FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Developing and Benchmarking a Molecular Dynamics Simulation using AutoPas

Nicola Fottner

FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Developing and Benchmarking a Molecular Dynamics Simulation using AutoPas

Entwicklung und Benchmarking einer Molekulardynamiksimulation mit AutoPas

Author:	Nicola Fottner
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 Nicola Fottner

Abstract

The C++ library AutoPas is meant to be included into simulation programs, and delivers optimal node-level performance for N-body problems with dynamic auto-tuning at run-time. In this thesis, a molecular dynamic simulation program was built utilizing the flexibility of AutoPas. It provides multiple methods and functionalities in order to create different molecular dynamic scenarios and was evaluated by simulating the spinodal decomposition phenomenon on the CoolMUC-2 cluster. The results show, that the performance of pairwise force calculations increases by exclusively using Structure of Arrays and AVX intrinsics. In general, the performance is inferior to ls1-mardyn.

Zusammenfassung

Die C++ Library AutoPas wird in Computersimulationen eingeschlossen, um mittels dynamischem Auto-Tunings zur Laufzeit optimale node-level Leistung für N-body Probleme zu erreichen. In dieser Bachelorarbeit wurde mithilfe der Flexibilität von AutoPas ein Program für molekulardynamische Simulationen erstellt. Das Program verfügt über mehrere Funktionalitäten, um verschiedene Simulationsszenarien zu erstellen und zu simulieren. Mittels der Reproduktion des spinodalen Phasenzerfalls auf dem CoolMUC-2 Kluster, wurde die Leistung des Programms evaluiert. Die Ergebnisse zeigen, dass die Kräfteberechnungen durch AutoPas unter der exclusiven Verwendung spezifischer Datenstrukturen und intrinsischer Funktionen schneller durchgefhrt werden. Insgesamt ist die Leistung des molekulardynamischen SImulationsprogramms schwächer, als die von ls1-mardyn.

Contents

1. Introduction and motivation

With the help of science we are able to understand the complexity of our environment through observation and critical thinking. Nevertheless, it can be extremely costly and complicated to analyse certain phenomenons on very small or very big scales from empirical observation. Therefore the popularity of simulations is increasing as a way of explaining unknown behaviour. On a cosmological scale, the behaviour of huge galaxies was illustrated by the IllustrisTNG simulations[AP17] that contribute to the understanding of stars and mass. As many-body problems cannot be solved analytically, molecular dynamic simulations are used to provide numerical solutions about microscopic systems. Their application is mainly found in biology or chemical engineering [AH15], as they allow the observation of complex behaviour of molecules and atoms. Besides efficient numerical algorithms, super computers are needed to simulate scenarios with a huge number of particles, such as separation of gases or the simulation of complete viruses, in e.g the simulation of the satellite tobacco mosaic virus (STMV)[FP]. With parallel and distributed computing methods and advanced programs, simulations of events with up to twenty trillion molecules are possible [TSH$^+$19]. The Leibniz-Rechenzentrum (LRZ) provides with the SuperMUC the opportunity to simulate such events.

As molecular dynamic simulations proceed N number of particles, the simulated problems are considered as N-body problems. Additionally, every object in a system theoretically interacts with every other object, so that the computational complexity is $\mathcal{O}(N^2)$. Reducing the complexity is achieved by lowering the number of particle interactions. This is most commonly done by only considering short-range interaction. Efficient algorithms are already established for those type of interactions [Pli95] and can reduce the computational complexity down to $\mathcal{O}(N)$.

Still in development by the project TalPas ("Task-based Load Balancing and Auto-tuning in Particle Simulations"), the AutoPas library provides multiple algorithms and tuning procedures in C++ in order to optimize the performance of huge programs, such as ls1-mardyn [ls1]. Due to its modular and extensible structure, it can easily be enlarged by developers. Furthermore, because of its simple interface, AutoPas can effortlessly be integrated into other particle-type simulations.

This flexibility of AutoPas will be the focus of this thesis. The library's functionalities are covered in depth and used to implement a molecular dynamic simulation program called "md-flexible". Moreover, its performance is compared to the ls1-mardyn program by reproducing the Spinodal Decomposition phenomenon.

2. Theoretical background

2.1. Physical and chemical Background

When talking about molecular dynamic simulations, it is important to understand the mathematical, physical and chemical laws behind the computational processes. While we need scientific laws to understand the interaction between the objects of a system, it is important to distinguish the different numerical techniques that are responsible for the time discretization. Furthermore, algorithms and data layouts are needed to efficiently compute the natural phenomenons. In the following paragraphs, those scientific principles will be presented and shortly explained, before moving on to the related code.

2.1.1. Newton's laws of motion

Newton's laws of motion were first published with Newton's Philosophiae Naturalis Principia Mathematica in 1687 [New87] and deal with explaining the motion of objects in relation to their exerted force. They are applicable on all physical scales, from atomical (mixed with Kepler's laws of planetary motion) to planetary systems [nwi].
First law: when no external force is exerted on a object, the object remains a constant velocity and is eather at rest, or moving in a staight line.
Second law: the acceleration a of a body is proportional to the net Force acting on the body and to the mass m of the Object.

$$F = a + m \tag{2.1}$$

Third law: "For every action (force) in nature there is an equal an opposite reaction." Therefore, between two body the exerted forces are the following:

$$F_A B = -F_B A \tag{2.2}$$

The forces are equal in magnitude but opposite in direction. [Bro99]
Newton's third law of motion can be used to cut the number of force interaction in a system by half. The native complexity still stays at $\mathcal{O}(N)$, but the computational time needed is halved.

2.1.2. Equation of state in a macroscopic system

As a modification of the ideal gas law Equation 2.3, the van der Waals equaltion of state estimates the behaviour of gases while considering them to not act.

$$PV = nRT \tag{2.3}$$

It describes the universal forces of attraction and gives a relation of the volume, the pressure and the temperature in kelvin in a macroscopic system.

$$\left(P + \frac{an^2}{V^2}\right)(V - bn) = nRT \tag{2.4}$$

In Equation 2.4, P is the pressure of the fluid, V is the volume of the fluid, n is the number of particles and T is the absolute temperature of the system and R is the gas constant. Furthermore, the constant b describes the volume excluded from V by one particle and the constant a describes the average strength of attractive force that exist between particles in a gas that increases when the attractive force between particles is increasing. The Lennard Jones potential uses this Equation 2.4 to simulate the attractive forces. Moreover, the repulsive forces are based in the Pauli exclusion principle [pau]. In the md-flexible simulation, those forces are modeled by the attractive Lennard Jones Potential.

2.1.3. Lennard Jones Potential

The Lennard-Jones-Potential U_{LJ} also known as the 12-6 potential is a mathematical model and was first proposed by John Lennard-Jones in 1924. It describes the potential energy of two interacting particles leading to the difference between attractive and repulsive forces of these particles depending on their separation [LJ24].

$$U_{\mathrm{LJ}} = 4\epsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right] \tag{2.5}$$

Here, r is the distance between two particles, σ is the finite distance at with the potential is zero, ϵ is the depth of the potential well.

As visualised in Figure 2.1, when the distance r between two particles is bigger than $1.12 \cdot \sigma$, the resulting forces are attractive until the potential energy reaches a minimum, so that the forces become repulsive and r increases again. This is a repeating cycle that accounts for the motion of the particle when no other external forces are present. The Figure 2.1 visualizes the potential and the inter-molecular forces

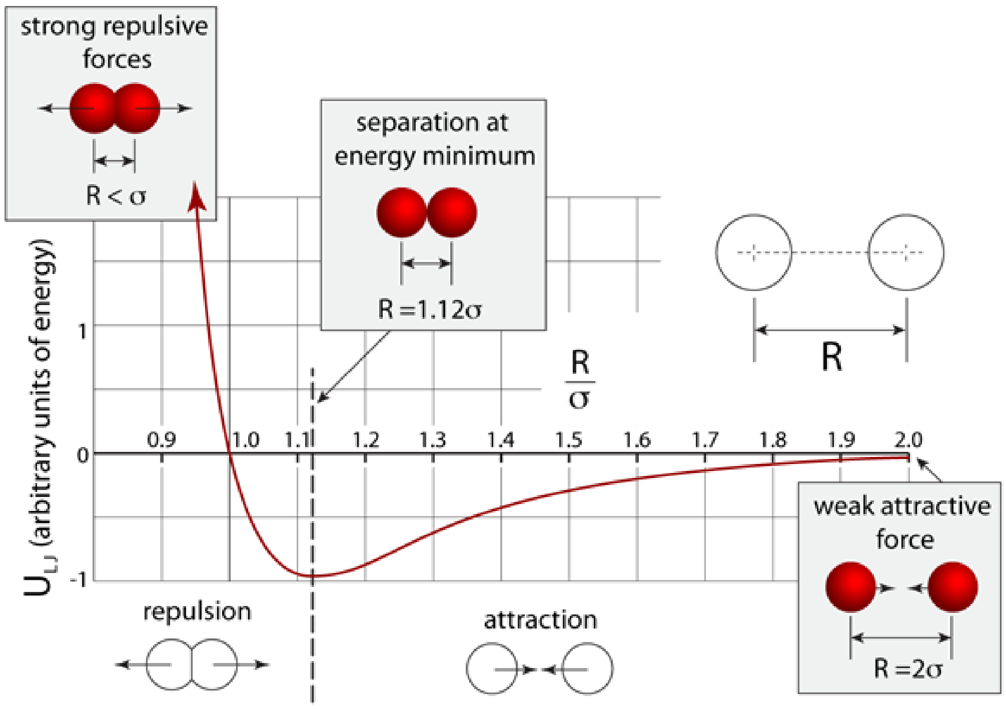


Figure 2.1.: The potential curve shows that if the seperation of the particles is situated left of the minimum they repel, otherwise they attract each other
Source: `http://atomsinmotion.com/book/chapter5/md`

The variables σ and ϵ in Equation 2.5 are specific to each particle, so that this potential cannot directly be applied to any pair of particles.
Accordingly, we need rules to compute the potential between particle pairs. There are different combination rules, and the choice of them can affect the accuracy of the simulation. The md-flexible simulation uses the Lorentz-Berthelot rules [Lor81][Ber98], that compute the resulting σ and ϵ by an arithmetic and a geometric mean respectively.

$$\sigma = \frac{\sigma_1 + \sigma_2}{2} \qquad \epsilon = \sqrt{\epsilon_1 \epsilon_2} \tag{2.6}$$

The simulation uses the gradient of the potential to compute the resulting forces acting on each particle [bmo]. To compute the forces of all particles with Equation 2.8, the program iterates pairwise through all particles. Here it is important to point out, that since the interacting force between particles diminishes with increasing distance, the potential between particles can be ignored. This is why, the user defines r_{cutoff} the maximum distance for which force interactions are being calculated.

$$F_{\mathrm{LJ}}(\overrightarrow{d_{\mathrm{i,i}}}) = \nabla_{\mathrm{d_{i,j}}} U_{\mathrm{LJ}}(d_{\mathrm{i,j}}) \tag{2.7}$$

$$F_{\mathrm{LJ}}(\overrightarrow{d_{\mathrm{i,j}}}) = \frac{24\epsilon}{|\overrightarrow{d_{\mathrm{i,j}}}|^2}\left[\left(\frac{\sigma}{|\overrightarrow{d_{\mathrm{i,j}}}|}\right)^6 - 2\left(\frac{\sigma}{|\overrightarrow{d_{\mathrm{i,j}}}|}\right)^{12}\right]\overrightarrow{d_{\mathrm{i,j}}} \tag{2.8}$$

In this model, $\overrightarrow{d_{\mathrm{i,j}}}$ is the vector between particles i and j, with $|\overrightarrow{d_{\mathrm{i,j}}}|$ its euclidean norm.

2.1.4. Kinectic energy and temperature

In a system, there are two forms of acting energies: Potential energy and kinectic energy. While potential energy is the energy stored by an object as a result of its position, kinectic energy describes the energy of motion. The kinectic energy of an object is proportional to its mass m and to its velocity v [Jai09].

$$E_k = \frac{1}{2}mv^2 \tag{2.9}$$

From this equation we deduct the average kinectic energy in a system:

$$E_{\text{kin}} = \sum\nolimits_{i=0}^{N} \frac{m_i < v_i, v_i >}{2} \tag{2.10}$$

Moreover, the temperature of a system is proportional to the average kinetic energy of that system. The relation between the temperature of a system and the kinectic energies between all particles are describe by Equation 2.11

$$T = \frac{2 \cdot \sum\nolimits_{i=0}^{N} \frac{m_i < v_i, v_i >}{2}}{dim \cdot N \cdot k_b} \tag{2.11}$$

In Equation 2.10 and Equation 2.11, m and v are the mass and velocity of the particle i. Furthermore dim is the number of dimensions in the system, N is the number of particles and k_B is the Boltzmann constant. The Boltzmann constant "is a physical constant that relates the average relative kinetic energy of particles in a gas with the temperature of the gas"[1]. To simulate heating or cooling processes, the simulation uses a thermostat, that scales the velocities of all particles in the system to obtain the desired temperature.

2.1.5. Brownian motion

The Brownian motion is the random motion of particles that results from the continuous collision with other particles. It was first observed by botanist Robert Brown in 1827 [Bro28] by observing the movement of pollen in water through a microscope. The Brownian motion of a single particle, is "the result of the thermal motion of molecular agitation of the liquid medium" [Hao19]. The smaller the particle and the higher the temperature in the system, the more the particles are displaced. This initial motion can be simulated in molecular dynamics by applying the Maxwell-Boltzmann distribution (Equation 2.12) in the first timesteps of the simulation using the thermostat that iterates over all particles in the system.

$$f(v) = (\frac{m}{2\pi k_B T})^{\frac{3}{2}} \cdot e^{-\frac{mv^2}{2k_B T}} \tag{2.12}$$

In Equation 2.12, m is the mass of the particle, k_B is the Boltzmann constant, T is the temperature and v is the velocity of the particle.

[1] `https://en.wikipedia.org/wiki/Boltzmann_constant`

2.2. Particle processing and time discretization

2.2.1. Time Discretization

The process of time discretization can be done with the application of different discretization methods. The basic idea, is to approximate the positions and velocities of all particles in every time step with different algorithms. Because of its simplicity and sufficient accuracy, the md-flexible simulation uses the Verlet-Störmer algorithm where the data for each particle is updated corresponding to its properties during the previous and current timestep [Swo82].

$$x(t^{n+1}) = x_i(t^n) + \Delta t \cdot v_i(t^n) + (\Delta t)^2 \frac{F_i(t^n)}{2m_i} \tag{2.13}$$

$$v(t^{n+1}) = v_i(t^n) + \Delta t \frac{F_i(t^n) + F_i(t^{n+1})}{2m_i} \tag{2.14}$$

Equation 2.13 and Equation 2.15 visualizes the computational procedure for the position x and velocity v of the particle i in relation to the acting forces F in timestep t of the *n-th* iteration with step size Δt.

2.2.2. Boundary conditions

For molecular dynamic simulations, as well as for any computational task, the resources are limited and the system has to fulfil a lot of constraints. As any imitation or simulation of natural phenomenons is processed in a limited domain, it is important that the simulation is modelling the rightful behaviour at the borders of that domain. Depending on the model, different boundary conditions can be implemented on each border of the domain. The most common boundary conditions are "Outflowing", "Reflecting", and "Periodic". For simulations that implement outflowing boundaries, the particles are disappearing when moving out of the borders. With reflecting boundaries, the particles won't vanish but will bounce off the borders.

Periodic boundaries can be defined as particles leaving the domain on one side and entering on the opposite side of the domain. This leads to particle interactions across borders and to an infinite domain. To implement reflecting and periodic boundaries, the domain is divided into three layers: inner, boundary and halo with the important property "interaction length" IT defined as followed:

$$IT = r_{cutoff} + r_{skin} \tag{2.15}$$

Here, $\mathrm{r_{cutoff}}$ represents the cutoff radius chosen in the Linked Cell algorithm Subsection 2.3.2 and the maximal distance of force interactions between the particles and r_{skin} represents the skin radius for verlet lists mentioned in Subsection 2.3.3.
The inner layer includes all particles with distance $> IT$ from all borders of the domain. The boundary layer contains all particles that are IT away from the boundary towards the innerlayer and $\mathrm{r_{skin}}$ towards the outside direction of the border. Finally, the halo layer contains all particles that are outside of the domain and $r_{skin}/2$ inside of the domain. To reflect the particles off a reflecting boundary, the common method is to copy the particle that has to be reflected onto the halo layer when it is getting closer to the border. As

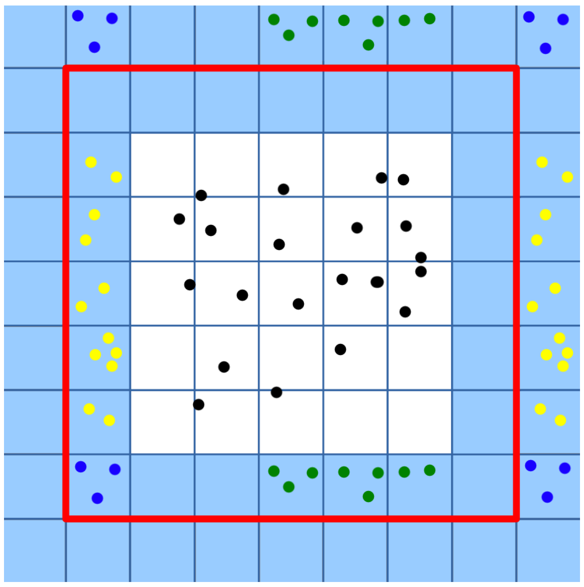


Figure 2.2.: Two dimensional boundary processing of particles by different layer cells. White cells are representing the inner layer, blue cells inside the red boundary are representing the boundaries layer and the halo layer is visualized by the cells outside of the red boundary. The copying of particles in the boundary cells to the halo cells is pictured by the different colors.

soon as the particle is moving outside of the domain, it is replaced by its appropriate halo particle, having the same properties as the original particle's, but with reverse velocity values. For periodic boundaries, as the particles interact across borders, the particles inside the boundary layer are interacting with respective particles in the halo layer. It is important to point out, that particles that are placed at the corners of the domain, have to be copied up the three times. This is visualized by Figure 2.2. To rightfully compute the halo particles, such that the memory consumption is not exploding, the AutoPas library is updating the halo particles every multiple time steps. For that, the "inner" domain and halo domain are divided into user defined sized cells. The management of cells is discussed Subsection 3.2.2. After simulating a new time step, new halo particles are computed, and halo particles from the previous computation are either deleted or cell-wise updated to their new position.

2.3. Molecular Dynamic Algorithms

2.3.1. Direct Sum

The direct sum algorithm is the most intuitive algorithm to compute the data of particles in a system. It is stating, that in a system of objects, all individual objects are interacting with each other. This is simply done by iterating over all particles in the system for every particle. By using the Newton's third law describes in Subsection 2.1.1, this method can be optimized, but nevertheless is very unefficient especially for large systems with a high number of particles. It has the most unnecessary calculations of particles of all three algorithms mentioned in Section 2.3. The complexity of a system implementing this method is $\mathcal{O}(N^2)$.

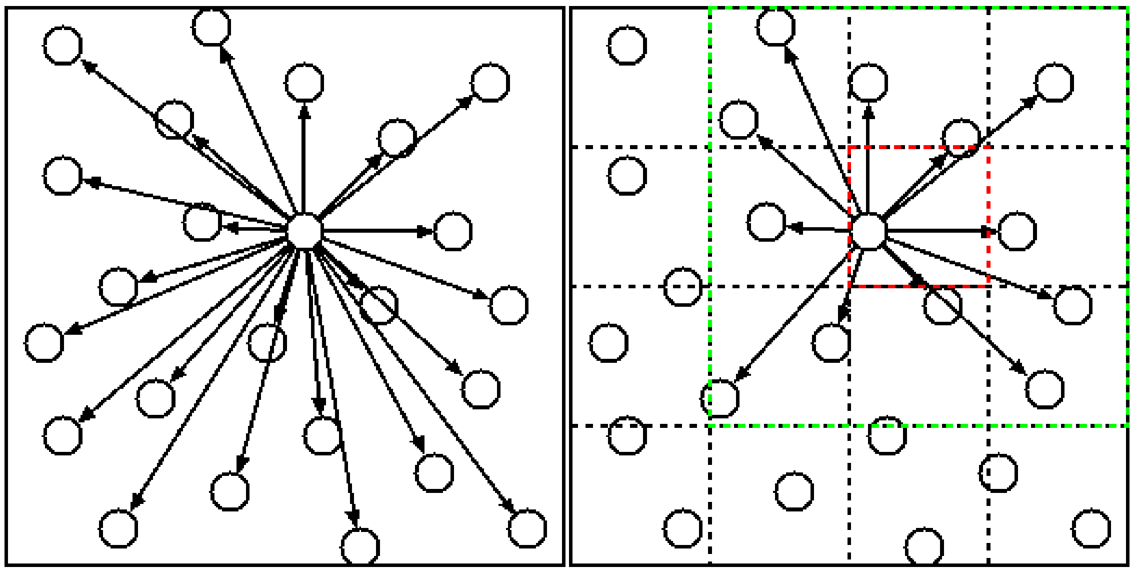

(a) Direct Sum (b) Linked Cell

Figure 2.3.: Particles interactions of Direct Sum and Linked Cell
modified from source: `https://en.wikipedia.org/wiki/Cell_lists`

2.3.2. Linked cell

The linked cell algorithm is one of the most used algorithm to proceed particles in molecular dynamic simulations. It divides the multidimensional domain into cells with adjustable but uniform border size c_{size}, and computes the data of the particles by iterating through the affected neighbouring cells. In most cases c_{size} is equal to r_{cutoff} so that the algorithm only needs to iterate through the neighbouring cells of the current cell. Regarding a three dimensional system, the linked cell algorithm would iterate over all 26 neighbour cells of the current cell. If c_{size} is smaller then r_{cutoff}, then it needs to iterate through more cells. Moreover, if c_{size} is bigger than r_{cutoff}, then in most cases, computational power is lost, because unnecessary particles are included into the process, mentioned in Subsection 2.1.3. The particle interactions are visualized in Figure 2.3.
With N the number of particle and r_{cutoff}, the computational effort of this method is

$\mathcal{O}(cN) \in \mathcal{O}(N)$. The linked cell algorithm is in between the direct sum and verlet list algorithms in terms of computational and memory overhead, but benefits most of the SoA data structure mentioned in Section 2.4. A full comparison of the methods is shown by Figure 2.5.

2.3.3. Verlet Lists

To compute the particles in a system using the verlet list algorithm, a data structure is created for every particle. Every particle contains a list of its neighbouring particles within r_{cutoff}. This list must be rebuilt every multiple time step as the particles are moving. To build these lists, it is necessary to iterate pairwise through all the particles and check all distances. The native computational effort for the build step is $\mathcal{O}(N^2)$, but can be improved with the linked cell algorithm to $\mathcal{O}(N)$ [ZYC04].

Furthermore, by multiplying the size of the list by $r_{verlet\text{-}skin}$, more particles will be included in the force calculations and the rebuild frequency can be diminished. Figure 2.4 visualizes the neighbour region for a single particle.

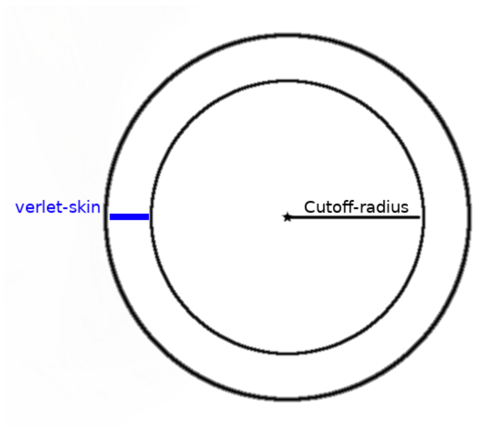


Figure 2.4.: Neighbour region of a verlet List

2.3.4. Comparison of the methods

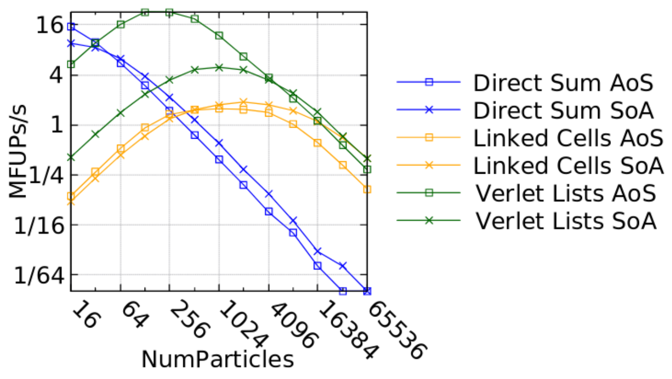
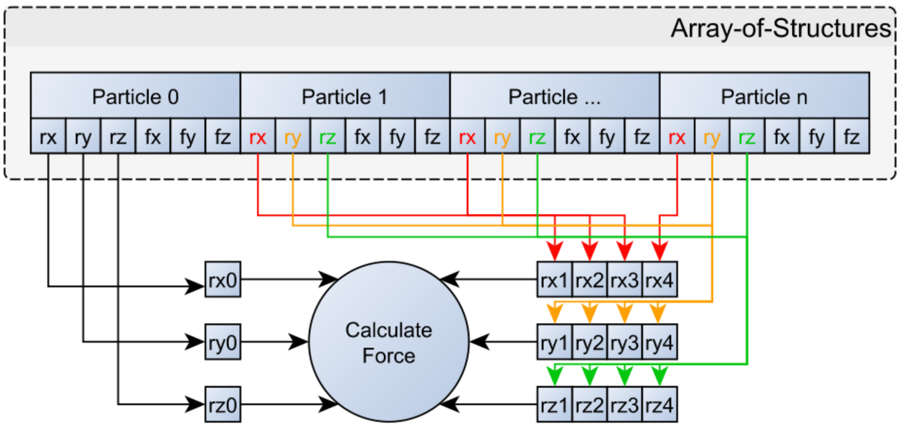


Figure 2.5.: Comparison of the Molecular Dynamic Algorithms tested on the Cool-MUC2 (mentioned in Section 6.1)
Source: [GST+19]

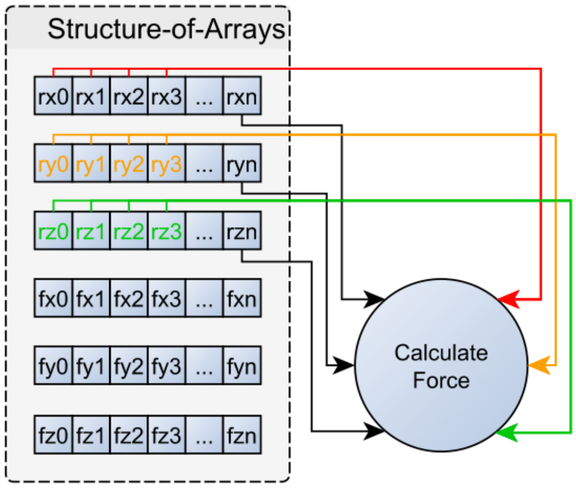
Figure 2.5 shows a comparison between the different molecular dynamic algorithms. In general, the usage of direct sum should be avoided for large scale scenarios, and the denser the scenario becomes, the more beneficial is the usage of the linked cell algorithm over the verlet lists.

2.4. Data Layouts

To efficiently compute the particles in the system, different data layouts can be chosen. Different data structures are used for accessing that data of the particles stored in the memory. Mostly vectors or arrays are used as containers to store the data corresponding to the particles. The Array of Structures (AoS) is the more intuitive one. As seen in Figure 2.6, for each particle, a data structure is set up with all the properties important to the calculation. This leads to multiple accesses on different memory locations being necessary to calculate the force interactions between the particles. However, it allows to easily manage the particles. Adding and deleting particles from a AoS data structure is easier than when using the Structure of Arrays (SoA). The SoA shown in Figure 2.7, has multiple advantages over the Array of Structures. The data structure separates each property of the particle used in the molecular simulation into different containers with size equal to the number of particles in the simulation. Data is therefore loaded efficiently as it is stored continuously in memory. To calculate the forces between the particles, the structure of arrays is also more efficient because of vectorization techniques mentioned in Section 3.2.5.
Moreover, as every particle need to be copied, the conversion from AoS to SoA is expensive.



[a]



[b]

Figure 2.8.: Data Layouts

3. Related codes

3.1. ls1-mardyn

ls1-mardyn is a highly optimized program to simulate enourmous domains in molecular dynamics. It was developped by the cooperation of four universities: the High Performance Center Stuttgard (University of Stuttgart), Laboratory for Engineering Thermodynamic (University of Kaiserslautern), the scientic computing faculty (Technical University of Munich), Thermodynamics and Energy Technology (University of Paderborn)[ls1]. It is built to compute parallely on supercomputing architectures and can simulate systems with up to twenty trillion particles [TSH+19]. In this thesis, the performance of ls1-mardyn with and without AutoPas is compared to the md-flexible simulation explained from the AutoPas library. This is done by computing the SpinodialDecomposition mentioned in Chapter 5.

3.2. AutoPas

The AutoPas library is an open-source project[1] and part of the "TaLPas"[2] project. It is meant to be included into simulation programs, and allows the optimization of N-body problems by dynamic auto-tuning at run-time in order to deliver the best node-level performance for the current state of the simulation. Though it doesn't include any native functionality to run on multiple nodes, it is possible to use the MPI functionalities of ls1-mardyn, or other simulation programs, to run an AutoPas instances on each node.
The mechanism of auto-tuning, described in Subsection 3.2.3, chooses the best configurations for the simulation by analysing the properties of the simulation in the current timestep. Most scientific simulations work on specific input with specific algorithms that are chosen before runtime. In practice it is hard to choose the right static properties to efficiently simulate variable scenarios. The Spinodal Decomposition, mentioned in Chapter 5, is such a scenario that describes the transformation from a homogeneous to a heterogeneous state. The AutoPas library fills this gap in Molecular Dynamic simulations with auto-tuning at run-time [GST+19]. In this thesis, most of the AutoPas library functionalities are included into the modular and flexible simulation program called "md-flexible", included as an example program inside the library, which allows the user to easily use the library's processes to build multiple simulation scenarios.

[1] `https://github.com/AutoPas/AutoPas`
[2] `https://wr.informatik.uni-hamburg.de/research/projects/talpas/start`

3.2.1. Structure of AutoPas

The AutoPas class is the main point of interaction between the user and the library. It abstracts all the functionalities of the containers and the tuning processes and additionally allows a pairwise iteration through the particles in the system. This procedure is further explained in Subsection 3.2.5. In order to be adaptable to multiple types of simulations, the class uses templates to be generic to the type of Particle cells and to the type of Particles used. The entire AutoPas object is built around C++ function templates. AutoPas divides the domain into cells. The containers provide an interface to handle the storage for the particles, and the functionality to add particles into the domain. Inside the domain, with help of iterators it is possible to iterate through the AutoPas container. Inside the containers are also implemented the different traversal methods. In case of molecular dynamics, a functor is used that calculates the Lennard Jones mentioned in Subsection 2.1.3. The particles, must inherit from a base class to work over the functionalities of the AutoPas library. On the restriction given by the "ParticleBaseclass", developpers can define their own particles and to implement different simulations. Furthermore to optimize a highly compute-intensive program at run-time, the library has different selectors with different tuning strategies are managed by the AutoTuner, mentioned in Subsection 3.2.3.

When a simulation program uses AutoPas to optimize its performance, it has to rightfully initialize the AutoPas object. This is done, when the user passes all necessary options for a auto-tuned simulation. The initialisation of the AutoPas object is shown by Listing A.2.

The AutoPas library can only compute the particles, if in all dimensions the domain is bigger than IT as defined in Equation 2.15. Therefore the user has to set the domain with "boxMin" and "boxMax" because the default values aren't working. After that, the user needs to call the *init()* function, displayed in Listing A.3, to initialize the auto-tuning process and the logicHandler according to the properties of the simulation before any other function is called on the AutoPas object.

3.2.2. Container management

The AutoPas library implements different type of containers implementing the molecular dynamic algorithms, mentioned in Section 2.3, to manage the particles in the system. All containers must inherit from the base class called "ParticleContainerInterface.h", that provides an interface for all the containers within AutoPas. It defines methods for addition and deletion of particles, accessing container properties, creating and iterating through particles, updating the container. All those functionalities are managed by the *LogicHandler* class, which calls specific functions on the AutoPas object. One of the functions is the *updateContainer()* function. This function can be called from the AutoPas object, and potentially updates the internal container by deleting halo particles, and resorting the owned particles into the appropriate cells. Besides, the *LogicHandler* manages all the function calls of the containers, to be called from the AutoPas instance.
Moreover, there are three different iterator behaviours, so that the container can iterate through the particles inside the container. The behaviours are: Iterating only through halo particles, iterating only through owned particles or iterating through both type of particles. By using the "begin(iteratorBehaviour)" function, the user can access the first particle of a container depending on their type and iterate through them with the *++* operator.

Furthermore, all container must also allow to iterate through particles of a certain region. In most cases, this is done by accessing specific cells in a cell base container. The accessing is done with the help of the "CellBlock3D" class, an interface designed to manage blocks of particle cells and provide a functionality to easily access the cells in the system by converting the three dimensional cell index to a one dimensional one. This class also provides a method to resize the cell size if needed by the AutoTuner.

3.2.3. Tuning

The AutoTuner receives the data from the simulation for auto-tuning from the containers. As all the containers are sharing a common interface, the AutoTuner follows the strategy pattern. This allows to select different configurations for the simulation at runtime for the containers. A configuration is dependent on the containers used, and is a combination of the following items:

- container
- traversal
- data Layout
- usage of Newton's third law
- cell size Factor

All available configurations of the system shape a "search space" from which the AutoTuner chooses the optimal configuration depending on the tuning strategy. Currently two different tuning strategies are implemented inside AutoPas: The *fullSearch* or the *bayesianSearch*. More tuning strategies can be implemented by inheriting from the "TuningStrategyInterface". The *fullSearch* strategy tests every possible configuration of the search space and then selects the optimum. The *bayesianSearch* predicts the best configuration option that will run through the program in the next step.
The tuning strategies are running when the simulation is in tuning phase. The procedure involves the following activities:

- For each available configuration, the AutoTuner collects multiple samples of statistical measurements. The number of samples collected depends on the user's input
- Comparing the samples with the selectorStrategy given by the user, the tuner deduces the optimal configuration having a high chance of delivering the optimal performance in the next iterations.

This procedure is repeated periodically. The period of applications on the system is specified by the user during the initialisation.
The AutoTuner is instantiated by the *init()* function call showed inside the AutoPas class Listing A.3. The selected configuration of traversals and data layouts will be applied with the next call of the iterate pairwise method. For the specific containers having a list of neighbouring particles, the tuner will rebuild the neighbouring lists.

3.2.4. Travelsals

The AutoPas library supports the use of OpenMP parallelization methods. When the tuner selects the right configuration for the AutoPas container, it chooses the best traversal for the current time-step. All cell based traversals are subclasses of the *CellPairTraversal* with inherits from the *TraversalInterface*. Most common used traversals are the *Sliced traversal* and the cell based traversals with the nomenclature *"C" "numberOfCells"*. The implemented algorithms are *"C01"*, *"C04"*, *"C08"* and *"C18"* and implement different orders to iterate through the cells. The traversals are being generated by the AutoTuner when calling the *iteratePairwise* method. The tuner chooses the right traversal accordingly to the current container, the pairwise functor, the data layout and the current configuration. Then the particles in the system are processed by the traversal.

The *sliced traversal* finds the longest dimension of the simulation domain and cuts it into so called "slices". Every thread then processes every slice as they are assigned following the round robin procedure. This means, that the slices are processed in a circular order. Each thread locks the cells on the boundary wall to the previous slice until the boundary wall is fully processed. Figure 3.1 shows the procedure on 3 threads, (colored in yellow, black and blue) working in parallel. The cells in red are synchronization cells. When the yellow thread is processing the cells, it locks the cells in red until the black thread has finished to process the first column of cells. This parallelization method is straight forward when the size of the cells is a least r_{radius}. But it is getting more complicated, when the cellsize is less then r_{radius}, because race conditions can happen on multiple overlaying cells therefore more cells must be synchronized.

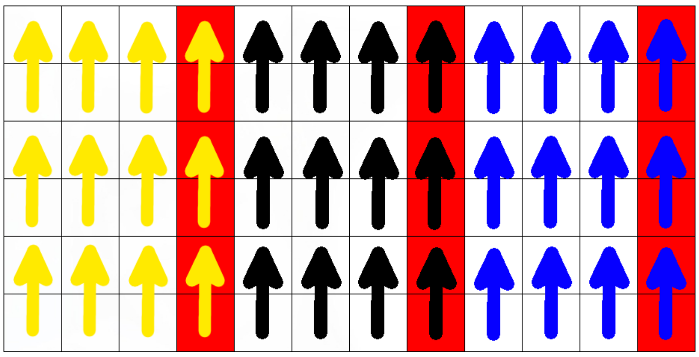


Figure 3.1.: Sliced traversal with 3 threads marked with yellow, black and blue. The red cells represent synchronization cells

For the cell based traversals with the nomenclature *"C" "numberOfCells"*. The procedure is as follows. The grid of cells in divided into cell cubes. For example, the *C08* traversal, subdivides the domain of cells into cube of cells with 8 cells in a three dimensional system. That means, for cell traversals with the nomenclature *"C" "numberOfCells"*, *"numberOf-Cells"* is the number of cells of the cubes in a three dimensional space. For 2D, the number of cells in a cube is divided into 2. After that all the cubes of cells are each assigned to one thread, all force interactions of the particles in the cube are being calculated. The thread processes the interactions of each cell with the other cells. This is shown by Figure 3.2.

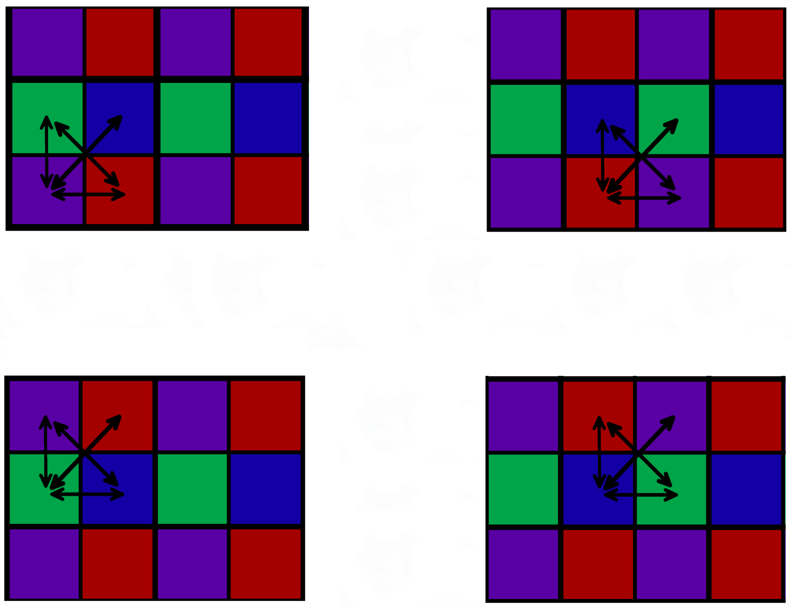


Figure 3.2.: Cutoff of a C08 traversal on 2D

The number of threads that are assigned to a domain is equal to the number of cubes that are generated by the traversal. The threads are processing the cubes independently from each other. As the direct sum container is only using two cells, one cell for halo particles and another cell for owned particles, it is possible to only compute the calculations either for all halo or for all owned cells. Therefore the direct sum algorithm is getting quickly inefficient when big systems need to be built with high number of particles. The efficiency can therefore be increased when using the linked cell container. Though the linked cell is working with a lot of cells, it is important to rightfully handle the cells close to each other.

3.2.5. Particle computation

To compute the interaction between the particles inside the simulation, AutoPas provides the functionality to iterate pairwise through all the particles. This is done by the function, listed in Listing A.4, which is called from the AutoPas object. In order to rightfully use this functionality, the used Functor must correlate with the particles in the simulation and needs to provide functions to calculate their forces interactions with AoS and SoA data types. This allows the user to adapt the AutoPas library to other types of simulations. For the md-flexible program, as force interactions are calculated with help of the Lennard Jones(LJ) potential, the "MoleculeLJ" particle is used. The elements needed for the procedure are described at Section 4.2. As the pairwise iteration over particles differs from container to container, the logic handler manages the iteratePairwise call as mentioned is Subsection 3.2.2.

Vectorization and SIMD instructions

AutoPas allows the computation of the particles under different data layouts mentioned in Section 2.4. While using the AoS (Array-of-structure) data type, it is not possible to process multiple particle datas with SIMD instructions, because of the definition of AoS. However, with the SoA (Struture-of-Arrays), as specific data of the particles is stored linearly in the memory, it is possible to speed up the computation of force interactions between the particles by vectorizing with SIMD instructions. The pairwise LJFunctor is using OpenMP SIMD directives to auto-vectorize the computation of the Lennard Jones forces between the particles. Though, as no function calls are allowed and very few math operations are possible inside a vectorized domain, resulting values of the mixing rules mentioned in Subsection 2.1.3 (necessary for the calculation of the Lennard Jones potential), had to be preloaded into vectors for the SIMD instructions in order to work.

4. Implementation of the Simulation

4.1. Structure of the Program

The AutoPas library is built to be as generetic as possible, to work with different particles, particle cells, pairwise functors and traversals. The md-flexible simulation is working with particles that are suited for the force calculations via the Lennard Jones Potential, those will be called MoleculeLJ. However, it can easily be adapted to other Particle types and particle cells as mentioned in Section A.1.
The settings for the simulation are being chosen either through the command line or with a Yaml configuration file. Further details about setting the simulation environment are discussed in Section A.2. The main point of interaction between the user and the md-flexible simulation is given by the "Simulation" class witch controls the simulation process.
First, the Simulation class processes the input given by the user and configures the AutoPas object accordingly. After initializing the simulation domain in accordance with the information of the particles provided inside the Yaml configuration file, it then proceeds the pairwise force calculation, the dicretization of time and if specified, applies the thermostat and periodic boundaries.
The calculation of the pairwise force interaction is done by calling the *iteratePairwise()* function of AutoPas, mentioned in Subsection 3.2.5, and needs the "ParticlePropertiesLibrary" when a scenario is simulated with multiple type of particles. The "ParticlePropertiesLibrary" map properties of the particles to their type and is further described in Section 4.2.
The movement of the particles is realized by the "TimeDiscretization" class. In order to simulate the behaviour for the position and velocity of all particles over time, the class uses the iterator functionality of AutoPas. Therefore the Verlet-Störmer algorithm, mentioned in **??** can easily be applied by creating an OpenMP parallel region and iterating over all particles inside the AutoPas container.
The periodic boundary functionality is implemented inside the "BoundaryCondition" class. By using the *updateContainer()* function, mentioned in Subsection 3.2.2, it copies the particles that are leaving the domain. Furthermore, the periodic boundary position change is done by shifting the positions of the leaving particles accordingly to their three dimensional position inside the domain, and adding to the AutoPas container.
As mentioned in Equation 2.11, the thermostat scales the velocity of all particles in a system, in order to get the desired temperature in the system. This is done by also utilizing the Iterator functionality of AutoPas.

4.2. Storage of particle properties

The md-flexible simulation uses "MoleculeLJ" particles because they provide the necessary data structure for the simulation to work displayed in Table 4.2.
Inside Table 4.2, *floatType* is a template parameter of the particle, and indicates the precision of the particle variable. Float for 32 bit, and double for 64 bit precision.

Data	**Data type**	**Size in memory in [Byte]**
Position	std::array<floatType,3>	double: 24 float: 12
Velocity	std::array<floatType,3>	double: 24 float: 12
Force of current time step	std::array<floatType,3>	double: 24 float: 12
Force of previous time step	std::array<floatType,3>	double: 24 float: 12
Owned	boolean	1
Particle Id	unsigned long	4
Type of particle	unsigned long	4

Table 4.1.: Elements of Molecule LJ particle

The boolean "Owned" defines whether the particle is inside the domain of the AutoPas object or not.
When simulating with 64 bit precision, the program needs to allocate 105 bytes of memory for every particle, compared to 57 bytes when using a 32 bit precision.
However, to compute the force interactions and the time discretisation, the algorithm needs to access further particle properties: Mass m, Epsilon ε, Sigma σ. The "Particle Properties Library" class, stores these properties for every particle type using the std::map container to easily access the data.

Property	**Data type**	**size in memory in [Byte]**
Mass m	floatType	double: 24 float: 12
Epsilon ε	floatType	double: 24 float: 12
Sigma σ	floatType	double: 24 float: 12

Table 4.2.: Data stored in Particle Properties Library

As those properties are not stored in every instance of the MoleculeLJ particle, 72 bytes or 36 bytes are spared for every particle depending on the accuracy of the data. This heavily reduces the needed memory for the simulation.

4.3. Object Generation with Yaml

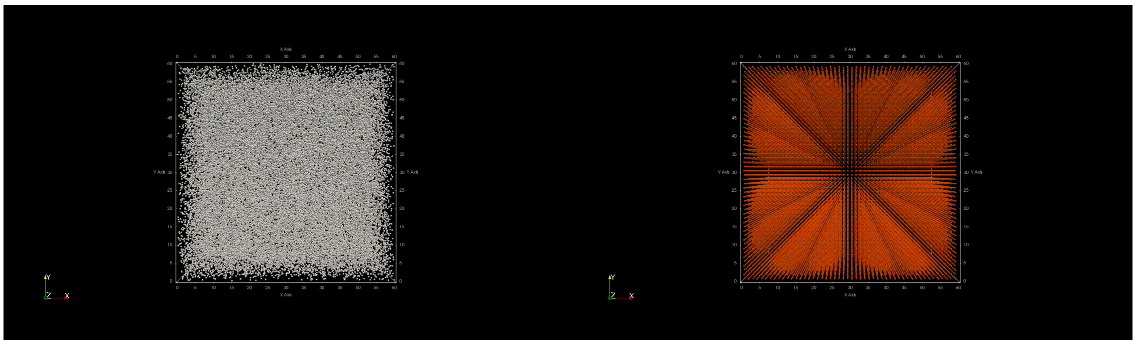
With the help of Yaml configuration files, the md-flexible simulation supports the functionality to generate multiple cubes or spheres in a 3 dimensional space. The cubes can either be filled with a grid of particles, or distributed either randomly or following the Gaussian distribution. An example of the configuration file is shown in Listing A.1.

5. Simulation Scenario : Spinodal decomposition

In order to compare the md-flexible program with ls1-mardyn, the spinodal decomposition phenomenon was performed respectively on both simulation programs.
In thermodynamics, spinodal decomposition is known as the mechanism for the rapid unmixing from a mixture of liquids or solids to form two distinct phases with different chemical compositions and properties [JBC94]. In contrast to the nucleation the formation of a new thermodynamical phase from an existing one where the transition occurs at discrete nucleation sites, the unmixing of the phase is more defined and occurs uniformly throughout the material. The decomposition happens in so called spinodal regions, and as there is no thermodynamic barrier to the reaction inside that region, the decomposition concludes from diffusion. Diffusion is the movement of a material from an area of high concentration to an area of low concentration [JK60]. The separation of the phase can be illustrated by the phase diagram.
In this thesis, the spinodal decomposition was performed on particles with mass = 1.0 u, which approximately corresponds to the chemical element of Hydrogen (with mass=1.008 u). To prepare the decomposition, a system is created where the particles are at equilibrium. Therefore, the simulated scenario consist of two phases. The equilibrium phase and the decomposition phase.

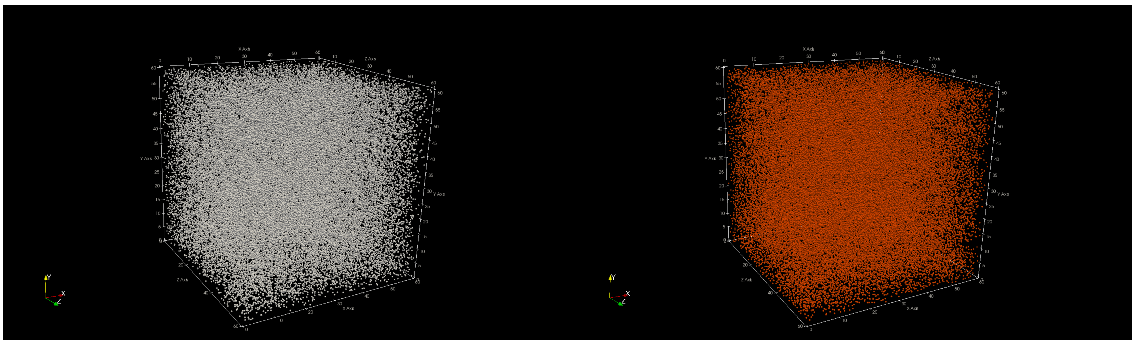
5.1. Equilibrium Phase

Phase equilibrium is a static condition witch exists between or within different states of phases (namely liquid, gas and solid). The stage of equilibrium means that the chemical potential of any object present in the system stays steady with time. That means that throughout the region of equilibrium, all physical and chemical properties of the particles are the same and the interaction between the particles are spacially uniform. It is "the absence of any tendency toward change on a macroscopic scale" [JS17].
To reach a state of equilibrium, and later simulate the spinodal decomposition on that state, 62 500 particles were initialized in a cubiq domain 60 nm in length. With the help of a thermostat the temperature of the system is hold to 1.4 K for 500 000 simulation time steps. The initialization of the particles is different for both simulating programs ls1-mardyn and md-flexible. ls1-mardyn generates a grid of particles with initial density 0.29 ρ. The md-flexible simulation program initilized the domain by filling it with randomly uniformly distributed particles. Figure 5.1 shows the initialisation and Figure 5.2 shows the final state of the equilibrium phase for both programs,
After 500 000 simulation time-steps we expect that the particles are evenly distributed. The output images produced by both programs are showing the expected behaviour.



(a) md-flexible (b) ls1-mardyn

Figure 5.1.: Initial state of Equilibrium phase



(a) md-flexible (b) ls1-mardyn

Figure 5.2.: Final state of Equilibrium phase

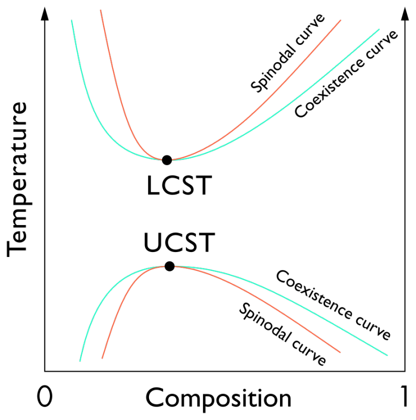


Figure 5.3.: Phase Diagram displaying spinodal curves, within the binodal coexistence curves and two critical points
Source: [spi]

5.2. Decomposition Phase

In order to simulate the second phase of the scenario, the temperature of the domain is dropped to 0.7 K. By decreasing the temperature, the state of the system is getting mechanically instable. This transition of the state is shown by Figure 5.3 and happens when the state is within one of the spinodal curve. Generally beneath the critical point UCST ("Upper critical solution temperature") or above the critical point LCST ("Lower critical solution temperature"). With the increase in time, the system is decomposing itself to a balance of liquid and gas. The balance of two distinct phases coexisting, is represented by the binodal curve, figured by the "Coexistence curve" in Figure 5.3. This procedure is simulated for 80 000 time steps by both simulation programs.
While for ls1-mardyn, dropping the temperature from 1.4 K to 0.7 K takes around 80 simulation steps, the md-flexible program is directly scaling the velocities in the system so that the desired temperature is attained at the first time step.
Right after the temperature dropped, we expect to observe the transition from a homogeneous to a heterogeneous system. More and more areas with low particle density are forming which represent the formation of gases contrary to building clusters of Hydrogen atoms.
With time, the topologie of the system is converging to a stable state.
Both programs are using periodic boundary on all borders to simulate the scenario.
The Figure 5.4, Figure 5.5 and Figure 5.6 are visualising this phenomenon simulated by both simulation programs.
The topologie of both programs is slightly different. In case of the simulation done by the md-flexible program, a main cluster of hydrogen molecules in the middle of the domain was built. Resembling a uneven sphere with outgoing arms. In comparison to the images from the ls1-mardyn simulation program, the clusters of particles are less connecting one another throughout the periodic boundaries. A difference between both simulations is expected, as both programs initialise differently the domain, and use different numerical algorithms to compute the data.

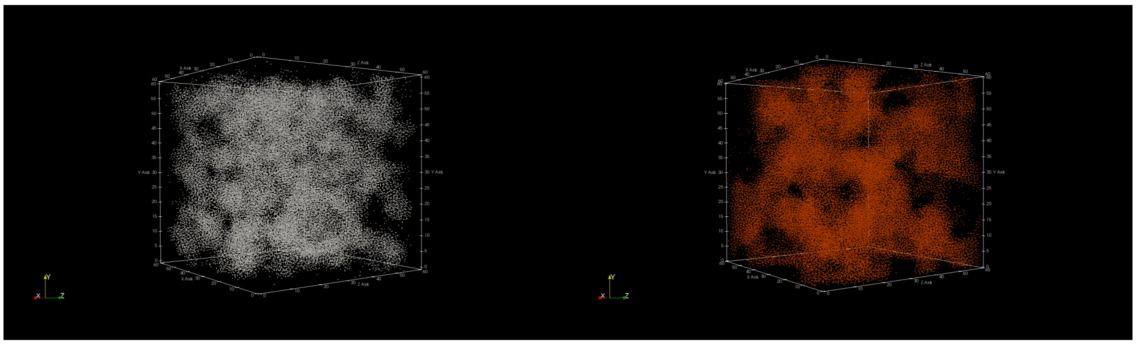

(a) md-flexible (b) ls1-mardyn

Figure 5.4.: Decomposition Phase at time-step: 20 000

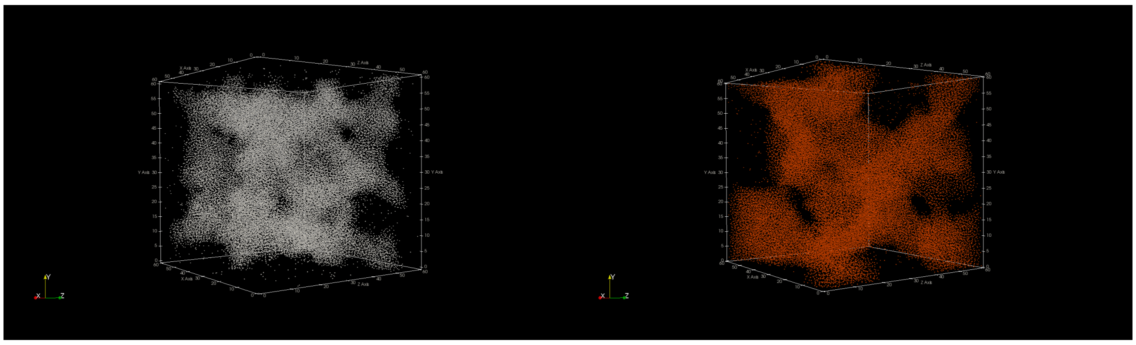

(a) md-flexible (b) ls1-mardyn

Figure 5.5.: Decomposition Phase at time-step: 40 000

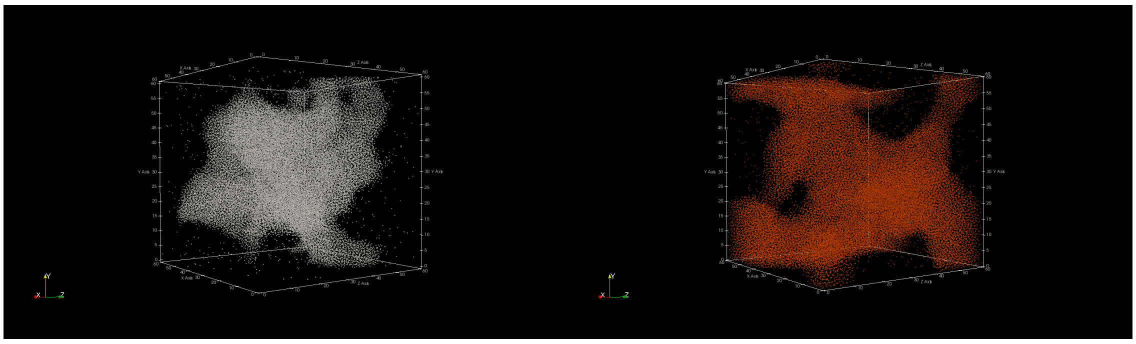

(a) md-flexible (b) ls1-mardyn

Figure 5.6.: Decomposition Phase at time-step: 80 000

6. Simulation results

6.1. Linux-Cluster CoolMUC-2

The LRZ Linux Clusters consist of several segments with different processors and different sizes of shared memory. The CoolMUC-2 is a cluster for both serial and parallel programming and was used to simulate the Spinodal Decomposition scenario with ls1-mardyn and md-flexible. It uses Haswell-based[1] nodes on the Intel Xeon E5-2697 v3 processor.

	CoolMuc-2
CPU	Intel Xeon E5-2697 v3
Cores per Node	28
Total Number of Cores	10752
Base frequency [GHz]	2.6
RAM per Node [GB]	64
Total Number of Nodes	384
Vector instruction set	AVX-2 (256 Bit)
Max. aggregated RAM [TB]	3.8

Table 6.1.: CoolMUC-2 Hardware

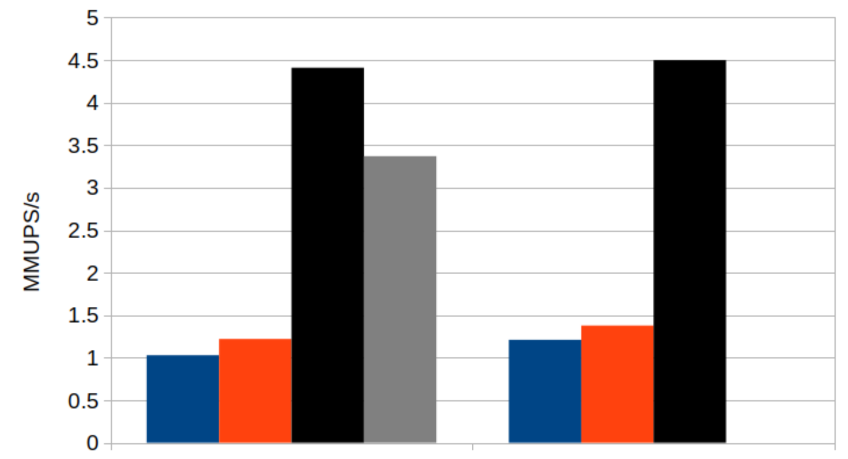
6.2. Benchmarking results

	ls1-mardyn	**md-flexible**
Allowed traversals	c08, sliced	all traversals
Allowed containers	linked Cell	linked Cell
Traversal selector strategy	fastest median	fastest Abs
Data layout	SoA	SoA
Tuning interval	1000	1000
Tuning samples	10	10

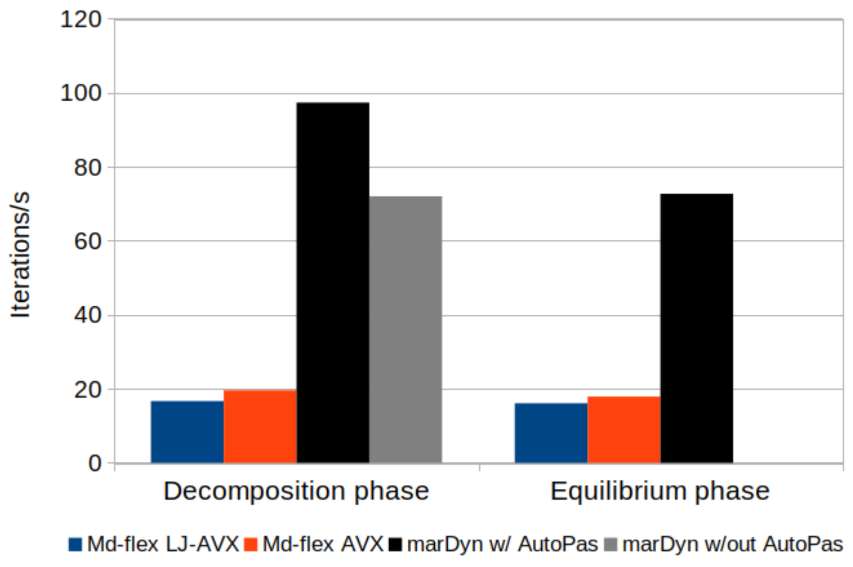
Table 6.2.: AutoPas options used for md-flexible and for auto-tuned ls1-mardyn execution

In order to benchmark the md-flexible simulation, it was run with two different type of Functors, the standard LJ Functor and the LJ Functor using AVX instrinsics (only implemented for SoA data layout). The performance measurements for both simulation programs are displayed by Table 6.4 for the equilibrium phase and by Table 6.3 for the decomposition phase.

[1] `https://en.wikipedia.org/wiki/Haswell_(microarchitecture)`



[a]



[b]

Figure 6.3.: Iterations/s and MMUPS/s for both simulation programs run on CoolMUC-2 with 28 threads

In average, the main loop of the md-flexible program is 4,8 times slower than the one of ls1-mardyn. The force calculations are by average 4,3 times slower. As Figure 6.1 and Figure 6.2 show, ls1-mardyn is a lot faster than md-flexible.
Furthermore, the force calculations are in average 24,5% faster with AVX instrinsics. When only observing the run by ls1-mardyn of the decomposition phase, AutoPas didn't improve the performance. Though, theoretically, as the state is very variable, the usage of the AutoPas library should better the performance. This is due because not the optimal options were chosen for AutoPas. But in general, the highly optimized ls1-mardyn program is faster. [GST+19].

	md-flexible		**ls1-mardyn**	
Build Options	LJ	LJ-AVX	w/out AutoPas	with AutoPas
Total time [sec]	4556,41	4097,37	821,703	1075,06
Force computation [sec]	2547,35	2045,50	513,81	563,84
Iterations/s	17,54	18,86	97,36	74,42
MMUPS/s	1,09	1,22	4,41	3,37

Table 6.3.: Performance measurements of Decomposition phase for both programs

	md-flexible		**ls1-mardyn**
Build Options	LJ	LJ-AVX	w/out AutoPas
Total time [sec]	31119,291	28030,801	6870,03
Force computation [sec]	15590,62	11192,89	4667,51
Iterations/s	16,12	17,85	72,04
MMUPS/s	1,211	1,378	4,502

Table 6.4.: Performance measurements of Equilibrium phase for both simulation programs

Furthermore, for the chosen scenario, when AutoPas can choose between AoS and SoA, it sometimes runs multiple time steps of the simulation with AoS, though it would be faster with SoA. The exclusive usage of SoA, leads that no expensive copying from AoS to SoA is performed and auto-vectorization methods are uses. This is confirmed by the measurements in Table 6.5: the force calculations are 8% faster when only using SoA.

	md-flexible	
Build Options	LJ with AoS, SoA	LJ with SoA only
Total time [sec]	4763,59	4556,41
Force computation [sec]	2767,69	2547,35
Iterations/s	16,67	17,54
MMUPS/s	1,03	1,09

Table 6.5.: Measurements of decomposition phase data layout usage: AoS, SoA vs SoA only

7. Conclusion

Based on the functionalities of the C++ library AutoPas, a molecular dynamic simulation program was built. Designed to be generic to the particles, it can be adapted to different types of simulations by implementing the appropriate pairwise functor responsible for the calculation of particle interactions. Currently, the pairwise force interactions are calculated with the help of the Lennard Jones potential. Therefore, suitable particles and an adapted library were implemented, both beeing used to store the data to be computed. The library reduces the memory overhead as it stores common properties for the same type of particles. Moreover, the Verlet-Störmer method and a thermostat were implemented in order to perform the movement of the particles and to keep the system of the simulated scenario on a desired temperature. Both functionalities use OpenMP parallelization methods to enable the execution on multiple cores.
In order to benchmark the performance of the simulation, it was compared to the massively parallel code of ls1-mardyn by using the CoolMUC-2 cluster. Therefore, the spinodal decomposition phenomenon was simulated on both platforms. Restricting the AutoPas configurations for md-flexible to only use Structure of Arrays accelerated the computations of the force interaction by 8%. Furthermore, the usage of AVX intrinsics increased the performance of the force calculations by 24%. Finally, the md-flexible code runs in average 4,8 times slower than ls1-mardyn under optimal configuration.
Following the results of this thesis, the performance of the developed simulation tool could be improved by optimizing the iterators and by merging numerical calculations over the data of the particles.

A. Appendix

A.1. Code modularity

The md-flexible simulation is designed to be generic to the following elements:

- Particles
- ParticleCells

By default, the md-flexible simulation is build with the particle type "MoleculeLJ" that correlates with the Functor calculating the Lennard Jones force interactions. It is important that the Functor is coherent with the particles used inside the simulation.
The needed steps to change the type of simulation are:

- changing the template parameters of the Simulation object inside the main function
- implementing a new pairwise Functor that works with the desired type of particles
- adapting the function call *calculateForces* inside the simulation loop to the new Functor

A.2. Setting simulation setting

The user can pass the simulation options for the md-flexible simulation by specifying a Yaml configuration file or by passing options through the command line. When passing the path to the yaml file, the user can override the settings of the yaml file with the command line. The syntax of the Yaml files is clear and easy to use . It allows the user to create scenarios with multiple Objects (Cubes of Spheres) in space. All setting for the simulation are displayed by Listing A.1 in the yaml syntax.

```
###AutoPas Options=
container:                          #string option
data-layout:                        #string option
selector-strategy:                  #string option
traversal:                          #string option
tuning-strategy:                    #string option
newton3:                            #string option
cell-size-factor:                   #string option
log-level: debug                    #string option
tuning-interval:                    #int option
tuning-samples:                     #int option
tuning-max-evidence:                #int option
verlet-rebuild-frequency:           #int option
verlet-skin-radius                  #double option
no-flops:                           #boolean option
log-file:                           #string option
vtk-filename:                       #string option
vtk-write-frequency:                #size_t option
###Simulation Options=
functor:                            #string option
delta_t:                            #double option
iterations:                         #size_t option
cutoff:                             #double option
###Thermstat Options=
Thermostat:
  initializeThermostat:             #boolean option
  initTemperatur:                   #double option
  numberOfTimesteps:                #size_t option
  target:
    targetTemperature:              #double option
    delta_temp:                     #double option
###Checkpointing =
checkpointFile:                     #string option: fullPath
###Objects generation Options =
Objects:
  CubeGrid:
    0:
      particle-type: 0              #size_t option
      particle-epsilon: 1.          #double option
      particle-sigma: 1.            #double option
      particle-mass: 1.             #double option
      particles-per-Dim: [10,10,10] #list of size_t
      particleSpacing: 0.5          #double option
      velocity: [10.,10.,10.]       #list of doubles
      bottomLeftCorner: [5.,5.,5.]  #list of doubles
```

```yaml
CubeGauss:
  0:
    particle-type: 0                  #size_t option
    particle-epsilon: 1.              #double option
    particle-sigma: 1.                #double option
    particle-mass: 1.                 #double option
    numberOfParticles: 100            #size_t option
    box-length: [8.,8.,8.]            #list of doubles
    distribution-mean: 2.0            #double option
    distribution-stddev: 5.0          #double option
    velocity: [0.,0.,0.]              #list of doubles
    bottomLeftCorner: [5.,5.,5.]      #list of doubles
CubeUniform:
  0:
    particle-type: 0                  #size_t option
    particle-epsilon: 1.              #double option
    particle-sigma: 1.                #double option
    particle-mass: 1.                 #double option
    numberOfParticles: 100            #size_t option
    box-length: [10.,10.,10.]         #list of doubles
    velocity: [0.,0.,0.]              #list of doubles
    bottomLeftCorner: [5.,5.,5.]      #list of doubles
Sphere:
  0:
    particle-type: 0                  #size_t option
    particle-epsilon: 1.              #double option
    particle-sigma: 1.                #double option
    particle-mass: 1.                 #double option
    center: [0.,0.,0.]                #list of doubles
    radius: 10                        #size_t option
    particleSpacing: 0.5              #double option
    firstId: 0                        #size_t option
    velocity: [5.,5.,5.]              #list of doubles
```

Listing A.1: Yaml configuration file

A.3. Important AutoPas functions

```
explicit AutoPas(std::ostream &logOutputStream = std::cout)
      : _boxMin{0, 0, 0},
        _boxMax{0, 0, 0},
        _cutoff(1.),
        _verletSkin(0.2),
        _verletRebuildFrequency(20),
        _tuningInterval(5000),
        _numSamples(3),
        _maxEvidence(10),
        _tuningStrategyOption(TuningStrategyOption::fullSearch
           ),
        _selectorStrategy(SelectorStrategyOption::fastestAbs),
        _allowedContainers(allContainerOptions),
        _allowedTraversals(allTraversalOptions),
        _allowedDataLayouts(allDataLayoutOptions),
        _allowedNewton3Options(allNewton3Options),
        _allowedCellSizeFactors
        (std::make_unique<NumberSetFinite<double>>(std::set<
           double>({1.}))) {
    _instanceCounter++;
    autopas::Logger::unregister();
    autopas::Logger::create(logOutputStream);
    autopas::Logger::get()->flush_on(spdlog::level::warn);
  }
```

Listing A.2: AutoPas constructor

```
  void init() {
    _autoTuner = std::make_unique<autopas::AutoTuner<Particle,
        ParticleCell>>(
        _boxMin, _boxMax, _cutoff, _verletSkin, std::move(
           generateTuningStrategy()),
        _selectorStrategy, _tuningInterval,
        _numSamples);
    _logicHandler =
        std::make_unique<autopas::LogicHandler<Particle,
           ParticleCell>>
        (*(_autoTuner.get()), _verletRebuildFrequency);
  }
```

Listing A.3: Initialisation function for AutoPas

```
template <class Functor>
void iteratePairwise(Functor *f) {
  static_assert(not std::is_same<Functor, autopas::Functor<Particle,
  ParticleCell>>::value, "The static type of Functor in iteratePairwise
  is not allowed to be autopas::Functor. Please use the "" derived type instead,
              e.g. by using a dynamic_cast.");
  if (f->getCutoff() > this->getCutoff()) {
    utils::ExceptionHandler::exception("Functor cutoff ({}) must not be larger
    than container cutoff ({})", f->getCutoff(), this->getCutoff());
  }
  _logicHandler->iteratePairwise(f);
}
```

Listing A.4: AutoPas iteratePairwise method

List of Figures

List of Tables

Bibliography

[AH15] Modesto Orozco Josep L Gelpi Adam Hospital, Josep Ramon Goni. Molecular dynamics simulations: advances and applications. November 2015.

[AP17] Lars Hernquist Volker Springel Rüdiger Pakmor Paul Torrey Rainer Weinberger Shy Genel Jill P Naiman Federico Marinacci Mark Vogelsberger Annalisa Pillepich, Dylan Nelson. First results from the illustristng simulations: the stellar mass content of groups and clusters of galaxies. December 2017.

[Ber98] D. Berthelot, editor. *Sur les m elanges des gaz.Comptes rendus hebdomadaires des s eancesde l'Acad emie des Sciences*, 1898.

[bmo] Basics of molecular dynamics. `http://phys.ubbcluj.ro/~tbeu/MD/C2_for.pdf`. Accessed: 2019-09-12.

[Bro28] R. Brown., editor. *A brief account of microscopical observations made in the months of June, July and August 1827, on the particles contained in the pollen of plants*, 1828.

[Bro99] Michael E. Browne. *Schaum's outline of theory and problems of physics for engineering and science*, page p. 58. McGraw-Hill Companies, 1999.

[FP] Larson SB McPherson A Schulten K Freddolino P, Arkhipov A. Satellite tobacco mosaic virus.

[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.

[Hao19] Tian Hao. *Electrorheological Fluids*, pages 235–236. Elsevier B.V.r, 2019.

[Jai09] Mahesh C. Jain. *Textbook of Engineering Physics (Part I)*, page 9. Phi Learning, 2009.

[JBC94] L. H. E. Kihlborg R. Metselaar M. M. Thackeray J. B. Clarke, J. W. Hastie. Definitions of terms relating to phase transitions of the solid state. *The Scientific Journal of IUPAC*, pages 577–595, 1994.

[JK60] P.J. Dunlop L.J. Gosting G. Kegeles J.G. Kirkwood, R.L. Baldwin. Flow equations and frames of reference for isothermal diffusion in liquids. *The Journal of Chemical Physics 33(5)*, pages 1505–1513, 1960.

[JS17] Michael Abbot M.T. Swihart J.M. Smith, Hendrick C Van Ness. *Electrorheological Fluids*, pages 30–35. McGraw-Hill Education, 2017.

[LJ24] J. E. Lennard-Jones. On the determination of molecular fields. *Proc. R. Soc. Lond.*, page 463–477, 1924.

[Lor81] H. A. Lorentz, editor. *Ueber die Anwendung des Satzes vom Virial in der kinetischenTheorie der Gase.Annalen der Physik*, 1881.

[ls1] ls1-mardyn. `http://www.ls1-mardyn.de/about.html`. Accessed: 2019-09-04.

[New87] Isaac Newton. *Philosophiæ Naturalis Principia Mathematica.* 1687.

[nwi] Wikipedia: Newton's law of motion. `https://en.wikipedia.org/wiki/Newton's_laws_of_motion`. Accessed: 2019-09-04.

[pau] Pauli exclusion principle. `https://en.wikipedia.org/wiki/Pauli_exclusion_principle`. Accessed: 2019-09-12.

[Pli95] Steve Plimpton. Fast parallel algorithmsforshort–range molecular dynamics. *Journal of Computational Physics, vol 117*, 1995.

[spi] Spinodal. `https://en.wikipedia.org/wiki/Spinodal`. Accessed: 2019-09-11.

[Swo82] William C.; H. C. Andersen; P. H. Berens; K. R. Wilson Swope. A computer simulation method for the calculation of equilibrium constants for the formation of physical clusters of molecules: Application to small water clusters. *The Journal of Chemical Physics. 76 (1)*, page 648, 1982.

[TSH$^+$19] Nikola Tchipev, Steffen Seckler, Matthias Heinen, Jadran Vrabec, Fabio Gratl, Martin Horsch, Martin Bernreuther, Colin W Glass, Christoph Niethammer, Nicolay Hammer, Bernd Krischok, Michael Resch, Dieter Kranzlmüller, Hans Hasse, Hans-Joachim Bungartz, and Philipp Neumann. Twetris: Twenty trillion-atom simulation. *The International Journal of High Performance Computing Applications*, Jan 2019.

[ZYC04] G.-R. Liu Z. Yao, J.-S. Wang and M. Cheng. Improved neighbor list algorithm in molecular simulations using celldecomposition and data sorting method. *Computer physicscommunications, vol. 161*, page 27–35, 2004.