS with Newton3 disabled

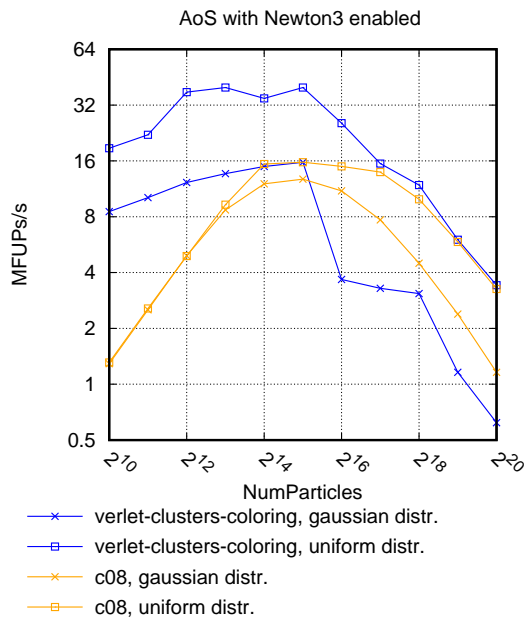


(c) CoolMUC2: SoA with Newton3 disabled

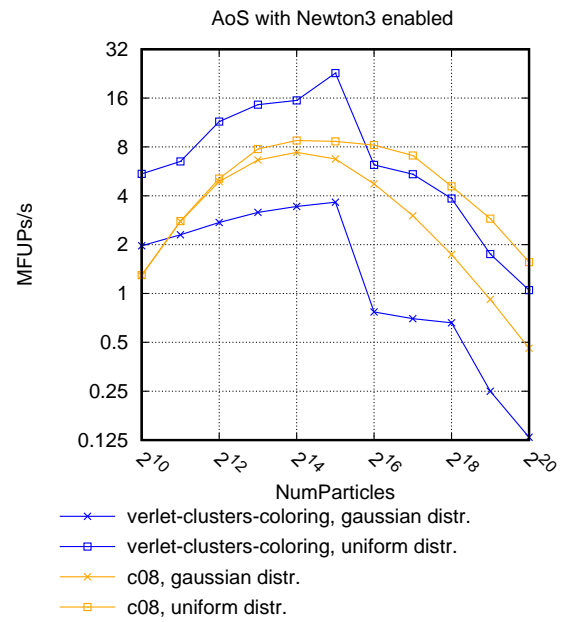
(d) CoolMUC3: SoA with Newton3 disabled

Figure 6.6.: Comparison of *Time To Solution* between *verlet-clusters-coloring* and *c08* without Newton 3.

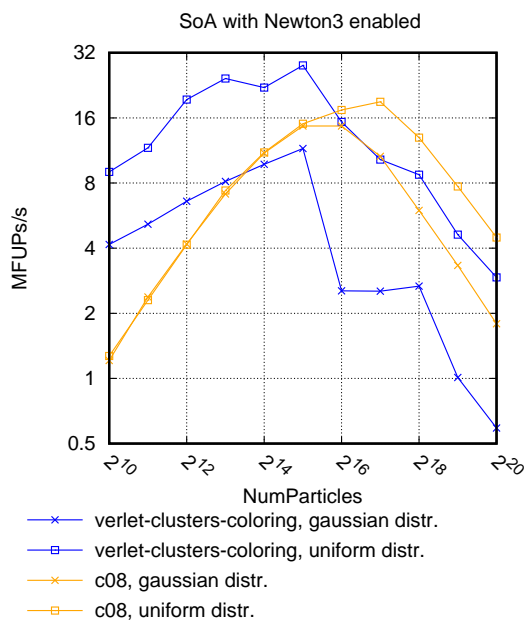
6. Experiments and Analysis



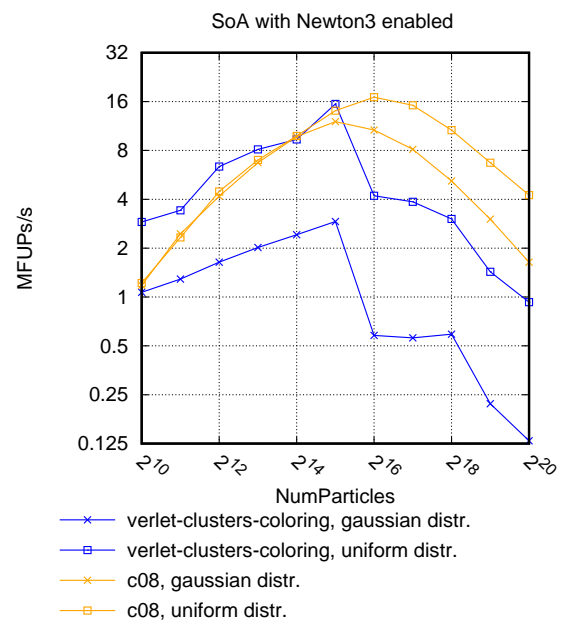
(a) CoolMUC2: AoS with Newton3 enabled



(b) CoolMUC3: AoS with Newton3 enabled



(c) CoolMUC2: SoA with Newton3 enabled



(d) CoolMUC3: SoA with Newton3 enabled

Figure 6.7.: Comparison of *Time To Solution* between *verlet-clusters-coloring* and *c08* with Newton 3.

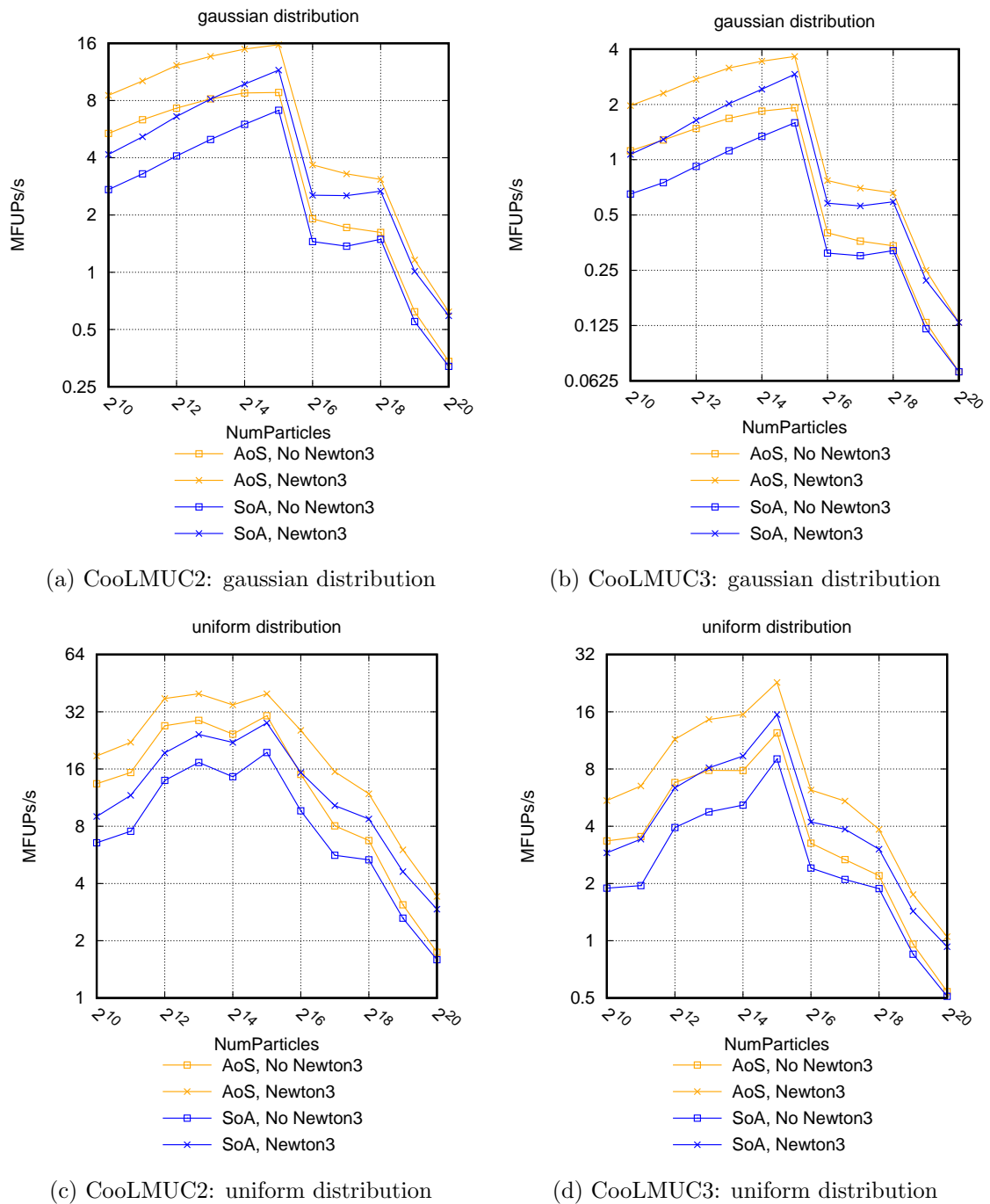


Figure 6.8.: Comparison of *Time To Solution* between all data layout and Newton 3 options of *verlet-clusters-coloring*.

The *Time To Solution* of the *verlet-clusters-coloring* traversal follows a curious course. No graph is smooth, but almost all have steps in it. This can be noticed in Figure 6.6 and Figure 6.7. Starting at 2^{15} particles, the performance decreases rapidly. Only between

2^{16} and 2^{18} particles it stays about the same. When comparing the uniform particle distribution on both machines Figure 6.8c and Figure 6.8d with the gaussian distributions Figure 6.8a and Figure 6.8b, one quickly notices that this behavior occurs way stronger on the gaussian distribution. In the uniform distribution, this behavior almost only appears on the CoolMUC3. This suggests that bad load-balancing is the cause for the steps since this is the main challenge of the gaussian distribution and the CoolMUC3 with more, but slower cores. However, further research is required to find the exact cause of this behavior. Subsection 6.4.4 describes a reason for why the performance starts to decrease at 2^{15} particles in all test runs, as it can be seen in Figure 6.6 and Figure 6.7.

From all configurations, the configuration with the AoS data layout that uses Newton's third law works best. Figure 6.6 and Figure 6.7 show that it is the only configuration capable of constantly outperforming the *c08* traversal with uniform particle distribution on the CoolMUC2. In Figure 6.8, it also outperforms all other configurations for both data layouts on both machines. As using Newton's third law for the `VerletClusterLists` container was the objective for this traversal, it is nice to see that this expectation was fulfilled. The SoA data layout only performs acceptably with small numbers of particles, as Figure 6.6c, Figure 6.6d, Figure 6.7c, and Figure 6.7d show. Especially for the gaussian data layout, the performance is very bad. Overall, AoS works faster than SoA for all configurations, as it can be seen in Figure 6.8. Furthermore, there is a wide difference between the two distributions in all graphs of Figure 6.6 and Figure 6.7. This suggests that the load-balancing in this traversal works not as good as hoped. The reason for this is probably the chunk size. Since in the gaussian distribution, most particles are centered in the middle, only a few color cells contain almost all particles. Since these are additionally neighbors and cannot be worked on in parallel, the load-balancing there is very bad. Although they are two very different machines, the traversal behaves nearly equally on CoolMUC2 and CoolMUC3.

6.4.4. Overall Interpretation for the `VerletClusterLists` Container

The `VerletClusterLists` container definitely holds more potential to use vectorization for Verlet Lists, as the `VarVerletListsAsBuild` container performs worse there. A problem that probably holds back performance for larger numbers of particles is the calculation of the tower side length. In all traversals of this container, the performance starts to decrease more rapidly after 2^{15} particles. The reader may verify that in Figure 6.2, Figure 6.4, Figure 6.6, and Figure 6.7. Figure 6.9 shows how the tower side length develops with increasing particles in the test simulations. It can be seen that the tower side length starts to be lower than the interaction length at almost exactly these 2^{15} particles. This conspicuity might be the reason for why the performance falls so rapidly at this point. In the future, other heuristics for calculating the tower side length for the `VerletClusterLists` container should be tested to overcome this.

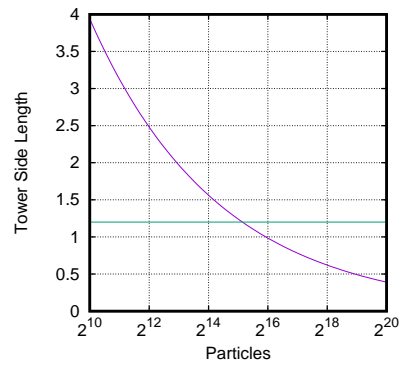
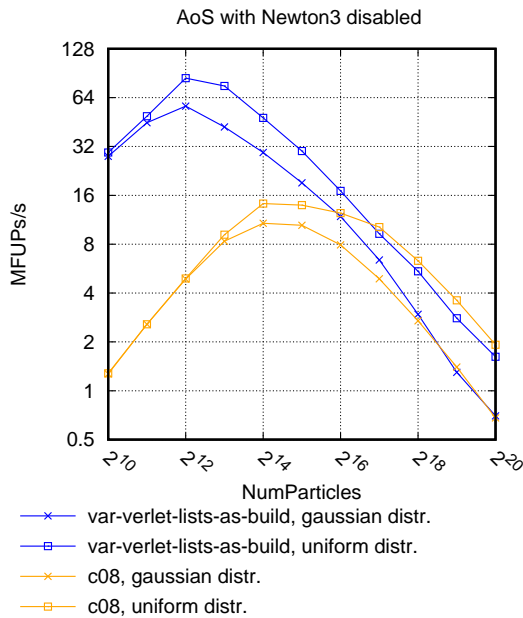
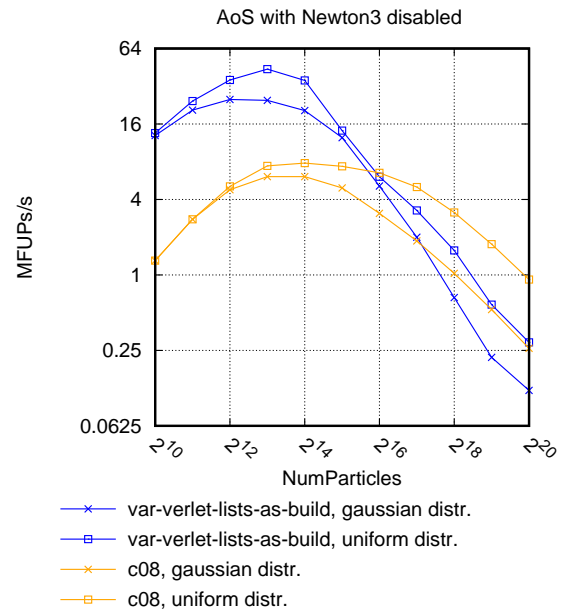


Figure 6.9.: Tower Side Length Depending on the Number of Particles in the Measurements.
The green line shows the interaction length.

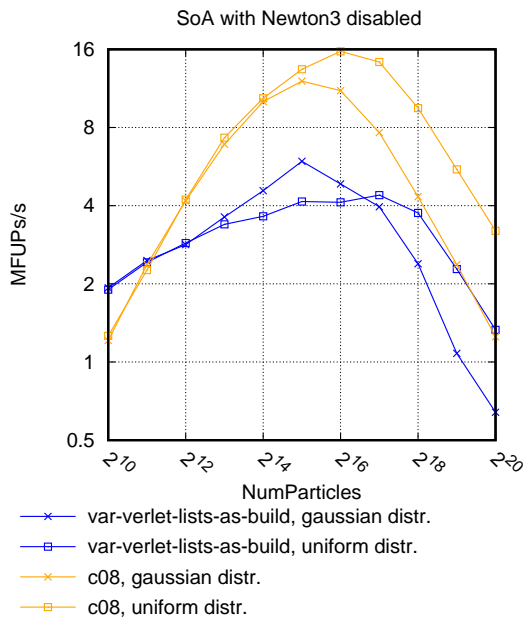
6.4.5. var-verlet-lists-as-build Traversal



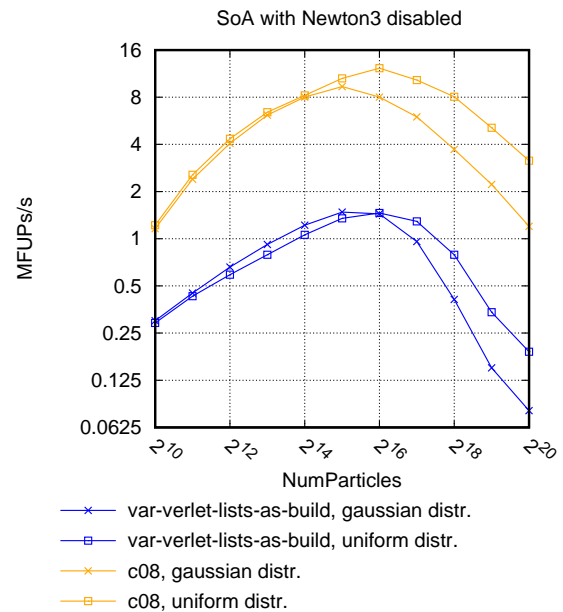
(a) CoolLMUC2: AoS with Newton3 disabled



(b) CoolLMUC3: AoS with Newton3 disabled

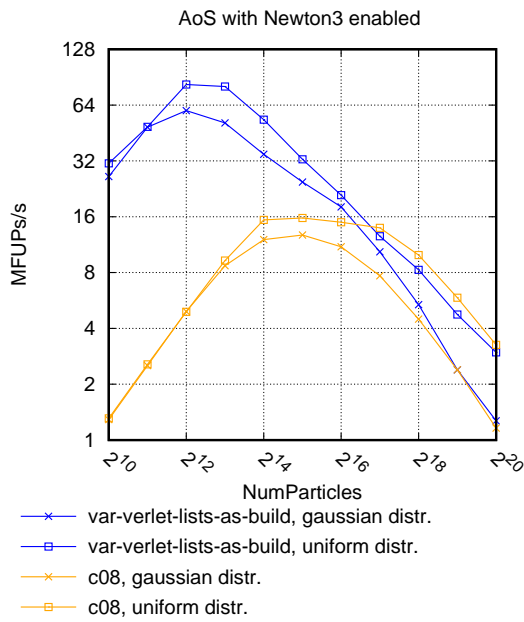


(c) CoolLMUC2: SoA with Newton3 disabled

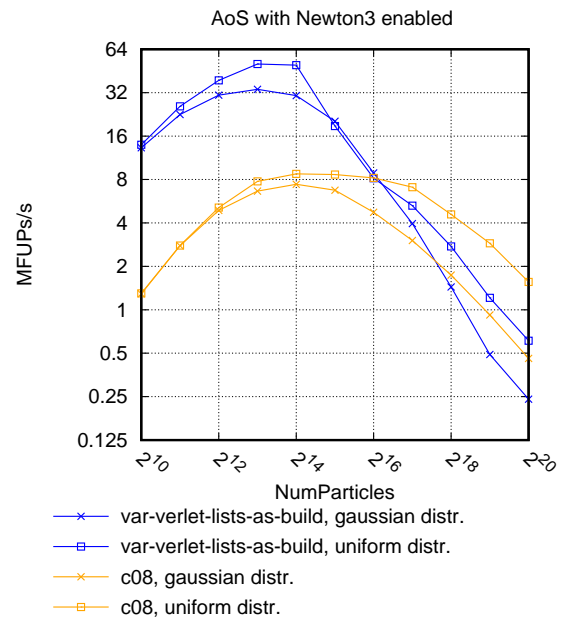


(d) CoolLMUC3: SoA with Newton3 disabled

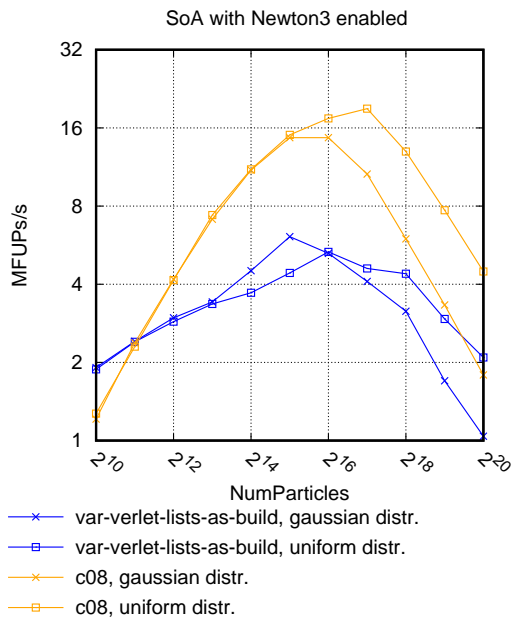
Figure 6.10.: Comparison of *Time To Solution* between *var-verlet-lists-as-build* and *c08* without Newton 3.



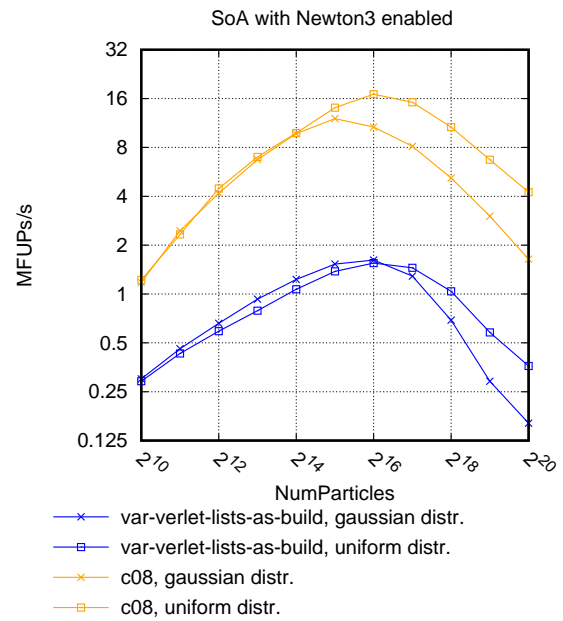
(a) CooLMUC2: AoS with Newton3 enabled



(b) CooLMUC3: AoS with Newton3 enabled



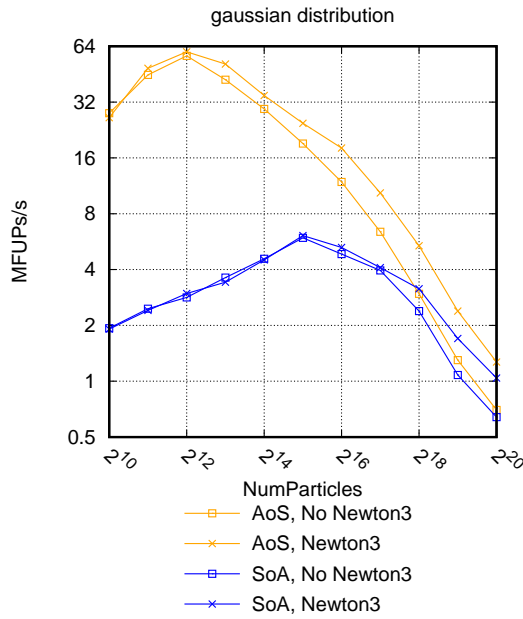
(c) CooLMUC2: SoA with Newton3 enabled



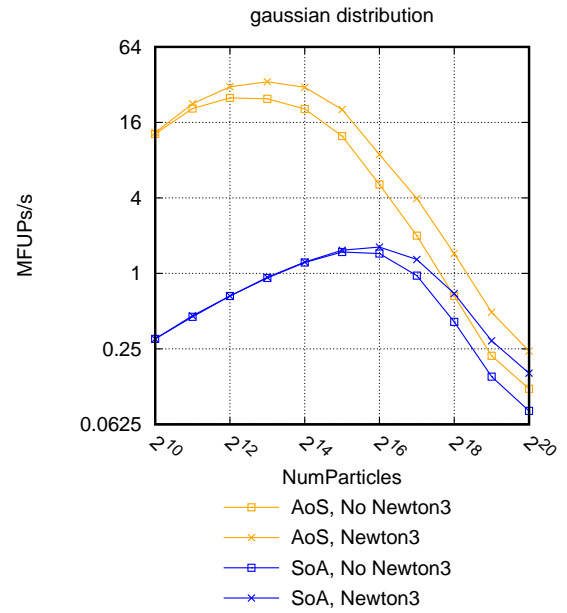
(d) CooLMUC3: SoA with Newton3 enabled

Figure 6.11.: Comparison of *Time To Solution* between *var-verlet-lists-as-build* and *c08* with Newton 3.

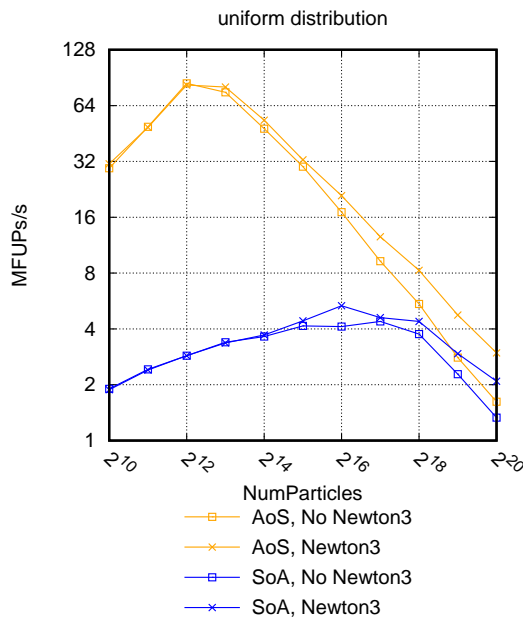
6. Experiments and Analysis



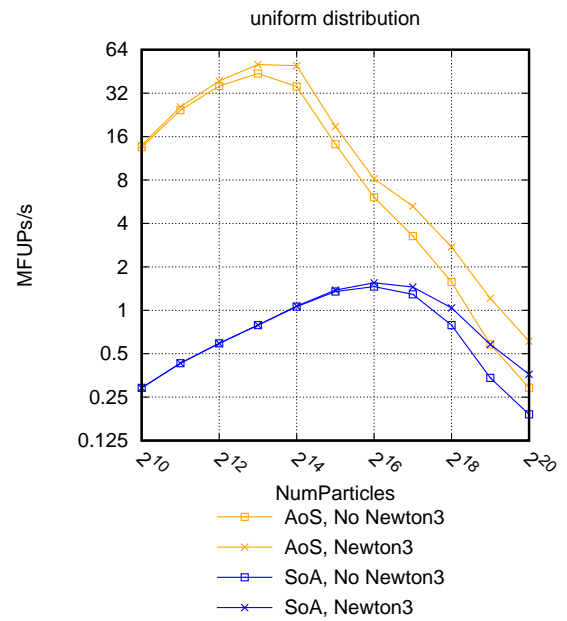
(a) CoolLMUC2: gaussian distribution



(b) CoolLMUC3: gaussian distribution



(c) CoolLMUC2: uniform distribution



(d) CoolLMUC3: uniform distribution

Figure 6.12.: Comparison of *Time To Solution* between all data layout and Newton 3 options of *var-verlet-lists-as-build*.

The *var-verlet-lists-as-build* traversal also works best for small to medium numbers of particles in the AoS data layout, as it can be seen in Figure 6.10a, Figure 6.10b, Figure 6.11a, and Figure 6.11a. In these cases, it outperforms the *c08* traversal. The SoA data layout is

not competitive at all at medium numbers of particles. With a very high number of particles, it closes the distance to the AoS variant a little, but performs worse in all test runs, as shown by Figure 6.12. The reason for this is not known. Since comparing Figure 6.10c and Figure 6.10d, as well as Figure 6.11c and Figure 6.11d, shows that the SoA data layout performs especially bad on the CoolMUC3 and the load-balancing is the same for both data layouts, the reason might be related to loading and extracting the SoA, or generating and accessing the SoA neighbor lists. The overall behavior of using Newton's third law or not is very similar, with using it having a noticeable performance gain of up to almost factor 2 with a higher number of particles, as visualized in Figure 6.12. The performance differences between both distributions, especially for higher numbers of particles, are mostly similar to the *c08* traversal, as shown in Figure 6.10 and Figure 6.11. This was expected as the load-balancing of both traversals is the same.

6.5. Strong Scaling

6.5.1. verlet-clusters Traversal

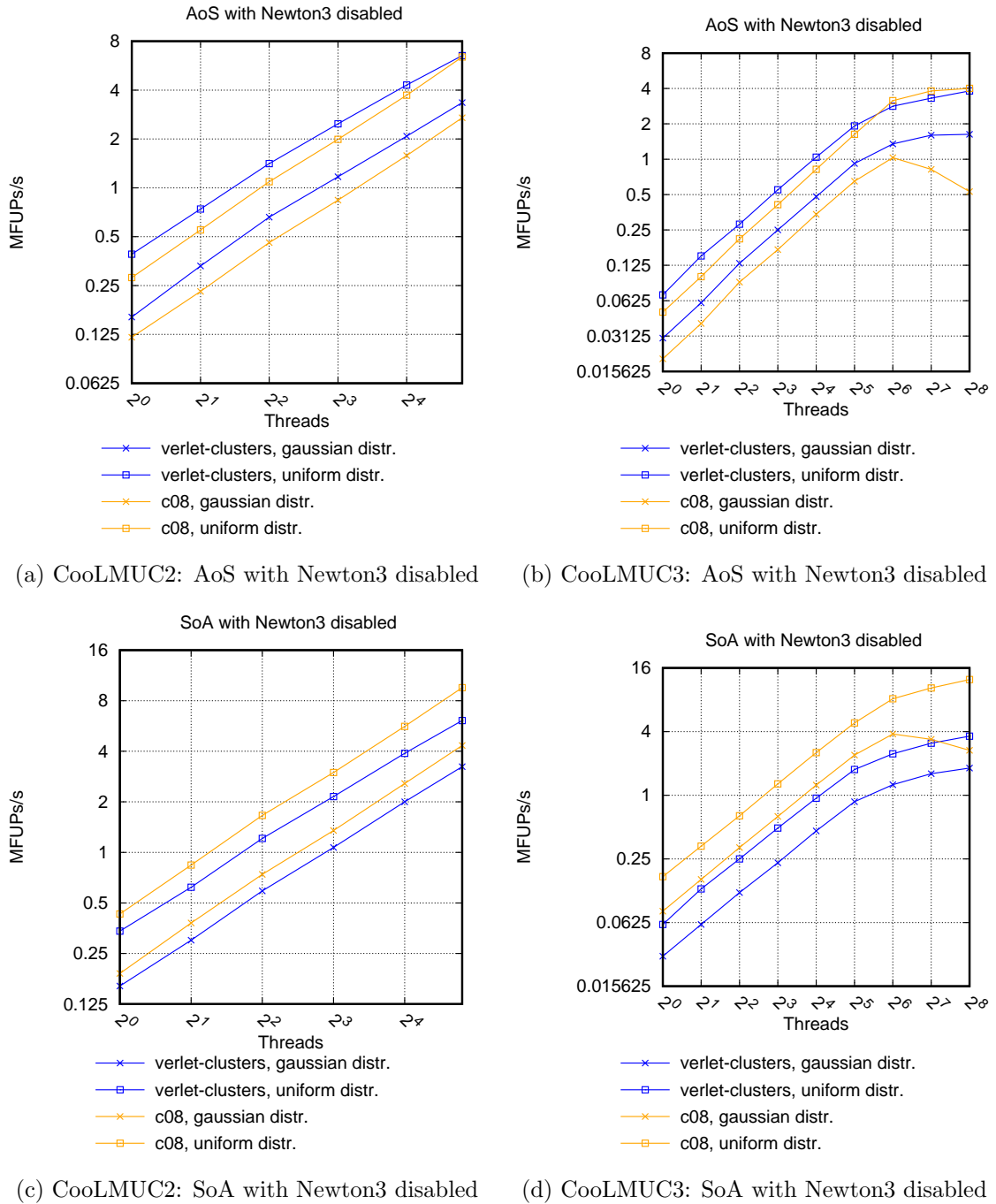
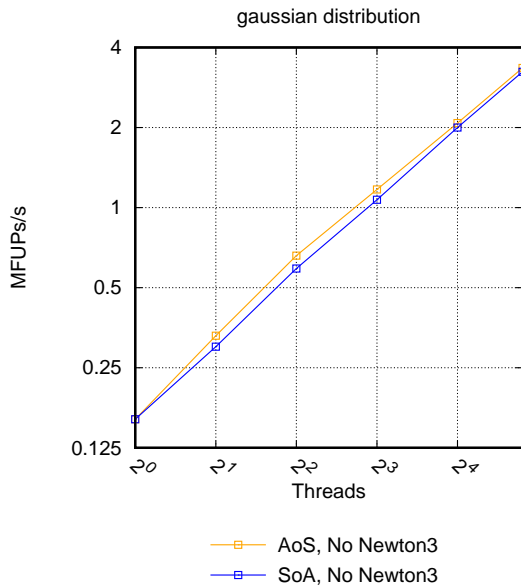
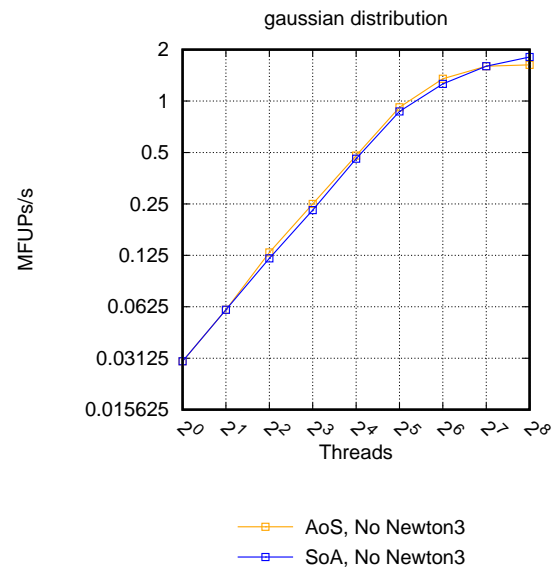


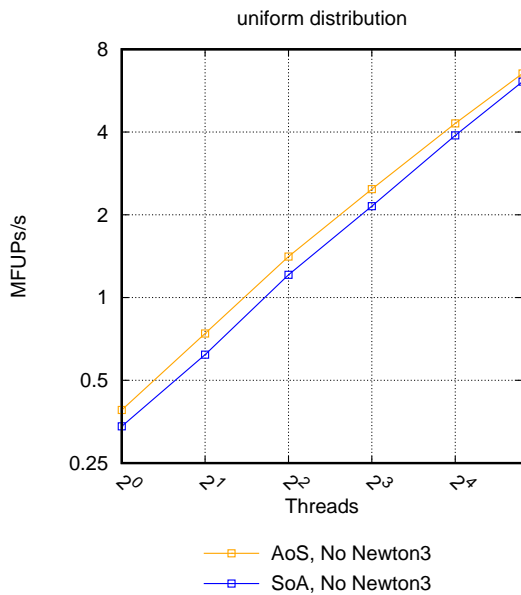
Figure 6.13.: Comparison of *Strong Scaling* between *verlet-clusters* and *c08*.



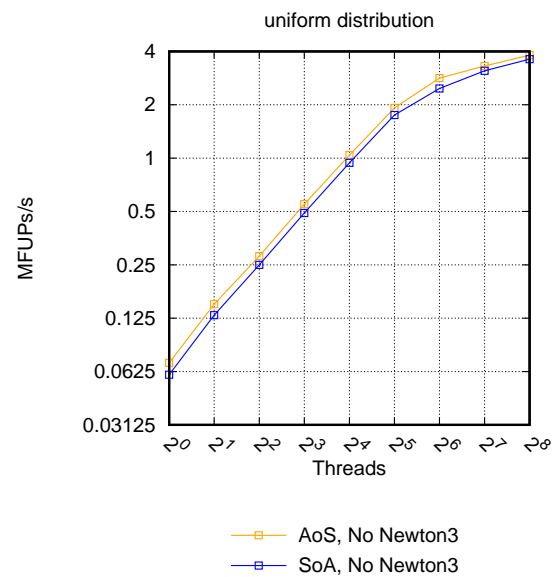
(a) CooLMUC2: gaussian distribution



(b) CooLMUC3: gaussian distribution



(c) CooLMUC2: uniform distribution



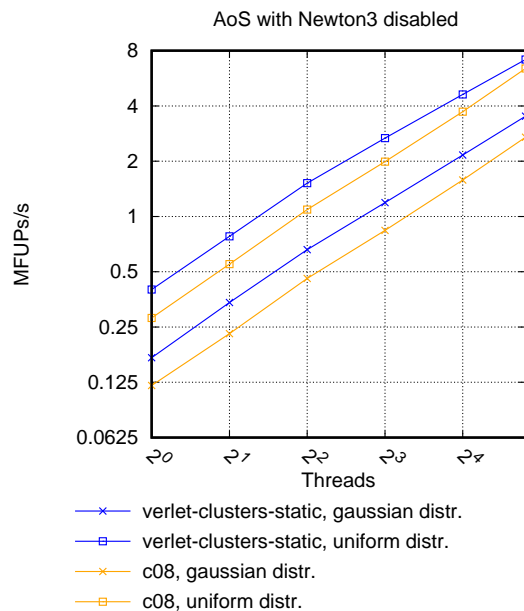
(d) CooLMUC3: uniform distribution

Figure 6.14.: Comparison of *Strong Scaling* between all data layout and Newton 3 options of *verlet-clusters*.

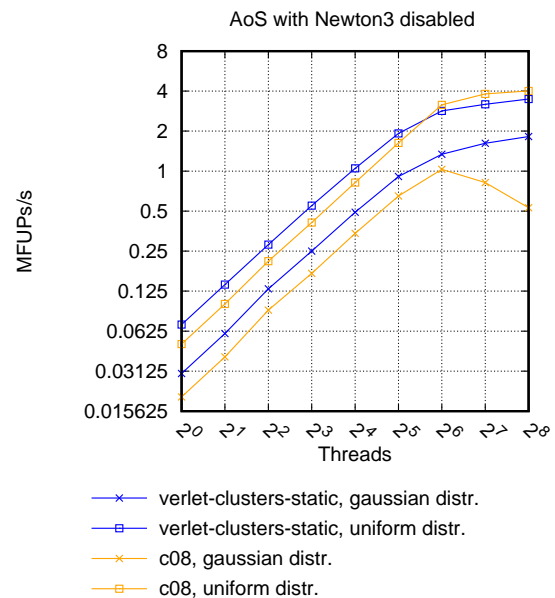
The *Strong Scaling* of the *verlet-clusters* traversal is quite good. On the CooLMUC2, it is linear in all test runs with a gradient slightly less than one, as shown in Figure 6.14a and Figure 6.14c. On the CooLMUC3, it starts equally but gets a little worse starting from 32

threads and more starting from 64, as it can be seen in Figure 6.14b and Figure 6.14d. At 64 threads, hyper-threading begins, so it is expected to get worse at this point. Nevertheless, the traversal's speed increases until the maximum of 256 threads for both distributions. As pictured in Figure 6.14b, the SoA layout even scales better to 256 threads than the AoS data layout. Compared to the *c08* traversal, Figure 6.13a and Figure 6.13b show that the performance and scaling for the AoS data layout and uniform distribution is competitive. When the particle distribution is gaussian, the *verlet-clusters* traversal is able to even better scale up to 256 threads on CoolMUC3 than the *c08* traversal, since it is able to benefit from hyper-threading. The *c08* traversal loses speed when attempting that. Figure 6.13b and Figure 6.13d illustrate that. While the overall performance of the SoA data layout is worse compared to the *c08* traversal, it scales equally good or better on the CoolMUC3, as it can be seen in Figure 6.13d and Figure 6.13c. Impressing are the nearly similar courses of the graphs for both distributions in all diagrams of Figure 6.13. The chunk sizes in this traversal are small enough to even scale well for 256 threads and the gaussian distribution.

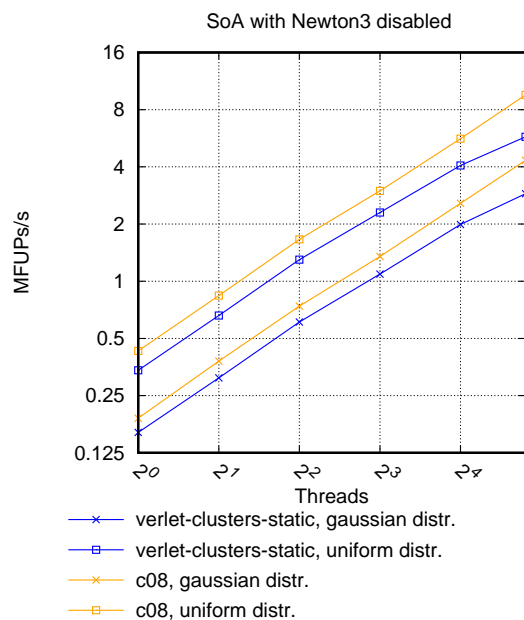
6.5.2. verlet-clusters-static Traversal



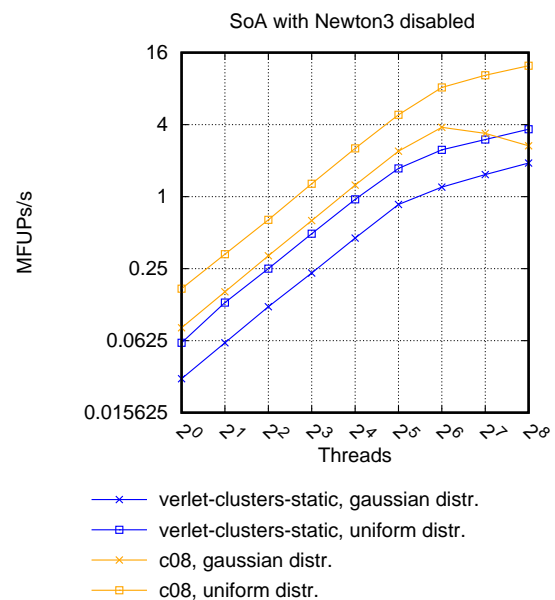
(a) CoLMUC2: AoS with Newton3 disabled



(b) CoLMUC3: AoS with Newton3 disabled

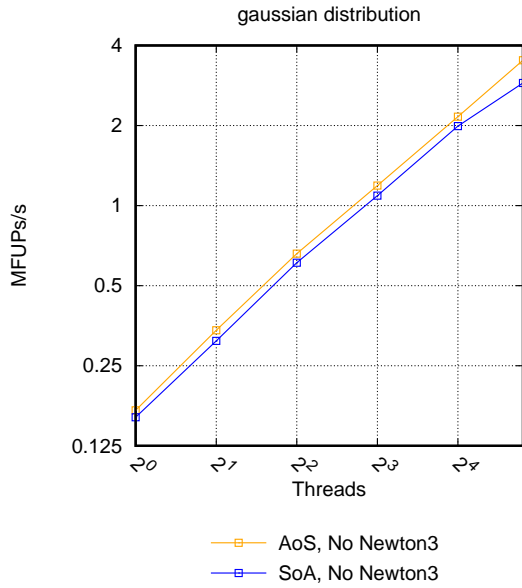


(c) CoLMUC2: SoA with Newton3 disabled

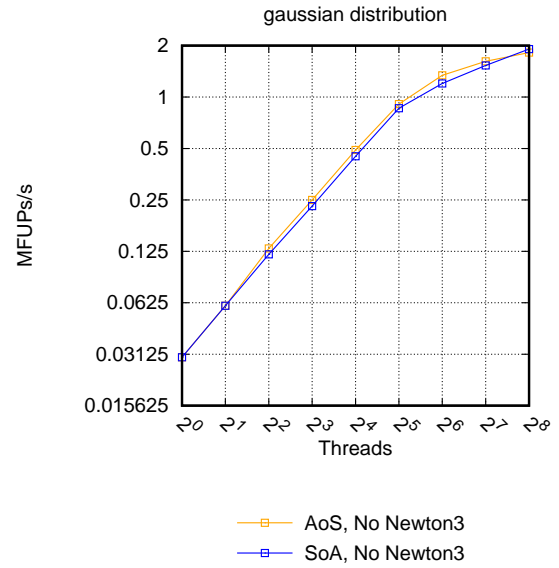


(d) CoLMUC3: SoA with Newton3 disabled

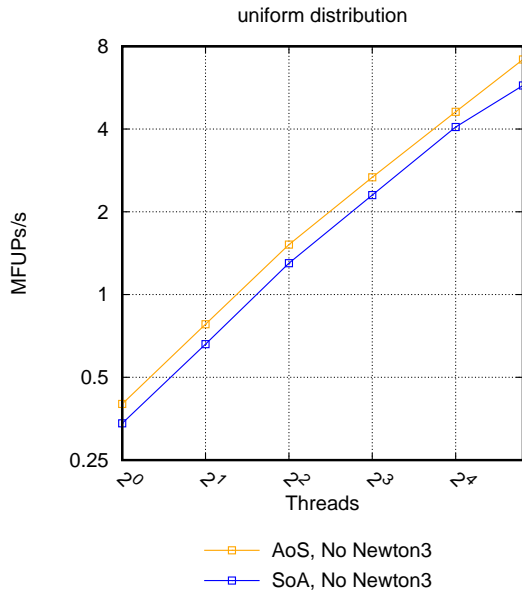
Figure 6.15.: Comparison of *Strong Scaling* between *verlet-clusters-static* and *c08*.



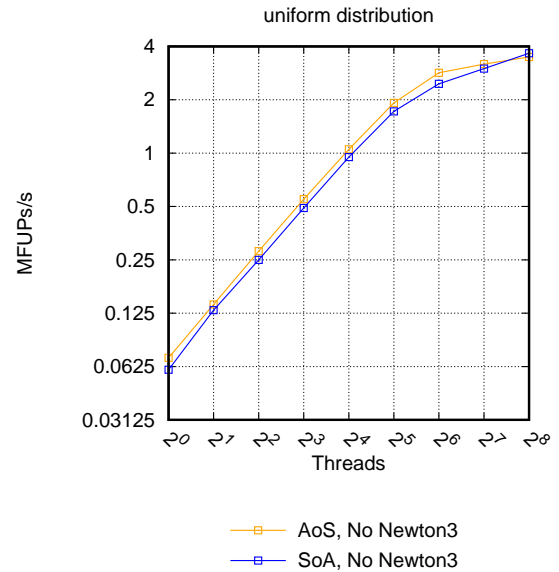
(a) CooLMUC2: gaussian distribution



(b) CooLMUC3: gaussian distribution



(c) CooLMUC2: uniform distribution



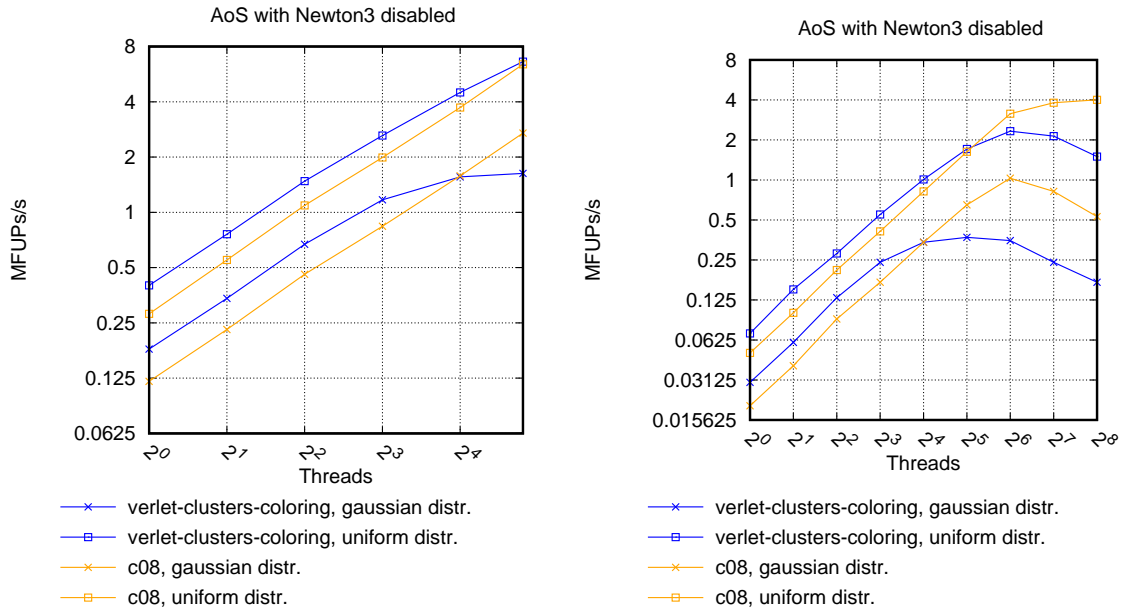
(d) CooLMUC3: uniform distribution

Figure 6.16.: Comparison of *Strong Scaling* between all data layout and Newton 3 options of *verlet-clusters-static*.

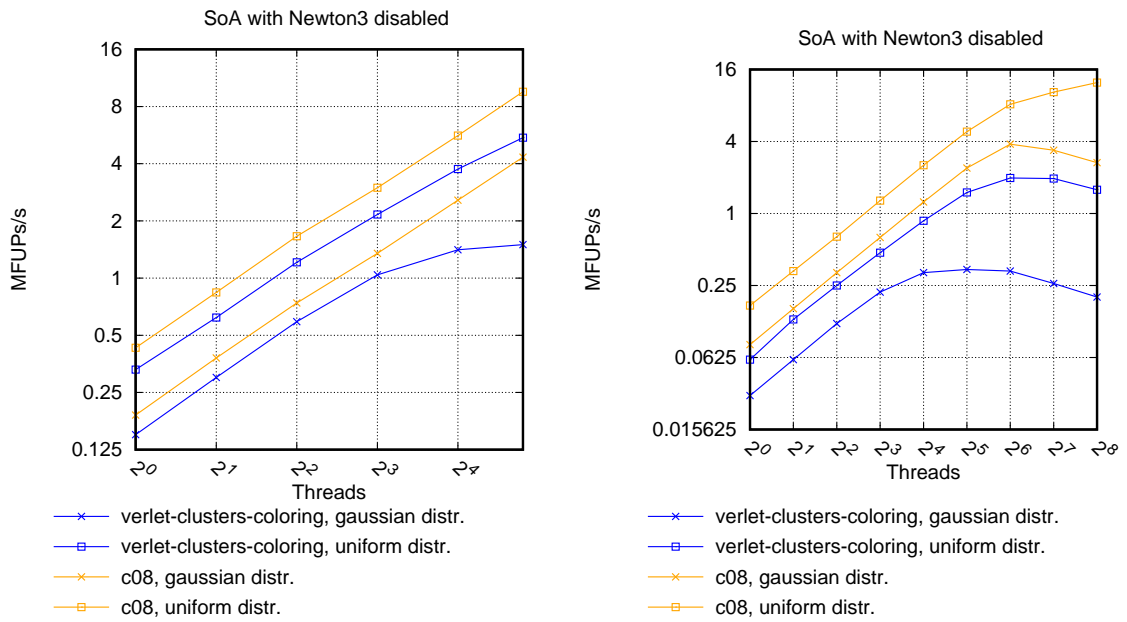
The strong scaling of the *verlet-cluster-static* is important to examine since it describes how well the static load-balancing is calculated for different numbers of threads. Figure 6.16 shows that it works. On CooLMUC2, all graphs are linear with a gradient slightly less

than one, as it can be seen in Figure 6.16a and Figure 6.16c. Even on CooLMUC3, it scales linearly until 32 threads and almost until 64 threads for both particle distributions, and also benefits from hyper-threading, as it is pictured in Figure 6.16b and Figure 6.16d. The overall scaling of the *verlet-clusters-static* traversal is very similar to the scaling of the *verlet-clusters* traversal. When comparing Figure 6.13 with Figure 6.15, almost no difference exists, although the first uses dynamic scheduling, and the second uses static scheduling. This confirms that the correct heuristic for the cost was chosen for generating the static cluster-thread distribution.

6.5.3. verlet-clusters-coloring Traversal

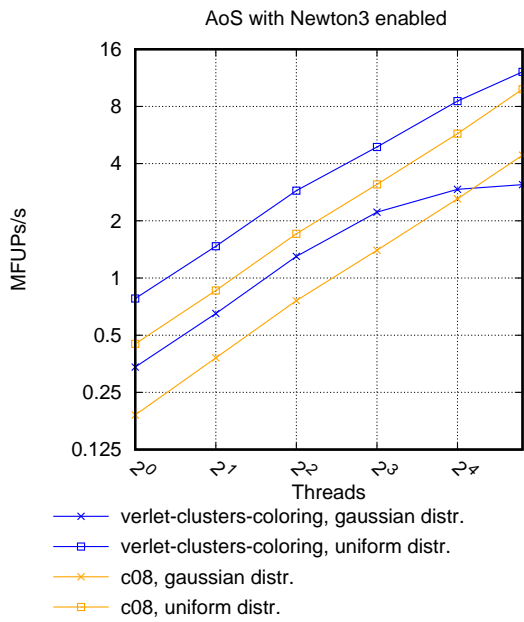


(a) CoolMUC2: AoS with Newton3 disabled (b) CoolMUC3: AoS with Newton3 disabled

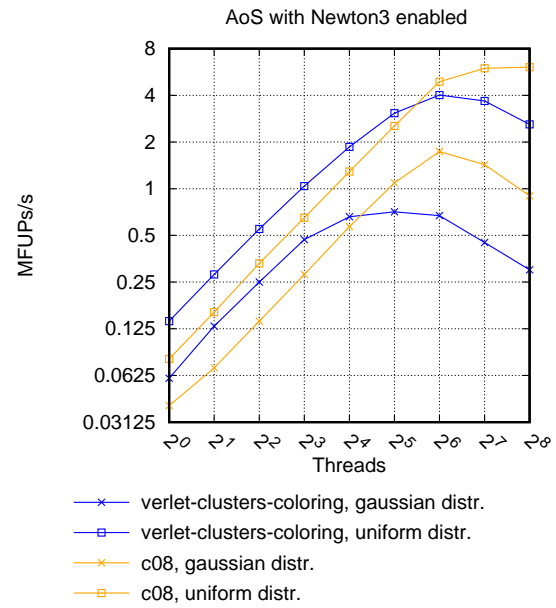


(c) CoolMUC2: SoA with Newton3 disabled (d) CoolMUC3: SoA with Newton3 disabled

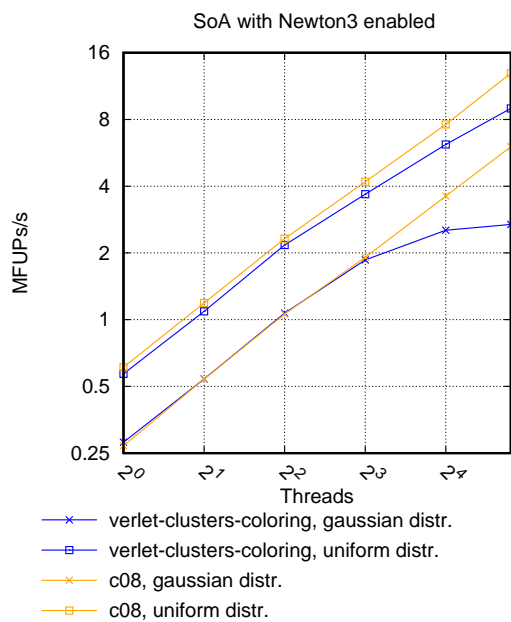
Figure 6.17.: Comparison of *Strong Scaling* between *verlet-clusters-coloring* and *c08* without Newton 3.



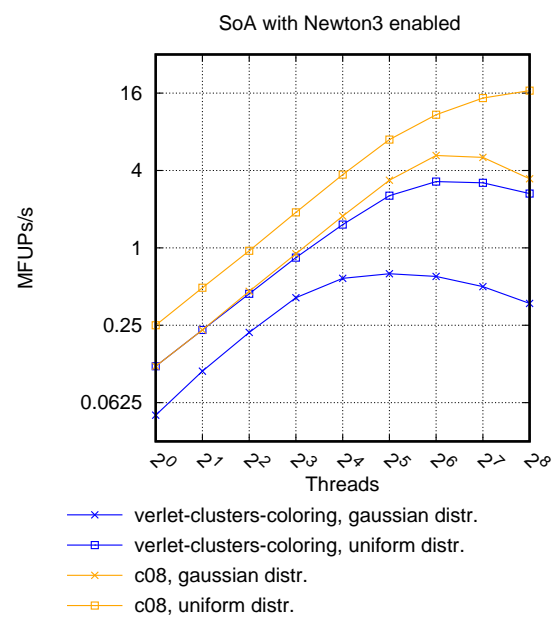
(a) CooLMUC2: AoS with Newton3 enabled



(b) CooLMUC3: AoS with Newton3 enabled

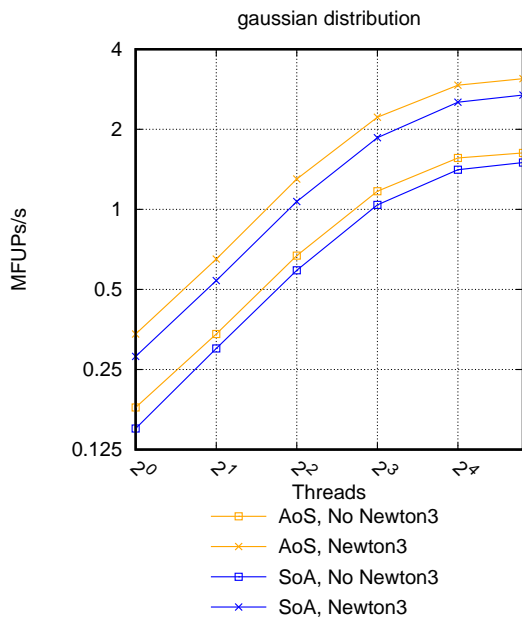


(c) CooLMUC2: SoA with Newton3 enabled

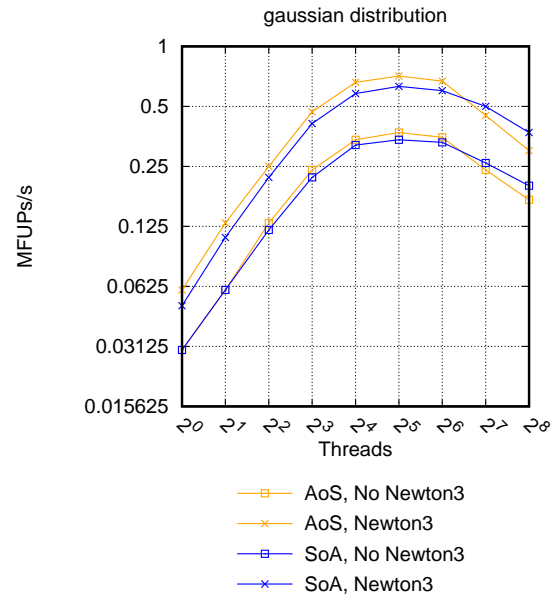


(d) CooLMUC3: SoA with Newton3 enabled

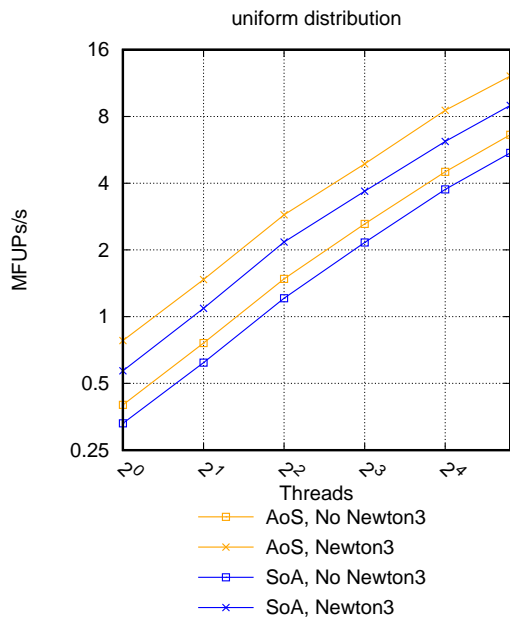
Figure 6.18.: Comparison of *Strong Scaling* between *verlet-clusters-coloring* and *c08* with Newton 3.



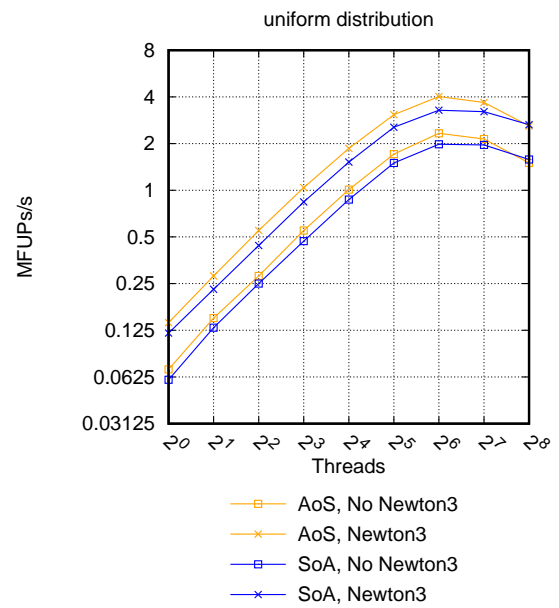
(a) CoolLMUC2: gaussian distribution



(b) CoolLMUC3: gaussian distribution



(c) CoolLMUC2: uniform distribution



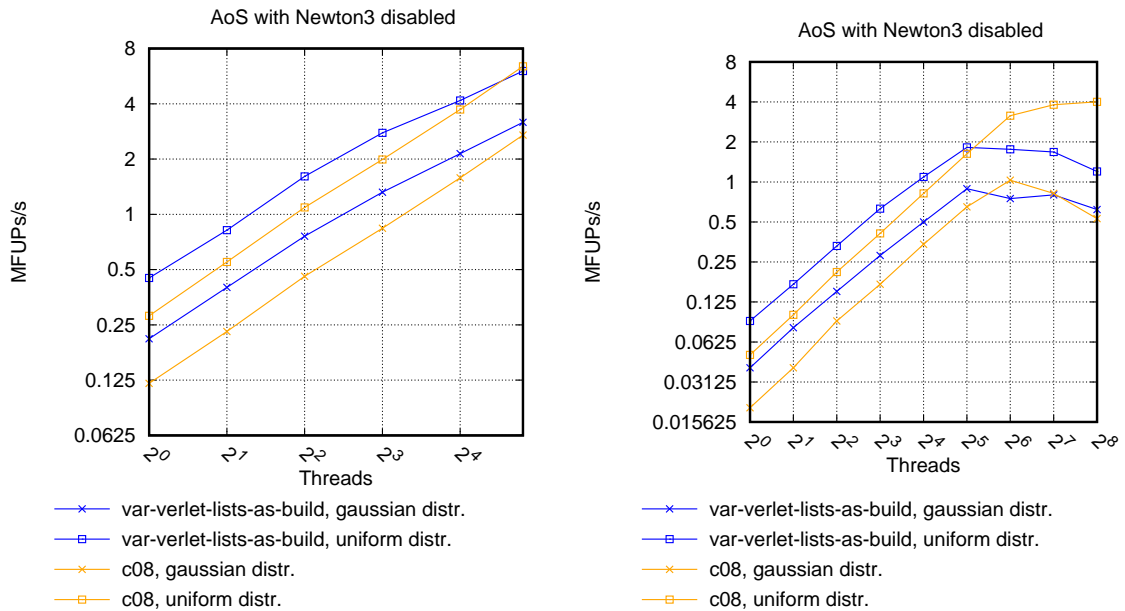
(d) CoolLMUC3: uniform distribution

Figure 6.19.: Comparison of *Strong Scaling* between all data layout and Newton 3 options of *verlet-clusters-coloring*.

In the *verlet-clusters-coloring* traversal, all data layout and Newton 3 options scale very similarly, with the SoA data layout having a slight advantage when hyper-threading is used in CoolLMUC3. This is demonstrated in Figure 6.19. Between both particle distributions,

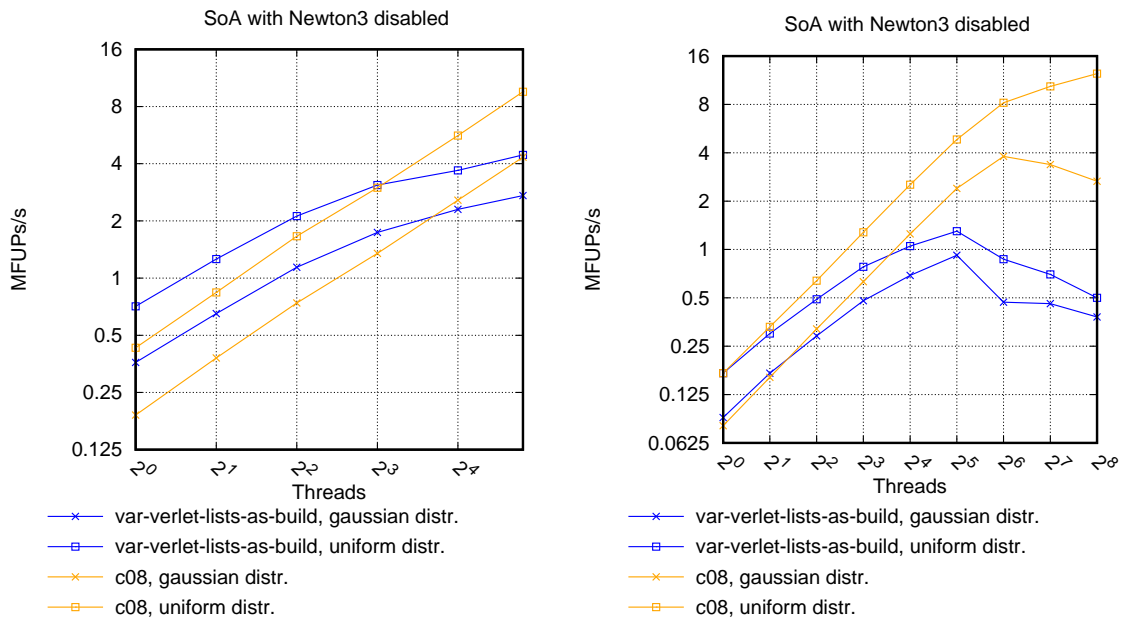
there is a big difference. In the uniform distribution, the traversal scales linearly up until 32 threads and almost until 64 threads, as it can be seen in Figure 6.19c and Figure 6.19d. Hyper-threading then decreases this performance. In the gaussian particle distribution, the traversal not even scales linearly until 16 threads on the CooLMUC2, as pictured in Figure 6.19a. On the CooLMUC2, it almost reaches its performance peak with 16 out of 256 threads and starts to lose performance after 32 threads. This makes this traversal not viable at all in this situation. The reason for this is most probably the big chunk size of this distribution, similar to the problem in Subsection 6.4.3. Comparing the traversal with the *c08* traversal, only the AoS data layout with uniform distribution on CooLMUC2 can somewhat keep up with the scaling, as it can be seen in Figure 6.17a, Figure 6.17c, Figure 6.18a, and Figure 6.18c. The configuration not using Newton's third law is slightly closer to the *c08* traversal. All other configurations are strictly worse than for the *c08* traversal, as it can be seen in Figure 6.17 and Figure 6.18.

6.5.4. var-verlet-lists-as-build Traversal



(a) CoolMUC2: AoS with Newton3 disabled

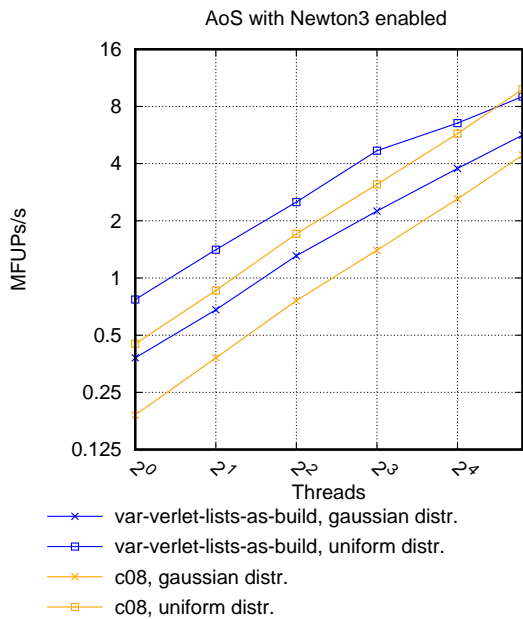
(b) CoolMUC3: AoS with Newton3 disabled



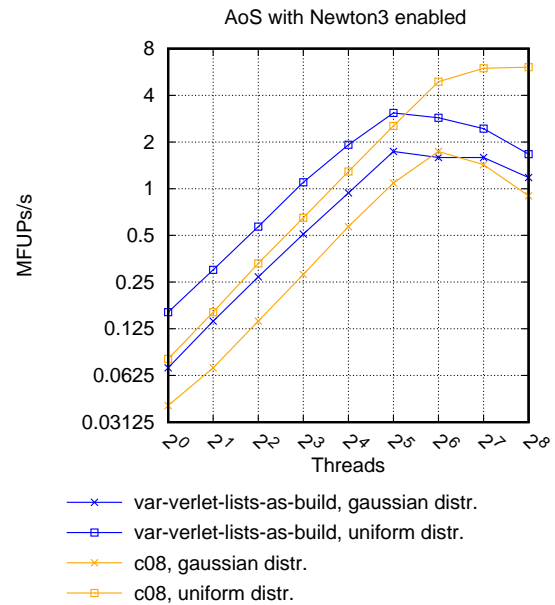
(c) CoolMUC2: SoA with Newton3 disabled

(d) CoolMUC3: SoA with Newton3 disabled

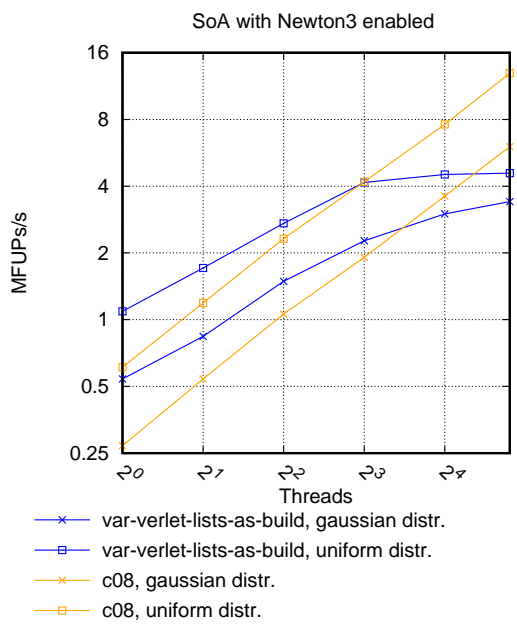
Figure 6.20.: Comparison of *Strong Scaling* between *var-verlet-lists-as-build* and *c08* without Newton 3.



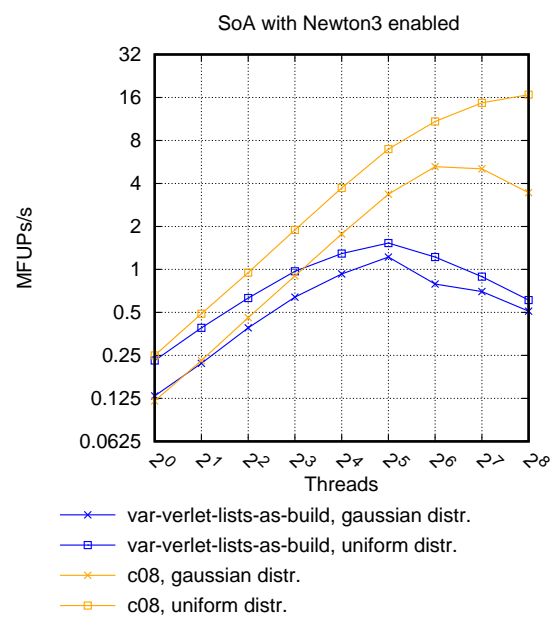
(a) CooLMUC2: AoS with Newton3 enabled



(b) CooLMUC3: AoS with Newton3 enabled

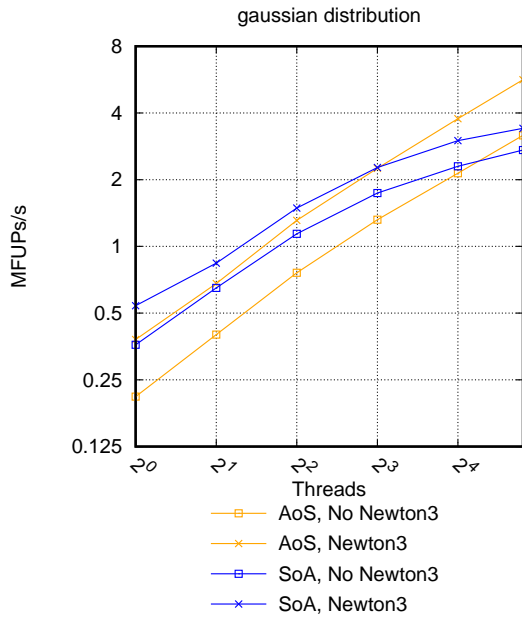


(c) CooLMUC2: SoA with Newton3 enabled

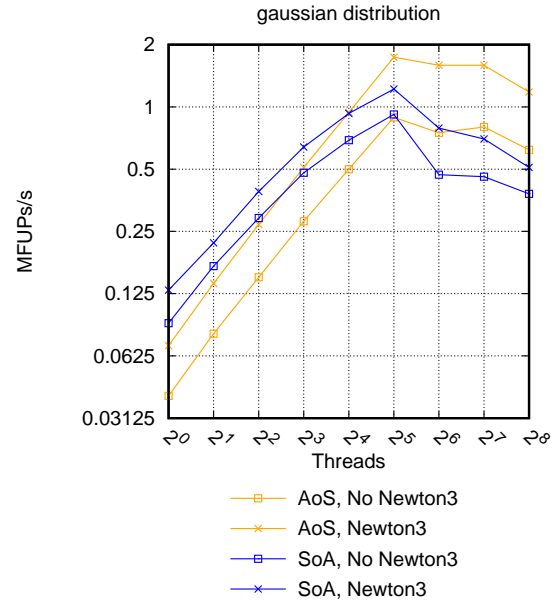


(d) CooLMUC3: SoA with Newton3 enabled

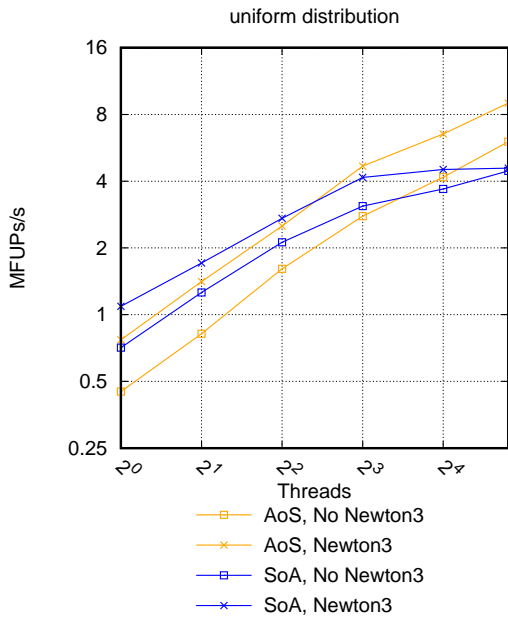
Figure 6.21.: Comparison of *Strong Scaling* between *var-verlet-lists-as-build* and *c08* with Newton 3.



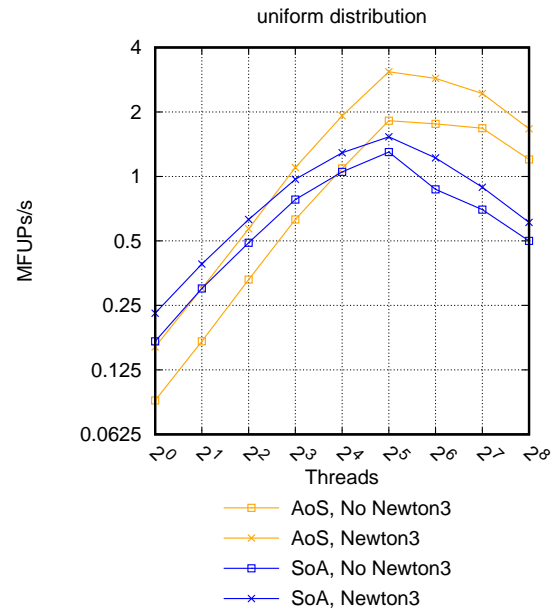
(a) CooLMUC2: gaussian distribution



(b) CooLMUC3: gaussian distribution



(c) CooLMUC2: uniform distribution



(d) CooLMUC3: uniform distribution

Figure 6.22.: Comparison of *Strong Scaling* between all data layout and Newton 3 options of *var-verlet-lists-as-build*.

In the *var-verlet-lists-as-build* traversal, the AoS data layout scales linearly on CooLMUC2 and until 32 threads on CooLMUC3, as it can be seen in Figure 6.20a, Figure 6.21a, Figure 6.20b, and Figure 6.21b. The SoA data layout behaves similar, but with a noticeably

lower gradient, as the diagrams in Figure 6.22 show. Its scaling is way better than its performance in the *Time To Solution* measurements in Subsection 6.4.5. Comparing Figure 6.22a with Figure 6.22c, and Figure 6.22b with Figure 6.22d shows that the load-balancing in this traversal seems to work properly, since the scaling with uniform and gaussian particle distributions looks very similar for both machines. However, using more than 32 threads on CoolMUC3 always seems to impair the performance, more drastically in the SoA layout, as Figure 6.22b and Figure 6.22d show. This behavior is very unexpected as this traversal is based on the *c08* traversal which scales linearly until 64 threads and is able to even benefit from hyper-threading with a uniform particle distribution. Figure 6.20b, Figure 6.20d, Figure 6.21b, and Figure 6.21d set both traversals in contrast. It requires further research to find the cause of this behavior.

Part III.

Conclusion

7. Summary

In this thesis, several new approaches for calculating the pairwise forces between particles were added to the C++ template library *AutoPas*.

For this, at first, some preparatory refactoring was done to the existing code. This included cleaning up old code through the removal of legacy template parameters and parameters from a central method in the container hierarchy. Furthermore, the traversal instantiation was centralized and the library was prepared for new implementations that use the SoA data layout. This was achieved by adding `SoAViews` and changing the neighbor list type given to one of the `SoAFunctor` methods in the `Functor` class. Furthermore, some unit tests have been generalized so that they are instantiated for all containers automatically.

Then, the new containers and traversals were developed, based on the Verlet Lists algorithm type. The `VerletClusterLists` container was rewritten to support the efficient use of the SoA data layout, be maintainable, and extensible. Three traversals were developed for this container. The first traversal, the *verlet-clusters* traversal, was the previous implementation's `iteratePairwise()` method. It distributes the towers of the container to the threads using dynamic scheduling, but could not use Newton's third law. The second traversal, called *verlet-clusters-static*, aimed to get rid of the scheduling overhead of the first traversal by calculating a static cluster-thread-distribution when rebuilding the neighbor lists. The third traversal *verlet-clusters-coloring* was developed to use Newton's third law. It uses a 6-way coloring of the towers to prevent data races of threads.

Additionally, a new container called `VarVerletListsAsBuild` was implemented. Its idea is to work on top of the `LinkedCells` of the library. The underlying container contains the particles and is used to build the neighbor lists. In order to allow developers to quickly add new Verlet Lists containers that work similar in the future a template for this called `VarVerletLists` was developed. It detaches the implementation of new neighbor list types from the container hierarchy and encourages code reuse. For the `VarVerletListsAsBuild` container, the *c08* traversal was used to build the neighbor list. This neighbor list remembers which thread added which pair of particles during which color. The traversal for this container, called *var-verlet-lists-as-build* uses this information to assign each thread the same pairs at the same time during the force calculation. This allows using Newton's third law for Verlet Lists.

Summed up, 12 new configurations were added, since all traversals support both the AoS and the SoA data layout, and the *verlet-clusters-coloring* and *var-verlet-lists-as-build* traversals support using Newton's third law. Finally, the performance of all these configurations was measured in different simulation settings and on different hardware, and their strengths and weaknesses were analyzed. These measurements showed that each traversal outperformed the previously available *c08* traversal in at least one environment, sometimes even many times over. Overall, efficiently using the SoA data layout for Verlet Lists and achieving good strong scaling with while utilizing Newton's third law remains a challenge. However, especially the `VerletClusterLists` container with the *verlet-clusters* and the

7. Summary

verlet-clusters-static traversal seems to be well suited for this problem. In the AoS data layout, the `VarVerletListsAsBuild` container with its traversal exploits all advantages the *c08* offers. On some machines, it is competitive even for a large number of particles, but it seems to outperform the *c08* traversal with a low to medium number of particles under all conditions. For these numbers of particles, the Verlet Lists algorithms are the better choice in *AutoPas*, compared to the Linked Cells algorithms.

8. Future Work

There are still many improvements that can be made for the containers and traversals developed in this thesis. The neighbor list building in the `VerletClusterLists` container takes much time. Optimizing it, for example through using Newton's third law, could potentially yield better performance. Finding a heuristic for the optimum number of threads building it instead of just taking all, might help, too. The tower side length for towers in the same container is probably also not optimal, as shown in Subsection 6.4.4. Developing other heuristics for that could also give better results. Synchronizing the tower side length with the cutoff might also bring benefits, especially to the *verlet-clusters-coloring* traversal, since its color cells depend on it. Furthermore, the static cluster-thread-distribution of the *verlet-clusters-static* traversal is not optimal, as the last thread gets considerably less work than the others. As a new traversal, we can imagine a region growing for the `VerletClusterLists` container, that works similar to the *verlet-clusters-static* traversal, but each thread gets a coherent region in the domain and is able to use Newton's third law for clusters interactions inside. This might also increase the cache efficiency compared to the coloring approach.

The `VarVerletListsAsBuild` container could be extended to support all traversals of the `LinkedCells` container, especially the other colorings, as described in Subsection 5.2.3. This would generate lots of new configurations and the benefits of each `LinkedCells` traversal would be used for a Verlet Lists traversal. Before implementing this, efficient techniques to reduce the search space for the auto-tuning have to be found, since this would enlarge it considerably.

List of Figures

2.1.	Lennard-Jones Potential with $\epsilon = \sigma = 1$. Source: [GKZ10]	3
2.2.	Visualization of the Linked Cells algorithm.	6
2.3.	C08 Traversal: Base Step Interactions	6
2.4.	Visualization of the VerletLists algorithm.	8
3.1.	UML Class Diagram of the interface of <i>AutoPas</i> and User Code that utilizes the library for own simulations.	12
3.2.	UML Class Diagram of the interfaces for all containers and traversals.	13
4.1.	UML Class Diagram of the new <code>TraversalInterface</code>	18
4.2.	UML Class Diagram of <code>SoAView</code> and <code>SoA</code>	19
5.1.	Clustering of particles in 2D	22
5.2.	UML Class Diagram of the <code>VerletClusterLists</code> container.	23
5.3.	Small Example of the Static Distribution of Clusters to Threads	26
5.4.	Color Cells covering Towers	27
5.5.	6-way Coloring of the Color Cells	27
5.6.	Base Step of the Coloring	27
5.8.	<code>VarVerletListsAsBuild</code> neighbor lists	28
5.7.	Order of towers for neighbor lists with Newton's third law	28
5.9.	<code>VarVerletLists</code> template	29
5.10.	<code>VarVerletLists</code> : Adding a new neighbor list	30
6.1.	Best and Worst Case Performance Change through the <code>VerletClusterLists</code> Refactoring.	34
6.2.	Comparison of <i>Time To Solution</i> between <i>verlet-clusters</i> and <i>c08</i>	35
6.3.	Comparison of <i>Time To Solution</i> between all data layout and Newton 3 options of <i>verlet-clusters</i>	36
6.4.	Comparison of <i>Time To Solution</i> between <i>verlet-clusters-static</i> and <i>c08</i>	38
6.5.	Comparison of <i>Time To Solution</i> between all data layout and Newton 3 options of <i>verlet-clusters-static</i>	39
6.6.	Comparison of <i>Time To Solution</i> between <i>verlet-clusters-coloring</i> and <i>c08</i> without Newton 3.	41
6.7.	Comparison of <i>Time To Solution</i> between <i>verlet-clusters-coloring</i> and <i>c08</i> with Newton 3.	42
6.8.	Comparison of <i>Time To Solution</i> between all data layout and Newton 3 options of <i>verlet-clusters-coloring</i>	43
6.9.	Tower Side Length Depending on the Number of Particles in the Measurements.	45

6.10. Comparison of <i>Time To Solution</i> between <i>var-verlet-lists-as-build</i> and <i>c08</i> without Newton 3.	46
6.11. Comparison of <i>Time To Solution</i> between <i>var-verlet-lists-as-build</i> and <i>c08</i> with Newton 3.	47
6.12. Comparison of <i>Time To Solution</i> between all data layout and Newton 3 options of <i>var-verlet-lists-as-build</i>	48
6.13. Comparison of <i>Strong Scaling</i> between <i>verlet-clusters</i> and <i>c08</i>	50
6.14. Comparison of <i>Strong Scaling</i> between all data layout and Newton 3 options of <i>verlet-clusters</i>	51
6.15. Comparison of <i>Strong Scaling</i> between <i>verlet-clusters-static</i> and <i>c08</i>	53
6.16. Comparison of <i>Strong Scaling</i> between all data layout and Newton 3 options of <i>verlet-clusters-static</i>	54
6.17. Comparison of <i>Strong Scaling</i> between <i>verlet-clusters-coloring</i> and <i>c08</i> without Newton 3.	56
6.18. Comparison of <i>Strong Scaling</i> between <i>verlet-clusters-coloring</i> and <i>c08</i> with Newton 3.	57
6.19. Comparison of <i>Strong Scaling</i> between all data layout and Newton 3 options of <i>verlet-clusters-coloring</i>	58
6.20. Comparison of <i>Strong Scaling</i> between <i>var-verlet-lists-as-build</i> and <i>c08</i> without Newton 3.	60
6.21. Comparison of <i>Strong Scaling</i> between <i>var-verlet-lists-as-build</i> and <i>c08</i> with Newton 3.	61
6.22. Comparison of <i>Strong Scaling</i> between all data layout and Newton 3 options of <i>var-verlet-lists-as-build</i>	62

List of Tables

- 6.1. Combinations of Particles and Iterations for *Time To Solution* Measurements. 32
- 6.2. Number of Threads used for each Machine in the *Strong Scaling* Measurements. 32

Bibliography

- [AMS⁺15] Mark James Abraham, Teemu Murtola, Roland Schulz, Szilárd Páll, Jeremy C. Smith, Berk Hess, and Erik Lindahl. Gromacs: High performance molecular simulations through multi-level parallelism from laptops to supercomputers. *SoftwareX*, 1-2:19 – 25, 2015.
- [AT87] M. P. Allen and D. J. Tildesley. *Computer Simulation of Liquids*. Clarendon Press, New York, NY, USA, 1987.
- [GKZ10] Michael Griebel, Stephan Knapek, and Gerhard Zumbusch. *Numerical Simulation in Molecular Dynamics: Numerics, Algorithms, Parallelization, Applications*. Springer Publishing Company, Incorporated, 1st edition, 2010.
- [GST⁺19] Fabio Alexander Gratl, Steffen Seckler, Nikola Tchipev, Hans-Joachim Bungartz, and Philipp Neumann. Autopas: Auto-tuning for particle simulations. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Rio de Janeiro, May 2019. IEEE.
- [IAAA⁺16] Ihsanullah, Aamir Abbas, Adnan M. Al-Amer, Tahar Laoui, Mohammed J. Al-Marri, Mustafa S. Nasser, Majeda Khraisheh, and Muataz Ali Atieh. Heavy metal removal from aqueous solution by advanced carbon nanotubes: Critical review of adsorption applications. *Separation and Purification Technology*, 157:141 – 161, 2016.
- [MCPR18] Ajay Muralidharan, Mangesh Chaudhari, Lawrence Pratt, and Susan Rempe. Molecular dynamics of lithium ion transport in a model solid electrolyte interphase. *Scientific Reports*, 8, 01 2018.
- [New87] I. Newton. *Philosophiae naturalis principia mathematica*. J. Societatis Regiae ac Typis J. Streater, 1687.
- [Ngu18] Jan Nguyen. Shared-memory parallelization of verlet lists. Bachelor’s thesis, Technical University of Munich, Dec 2018.
- [PC17] Sanwardhini Pantawane and Niharendu Choudhury. Molecular dynamics simulation study of carbon nanotubes in coarse-grained water: Effect of cnt diameter. *AIP Conference Proceedings*, 1832(1):040003, 2017.
- [PGL⁺18] Ugo Perricone, Maria Rita Gulotta, Jessica Lombino, Barbara Parrino, Stella Cascioferro, Patrizia Diana, Girolamo Cirrincione, and Alessandro Padova. An overview of recent molecular dynamics applications as medicinal chemistry tools for the undruggable site challenge. *Med. Chem. Commun.*, 9:920–936, 2018.

- [Ver67] Loup Verlet. Computer "experiments" on classical fluids. i. thermodynamical properties of lennard-jones molecules. *Phys. Rev.*, 159:98–103, Jul 1967.