DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Shared-Memory Parallelization of Verlet Lists

Nguyen Jan

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Shared-Memory Parallelization of Verlet Lists

Shared-Memory Parallelisierung von Verlet Listen

Author:	Nguyen Jan
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisor:	Steffen Seckler, M.Sc. (hons)
Submission Date:	Dec 17, 2018

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

Munich, Dec 17, 2018 Nguyen Jan

Abstract

This thesis deals with different algorithms for molecular dynamics simulations. This paper focuses on variants of the Verlet lists. The presented algorithms are implemented in the C++ library AutoPas. For acceleration, the library uses shared-memory parallelization. We ran performance tests of the various implemented methods on the CoolMUC-2 system of the Leibniz Supercomputing Centre. From which we learned that performance, unfortunately, does not grow linearly with the number of threads used. How well we can scale depends on the algorithm used.

Contents

Abstract iii

1 Introduction 1

2 n-Body problem 2

3 Data structures 5
3.1 General 5
3.2 Direct Sum 5
3.3 Linked Cells 5
3.4 Verlet List 6
3.5 Verlet Cluster List 7

4 Computer Architecture 10
4.1 General 10
4.2 Cache 10
4.3 SIMD 11

5 Parallelization 13
5.1 General 13
5.2 Traversals 13
5.2.1 C01 13
5.2.2 C18 13
5.2.3 C08 14
5.2.4 Sliced 14
5.2.5 Theoretical maximum number of threads. 16

6 Implementation 17
6.1 Linked Cells 17
6.2 Verlet Lists 17
6.3 Verlet Cluster Lists 18
6.4 OpenMP 20
6.5 Template 23

7 Performance comparison **24**
7.1 Performance measurments 24
7.2 Overhead 24
7.3 Potential neighbours 27
7.4 Cache 28

8 Conclusion **30**

9 Future works **31**
9.1 AoS and SoA 31
9.2 Verlet Cluster 32
9.3 Auto tuner 32

Bibliography **33**

1 Introduction

The main reason for this work is to solve the n-body problem. We want to calculate the interactions between any number of objects. This is a common problem in molecular dynamics (MD). The goal is to run simulations on a molecular level. These allow us to analyze physical, biological and chemical processes. The complexity of a simulation usually grows quadratically with the number of particles simulated. For this reason, we use numerical methods to calculate large simulations. This paper shows some solutions implemented in AutoPas.

AutoPas is a C++ library developed at the Technical University of Munich (TUM) by the Chair of Scientific Computing in Computer Science (SCCS). It is versatile in its use. The library allows the use of own particle classes and self-defined interactions between particles. This makes it applicable in various scientific fields. The user can also specify which parallelization method to use to get an optimal result for his specific simulation. An automated selection is still in development. A modified version of the algorithm used by the GROMACS molecular simulation package is also available [PH13]. GROMACS distributes work to the GPU or to parallel computers using MPI. AutoPas, on the other hand, is currently specialized in shared-memory parallelization.

This paper begins with an introduction to the n-body problem. It follows the theoretical construction of the data structures used to solve the problem. Chapter 4 is an introduction to computer architecture, so it is clear why we make certain implementation decisions. In the next chapter, we show some possibilities for parallelization. The subsequent chapter deals with the detailed implementation in AutoPas, followed by some performance measurements.

2 n-Body problem

The interactions are often dependent on the distance from each other, such as the Coulomb or gravitational forces.

$$F_C = \frac{1}{4\pi\epsilon_0}\frac{q_1 q_2}{r^2} \tag{2.1}$$

$$F_G = G\frac{m_1 m_2}{r^2} \tag{2.2}$$

For MD the most common force estimation between particles is based on the Lennard-Jones potential shown in Equation 2.3 [All04]. The derivative of this is the resulting force Equation 2.4. A possible plot is shown in Figure 2.1. This force is an approximate combination of different repelling and attracting forces two particles enact on each other at range r. At short range, overlapping electron hulls lead to a repulsive force. At long range the van der Waals forces attract them. These forces are dependent on the types of pair considered, which results in different constants ϵ and σ in our formula. A negative force signifies that they attract each other whereas a positive value means that the total force pushes them away.

$$U_{LJ} = 4\epsilon\left((\frac{\sigma}{r})^{12} - (\frac{\sigma}{r})^{6}\right) \tag{2.3}$$

$$F_{LJ} = \frac{24\epsilon}{r}\left(2(\frac{\sigma}{r})^{12} - (\frac{\sigma}{r})^{6}\right) \tag{2.4}$$

We can determine the resulting acceleration, future speed, and position by solving the following equations.

$$a = \frac{F}{m} \tag{2.5}$$

$$v = v_0 + \int a\,dt \tag{2.6}$$

$$x = x_0 + \int v\,dt \tag{2.7}$$

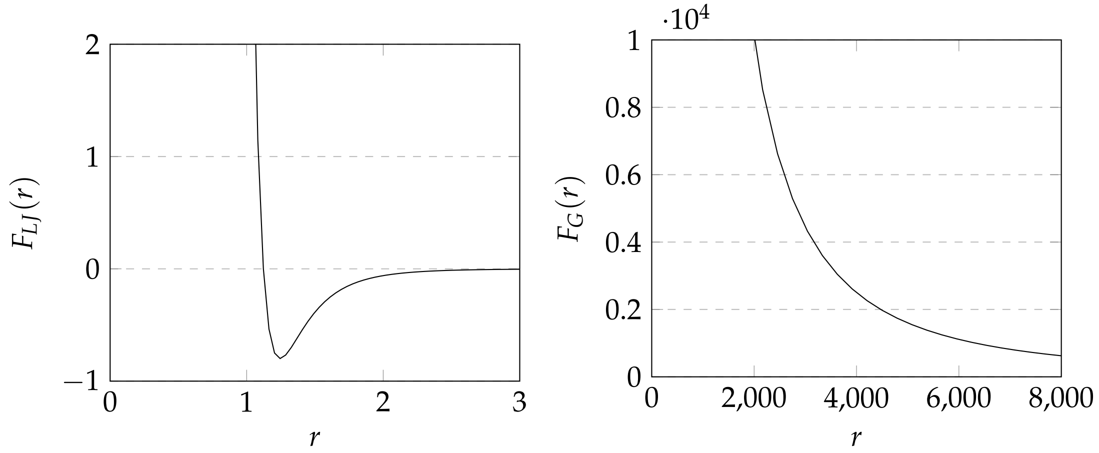

Figure 2.1: Example of Lennard Jones Potential (left) and gravitational force (right)

The problem is that the acting forces change as the particle move. This again has an influence on the movement. Solving this differential equation analytically is near impossible as the number of objects increases. That's why we're taking a numerical approach. One possible integration method is Störmer-Verlet, which assumes that within a sufficiently small time step Δt the velocity changes only linearly in time [HLW03]. This can be shown by the Taylor expansion of our equations.

$$v(t + \Delta t) = v(t) + \Delta t \frac{d}{dt} v(t) + O(\Delta t^2) \tag{2.8}$$

Ignoring the last summand leads to the explicit Euler method. We can also use the Taylors theorem backward leading to the implicit Euler method.

$$v(t) = v((t + \Delta t) - \Delta t) = v(t + \Delta t) - \Delta t \frac{d}{dt} v(t + \Delta t) + O(\Delta t^2) \tag{2.9}$$

Combining both leads to the following commonly used equation.

$$v(t + \Delta t) = v(t) + \frac{\Delta t}{2} (\frac{d}{dt} v(t) + \frac{d}{dt} v(t + \Delta t)) + O(\Delta t^2) \tag{2.10}$$

$$v(t + \Delta t) \approx v(t) + \frac{\Delta t}{2} (a(t) + a(t + \Delta t)) \tag{2.11}$$

To calculate the position we can even use an additional Taylor polynomial.

$$x(t + \Delta t) = x(t) + \Delta t \frac{d}{dt} x(t) + \frac{\Delta t^2}{2} \frac{d^2}{d^2 t} x(t) + O(\Delta t^3) \tag{2.12}$$

$$x(t+\Delta t) \approx x(t) + \Delta t v(t) + \frac{\Delta t^2}{2} a(t) \tag{2.13}$$

But all in all, we can only calculate the position to the chosen timestep. So we repeat this trick with the new positions until the timesteps sum up to our desired value. The calculation in each step is not difficult, but it can become very extensive. With n particles the number of pairs is $n(n-1)$. Which we all have to consider in order to calculate the acting force. Multiplied by the number of time steps t_n, we can evaluate the scale.

$$\#\text{Calculations} = n(n-1)t_n \tag{2.14}$$

One way to reduce the calculations needed is the exploitation of symmetry. Newton thirds law states that if one body applies force to a second, the second will also apply a force to the first one. Both this forces are equal in strength. But they act in opposite directions, such that the sum of them is 0. This allows us to cut the number of pairs in half.

$$\#\text{Calculations}_{Newton3} = \frac{1}{2} n(n-1)t_n = O(n^2 t_n) \tag{2.15}$$

The calculation still grows quadratically with the particles. That's why we're making another estimation. We see that our force functions go towards zero with increasing distance. We, therefore, define a cutoff radius r_c and assume that particles further away than this radius have a negligible influence on each other. Ignoring these pairs lead to a significant boost in performance.

3 Data structures

3.1 General

There are different possibilities to make use of the cutoff radius. The most common data structures are Linked Cells and Verlet lists. This paper deals mainly with the second. We also introduce the implementation of GROMACS.

3.2 Direct Sum

The first thought would be to test all particle pairs to see if they lie within the cutoff and ignore those who aren't. This leads us to the algorithm 1. We can skip some force calculations but we still have to calculate distances for a quadratic amount of pairs.

Algorithm 1 Direct sum iteration with Netwon 3

1: **for** i from 0 to $n_{Particles}$ **do**
2:     **for** j from $i + 1$ to $n_{Particles}$ **do**
3:         $distSquare \leftarrow$ squared distance between particle i and particle j
4:         **if** $distSquare < cutoffSkinSquare$ **then**
5:             $f \leftarrow$ force particle i enacts on particle j
6:             Apply force $-f$ to particle i
7:             Apply force f to particle j

3.3 Linked Cells

The Linked Cells data structure offers a way to also get rid of some distance calculations [TA87]. Figure 3.1 visualizes the main idea. We define a domain which should contain all particles. We use a grid to subdivide this domain into cells and group all particles according to the cell in which they are contained. When searching for neighbors, we can skip all cells that are completely outside the cutoff. It is advantageous to choose the side length of the cells to be at least as long as the cutoff radius. This way only adjacent cells are close enough. Meaning in a 3D space we only have to test all

particles in the same cell and in the 26 adjacent cells. The number of particles we have to test is solely dependent on the chosen cell size and the density of the particles ρ. Ideally, the side length of the cells is equal to the cutoff. This would lead to a runtime of $O(n_{Particle}) + O(n_{Particle} \cdot \rho \cdot (r_c)^3)$ per iteration. The first summand results from the distribution of the particles on the cells. So if we have an extremely low-density situation, we waste most of our time building our structure. We may define a rebuilding period $T_{rebuild}$ to distribute the cost over multiple iterations. Over this period, particles may jump from one cell to another. This could lead to a new pair that has been ignored so far. We will ignore this pair until the rebuild, which of course falsifies the simulation.

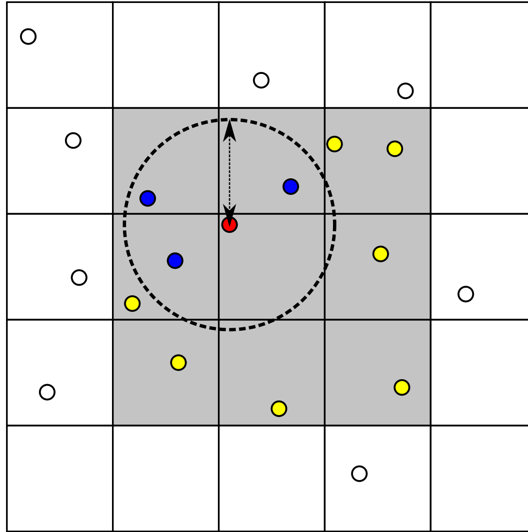

Figure 3.1: Linked Cells. Blue Particles are in the cutoff radius of the red one. Yellow ones will be tested but are too far away.

3.4 Verlet List

The Verlet lists structure stores for each particle a list of all its neighbors [Ver67]. The neighbors are all particle in the cutoff radius and also all that could be in $T_{rebuild}$ steps. To achieve this we have to think about the expected velocities. If we know the maximum speed v_{max} that every particle can't exceed, we also know the maximum distance they can travel over a given time. Any two particles cannot reduce their distance to each other more than r_s. Which is the case if they directly fly to each other with maximal speed.

$$r_s = 2 \cdot T_{rebuild} \cdot \Delta t \cdot v_{max} \tag{3.1}$$

We call this radius the skin. Adding the skin to our cutoff defines a ball which contains all our neighbors. Figure 3.2 shows the possible extent of using a skin. We could always calculate the ideal skin for the given situation. But for performance reasons, a fixed skin is often chosen.

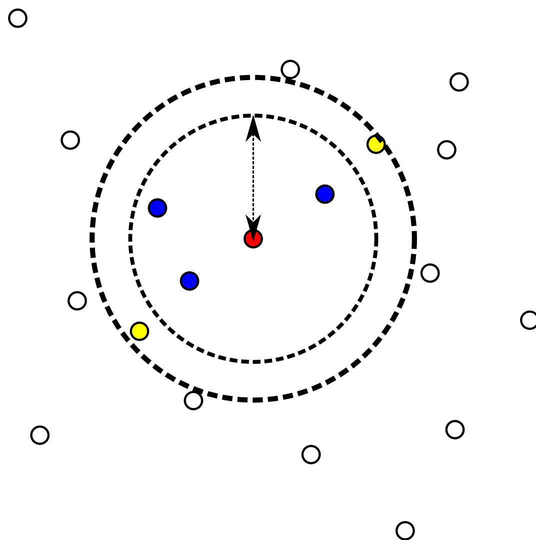

Figure 3.2: Verlet list. Blue and yellow particles are inside the neighbor list of the red one. Only blue ones are inside the cutoff radius.

Nonetheless to create the Verlet lists we have to check the distances for each pair. To avoid this quadratic runtime we use our prior solution, the Linked Cells. So in summary we have a runtime of $O(n_{Particle}) + O(n_{Particle} \cdot \rho \cdot (r_c + r_s)^3)$ for the rebuild. In all other steps, we just have to iterate through the Verlet lists. This has a runtime of $O(n_{Particle} \cdot \rho \cdot (r_c + r_s)^3)$. For extremely sparse situation this may be insignificantly low compared to the rebuild steps.

3.5 Verlet Cluster List

An invariant of the Verlet lists is used by GROMACS. They group particles into clusters of a fixed size $n_{cluster}$. Instead of saving particle pairs, we create a Verlet list of clusters. When two clusters interact with each other, we calculate the interactions between all particle in one cluster with the ones in the other. This means one cluster pair equals to $(n_{cluster})^2$ particle pairs. To generate this Verlet lists we need to define the distance between two clusters, which ensure we don't lose any particle pairs. That is why we include a cluster pair as soon as at least one object in a cluster interacts with any one

of the others. The left example in Figure 3.3 shows how one can visualize this. The red area is the union of four spheres around the four red particles in the same cluster. Meaning the implementation had to test distances to the cluster by calculating the distance to each of the particles. To reduce this overhead we can draw the axis-aligned bounding box of the cluster as shown in the right side of our Figure 3.3. We can calculate the distance of two boxes with Algorithm 2. Due to the properties of our boxes, we can obtain the distance in each dimension independently and sum them up into one distance vector. This simplification obviously can lead to additional pairs like in our example. The green bounding box is within the radius of the cluster, although no particle pair is within range of each other. But in a much denser example, many clusters would be completely in our cutoff radius. Which makes this overhead more and more negligible.

Algorithm 2 Distance of two bounding boxes

1: **function** DISTANCE($bb1min, bb1max, bb2min, bb2max$)
2: $d \leftarrow$ dimensions
3: $v \leftarrow$ zero vector of dimension d
4: **for all** i from 0 to d **do**
5: **if** $bb1max[i] < bb2min[i]$ **then**
6: $v[i] \leftarrow bb1max[i] - bb2min[i]$
7: **else if** $bb1min[i] > bb2max[i]$ **then**
8: $v[i] \leftarrow bb1min[i] - bb2max[i]$
return Length of v

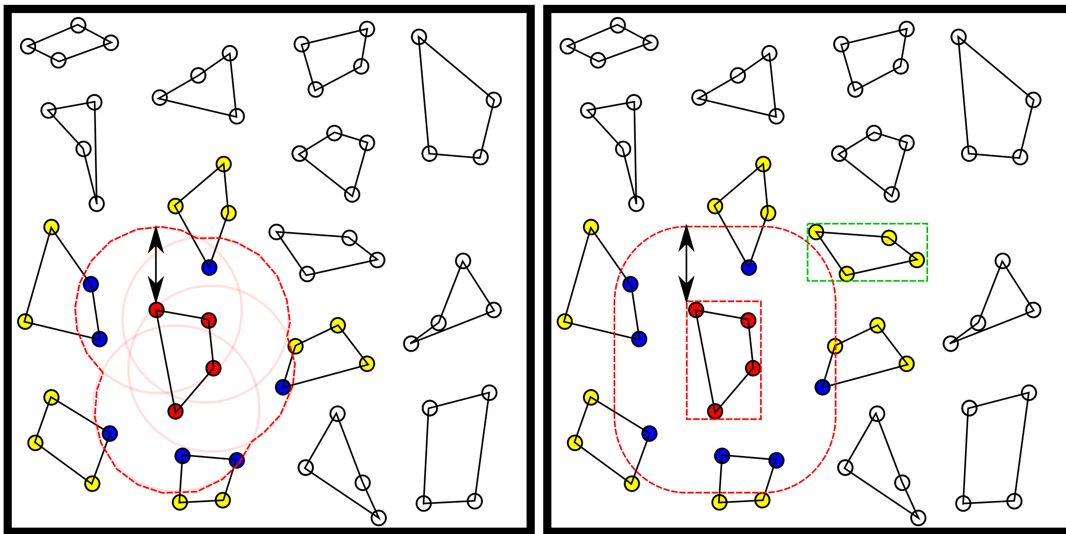

Figure 3.3: Verlet cluster list (left) and with bounding boxes (right). Blue and yellow particles are inside the neighbor list of the red ones. Only blue ones are inside the cutoff radius.

4 Computer Architecture

4.1 General

Before we can think about further implementations, we must consider what our used hardware is capable of. In this paper, we only deal with shared-memory parallelization of multicore nodes. We use one main memory which multiple CPU-cores can access.

4.2 Cache

To do so all cores must be able to access the shared memory. Unfortunately, the main memory can't handle multiple read- or write-instructions at once. So if multiple processors want to fetch data they have to do so sequentially. To avoid unnecessary data accesses, all processors have their own memory, the cache [HP06]. This is a temporary place for data, so the processor can work with them without requiring the help of the main memory. When attempting to access data, first check whether it is already in the cache. If not they must load the data from the main memory. If the data has been loaded before they do not need to access the main memory. But it can occur that a process accesses data that is already in another cache. Since these could be modified, they must be written back to the main memory before they can be read. The process would have to wait depending on the transmission speed. What makes matters worse is the fact that only blocks of fixed size (Cache lines) can be transferred. This design choice is due to spatial locality. We assume that if someone accesses data, he will probably access adjacent data in near future. This may cause data to be stored on caches that are not used at all. Different processes not only hinder each other when working on the same data but also when the used cache lines overlap. This problem is called false sharing.

Nevertheless, caches are usually advantageous. Therefore another cache level is often used as seen in Figure 4.1. This cache is also shared between the cores but is smaller than the main memory. This way false sharing is less impactful because the transmission between caches is way more efficient.

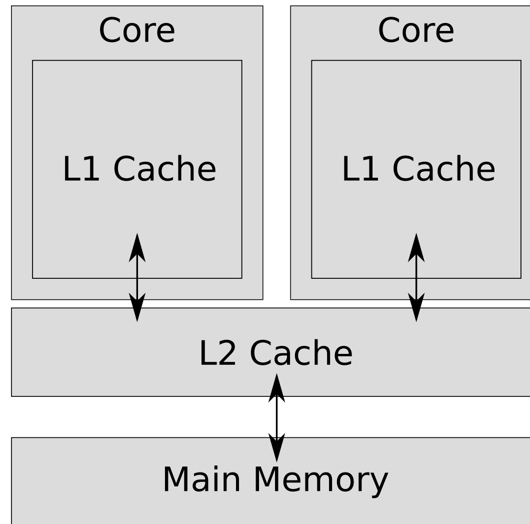

Figure 4.1: Cache with two layers.

4.3 SIMD

Another way to reduce the instructions needed is vectorization. Normally simple operations like addition need multiple instructions. Two for loading both summands into the registers of the CPU. One for the addition and one to store the solution back into the memory. It's often the case that we have to repeat the same instruction for different values. For example, if we want to add two vectors of any dimension, we have to repeatedly add up each component. We call this single instruction, multiple data (SIMD) [PH13]. Modern CPUs are optimized for this problem. They can load and store multiple values in one instruction. The same even applies to operations as seen in Figure 4.2. To load multiple data efficiently the data has to be aligned and consecutive in the memory. Otherwise, we have to use multiple loads and store instructions as seen in Figure 4.3. That's why the data layout of given data should be considered as this can lead to a significant difference in performance.

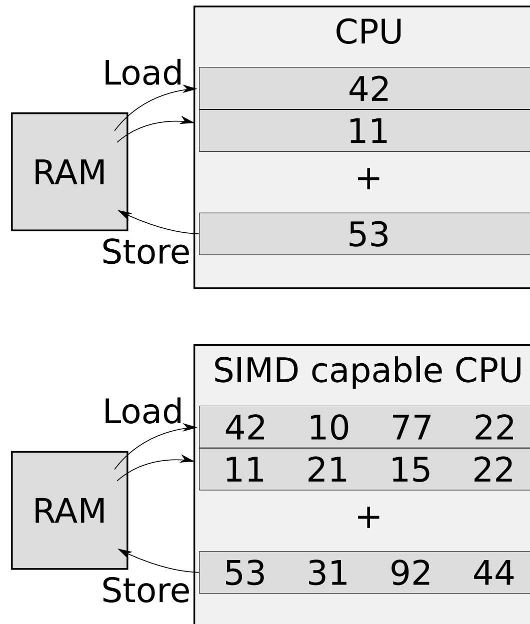

Figure 4.2: Advantage of SIMD capable CPUs: multiple operations can be performed with the same amouf of instructions compared to non-simd-capable CPUs

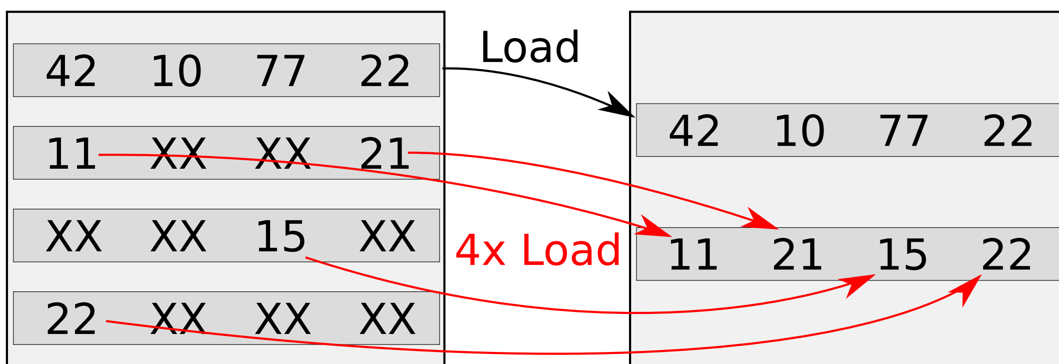

Figure 4.3: Advantage of consecutive data: Only one load instruction is needed. Unaligned values have to loaded individually

5 Parallelization

5.1 General

As simulations get more extensive, it is strongly advisable to distribute the calculations across several threads. However, as with any implementation of parallelism, it is vital to be aware of data dependency. In order to calculate the force between a pair of particles, various properties of both particles, such as position or mass, must be read. But solely the variables representing the cumulative forces on each particle require a read and write access. Hence, race conditions can only occur if multiple threads write to the same force variable of a particle. To avoid this two points have to be considered. How to build the Verlet lists of each particle and how to distribute the lists amongst the threads. To do this we use some methods of linked cells and adapt them to Verlet lists. In order to avoid collisions, we use traversals. These define how interactions between cells can be parallelized. They group the calculations together such that one group cannot cause collisions with the others. This allows us to distribute the groups amongst threads. We may not be able to divide up all the work and have to add a few synchronization steps.

5.2 Traversals

5.2.1 C01

One way to guarantee thread safety is not to utilize the third Newtonian law. For this purpose, the neighbor list of each particle must contain all particles in the cutoff radius. This means that the total force of a particle can be calculated using only its neighbor list and thus parallel to the others. As a result, all lists can be distributed freely among the threads. A disadvantage is that all pairs have to be stored and calculated twice.

5.2.2 C18

To get rid of the redundant pairs we need to apply Newton's third law. But we have to consider that now both particles of a pair need write access. The C18 traversal can be utilized here to avoid collisions. It uses the Linked-Cells structure to add the

appropriate particles to the neighbor lists. Let the cell ID of a particle be the index of the cell containing it. Then each list should only contain neighbors with a greater cell ID. This ensures storing each pair exactly once. Each pair is contained in the Verlet lists of the particle with a lower cell ID. To traverse over the particles we use 18 synchronization steps. In each step, we choose a distinct subset of cells and distribute them over the thread. We select the subsets in such a way that different threads may never access the same cell simultaneously. For visualization, we color the cells in the same subset in the same color. Thus, we need to ensure that no two cells of the same color access the same cell. Figure 5.1 shows the coloring in 2D. It also indicates the accesses of the yellow cells.

5.2.3 C08

To reduce the number of colors the C08 traversal is used. It reduces the synchronization steps to 8, but for that, we have to rethink a bit. When we process a cell, we don't just work on the particles inside that cell. This cell is considered as the base cell. We calculate interactions between cells that are relative to the base cell. It must be guaranteed that all necessary interactions have been calculated after all base cells have been processed. And all base cells of the same color do not access the same cells. The Figure shows the 2D solution to this problem which needs 4 colors. Unfortunately, this method cannot easily be applied to Verlet lists. The pairs between two cells must be known. This means that the Verlet lists cannot be stored per particle. Instead, you can, for example, save one Verlet list per cell pair.

5.2.4 Sliced

Another way to avoid race conditions is to use locks. We select a dimension and divide the cells in this direction. In each of these slices, a thread can now work. Collisions can only occur at the boundary of these regions. That's why we put a lock on one side. The thread may only calculate cells at the boundary when the other one has finished the critical area and releases the lock. Obviously to optimize this method the number of slices should be equal to the number of threads. To minimize the synchronization area, we select the dimension that is the longest. Figure 5.2 shows a possible run in 2D space. Each thread can start to work on the brighter areas. And if the neighbor releases the lock, the grayed areas too. That's why each thread should start at the edge to release the locks as soon as possible.

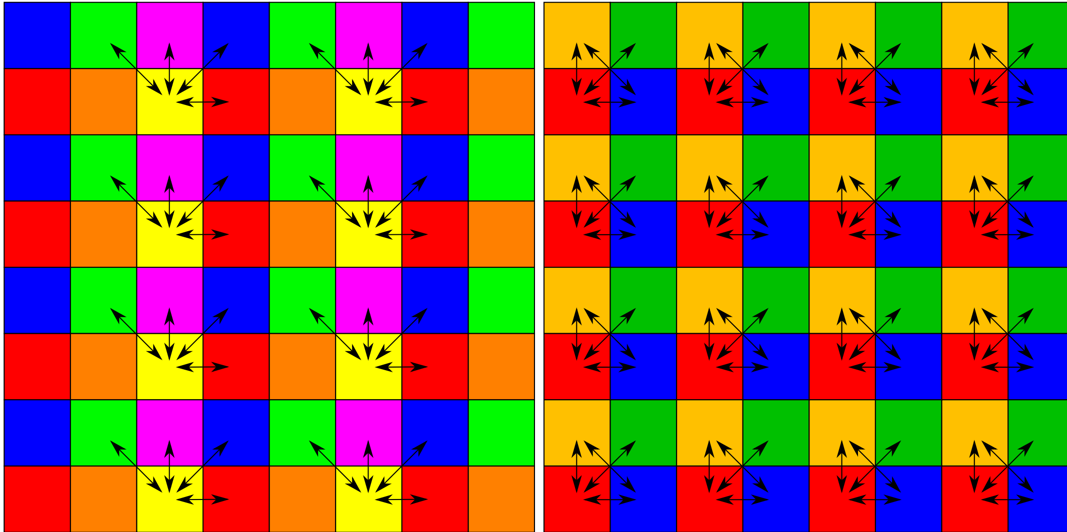

Figure 5.1: C18 Traversal (Left), C08 Traversal (right)

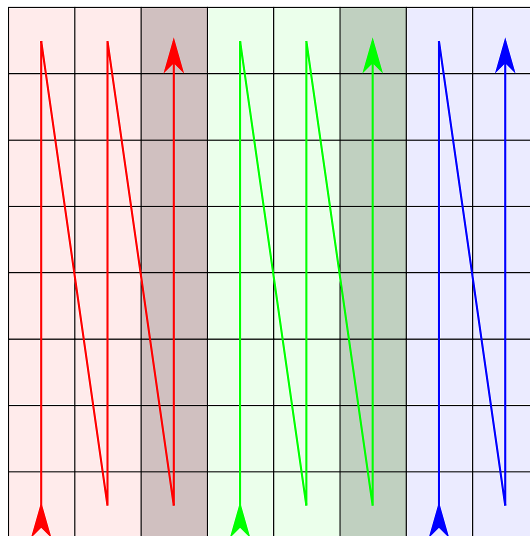

Figure 5.2: Sliced Traversal. The arrows show a possible processing order of the threads.

5.2.5 Theoretical maximum number of threads.

Let's make the utopian assumption that more threads always lead to better performance. Then it would be crucial to know how many can run at the same time. The color schemes assign approximately the same number of cells to each color. Let d be the number of cells in each dimension and c the number of colors. The maximum number of usable threads is then:

$$n_{color} = \frac{d^3}{c} \tag{5.1}$$

The slicing algorithm can only divide the work in one direction. At the same time, there must be at least one non-critical area between each two synchronization areas. Otherwise, the threads would wait for each other and not advance.

$$n_{sliced} = \frac{d}{2} \tag{5.2}$$

That is if the dimension is greater than 3, n_{sliced} is less than n_{color} for all traversals.

6 Implementation

6.1 Linked Cells

When implementing Linked-Cells the most important question is, how to store the particles. Firstly there are two options. Saving the particle in a global list or create a list for each cell. The second option indicates which particle is in which cell. This is not the case for the first. So every cell still needs a list containing references to the global list. That is why we opted for the second option. Now we have to decide which data structure to use. We cannot simply use arrays because the number of particles within a cell is unknown. Usually, one chooses array lists or linked list. The main advantage of linked lists is that when iterating through, items can be deleted in constant time. This would allow particles that have moved to another cell to be placed in the correct list. The problem is that their construction requires more memory. Each element also contains a pointer to the next one. Because we allow a vast amount of particles this can lead to a significant difference in memory usage. This also leads to a worse cache locality which we will cover later. In addition, the order of particles in the list is not important to us. This allows us to delete each element in an array list in constant time by swapping it with the last element in the list first. So we have no disadvantage using array lists.

6.2 Verlet Lists

Since Verlet lists use the same cells for traversing, we store the particles within a Linked-Cells structure. For the lists, we have also decided to create one for each cell. Due to parallelization, it is necessary that the corresponding list for each particle can be identified with ease. We want to give users of AutoPas freedom to implement their own particles. That's why we don't store the lists inside the particles. We have to store them separately and provide a projection of particles to the lists. Particles can easily be distinguished by their index within a cell. Intuitively we create an array list for the neighbor lists in such a way that the particles and their lists use the same indices. To construct the lists we use the traversals of the Linked Cells data structure. If it doesn't matter which particle contains a pair, we can use any traversal. Normally we pass a

force functor to the traversal, but we can also pass functors to construct the Verlet Lists, as shown in Algorithm 3. The second functor adds the pair to both Verlet lists. It's used for traversals which don't use Newton's 3rd law. Accordingly, the other functor only adds the pair to the neighbor list of the first particle and is used for traversals which do use Newton's law. The C18 traversal will call this function whereas the first particle will be the one with a lower cell ID. This means that the particle with lower cell ID will contain the pair in its neighbor list.

Algorithm 3 Verlet construction functors

```
1: function AddToVerlet(particle1, particle2)
2:     distSquare ← squared distance between particle1 and particle2
3:     if distSquare < cutoffSkinSquare then
4:         index ← index of particle1 in cell
5:         Add (particle1, particle2) to VerletList[index]

1: function AddToVerletBi(particle1, particle2)
2:     distSquare ← squared distance between particle1 and particle2
3:     if distSquare < cutoffSkinSquare then
4:         index1 ← index of particle1
5:         index2 ← index of particle2
6:         Add (particle1, particle2) to VerletList[index1]
7:         Add (particle2, particle1) to VerletList[index2]
```

6.3 Verlet Cluster Lists

Currently the cluster lists are implemented only with some simplification. We estimate the particle density ρ by dividing their number by the volume of our domain. This allows us to estimate the volume that contains a cluster. We try to split the domain in cubes of same size such that each cube contains one cluster. The expected side length of these is:

$$l_{Grid} = \sqrt[3]{\frac{n_{Cluster}}{\rho}} = \sqrt[3]{\frac{n_{Cluster} V_{Domain}}{n_{Particles}}} \tag{6.1}$$

We create a 2D grid in x- and y-direction with this side length to divide our domain into cells. Interestingly if we assume a cubic domain, the number of cells in one dimension d is only dependent on the number of particles and the chosen size of clusters.

$$d = \frac{l_{Domain}}{l_{Grid}} = \frac{l_{Domain}}{\sqrt[3]{\frac{n_{Cluster}(l_{Domain})^3}{n_{Particles}}}} = \sqrt[3]{\frac{n_{Particles}}{n_{Cluster}}} \tag{6.2}$$

To find out which particles are in the same cluster we create a list for each cell. This list contain all particle in the corresponding cell. We sort this lists by the z-position of the particles. Every $n_{Cluster}$ particles within a list now form a cluster. Figure 6.1 shows a visualization of this method. The x and y dimensions of the boxes are fixed. We now try to get 4 particles each into one box with the last dimension.

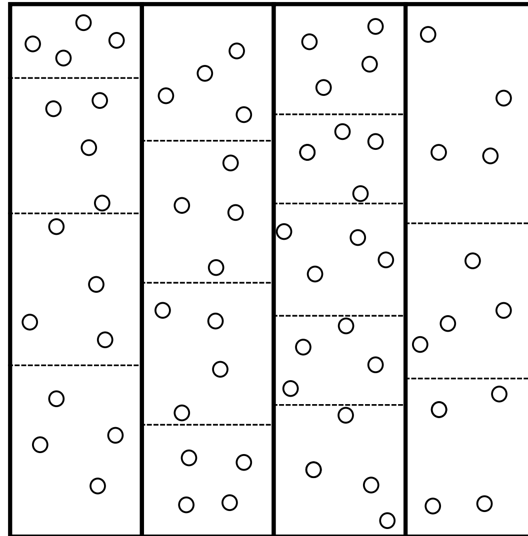

Figure 6.1: Verlet Cluster. Each 4 particle in the same box form a cluster.

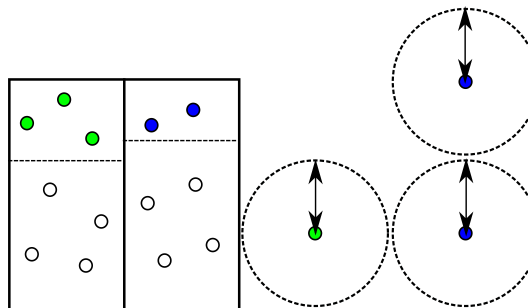

Figure 6.2: Verlet Cluster dummy particles. One green particle added to the upper left cluster. Two blue particle added to the upper right cluster.

Because not every cell will contain a multiple of the cluster size we add dummy particles until it does. We position these outside of the interaction radius of all other particles, as seen in Figure 6.2, so we don't falsify our results. We can estimate the

maximal number of dummy particles possible with our previous assumptions.

$$n_{\text{max dummy}} = (n_{cluster} - 1)d^2 \tag{6.3}$$

This leads to a proportion of:

$$\frac{n_{\text{max dummy}}}{n_{Particle}} = \frac{(n_{cluster} - 1)(\frac{n_{Particles}}{n_{Cluster}})^{\frac{2}{3}}}{n_{Particle}} = (n_{cluster} - 1)(n_{Cluster})^{-\frac{2}{3}}(n_{Particle})^{-\frac{1}{3}} \tag{6.4}$$

Obviously, more particles lead to more dummy particles. But Equation 6.4 shows that the dummies are becoming less and less relevant as the size of the simulation increases. For example, the percentage of dummies in a system with 540000 particles and a cluster size of 4 is in worst case 1.46%.

As announced, we calculate the cluster pairs slightly differently. The first function in Algorithm 4 shows the rough process of the original. With the help of the cutoff and the side length d_{Grid}, we can determine the maximum index distance $cutoffCells$. Two indices further away than this value cannot be within range. Otherwise, we'll have to calculate the exact distance between the two clusters. GROMACS has calculated a minimum bounding box for each cluster for this purpose. Instead, we calculate the distance in x- and y-direction using the cells. With the help of the cell indices, we know how many cells lie between two clusters. Multiplied by the side length, this results in the minimum distance. The second function in Algorithm 4 shows our approach. We also use the fact that the XY distance between two cells remains fixed. To do this, we iterate through the cells first, avoiding having to recalculate this value over and over again. But we still have to calculate the distance in Z-direction normally. This is however much simpler because the particles are ordered in this direction. Consequently, the first particle in the cluster indicates the lower limit and the last the upper limit. We essentially use boxes which are larger in the x- and y-dimension than bounding boxes, as seen in Figure 6.3.

6.4 OpenMP

We use the OpenMP directive for parallelization [Ope15]. Directives are used to extend the existing code. They are not conventional functions that are executed, but markers for the preprocessor. This allows us to write the program without having to focus on parallelization. After that, we can easily extend the code with OpenMP. For example, we can combine multiple loops with the collapse command. In this way, the code remains clear and structured. Comparing Listing 6.1 and 6.2 we can see the advantages.

Algorithm 4 Construct Verlet cluster lists

```
1: function REBUILD
2:     cutoffCells ← cutoff/d_Grid
3:     for all cluster (x_i, y_i, z_i) do
4:         for all x_j in range cutoffCells of x_i do
5:             for all y_j in range cutoffCells of y_i do
6:                 for all z_j in cell (x_j, y_j) do
7:                     d ← distance vector between (x_i, y_i, z_i) and (x_j, y_j, z_j)
8:                     if d_x^2 + d_y^2 + d_z^2 < cutoffSquared then
9:                         Add ((x_i, y_i, z_i), (x_j, y_j, z_j)) to list
```

```
1: function REBUILDSIMPLIFIED
2:     cutoffCells ← cutoff/d_Grid
3:     for all cell (x_i, y_i) do
4:         for all x_j in range cutoffCells of x_i do
5:             d_x ← max(0, abs(x_i − x_j) − 1) ∗ d_Grid
6:             for all y_j in range cutoffCells of y_i do
7:                 d_y ← max(0, abs(y_i − y_j) − 1) ∗ d_Grid
8:                 dSqr_xy ← d_x^2 + d_y^2
9:                 if dSqr_xy < cutoffSquared then
10:                    for all z_i in cell (x_i, y_i) do
11:                        for all z_j in cell (x_j, y_j) do
12:                            d_z ← distance in z of (x_i, y_i, z_i) and (x_j, y_j, z_j)
13:                            if dSqr_xy + d_z^2 < cutoffSquared then
14:                                Add ((x_i, y_i, z_i), (x_j, y_j, z_j)) to list
```

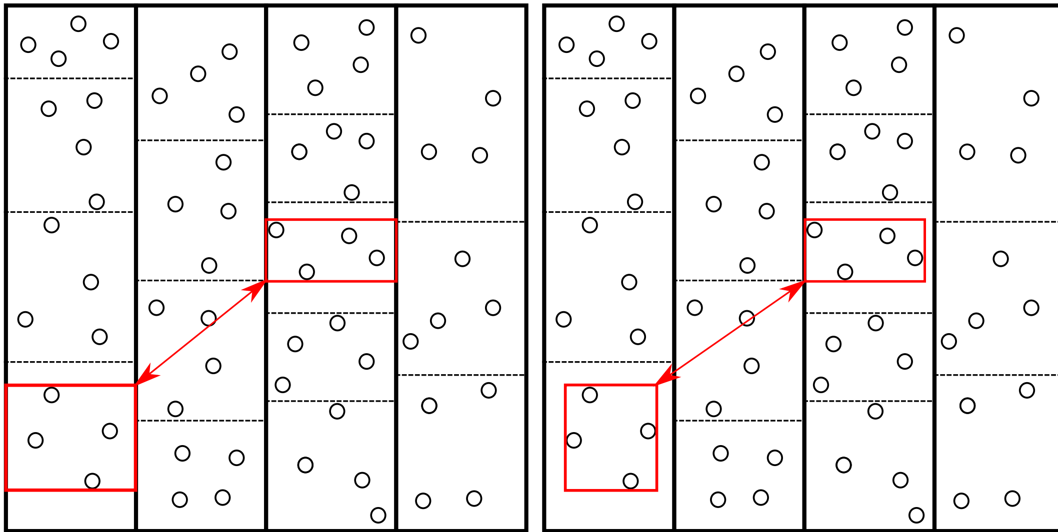

Figure 6.3: Comparison between simplified boxes (left) and bounding boxes (right). The simplified boxes lead to a smaller or equal distance.

We didn't even implement concurrency but unrolling the loops has already severely impaired readability.

```
#pragma omp parallel for collapse(3)
for (int x = 0; x < xmax; x++) {
  for (int y = 0; y < ymax; y++) {
    for (int z = 0; z < zmax; z++) {
      DoSomething(x,y,z);
    }
  }
}
```

Listing 6.1: OpenMP nested for

```
for (int i = 0; i < xmax*ymax*zmax; i++) {
  z = i / (xmax * ymax);
  y = (i - z * (xmax * ymax)) / zmax;
  x = i - xmax * (y + ymax * z);
  DoSomething(x,y,z);
}
```

Listing 6.2: Collapsed nested for

Note that directives like OpenMP solely specify a standard, so programmers don't have to study a different language for different compilers. But the exact implementation must be determined for each compiler individually. Thus, with the same code and environment but different compilers, performance differences may occur. In our case, we consistently use the GNU Compiler Collection (GCC) for benchmarking.

6.5 Template

AutoPas allows the implementation of custom particles. But instead of using dynamic polymorphism through inheritance, we use static polymorphism through templates [VJG18]. When compiling, one class is created per template argument used. This makes the compiled code larger and error messages harder to read. But the problem with inheritance is that the base class must offer virtual functions. These can only be called via an indirection. Since we often use these functions, this is a considerable performance loss. This is why we prefer static polymorphism.

7 Performance comparison

7.1 Performance measurments

The following tests were performed on the CoolMUC-2 system of the LRZ Linux Cluster. Each node consists of 28-core Haswell nodes and has 64 GB RAM available. The performance is measured in force updates per seconds (FUPS).

7.2 Overhead

Figure 7.1 compares the various calculation options depending on the number of threads used. Both scenarios use the same density of molecules. With the smaller setup, one sees that a larger number of threads has a negative impact on traversals with synchronization steps. Only a few cells can be distributed in each step, but all threads must be synchronized in the end. At a certain point, it no longer makes sense to distribute the work any further. But every additional thread has to be managed nonetheless, which only results in additional overhead. Although C01 must treat each pair twice, it resulted in a better runtime. It only has to synchronize all threads once and can distribute more work than the others. As seen in Figure 7.3 the number of cells in the other example is significantly greater. The FUPS using a single thread roughly corresponds to the first example. But with the extra work, all traversals scale better than before. This time C18 outperforms the others. The effect of overhead becomes clear when we influence scheduling. All cells are divided into groups the size of the chunk size. With a chunk size of 1, all cells are distributed to the threads individually. With a value of 2, we distribute groups consisting of 2 cells. So the number of groups is about half of the number of cells. Increasing the value thus leads to less overhead, but the work can be spread less precisely. In a sparse grid, the work per cell is low, so the chunk size mainly affects the overhead. Figure 7.2 shows the performance of Verlet lists with different chunk size. The value does not influence the algorithm, only the scheduling of the threads. Thus the resulting work of using the threads. We can see that increasing the chunk size has a great impact on the performance. Our tests confirm our assumptions. But we also see that a higher density reverses this effect. Since we can’t distribute the work as accurately as we used to, individual threads often

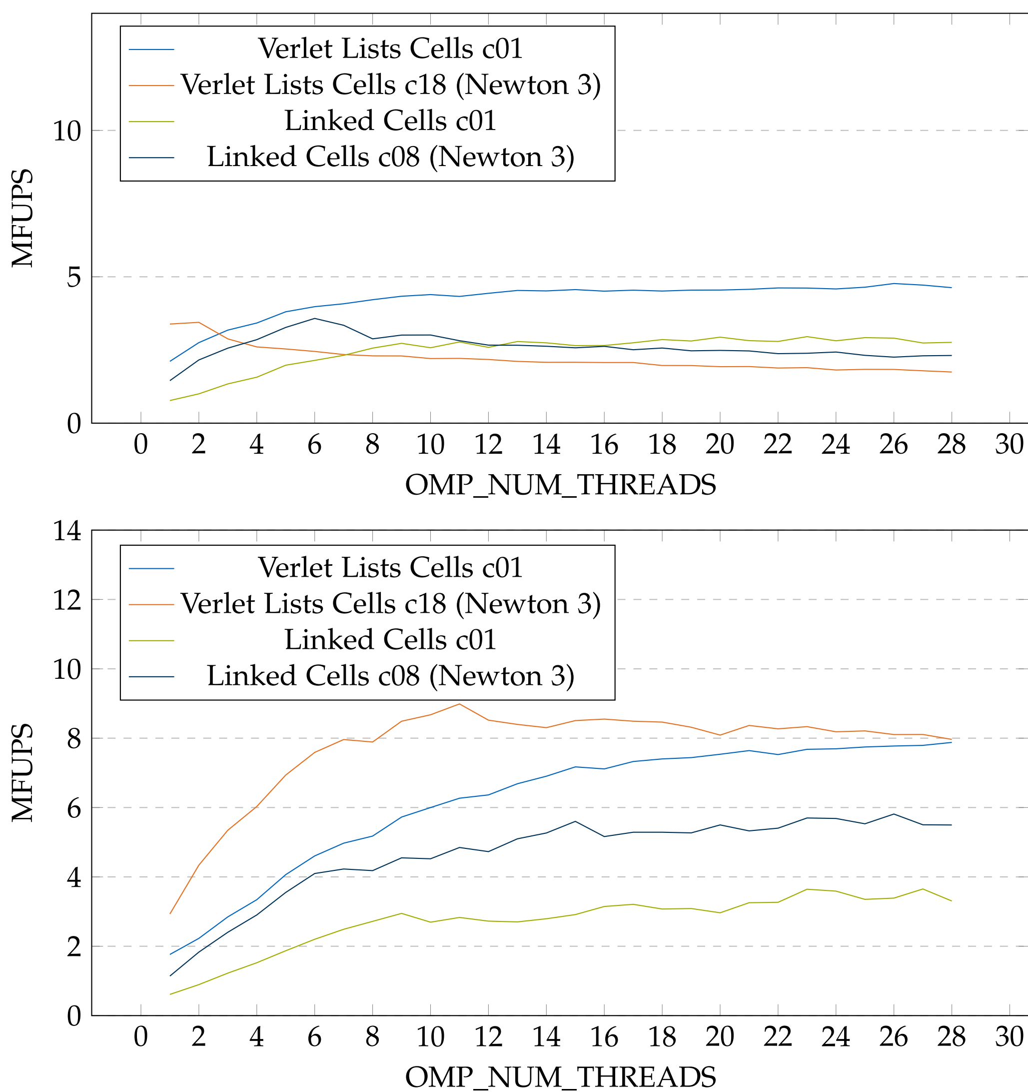

Figure 7.1: Force updates per second of traversals in dependence of number of threads used. 500 Molecules in a box with edge length 0.15 (Top). 4000 Molecules in a box with edge length 0.3 (Bottom).

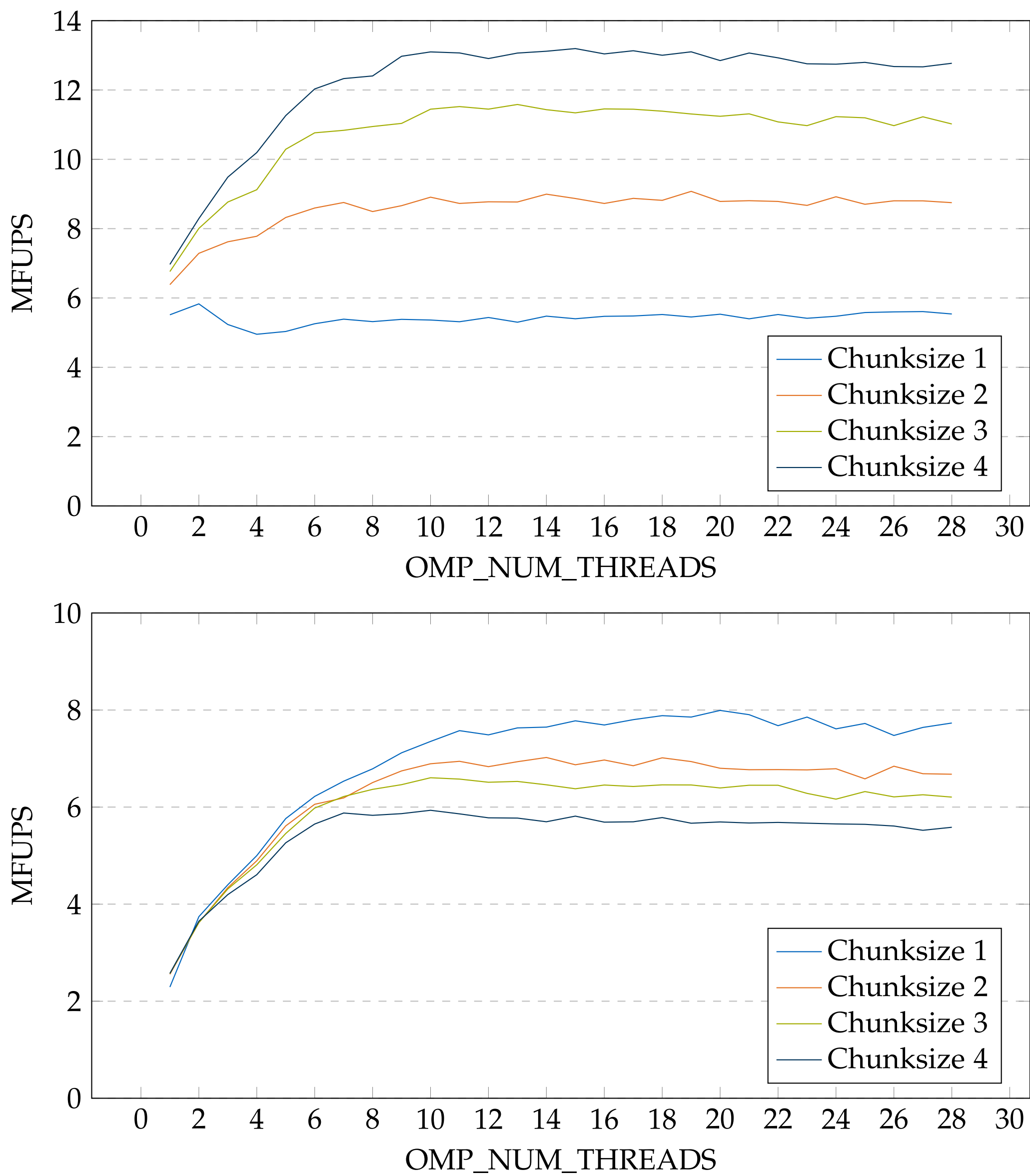

Figure 7.2: Force updates per second of verlet lists cells with different chunksize in dependence of number of threads used. We use c18 with newton 3. 10000 Molecules in a box with edge length 0.90 (Top). 4000 Molecules in a box with edge length 0.30. (Bottom)

Box size	0.15	0.3	0.45	0.6	0.75
Linked Cells	7x7x7	12x12x12	17x17x17	22x22x22	27x27x27
Verlet Lists	5x5x5	9x9x9	13x13x13	17x17x17	21x21x21

Figure 7.3: The number of cells used including halo cells

have more work to do than others. For small examples, we don't even have enough work to distribute. Because each thread has to be synchronized in the end, the overall runtime solely depends on the thread which takes the longest. But it's difficult to tell when which of the two cases occurs. And it's just as hard to know which chunk size is ideal in a given case.

7.3 Potential neighbours

The Linked-Cells algorithm reduces the number of potential pairs tremendously in comparison to checking all pairs. Ideally, for each particle, we only check the particles in the cutoff radius. But in the case of linked cells, we check all particles in adjacent cells. Apart from the edge cases, this corresponds to a cube with a side length of 3 times the cutoff radius. This translate to a volume of:

$$V_{LC} = (3r_c)^3 = 27(r_c)^3 \tag{7.1}$$

Therefore a skin of 30% increases this volume by 119,7%. Assuming a fixed density ρ, we have to test roughly twice as many pairs of particles to build our neighbor lists.

$$\#\text{Pairs LC} = n_{Particles} \cdot n_{\text{Neighbors LC}} \approx n_{Particles} \cdot V_{LC} \cdot \rho = n_{Particles} \cdot 27\rho(r_c)^3 \tag{7.2}$$

But in our example, we only have to do this once every 20 iterations. In all other iterations, we just can use the Verlet lists instead. We checked the distance of all pairs beforehand. So the Verlet-List contains all particles within distance cutoff plus skin. As a simplification, we choose the skin to be the cutoff multiplied by a skin factor s. Each Verlet list, therefore, contains all particles in a ball with volume:

$$V_{Verlet} = \frac{4}{3}(sr_c)^3\pi \tag{7.3}$$

The factor between the Verlet list and Linked-Cells is as a result:

$$\frac{\#\text{Pairs LC}}{\#\text{Pairs Verlet}} = \frac{n_{Particles} \cdot V_{LC} \cdot \rho}{n_{Particles} \cdot V_{Verlet} \cdot \rho} = \frac{27(r_c)^3}{\frac{4}{3}(sr_c)^3\pi} = \frac{81}{4s^3\pi} \tag{7.4}$$

In our case, the skin factor of 130% yields approximately 3 times the pairs. This means that the Verlet algorithm only has to check one third of the pairs of the Linked Cells algorithm. With the right settings the saved time can balance out the extra work in each rebuild-step. The ideal skin factor for a fixed rebuild period $T_{rebuild}$ and highest relative velocity between any two particle Δv_{max} is:

$$s = \frac{2 \cdot \Delta v_{max} \Delta t T_{rebuild}}{r_c} \tag{7.5}$$

Plugging this into Equation 7.3, we see that the number of pairs scales cubic with $T_{rebuild}$. Thus we have an unknown time consumption t_{Pair}, which scales cubically. And the equally unknown time $t_{Rebuild}$ that the rebuilding requires and a fixed expense $t_{General}$ on which we have no influence. This enables us to estimate the amortized runtime.

$$t_{Iteration} = t_{Pair} \cdot (T_{Rebuild})^3 + \frac{t_{Rebuild}}{T_{Rebuild}} + t_{General} \tag{7.6}$$

The theoretical minimum of which would be at:

$$T_{Min} = \sqrt[4]{\frac{t_{Rebuild}}{3 \cdot t_{Pair}}} \tag{7.7}$$

But finding this value for given situation is not trivial.

7.4 Cache

The used traversals ensure that different threads don't write onto the same data at the same time. Therefore, the threads should be able to work independently of each other apart from the synchronization steps. The problem here is false sharing. Figure 7.4 shows the difference of cache-efficient programming. In general, storing cluster pairs leads to more particle-pairs than normal Verlet-lists. Using a low density we can see that the GROMACS algorithm is always slower because of the increased overhead. But in the much denser example, it dominates the others. We use the simple C01 to iterate through the cluster list. But the difference is that we sorted the particles by their position. We moved the particles in such a way, that particles in a cluster are adjacent in memory space. Since each cluster is processed one after the other, we use the assumed spatial locality in this case. In addition, the clustering algorithm has many more cells due to the high density. This way we can distribute the work better. The algorithm scales much better and remains nearly linear.

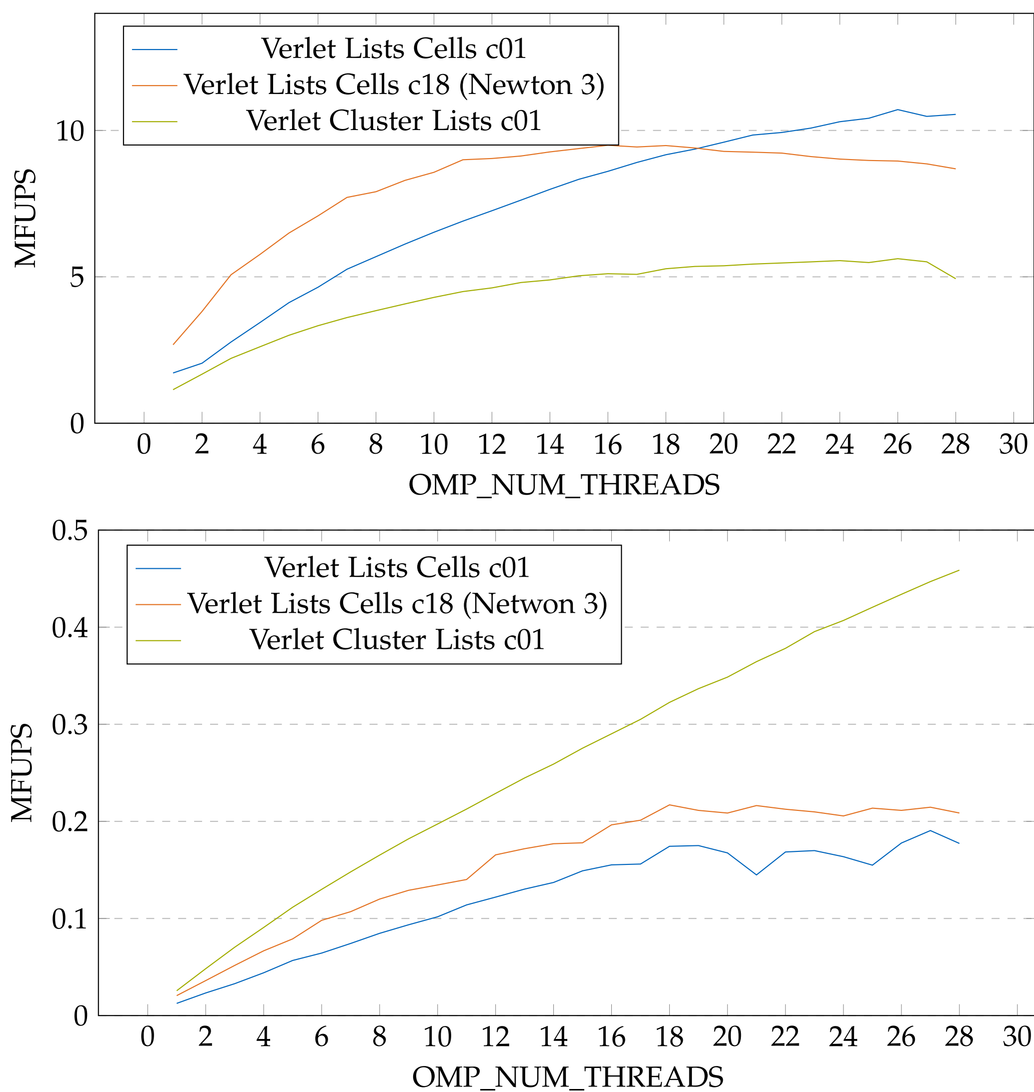

Figure 7.4: Force updates per second of traversals in dependence of number of threads used. 4000 Molecules in a box with edge length 0.3 (Top). 540000 Molecules in a box with edge length 0.3 (Bottom).

8 Conclusion

We have introduced many data structures and parallelization possibilities. The three data structures presented have already been implemented in AutoPas. For small cases, we also allow the calculation of all pairs so that no overhead occurs. The implementation using OpenMP enables additional setting options using environment variables. For example, by defining OMP_NUM_THREADS its possible to adjust the number of threads used. The performance, unfortunately, does not increase steadily with the number of threads used. Any combination of data structure and traversal has its own performance limit. It strongly depends on how the simulation looks like and which hardware is available. Therefore it is most important to offer as many algorithms as possible. This allows each user to choose the most suitable option for his case. That's why AutoPas is still in development and constantly testing new algorithms.

9 Future works

9.1 AoS and SoA

Our introduced algorithms don't make use of the SIMD capability of most CPUs. As we have already explained, vectorization requires attention to the layout of the data. In order to avoid unnecessary load instructions, we must be able to merge multiple into one. So the x-component of the position of each particle must be in memory consecutively. The same applies to all other properties. This form of storage is called Structure of Arrays (SoA). So far we have used Arrays of Structures (AoS). Figure 9.1 shows an example of objects defined only by their position and mass. While iterating the particles, the right layout allows us to process four particles at once.

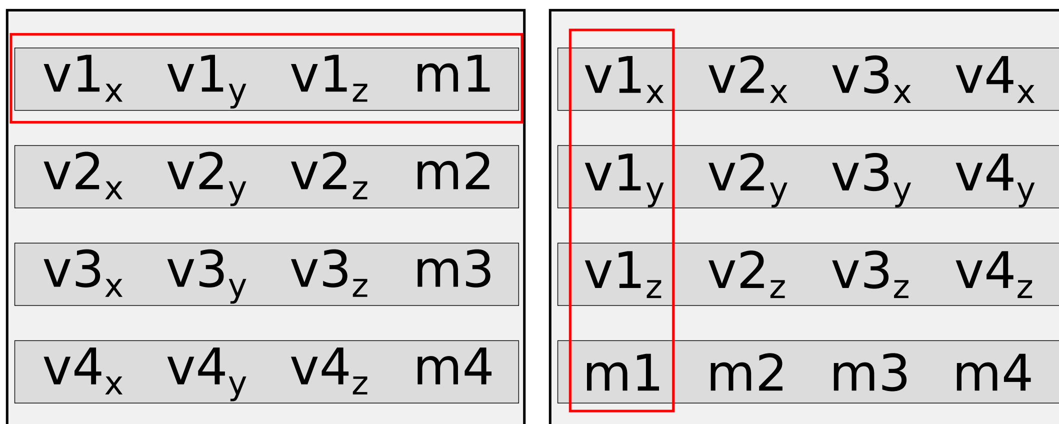

Figure 9.1: Comparison between AoS (left) and SoA (right).

This is especially helpful for the GROMACS algorithm. We can adjust the cluster size to the SIMD length. This allows a CPU to work cluster by cluster. This is because the data structure rearranges the particles. We make sure that all particles in a cluster are stored consecutively in memory. In normal Verlet lists, each neighbor list contains only a pointer to the neighbor. The neighbors themselves can lie widely distributed in the memory.

9.2 Verlet Cluster

The current implementation of the GROMACS algorithm is only temporary. Since this still has proven successful, we may implement it completely. This includes bounding boxes for more accurate distance estimation and closer observation of clusters at the edge of the cutoff. If there is a possibility that a cluster pair does not contain particle interactions, we check all pairs to see if this is the case. GROMACS uses the GPU to prune such pairs. Whether the trick is also suitable for shared-memory parallelization still has to be verified.

9.3 Auto tuner

We have identified some factors that influence the performance of the respective methods. The key values of a simulation are the particle density, the size of the domain and the cutoff radius. Yet it is still unclear how we can make the optimal selection from this information. That's why an auto tuner for AutoPas is under development. There are several settings that can be tested. This starts with the selection of the data structure and the traversal. Up to the choice of scheduling including the chunk size. The task of the auto-tuner is to determine the optimal setting by performing test runs. If we have to calculate many iterations, we can use the first ones to try out different settings. We measure the runtime to determine the performance of each run. With these measurements, we can estimate which setting is most suitable.

Bibliography

[All04] M. P. Allen. *Introduction to molecular dynamics simulation*. 2004.

[HLW03] E. Hairer, C. Lubich, and G. Wanner. "Geometric numerical integration illustrated by the Störmer–Verlet method." In: *Acta Numerica* 12 (2003), pp. 399–450. DOI: 10.1017/S0962492902000144.

[HP06] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006. ISBN: 0123704901.

[Ope15] OpenMP Architecture Review Board. *OpenMP Application Program Interface*. 2015.

[PH13] S. Páll and B. Hess. "A flexible algorithm for calculating pair interactions on SIMD architectures." In: *Computer Physics Communications* 184 (2013), pp. 2641–2650.

[TA87] D. Tildesley and M. Allen. *Computer Simulation of Liquids*. Clarendon, 1987.

[Ver67] L. Verlet. "Computer "Experiments" on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules." In: *Phys. Rev.* 159 (1 July 1967), pp. 98–103. DOI: 10.1103/PhysRev.159.98.

[VJG18] D. Vandevoorde, N. M. Josuttis, and D. Gregor. *C++ templates : the complete guide*. Boston : Addison-Wesley, 2018.