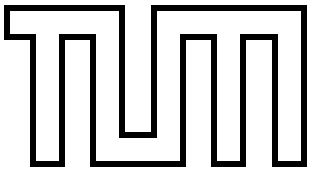


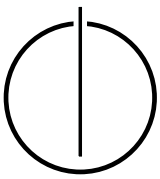
FAKULTÄT FÜR INFORMATIK

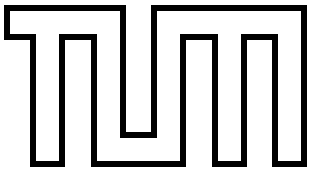
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Engineering Science

Evalulation and Adaptation of the Flexible Node-Level Library AutoPas in Molecular Dynamics Simulations

Raffael Düll





FAKULTÄT FÜR INFORMATIK

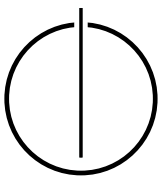
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Engineering Science

Evalulation and Adaptation of the Flexible Node-Level Library AutoPas in Molecular Dynamics Simulations

Evaluierung und Anpassung der flexiblen Node-Level Bibliothek AutoPas für Simulationen im Bereich der Molekulardynamik

Author:	Raffael Düll
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisor:	Fabio Alexander Gratl, M.Sc.
Date:	2nd of August 2018



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

Munich, 2nd of August 2018 Raffael Düll

Abstract

The aim of this Bachelor's thesis is to analyse the performance and the scalability of the developing AutoPas library for molecular dynamics. A testing program which supports pairwise intermolecular interactions for atoms and rigid molecules has therefore been developed . The evaluation of the library is performed on thin nodes of the SuperMUC phase 1 and on CoolMUC-3 at the LRZ Linux-cluster. The main aspects of this thesis where OpenMP parallelisation on a single node and vectorisation by SIMD instructions.
It has been shown that the efficiency of the force calculations with the AutoPas library improves with adequate vector instructions. At node-level, AutoPas scales well if few cores are used, but the execution time does not decrease as rapidly as expected for a higher number of threads. The overall performance remains below the performance of the highly scalable program ls1-mardyn.

Zusammenfassung

Das Ziel dieser Bachelorarbeit ist die Leistungs- und Skalierbarkeitsanalyse der Bibliothek AutoPas im Bereich der Molekulardynamik. Zu diesem Zwecke wurde ein einfaches Programm entwickelt, welches intermolekuläre Kräfte zwischen Atomen und Molekülen unterstüzt. Thin nodes auf der Phase 1 des SuperMUC und der CoolMUC-3 des Linux-Clusters am LRZ werden verwendet, um die Evaluierung von AutoPas durchzuführen. In erster Linie betreffen die Untersuchungen OpenMP Parallelisierung auf Knoten-Ebene und SIMD Vektorisierung. Die Effizienz der Kraftberechnungen mit der AutoPas Bibliothek kann dank einer angepassten Vektorisierung deutlich verbessert werden. Bei der Verwendung von wenigen Prozessoren skaliert AutoPas sehr gut, doch für eine höhere Anzahl an threads verbessert sich die Laufzeit weniger als erwartet. Die allgemeine Leistung des Programms bleibt allgemein unter der des stark skalierbaren Programms ls1-mardyn.

Contents

1. Introduction and motivation

The investigation of chemical processes has historically been conducted through experiments. However, the growing knowledge in chemistry, the motivation to analyse more complex relations and the impossibility to directly observe events at a molecular scale increase the importance of molecular dynamics (MD) simulation. The continuous increase in the precision and the performance of computers and the increasing availability of HPC supercomputers to run simulations at very large scales improve the results of MD simulations and their popularity in chemical research. They fill the gap between nanoscale interactions at a molecular level and macroscopic observations of chemical entities. Thus, they make the verification of the accuracy of theoretical models and the prediction of the behaviour of particular molecule configurations possible [BBB+14].
Molecular dynamics simulations can be considered as N-body problems in which each atom is represented as a point mass which interacts with all other points. Therefore, in the general case, the growth of the computational complexity is quadratic with respect to the number of simulated elements. The main challenge of MD simulations is to provide an efficient and scalable environment without any disproportionate loss of accuracy.
To that effect, a new C++ library for molecular dynamics named AutoPas is currently under development at the chair for Scientific Computing in Computer Science at the Technical University of Munich. It provides a flexible environment in order to support a broad range of possible scenarios. The user has to specify the underlying particle type, the occurring pairwise interactions between particles and the overall structure of the simulation tool. In particular, AutoPas handles the particle traversals and the pariwise interactions which are decisive for the overall performance of the simulation. Moreover, it includes OpenMP parallelisation and SIMD vectorisation in order to maximise the use of the available hardware resources on a single node.
In this thesis, the implementation of a molecular dynamics program, named MolSim, based on the AutoPas library and a consequent node-performance analysis are provided. The program supports single-site atoms as well as multi-site molecules which can move freely in the simulation domain and may experience a uniform gravitation field. The pairwise interactions between particles can be derived from a Lennard-Jones, an electrostatic, a gravitational or a harmonic potential. After the explanation of the theoretical foundations of a molecular dynamics simulation in Chapter 2, a brief overview of ls1-mardyn and AutoPas is given in Chapter 3. Thereafter, the implemented program is drafted in Chapter 4 with the comparison of different implementation alternatives to ensure a good performance of the code. Finally, the results of node-level experiments to evaluate the performance and the scalability of the program and of the integrated AutoPas library are presented in Chapter 5.

2. Theoretical background

2.1. Chemical background

As molecular dynamics simulations describe the behaviour of chemical entities, pairwise interactions between particles are the basis of many simulation tools. The forces acting on each particle are typically derived from potentials, such as the Lennard-Jones potential or the electrostatic potential. Moreover, energy conservation and Brownian motion improve the accuracy of the simulation. Since molecular dynamics occur at very small scales, meaningful unit systems are necessary.

2.1.1. Lennard-Jones potential

The main interaction between molecules in a fluid or gas is represented by the Lennard-Jones potential.

Spontaneous and inhomogeneous charge distributions in a single atom can cause a local polarisation. Consequently, adjacent elements orient along the same axis, and an induced dipole with weak attractive forces, which are called London dispersion forces, occurs [EL30].

According to the Pauli exclusion principle, two electrons with the same spin cannot occupy the same quantum space. Therefore, if two identical electron orbitals overlap, repulsive forces occur between the two electrons [Dys67], thus particles are kept apart when they get too close.

These pairwise interactions can be combined to build up a simple mathematical model, the Lennard-Jones potential [Jon24], which is given by:

$$U_{LJ}(r) = 4\epsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right] \tag{2.1}$$

In this model, r is the distance between two particles, σ is the distance at which the potential cancels out, and ϵ corresponds to the potential well, or the minimum reached potential. The Figure 2.1 provides an example of the Lennard-Jones potential which indicates the values of σ and ϵ. For diminishing intermolecular distances, the potential increases very fast, while for increasing distances, the potential converges to zero. The potential well is located at a distance of $\sqrt[6]{2} \cdot \sigma$.

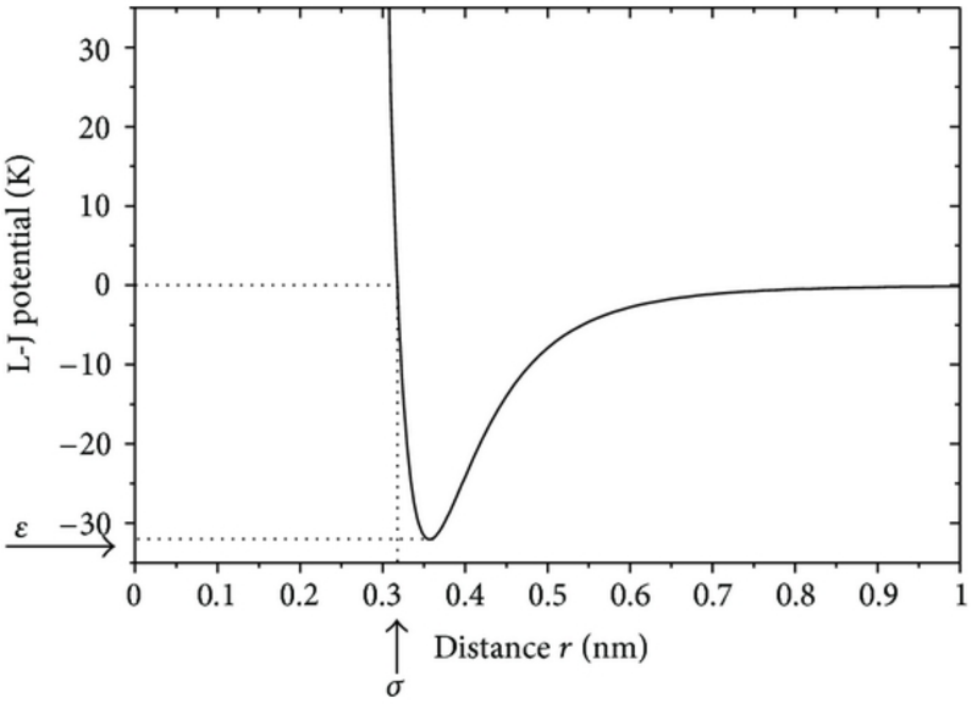


Figure 2.1.: Example of the Lennard-Jones potential of the carbon-hydrogen interaction.
Source: [KML13]

Nevertheless, this potential cannot be directly applied to any given two particles, since the variables σ and ϵ are specific to each chemical entity. As a matter of fact, if two different particles are considered, combining rules are needed to compute these parameters from the particle data. A simple method to achieve this is given by the Lorentz-Berthelot rules [Lor81][Ber98]. According to this rule, σ can be computed by an arithmetic mean and ϵ by a geometric mean.

$$\sigma = \frac{\sigma_1 + \sigma_2}{2} \tag{2.2}$$

$$\epsilon = \sqrt{\epsilon_1 \epsilon_2} \tag{2.3}$$

Here, σ_1 and ϵ_1 (resp. σ_2 and ϵ_2) correspond to the characteristics of the first (resp. second) particle.

The effective force acting on each particle can be computed with the gradient of the potential. Therefore, the Lennard-Jones interaction induces the following force with $\vec{r}$ being the vector between both particles and $|\vec{r}|$ its euclidean norm:

$$F_{LJ}(\vec{r}) = \nabla_r U_{LJ}(r) \tag{2.4}$$

$$F_{LJ}(\vec{r}) = \frac{24\epsilon}{|\vec{r}|^2} \left[\left(\frac{\sigma}{|\vec{r}|} \right)^6 - 2 \left(\frac{\sigma}{|\vec{r}|} \right)^{12} \right] \vec{r} \tag{2.5}$$

The force has the same direction as the distance vector for $|\vec{r}| > \sqrt[6]{2} \cdot \sigma$, therefore the particles attract each other. On the other hand, for $|\vec{r}| < \sqrt[6]{2} \cdot \sigma$, the force is repulsive.

2.1.2. Electrostatic potential

Electrostatic potentials between two charged particles lead to another very important type of intermolecular interactions. The charge distribution in an atom is neglected, and atoms are represented as point charges. Then, the force between two atoms with respective charges q_1 and q_2 can be approximated by Coulomb's law [Cou85]:

$$F_C(\vec{r}) = k_C \frac{q_1 q_2}{|\vec{r}|^2} \cdot \frac{\vec{r}}{|\vec{r}|} \tag{2.6}$$

In a vacuum, the value of Coulomb's constant is $k_C = 8.99 \cdot 10^9 N m^2 C^{-2}$. If the assumption of a vacuum does not hold, k_C depends on the permittivity ϵ of the considered medium by the relation $k_C = \frac{1}{4\pi\epsilon}$

A particle can be either charged itself, e.g. in the case of an ion, or it can possess partial charges in its structure. One kind of partial charge has already been mentioned in connection with the London dispersion in the Subsection 2.1.1. Nevertheless, they occur inside of an atom, and as atoms are considered as point charges, they will be neglected in the context of electrostatic potentials.

In the case of molecules composed of elements with different electronegativities, polarised covalent bonds arise. The atom with the higher electronegativity has a stronger attraction with the bonding electrons and gets a negative partial charge whereas the atom with the lower electronegativity gets a positive partial charge. One can compute the partial charge S_X between two atoms X and Y if one knows the main group number G_X, the number N_X of unbounded electrons, the number of electrons B_X participating in the bond and the values of the electronegativities χ_X and χ_Y [Cla04].

$$S_X = G_X - N_X - B_X \cdot \frac{\chi_X}{\chi_X + \chi_Y} \tag{2.7}$$

2.1.3. Harmonic Potential

A very basic simulation of large molecules such as membrane proteins could be a 2-dimensional mesh of elastically bounded atoms. Each constituting atom is linked to its direct mesh neighbours and its diagonal neighbours, as depicts Figure 2.2. Therefore, for each particle, a total of eight interactions account for the calculation of the total exerted force on a molecule. It can be combined with a repulsive Lennard-Jones potential, which only takes into account interactions for distances inferior to $\sqrt[6]{2}\,\sigma$ to prevent collisions of the membrane atoms.

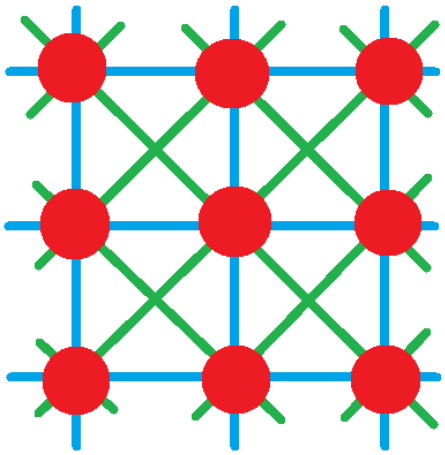


Figure 2.2.: Example of the set-up for a membrane. Blue bonds represent interactions between direct neighbours and green bonds represent interactions between diagonal neighbours.

The bond between two adjacent atoms can then be represented by a spring with stiffness k and rest length r_0[Hoo78]]. The common force in a spring is given by:

$$F(\vec{r}) = k \cdot (\vec{r} - \vec{r_0}) \tag{2.8}$$

Consequently, for given mesh length σ_0 and stiffness k, the membrane forces between a particle and a direct neighbour is given by:

$$F(\vec{r}) = k \cdot (|\vec{r}|_2 - \vec{\sigma_0}) \tag{2.9}$$

and for a diagonal neighbour by:

$$F(\vec{r}) = k \cdot \left(|\vec{r}|_2 - \sqrt{2}\vec{\sigma_0}\right) \tag{2.10}$$

2.1.4. Energy conservation

In an isolated system without any mass or energy transfer to and from the outside, the total amount of energy remains constant over time. If only particle dynamics are considered, and therefore chemical reactions are not simulated, the total energy of the system corresponds to the thermal energy which is equal to the sum of the kinetic and rotational energies of all molecules.

$$E_{therm} = \frac{f \cdot N}{2} k_B T = E_{kin} + E_{rot} \tag{2.11}$$

$$E_{kin} = \sum_{i=1}^{N} \frac{m_i |\vec{v_i}|_2^2}{2} \tag{2.12}$$

$$E_{rot} = \sum_{i=1}^{N} \frac{1}{2} \vec{\omega_i}^T I_i \vec{\omega_i} \tag{2.13}$$

The parameters are:

- f is the number of degrees of freedom of the system particles
- N is the total number of particles
- T is the system temperature
- $\vec{v}_i$ is the velocity of a particle i
- m_i is the mass of a particle i
- $\vec{\omega}_i$ is the angular velocity of a particle i
- I_i is the inertia tensor of a particle i
- $k_B = 1.381 \cdot 10^{-23} J \cdot K^{-1}$ is the Boltzmann constant

To ensure the physical correctness of the simulation regarding energy conservation, the actual total mechanical energy of the system shall be regularly computed, since it might vary slightly due to numerical errors, and the velocities of the particles shall consequently be adjusted so that the mechanical energy matches the set thermal energy of the system. The factor β to scale the velocities and angular velocities is computed as following [GKZ07a].

$$\beta = \sqrt{\frac{E_{init}}{E_{kin} + E_{rot}}} \tag{2.14}$$

These recurrent velocity adjustments can also be used to simulate cooling or heating processes by a slow increase or decrease of the temperature in the computation of the total thermal energy.

2.1.5. Brownian motion

In 1827, Robert Brown observed spontaneous and random movements of particles in fluids, without any exterior force effect [Bro28]. These movements change over time without any explicable pattern and the excitement strengthens with an increase of the temperature. In molecular dynamics, Brownian motion can be simulated by applying a Maxwell-Boltzmann distribution to the velocity of the particle at the beginning of the simulation [GKZ07b]. The velocity distribution for one component v_i is given by:

$$f(v_i) = \left(\frac{m}{2\pi k_B T}\right)^{\frac{3}{2}} e^{-\frac{m v_i^2}{2 k_B T}} \tag{2.15}$$

$$f(v_i) = \left(\frac{b}{\pi}\right)^{\frac{3}{2}} e^{-b v_i^2} \tag{2.16}$$

The Brownian factor b can be arbitrarily chosen if a thermostat is used, since the velocity will be scaled according to the system temperature.

2.2. Particle dynamics

2.2.1. Newton's third law of motion

According to Newton's third law of motion, for any force acting from one body to another, there is an equal reaction in the exact opposite direction acting on the second body [New66].

Therefore, it is sufficient to consider each pair only once for the computation of pairwise interactions in molecular dynamics, since the computed force can be added to the first particle and be subtracted from the second particle.

$$F_{ij} = -F_{ji} \tag{2.17}$$

2.2.2. Rigid-body moves

If the system is based on individual atoms, which can be considered as point masses, it is sufficient to consider the forces acting on the particle. However, to simulate molecules, which can be approximated as a rigid body of several point masses that represent each one constitutive atom of the molecule, rotational movements cannot be neglected. Every force acting on a molecule at a non-zero distance from its mass centre induces a torque and consequently a rotation. The position x_m of the mass centre is computed as the sum of the weighted position x_i of each site with mass m_i.

$$x_m = \sum_{i=1}^{N_{particles}} m_i x_i \tag{2.18}$$

The torque τ is expressed as the sum of crossproducts of the force F_i and the distance from the mass centre $r_i = x_i - x_m$.

$$\tau = \sum_{i=1}^{N_{particles}} r_i \times F_i \tag{2.19}$$

Moreover, the torque is the derivative of the angular momentum L, which is the linear transformation of the angular velocity ω by the inertia tensor I. Thus, the angular velocity of a molecule can be computed from the torque in two steps.

$$L = \int \tau \, dt \qquad \omega = I^{-1} L \tag{2.20}$$

The inertia tensor

The inertia tensor or moment of inertia describes the factor between the torque and the angular acceleration of a rigid body, thus it is similar to the mass in the relation between force and acceleration ($F = ma$). It depends on the mass distribution of the body and is, in general, represented by a symmetric 3×3 matrix. For a body of N point masses, its entries can be easily determined from the distance $[x_i, y_i, z_i]^T$ of each site to the mass centre.

$$
\begin{aligned}
I_{xx} &= \sum_{i=1}^{N} m_i(y_i^2 + z_i^2) \qquad & I_{xy} = I_{yx} &= -\sum_{i=1}^{N} m_i x_i y_i \\
I_{yy} &= \sum_{i=1}^{N} m_i(x_i^2 + z_i^2) \qquad & I_{xz} = I_{zx} &= -\sum_{i=1}^{N} m_i x_i z_i \\
I_{xx} &= \sum_{i=1}^{N} m_i(x_i^2 + y_i^2) \qquad & I_{yz} = I_{zy} &= -\sum_{i=1}^{N} m_i y_i z_i
\end{aligned}
$$

The formula for the angular velocity in [2.19] requires the inverse of the inertia tensor. For a 3×3, it can be be directly calculated.

$$
I^{-1} = \frac{1}{\det(I)} \begin{pmatrix} I_{yy}I_{zz} - 2I_{yz} & I_{xz}I_{yz} - I_{zz}I_{xy} & I_{xy}I_{yz} - I_{xz}I_{yy} \\ I_{yz}I_{xz} - I_{xy}I_{zz} & I_{xx}I_{zz} - 2I_{xz} & I_{xz}I_{xy} - I_{xx}I_{yz} \\ I_{xy}I_{yz} - I_{xz}I_{yy} & I_{xz}I_{xy} - I_{xx}I_{yz} & I_{xx}I_{yy} - 2I_{xy} \end{pmatrix} \tag{2.21}
$$

Principal axis theorem

In actual fact, the inertia tensor is not always invertible. For instance, H_2 is composed of two sites and has only two rotational degrees of freedom, since a torque in the direction of the axis of the molecule would not affect its rotational behaviour. As a matter of fact, any rigid body built of aligned particles will have an inertia tensor of rank 2, which is therefore not invertible.
This issue can be solved by a transformation of the torque and the angular momentum into the body frame, in which the coordinate system corresponds to the principal axis of the body. A torque along a principal axis will only affect rotations along the same axis, hence the inertia tensor in the body frame is a diagonal matrix [Syl52]. The inverse is then also a diagonal matrix and its entries are equal to the inverses of all non-zero elements of the inertia tensor. The columns of the rotation matrix from the current frame to the body frame correspond to the normalised eigenvectors of the current inertia tensor and the entries of the diagonal inertia tensor correspond to its eigenvalues. For a symmetric 3×3 matrix, the eigenvalues and eigenvectors are always real and can be computed by an eigenvalue

algorithm, which is sketched below [Smi61].

Algorithm 1: The eigenvalue algorithm [Smi61]

Input: normal 3×3 matrix A
Output: Eigenvalues $\lambda_1, \lambda_2, \lambda_3$

Function `getEigenvalues(`A`):`
  if A *is diagonal* **then**
    return $diag(A)$
  else
    $m \leftarrow trace(A)/3$
    $p_1 \leftarrow A_{12}A_{21} + A_{13}A_{31} + A_{23}A_{32}$
    $p_2 \leftarrow (A_{11} - m)^2 + (A_{22} - m)^2 + (A_{33} - m)^2 + 2p_1$
    $p \leftarrow \sqrt{p_2/6}$
    $B \leftarrow p^{-1}(A - mI)$
    $r = det(B)/2$
    `// Due to numerical errors, r may not be in the interval [-1, 1]`
    if $r < -1$ **then**
      $\phi \leftarrow \pi/3$
    else if $r > 1$ **then**
      $\phi \leftarrow 0$
    else
      $\phi \leftarrow \arccos(r)/3$
    end
    $\lambda_1 \leftarrow m + 2p\cos(\phi)$
    $\lambda_2 \leftarrow m + 2p\cos(\phi + 2\pi/3)$
    $\lambda_3 \leftarrow 3m - \lambda_1 - \lambda_2$
    return $\lambda_1, \lambda_2, \lambda_3$

An eigenvector associated to the eigenvalue λ is the crossproduct of two linearly independent columns of $(A - \lambda I)$. If no such columns exist, any perpendicular vector to the column space can be taken as eigenvector to λ.

2.2.3. Quaternion representation of rotations

In addition to the position, the orientation of a particle has to be considered in order to correctly handle molecule rotations. There exist three widespread methods to represent rotations of rigid bodies: rotation matrices, the axis-angle representation and quaternions. Firstly, rotation matrices allow a very straightforward computation of coordinates from an initial orientation. However, they require storage for nine entries for a three-dimensional molecule, which would require large amounts of memory for simulations with a large amount of molecules.
Secondly, the angle-axis is a very intuitive representation of a rotation and only needs four values, divided in three for the axis and one for the rotated angle. However, computationally intensive trigonometric functions are needed to retrieve a rotated point's coordinates and

the risk of a gimbal lock comes up. A gimbal lock appears if two rotation axes reach a parallel orientation and the molecule loses one rotational degree of freedom.
Lastly, unit quaternions (or versors) provide a convenient and numerically stable representation of orientations with only four numbers. A quaternion $q = a + bi + cj + dk$ is composed of a real and three imaginary parts, where the complex units i, j, k represent a set of base units of a three-dimensional vector space. If $(\vec{u}, \alpha)$ is the axis-angle representation of a rotation with an angle α and a unitary rotation axis $\vec{u}$, the rotation quaternion can be expressed by:

$$q = \cos\frac{\alpha}{2} + u_1 \sin\frac{\alpha}{2} i + u_2 \sin\frac{\alpha}{2} j + u_3 \sin\frac{\alpha}{2} k \tag{2.22}$$

Let $\vec{p}$ be the position of a point P and r the quaternion given by $r = 0 + p_1 i + p_2 j + p_3 k$ The rotation of $\vec{p}$ by $(\vec{u}, \alpha)$ is then equal to $r' = qr\bar{q}$, where $\bar{q}$ is the complex conjugate of q.

Thus, the rotation of any vector $\vec{p}$ by a unit quaternion $q = a + bi + cj + dk$ is computed explicitly by:

$$\vec{p}' = R\vec{p} \tag{2.23}$$

$$\begin{pmatrix} p'_1 \\ p'_2 \\ p'_3 \end{pmatrix} = \begin{pmatrix} a^2 + b^2 - c^2 - d^2 & 2bc - 2ad & 2ac + 2bd \\ 2ad + 2bc & a^2 - b^2 + c^2 - d^2 & 2cd - 2ab \\ 2bd - 2ac & 2ab + 2cd & a^2 - b^2 - c^2 + d^2 \end{pmatrix} \begin{pmatrix} p_1 \\ p_2 \\ p_3 \end{pmatrix} \tag{2.24}$$

The inverse rotation is given by $r = q^{-1} r' q$ or, using the matrix-vector notation, $\vec{p} = R^T \vec{p}\,'$.

If an angular velocity ω is applied to a body oriented by the quaternion q, the rate of change of its alignment is given by [Eck14]:

$$\frac{\partial q}{\partial t} = Q\hat{\omega} \tag{2.25}$$

with $\hat{\omega} = \begin{pmatrix} 0 \\ \omega_1 \\ \omega_2 \\ \omega_3 \end{pmatrix}$ and $Q = \begin{pmatrix} a & -b & -c & -d \\ b & a & -d & c \\ c & d & a & -b \\ d & -c & b & a \end{pmatrix}$. Again, we have $q = a + bi + cj + dk$.

2.2.4. Discrete time integration

Newton's second law of motion [New66] describes forces as the product between the mass and the acceleration. Thus, the acceleration, its first anti-derivative, the velocity, and its second anti-derivative, the position, can be deduced from the forces acting on a particle.
Taylor expansions are used to represent the discrete time integration for a single particle at time t with a time step Δt. First, we will express the velocity in the next time step in function of the force at $t + \Delta t$, the force at t and the velocity at t using two Taylor

expansions of second order at $t + \Delta t$ and t.

$$v(t + \Delta t) = v(t) + \Delta t \frac{\partial}{\partial t} v(t) + \mathcal{O}(\Delta t)^2 \tag{2.26}$$

$$v(t) = v(t + \Delta t) - \Delta t \frac{\partial}{\partial t} v(t + \Delta t) + \mathcal{O}(\Delta t)^2 \tag{2.27}$$

$$v(t + \Delta t) - v(t) = v(t) - v(t + \Delta t) + \Delta t \left(\frac{\partial}{\partial t} v(t) + \frac{\partial}{\partial t} v(t + \Delta t) \right) + \mathcal{O}(\Delta t)^2 \tag{2.28}$$

$$v(t + \Delta t) = v(t) + \frac{\Delta t}{2} \left(\frac{\partial}{\partial t} v(t) + \frac{\partial}{\partial t} v(t + \Delta t) \right) + \mathcal{O}(\Delta t)^2 \tag{2.29}$$

Since $F(t) = m \cdot a(t) = m \cdot \frac{\partial}{\partial t} v(t)$, we get:

$$v(t + \Delta t) = v(t) + \frac{\Delta t}{2m} (F(t) + F(t + \Delta t)) + \mathcal{O}(\Delta t)^2 \tag{2.30}$$

Moreover, we can compute the position with one Taylor expansion of third order at $t + \Delta t$ and the law $v(t) = \frac{\partial x(t)}{\partial t}$.

$$x(t + \Delta t) = x(t) + \Delta t \frac{\partial}{\partial t} x(t) + \frac{\Delta t^2}{2} \frac{\partial^2}{\partial^2 t} x(t) + \mathcal{O}(\Delta t)^3 \tag{2.31}$$

$$x(t + \Delta t) = x(t) + \Delta t v(t) + \frac{\Delta t^2}{2m} F(t) + \mathcal{O}(\Delta t)^3 \tag{2.32}$$

$$\tag{2.33}$$

This numerical method to compute the position and the velocity of particles is called Verlet-Störmer [Ver67] and is preferred to other integration schemes such as central differences because of its time-reversibility and symplecticity, or its ability to integrate a Hamiltonian system [GKZ07c].

In rigid-body dynamics, the angular momentum j can be computed from the torque τ similarly to the velocity and the attitude q by a simple forward Euler scheme.

$$j(t + \Delta t) = j(t) + \frac{\Delta t}{2} (\tau(t) + \tau(t + \Delta t)) \tag{2.34}$$

$$q(t + \Delta t) = q(t) + \Delta t \frac{\partial q}{\partial t} \tag{2.35}$$

2.3. Algorithms for N-body simulations

The computation of Lennard-Jones and electrostatic interactions requires the consideration of every possible particle pair in the system. However, with the assumption that forces can be neglected if two particles are distant enough, it is possible to considerably reduce the computational effort as the number of relevant particle pairs diminishes.
As already discussed in Equation 2.5, the Lennard-Jones potential comprises a short-range repulsive force, which decreases by a factor r^{-12}, and a long-range attractive force, which decreases by a factor r^{-6} for an increasing distance r between two particles. For instance, at a

given σ, the Lennard-Jones force applied at a distance $r = 5\sigma$ is only $F_{r=5\sigma} = -1.28 \cdot 10^{-4} F_{r=\sigma}$ of the resulting force at a distance $r = \sigma$ and at $r = 10\sigma$ we have: $F_{r=10\sigma} = -2.00 \cdot 10^{-6} F_{r=\sigma}$. Thus, the Lennard-Jones force can be neglected for two particles with a relative distance r exceeding a defined cut-off radius r_c.
Nevertheless, this statement does not hold for Coulomb forces, which only decrease to the second power of the distance. In fact, a cut-off radius can only be applied to the electrostatic potential for uncharged, polarised particles. In this case, the Coulomb interaction determines the orientation of adjacent molecules, and long-range forces between two particles vanish since the sum of all partial charges in a molecule is zero.

2.3.1. Direct sum

The most intuitive way to apply pairwise interactions is to iterate twice over all particles in the system such that every particle pair is only considered once according to Newton's Third Law. An outline of such an algorithm is given in Listing 2.1.

```
1 for(i=0; i<NumParticles; ++i){
2   for(j=i+1; j<NumParticles; ++j){
3     //check cut-off radius
4     if (distance(Particle[i], Particle[j]) <= r_cutoff){
5       ApplyPairwiseInteraction(Particle[i], Particle[j]);
6     }
7   }
8 }
```

Listing 2.1: Example of a direct sum iteration for pairwise interactions using Newton's Third Law

The complexity of this method is $\mathcal{O}(N^2)$, which leads to very low performances for high numbers of particles.

2.3.2. Linked cells

The simulation domain is subdivided in equally sized cells which contain the system particles such that the edge length of the cells is equal to the cut-off radius of the pairwise interaction. In this case, a particle only interacts with the particles located in the same cell and in the adjacent cells. In two dimensions, nine neighbouring cells have to be considered and in three dimensions, a cube of 27 cells contain relevant particles. Each cell contains $N_{particles}/N_{cells}$, and if the particle density is considered constant in the domain and if the number of cells is constant, the complexity of this method is given by $\mathcal{O}(N)$. As depicted in Figure 2.3, it is possible to use Newton's third law and only consider half of the adjacent cells. For instance, for a particle in the red cell, it is sufficient to compute the interactions with the particles in the red and in the blue cells. Indeed, the interactions with the particles in the green cells are computed when a green cell is considered in the centre of the traversal cube.
To better approximate the geometry of the cut-off sphere, it is possible to reduce the edge length to any fraction of r_{cutoff}. This increases the number of cells per particle, but reduces the number of unused particles in the neighbouring cells and the consequent number of superfluous cut-off radius checks.

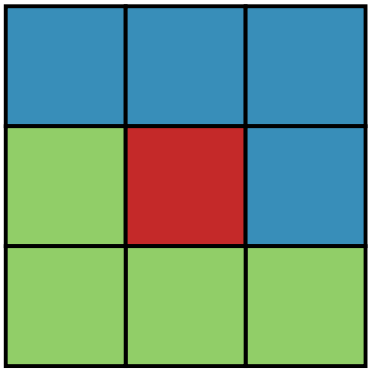


Figure 2.3.: Example of the relevant linked cells for a particle located in the central cell

2.3.3. Verlet lists

Loup Verlet [Ver67] proposed in 1967 to associate to each particle a neighbours list containing references to all particles located within an arbitrary cut-off radius around the particle. Therefore, for a given molecule, all relevant particles for a pairwise interaction can be easily accessed through the neighbour's list, and the complexity reduces to $\mathcal{O}(N)$. However, at each iteration, particles move and might leave the region covered by the neighbour's list or other particles might enter the region without being added to the neighbour's list. To prevent such issues, an update of all Verlet lists is required at each iteration, which again has a complexity of $\mathcal{O}(N^2)$. If each Verlet list contained, additionally to the particles strictly located inside the cut-off radius, particles in a given buffer zone from the neighbour's region, thus particles placed at a distance inferior to $r_{cutoff} + r_{buffer}$ from the original particle, the update rate can be reduced to the theoretical minimal time a particle would need to cross the buffer zone.

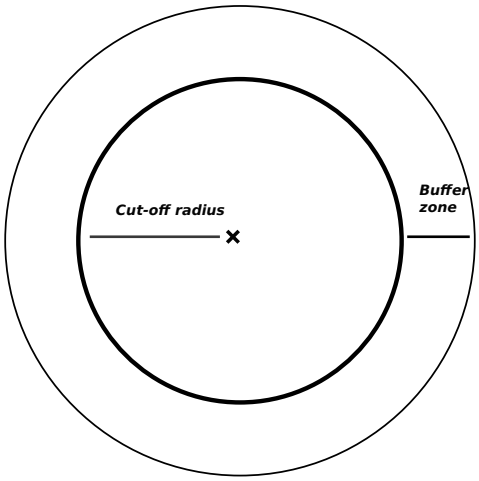


Figure 2.4.: Neighbour's region of a Verlet list with buffer zone

In a further extent, it is possible to combine Verlet neighbour's lists and linked cells. The particles are contained in a linked cells domain and every particle has a list of the references of all Verlet neighbours. As two spatially close particles can be easily accessed by the linked cells data layout, the complexity of the update of the Verlet neighbour's list reduces to $\mathcal{O}(N)$ as in the linked cells traversal. The particle pair iteration for the actual force computation is still performed through Verlet neighbour's list as they minimise the number of useless cut-off distance checks.

3. Related codes

3.1. ls1-mardyn

The molecular dynamics simulation program ls1-MarDyn has been jointly developed by the High Performance Computing Center Stuttgart (HLRS), the Laboratory of Engineering Thermodynamics at the University of Kaiserslautern (TLD), the Chair of Scientific Computing in Computer Science at the Technical University of Munich (SCCS) and Thermodynamics and Energy Technology at the University of Paderborn (ThEt). The name ls1-MarDyn stands for Large Systems 1: MoleculAR DYNamics [BBB+14].
It supports simulations on large domains with up to twenty trillions particles [ST18], is highly scalable and is adapted to run on several high-performance computing architectures. Therefore, ls1-mardyn is particularly suitable for simulations that require the representation of large numbers of particles such as nucleation in supersaturated vapours or fluid flow through nanoporous membrane particles. However, it is limited to rigid molecules, only supports simulations at a constant volume and cannot represent long-range electrostatic interactions with ions. In the scope of this thesis, the implemented simulation code MolSim with the AutoPas library is compared with ls1-mardyn with respect to performance and scalability.

3.2. AutoPas

AutoPas is a developing tool which provides an environment to run molecular dynamics simulations. It manages the memory, handles the particle traversals and includes tools for vectorisation and parallelisation. The user has only to specify the particle type and its properties such as pairwise functors to compute inter-molecular interactions.

3.2.1. The AutoPas interface

The instantiation of the AutoPas class acts as a particle container which handles the provided data. It is initialized as a cuboidal domain of given size with an underlying data structure, which can at the moment be either direct sums, Verlet neighbour's lists or linked cells, and a cut-off radius r_{cutoff}, beyond which pairwise interactions between particles are neglected. Additionally, it enables the addition and the deletion of halo particles, which are located in a r_{cutoff} -thick layer beyond the domain limits. It is equipped with a pairwise iterator, which iterates through all particles pairs according to the specified container as described in Section 2.3. and applies the provided functor to the particle pair. If the linked cells data layout is used, the container must be updated before the pairwise iterator can be applied if particles moved and changed their assigned cell. Finally, it includes a particle iterator and a region particle iterator which iterates over all particles in the domain or in a desired sub-domain.

3.2.2. Vectorisation in AutoPas

Pairwise functors in AutoPas can be executed on two different ways, the first using arrays of structures (AoS) and the other structure of arrays (SoA). AoS can be imagined as an array of particles which contain the values of the properties as illustrated in Figure 3.1. On the other hand, SoA contains all properties in separate arrays such that the values at the same index return the properties of the identical particle as can be seen in Figure 3.2. AoS functors take as input two instantiations of the particle class and directly adjust the forces and eventually other properties of the provided particles according to the interaction implemented in the functor. AoS functors take as input two instantiations of the particle class and directly adjust the forces and eventually other properties of the provided particles according to the interaction implemented in the functor.

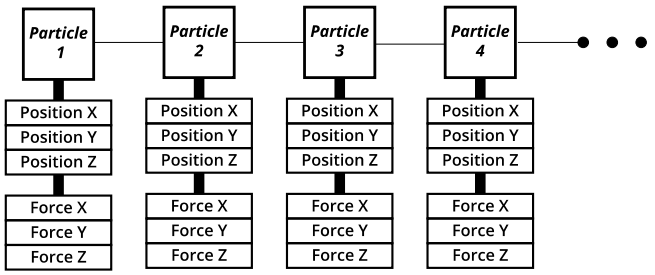


Figure 3.1.: Model of an array of structures (AoS) with particles containinga position and a force vector.

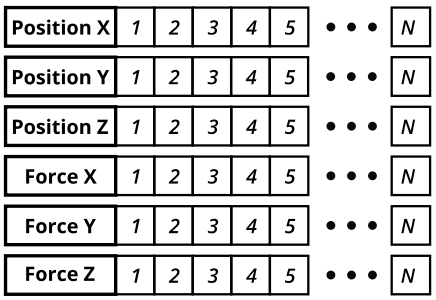


Figure 3.2.: Model of a structure of arrays (SoA) with six components and N particles.

SoA functors rely on arrays which contain each the values of one characteristic of all particles. For instance, a basic functor would require three arrays for the particle positions and three arrays for the components of the force vector, and the length of each array is equal to the number of considered particles. Therefore, the data is first extracted from the particle cells or Verlet neighbour's lists and then stored in the corresponding arrays. As all operations in the vector are now directly performed on arrays, it is possible to vectorise the force computations with SIMD for example.

3.2.3. Parallel cell traversals

The AutoPas class provides OpenMP parallelisation options for the simultaneous processing of linked cells. If Newton's third law is used, the computation of an interaction between two particles affects the force vectors of both particles, and as interactions happen between adjacent cells, race conditions can appear if two threads write in the same cell. Synchronisation cells, which will be handled last, are therefore needed to avoid such race conditions. Two different parallel cell traversals have so far been implemented in AutoPas: the sliced and the C08 traversal.

Sliced traversal

A domain is subdivided in a mesh of $n \times m \times l$ cells and a number of threads $n_{threads}$ is given. The dimension with the highest number of cells is taken and divided in $n_{threads}$ equally sized chunks of cells. Each chunk corresponds to a parallel region and is separated from the next chunk by one row of synchronisation cells. A two-dimensional example of a sliced traversal with three parallel threads is given in Figure 3.3. All cells connected by arrows of the same colour represent one parallel task that can be simultaneously processed. The hatched cells are synchronisation cells which are locked as long as the first column of the following thread has not been finished. For instance, the blue hatched cells are unlocked when the red thread begins with the second column. For a given cell mesh, the maximum number of threads is thus given by $n_{threads} = \max(n, m, l)/2$, since the smallest possible layout is composed of alternating columns of parallel regions and synchronisation cells.

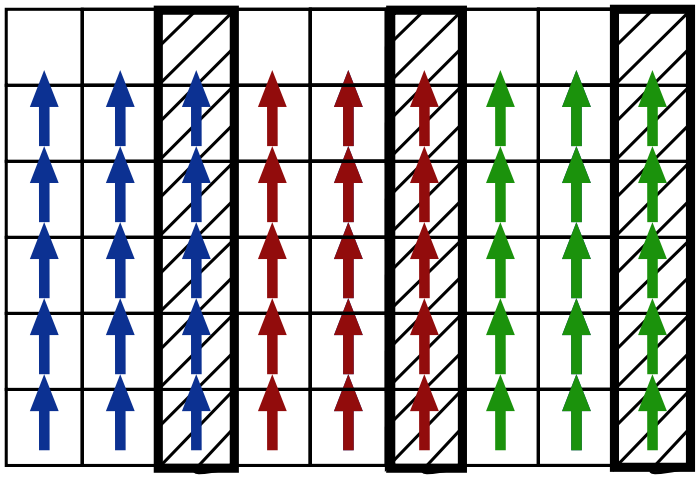


Figure 3.3.: Set up for a sliced traversal with three threads in blue, red and green with hatched synchronisation cells

If the cell edge length is smaller than the cut-off radius, the number of necessary adjacent cells for the force computation increases and the parallelisation gets more complex as more synchronisation cells are required.

C08 traversal

In the C08 traversal, the cell mesh is subdivided in cubes with an edge length of two cells. Starting from one cell, such as the yellow cell in Figure 3.4, all interactions between particles, represented by black arrows, in the corresponding cube are calculated. Each cube can be processed independently from the others, and after all forces in all cubes have been computed, the subdivision is repeated starting from another cell, such as the green cell. Thus, in 2D, four synchronisation steps and in 3D eight synchronisation steps are required. The maximum number of threads in the C08 traversal is given by the maximal number of cubes that can be built simultaneously in the domain, in 2D, it is thus possible to have $n_{threads,2D} = n \cdot m/4$ and in 3D there are at most $n_{threads,3D} = n \cdot m \cdot l/8$.

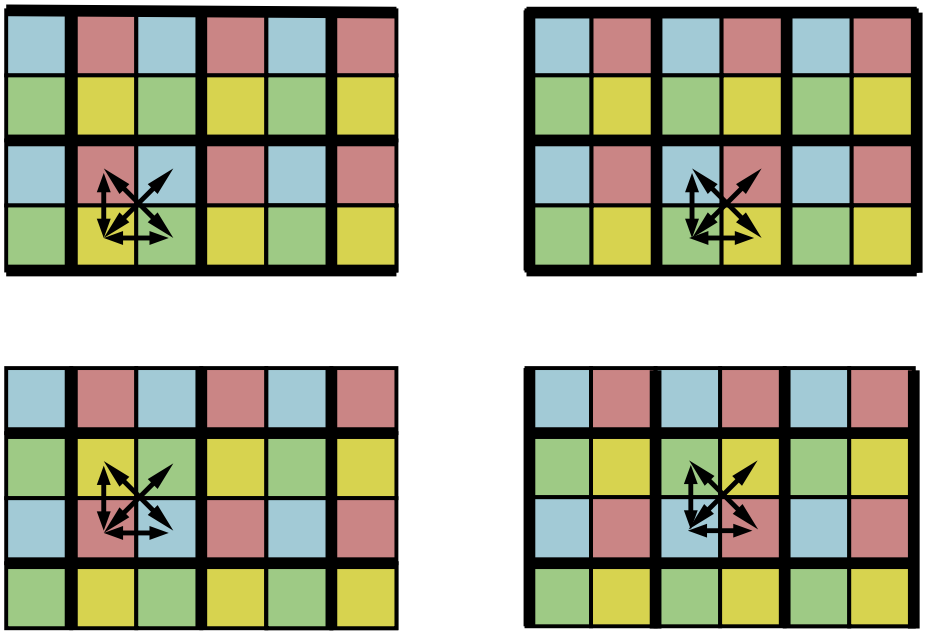


Figure 3.4.: Set up for a C08 traversal.

The task assignment follows a dynamic scheduling, thus after the execution of all computations in one cube, the thread gets the next available cube and no overhead is produced at the task barriers. This makes the C08 traversal particularly suitable for domains with unevenly distributed particles.

4. Implementation of the simulation tool

4.1. Stucture of the program

The first step in the execution of the simulation program is the input of the simulation settings, such as the time step, the kind of boundary conditions or the thermostat parameters which are provided in an XML-file. In a second step, particles are created by means of a particle generator, which can create arbitrary meshes of equally-distanced particles, spheres or single particles at defined positions. The chemical and physical characteristics such as the σ and ϵ values and the number and relative position of sites in a molecule and the desired particle generator are specified in an additional XML-file. The separation of the input data in two separate files allows for higher flexibility in the choice of the simulation. For instance, it is possible to easily change the simulation parameters on the same particle data, and on the other hand, run the same simulation on different input data. An overview of the detailed parameters of the input files can be found in the Appendix B.

```
1  /***************PREPROCESSING**************/
2  SetSimulationParameters(Settings.xml);
3  //initialise the particle container
4  auto* Container = new AutoPas<MoleculeMS , FullParticleCell<MoleculeMS>>();
5  Container->init(BoxMin, BoxMax , rcutoff, ContainerType );
6  GenerateParticles(ParticlesInput.xml, Container) ;
7  applyBrownianMotion(Container);
8  applyThermostat(Container);
9  size_t iteration=0;
10 /*************MAIN LOOP*****************/
11 for(double current_time = 0; current_time< end_time ; current_time +=
       time_step){
12   if(iteration==1) Timer.start();
13   if(Container->isContainerUpdateNeeded()) Container->updateContainer();
14   resetForces();
15   // apply single functors
16   for(auto particle_iterator = Container.begin(); particle_iterator.isValid();
         ++particleiterator){
17     for (int i = 0; i < NumSingleFunctors; ++i ){
18       SingleFunctors[i] (*particle_iterator);
19     }
20   }
21   applyBoundaryConditions(Container);
22   // apply pairwise functors
23   for(int j = 0; j < NumPairwiseFunctors; ++j ){
24     Container->iteratePairwise(PairwiseFunctors[i], DataLayout);
25   }
26   PartialTimeIntegrations(Container);
27   applyThermostat(Container);
28   OutputWriter(Container);
```

```
29    ++iteration();
30  }
31
32  /**********POSTPROCESSING***************/
33  double total_time = Timer.end();
34  WriteMUPS(total_time, NumParticles , iteration);
35  WriteFLOPS(Container, PairwiseFunctors);
```

Listing 4.1: Main structure of the simulation tool

Once the simulation parameters have been set, Brownian motion is applied to the particles and the resulting velocities are scaled by a thermostat. The main time loop is then started, and at the beginning of each iteration, the forces are reset to zero, the eventual linked cells container is updated, and single functors that only act on single particles, such as gravitation and electric fields or membrane forces are applied to all particles. Optional boundary conditions can then be applied to particles in proximity of the domain border. Thereafter, the desired pairwise iterator, which can use either direct sums, linked cells or Verlet's neighbour lists, is executed with a variable number of bi-molecular functors. After the computation of all occurring forces and consequent torques, partial time integrations as described in Subsection 2.2.4 determine velocities, positions, and, in the case of molecules, the angular velocity and the orientation. As the partial integrations consider each particle on its own, they can be easily parallelised without the danger of the incidence of race conditions. Finally, a thermostat can be applied to rescale the velocities, the particle data can be plotted to a VTK-file for the visualisation or to a CSV-file to get velocity profiles, and the iteration and time counters are incremented. A pseudo-code of the main program is given in Listing 4.1.

After the completion of the time iterations, the molecule updates per second (MUPS), thus the product of the total amount of particles and the number of iterations over the runtime of the main loop, and the floating point operations per second (FLOPS) are computed according to Equation 4.2 to provide data for the performance analysis.

$$MUPS = \frac{N_{particles} \cdot N_{iterations}}{t_{execution}} \tag{4.1}$$

$$FLOPS = \frac{N_{operations}}{t_{execution}} \tag{4.2}$$

4.2. Representation of the particle data

Each particle is instantiated by a molecule class, which comprises the specific physical characteristics of the particle. They include variables such as the current position, velocity and the experienced force of general particles, as well as the orientation, angular velocity and resulting torques in rotating molecules. Moreover, the values of the forces and torques from the previous time step are stored, since they are required by the partial integration scheme presented in the Subsection 2.2.4. In addition to these variable parameters, a unique index is assigned to the particle.
General properties of a particle type, such as the mass or the value of σ and ϵ, are not stored in the molecule class to avoid redundant information but in a separate particle type class, which associates all properties of a molecule type to a unique ID. Furthermore, the combined

ϵ_{mixed} are preliminarily computed with the Berthelot mixing rule of Equation 2.3 for all possible pairs of $\epsilon_{particle}$ to avoid the repeated execution of the expensive calculation of a square root in each call of the pairwise Lennard-Jones functor. The molecule class itself only stores the unique type ID to retrieve the element properties from the particle type class.
Finally, the configuration of a molecule is provided in a molecule type class. It contains the relative positions of the constituting atoms to the mass centre of the molecule as specified in the input XML-file. Each site is also associated to a type ID, so the specific properties of each constituting atom can be used. As described in Subsection 2.2.2, the molecule type class holds the inertia tensor of the molecule in the initial frame, its inverse if it is defined, its principal values in the body frame and the inverses of the non-zero principal values. These specifications can be accessed from the molecule class by a pointer to the corresponding instantiation of molecule type class.
The required memory space for one particle is resumed in Table 4.2. If a reasonable number of different particle and molecule types is used, the memory needed by these classes can be neglected compared to the particle data.

Element	used datatype	required memory [Byte]
Position	vector of 3 double	24
Velocity	vector of 3 double	24
Force from the current iteration	vector of 3 double	24
Force from the previous iteration	vector of 3 double	24
Orientation	quaternion of 4 double	32
Angular velocity	vector of 3 double	24
Torque from the current iteration	vector of 3 double	24
Torque from the previous iteration	vector of 3 double	24
Unique index	unsigned long integer	8
Particle Type	integer	4
Reference to the molecule type	pointer	8
	Total:	**220**

Table 4.2.: Memory requirements of one particle

4.2.1. Computation of the inverse inertia tensor

According to Subsection 2.2.2, the angular velocity can be computed on two different manners: either by a direct matrix-vector multiplication of the angular momentum by the inverse inertia tensor or a prior rotation of the molecule into the body frame, and then a component-wise division of the angular momentum vector by the respective principal values. The first method can only be applied to molecules with three rotational degrees of freedom, while the computation in the body frame can be applied to any kind of particles.
In both cases, the angular momentum is first rotated into the initial frame, as provided by the input file. The inertia tensor and its inverse are then only computed once per molecule type at the beginning of the execution. According to Equation 2.23, each rotation by a

quaternion needs 21 additions and 25 multiplications. The so calculated angular velocity has to be rotated back into the current orientation of the molecule. Therefore, he transformation of the system into a frame where an inverse inertia tensor can directly be applied requires 42 additions and 50 multiplications forth and back. On the other hand, it is also possible to get the inverse inertia tensor for the current layout of the particle. It is retrieved by applying the quaternion rotation to the inverse inertia tensor of the initial frame using in total 63 additions and 75 multiplications. In conclusion, the transformation of the angular momentum and velocity requires less computational steps than the transformation of the inverse inertia tensor.
The aim of the following paragraph is to compare the speed of the computation of the angular velocity in the body frame and in the initial frame for a water molecule with three rotational degrees of freedom as described in Subsection 5.6.1 . Both methods are tested in a serial execution with a cuboid of 10000 molecules in a linked-cells container with a Lennard-Jones functor for molecules and for ten time iteration steps. As the simulation in the body frame induces a 7% longer runtime than in the initial frame, the second is the preferred method for the computation of the angular velocity of a suitable molecule. This can be explained by the number of required transformations: the direct computation in the initial frame requires one matrix-vector multiplication, though two vector rotations, which correspond each to a matrix-vector multiplication, are needed to reach the body frame in addition to the cheap division of the components by the principal values of the inertia tensor. Thus, if the input file does not specify anything else, the program automatically choses the initial frame computation for a molecule with an invertible inertia tensor and the body frame for any other type of molecule.

4.3. Boundary conditions

4.3.1. Reflective

Some simulations require particles to stay in the domain at the same position near the boundary. For example, the simulation of a gravity-induced mixing process of two fluids with different densities on a plate needs a support to prevent the particles from "falling down". This can be achieved with a reflective boundary on the lower face of the domain.
The repulsive Lennard-Jones potential, or Pauli repulsion forces as stated in Subsection 2.1.1, describes the force acting on particles in the neighbourhood of the domain delimitation. A halo particle is generated at a symmetric position if a particle is located at less than $\frac{1}{2}\sqrt[6]{2}\,\sigma$ from the boundary, as shown in Figure 4.1. A particle that reaches the boundary is then reflected and stays in the simulation domain.

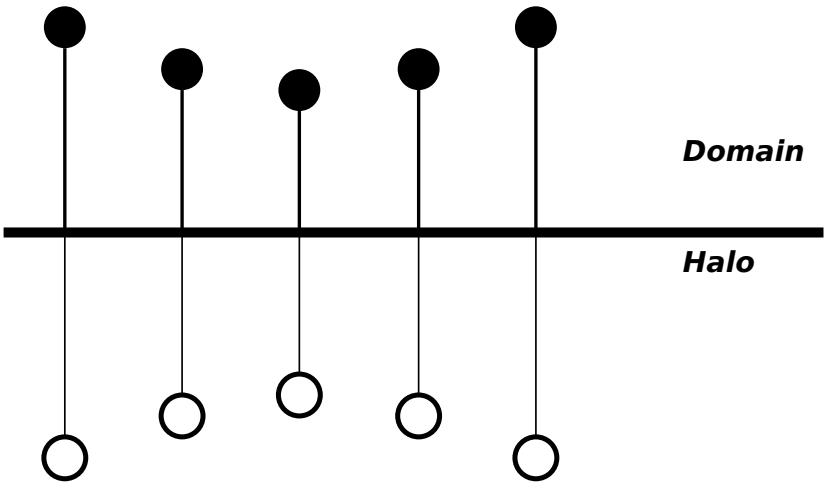


Figure 4.1.: Example of the repulsion of a particle at a reflective boundary condition

4.3.2. Periodic

To represent a behaviour on very large domains, it can be convenient to run simulations on a smaller domain and to extend it periodically. For instance, if one is interested in phenomena within a liquid, which only interacts with itself and is not influenced by exterior factors, any events or interactions at the domain boundary of the simulation would lead to undesired results. One solution to this issue is periodic boundaries, which let the simulation represent an infinite domain.
These boundary conditions enable particles next to the domain limit to interact with the particles located on the opposite side of the domain. Additionally, the area around an edge interacts with the area of the diagonally opposite edge and the same holds for the corners of the domain. Since particles only interact if their relative distances are below the given cut-off radius, it is sufficient to compute the forces transmitted through the domain boundaries for particles located within this cut-off radius from the boundary. Moreover, if a particle leaves the domain on one side, it reappears on the opposite side, i.e. two of its coordinates remain constant and the third is incremented or decremented by the domain length. To compute the transmitted forces, three different implementations have been tested. The first function, named *computePBC*, directly calculates the forces between particles on opposite sides after an iteration over all particles in proximity of the boundary. The second, *computeNeighboursPBC*, is similar to the first but only iterates over the particles at a distance less than r_{cutoff} along the axis spanning the periodic faces of the domain using the AutoPas region iterator. Lastly, the third function, called *applyPBC*, creates halo particles placed behind the boundary, as explained in Figure 4.2.

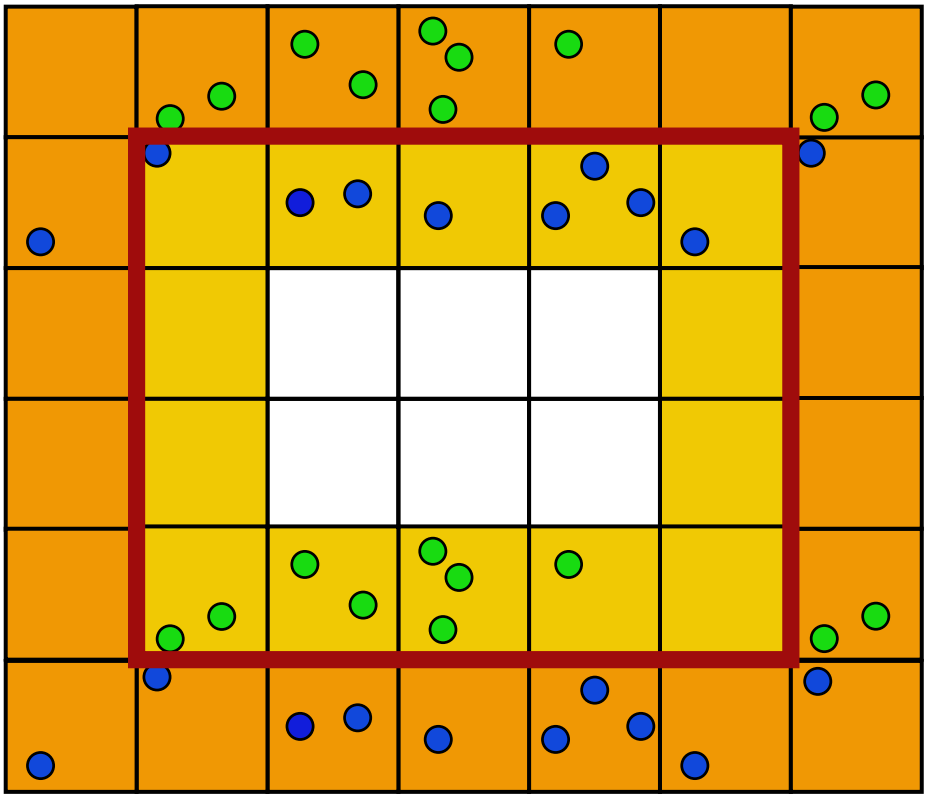


Figure 4.2.: Example of the creation of halo particles as implemented in *applyPBC*
The domain delimiter is the red line. The orange cells are the halo layer and the yellow cells are the boundary layer of the domain, which are considered when periodic interactions are computed. Every particle in a boundary cell is copied into the corresponding halo cell on the opposite side of the domain. For instance, the blue particles from the upper boundary layer are copied into the lower halo layer and vice-versa for the green particles. A special attention needs to be paid to corner cells, such as the lower-left boundary cell with two green particles, which is copied three times: to the top, to the right, and to the diagonally opposite cell

The last function requires the instantiation and the deletion of new particles at each iteration, but benefits from the linked cells traversals and its possible parallel execution, therefore, the complexity of this method is $\mathcal{O}(N_{particles})$. The second method only iterates over an approximately constant number of particles on the opposite side, so its complexity can be given by $\mathcal{O}(N_{particles})$, and it does not need the creation of halo particles. The first should yield the worst results since it acts as a direct sum and has, therefore, a complexity of $\mathcal{O}(N_{particles}^2)$.
To compare the three implementations, we consider a domain composed of 5832 cells (18 per dimension) and periodic boundary conditions in all three directions. Subsequently, 1740 cells are located next to a boundary and their particles are considered in the computation of the transmitted forces. This domain is then uniformly filled with a variable number of particles and the molecule update rates per second (MUPS) have been measured for each

scenario and both methods. The simulation is run on a single core with one OpenMP thread using the linked cells traversal. The results are shown in Figure 4.3.

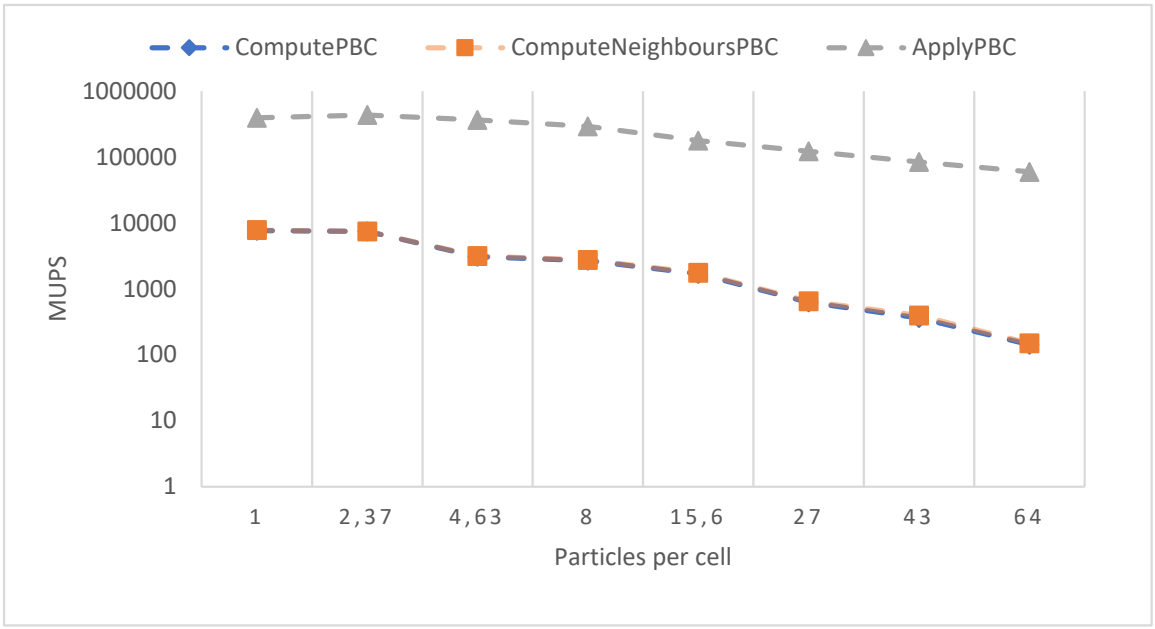


Figure 4.3.: Molecule updates per second for different particle densities using the three methods to apply periodic boundary conditions The simulation is run on a single thread with SoA arrays but without vectorisation in a domain of 5832 cells. Therefore, the total number of particles varies from $5.8 \cdot 10^3$ to $3.7 \cdot 10^5$

We can observe that for all methods, the performance decreases with the increase of the particle density. This is mainly explained by the fact that at a higher density, more particles lie within the cut-off radius of every single particle and more pairwise interactions have to be computed. However, the speed of *ApplyPBC* is much higher than the two other methods. *ComputeNeighboursPBC* is even slightly worse than *ComputePBC*, which reaches on average 96.7% the MUPS of *ComputeNeighboursPBC*. This is due to a loose implementation of the region iterator, which checks all particles in the domain whether they lie in the requested region or not. Thus, the expected computational speed of $\mathcal{O}(N)$ deteriorates to $\mathcal{O}(N^2)$. For an increasing particle density, the MUPS decrease faster using these two methods compared to *ApplyPBC*, with a relation of 2% for the lowest density and 0.2% for the highest density. This is mainly due to the $\mathcal{O}(N^2)$ complexity of the two first methods.
In conclusion, only the *ApplyPBC* method yields a satisfying performance. It has been further adapted to an OpenMP parallelisation. First, a list of all boundary particles is built in a serial process. The instantiation of halo particles is then achieved in various parallel threads by a static scheduling from the list elements.

5. Simulations results

5.1. Simulation hardware

5.1.1. SuperMUC

SuperMUC[1] is the name of the supercomputer at the Leibniz-Rechenzentrum (LRZ) in Garching. It builds on more than 155000 cores and has a peak performance of 3 Petaflop/s reached in June 2012 [NWA13]. It consists of two phases: the first comprises thin nodes and fast nodes while the second phase builds on Haswell nodes. In the context of this thesis, only thin nodes on SuperMUC phase 1 (SuperMUC-1-tn) are used to perform simulations and will be presented in further detail.
The node islands build on the IBM System x iDataPlex dx360M4 with Sandy Bridge-EP Xeon E5-2680 8C processors, which have a nominal frequency of 2.7 GHz. In total, there are 18 islands containing 512 nodes each. Every node has two processors with eight cores each. Therefore, all thin islands contain a total 147 456 cores. Intel hyper-threading allows simultaneous computation of two threads on one core. As the scope of this thesis is at node-level, maximal number of parallelisable tasks that can be simultaneously executed is 32.

5.1.2. CoolMUC

CoolMUC[2] is part of the Linux-cluster at the LRZ dedicated for smaller projects. The recent system CoolMUC-3 has been set up in September of 2017 and has been designed for highly-vectorisable and parallel tasks. It builds on the "Knights Landing" many-core processor Xeon Phi 7210-F with a nominal core frequency of 1.7 GHz. Each node contains 64 cores and is capable to handle up to 4 hyper-threads per core, therefore the maximal number of tasks executable in parallel is 256 per processor. CoolMUC-3 therefore offers a good opportunity to strong-scale the parallelisation of the simulation at node level.

5.1.3. Comparison

The SuperMUC-1-tn has a better core performance than the CoolMUC-3. For example, a simple, serial molecular dynamics simulation with approximately $2 \cdot 10^6$ molecules is about 3.5 faster on the SuperMUC-1-tn than on the CoolMUC-3. However, the Knights Landing architecture allows for a better scalability for multi-thread simulations at node-level. Figure 5.1 depicts the speedup $S(p)$, the ratio of the performance using p threads over the performance of a serial execution for different values of p. Notwithstanding, the ideal speedup $S(p) = p$ is by far not reached for high numbers of threads because of remaining

[1] https://www.lrz.de/services/compute/supermuc/systemdescription/
[2] https://www.lrz.de/services/compute/linux-cluster/overview/

serial code that cannot be parallelised and large overheads when parallel tasks are assigned. The performance drop on SuperMUC-1-tn between 16 and 32 threads is caused by hyper-threading, as each node only comprises two processors that can handle up to 16 threads each. All parallel threads perform similar computations and two hyper-threads require the same resources of the common processor. Thus, additional overhead is produced by the simultaneous use of one core for two tasks and the overall performance is affected.

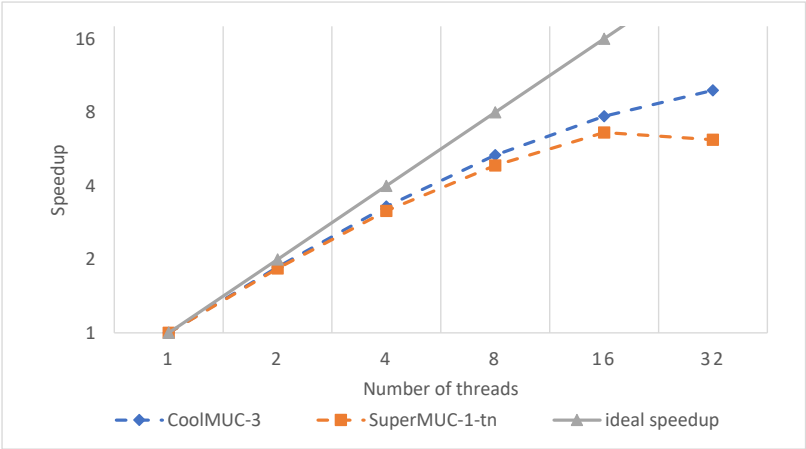


Figure 5.1.: Speedup on the CoolMUC-3 and the SuperMUC-1-tn Simulation of a cubic grid with $2.1 \cdot 10^6$ Lennard-Jones atoms and $r_C/\sigma = 5.0$. A native vectorisation and a C08 traversal are applied.

5.2. Comparison to the 2017 world record

5.2.1. Description of the world record scenario

The aim of this section is to compare the performances of the implemented program to the 2017 world record by ls1-MarDyn at a single node, the simulation tool before including the AutoPas library, referred to as MolSim, and the actual molecular dynamics tool using the functionalities of the AutoPas library.
For the sake of the best comparability, the same input configuration is used as in the 2017 single-node world record simulation [ST18]. It is based on a cubic domain uniformly filled with single-site particles configured in a cubic body-centred net. For the actual program, this layout is obtained by creating two cuboids with the same mesh length l that are placed with a relative offset of $l/2$ in all dimensions. In Figure 5.2 the construction of such a grid is drafted. Two cuboids, which are represented in red and in blue, are organised such that each particle lies in the middle of a cube formed by particles of the other cuboid. Figure 5.3 provides an example of such a cubic grid with 16000 particles. Three different scenarios are compared in which the domain is split in linked cells with a respective dimensionless edge length of $r_C/\sigma = 2.5$, $r_C/\sigma = 3.5$ and $r_C/\sigma = 5.0$. Accordingly, each linked cells contains 12, 33 or 98 particles. A sliced traversal as described in Subsection 3.2.3 iterates over all cells in the domain.

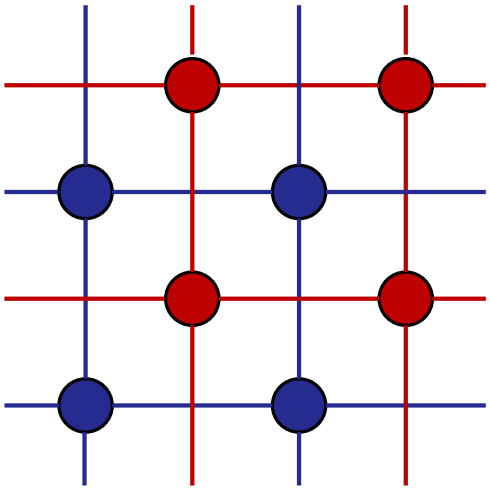


Figure 5.2.: Construction of a two-dimensional cubic grid.

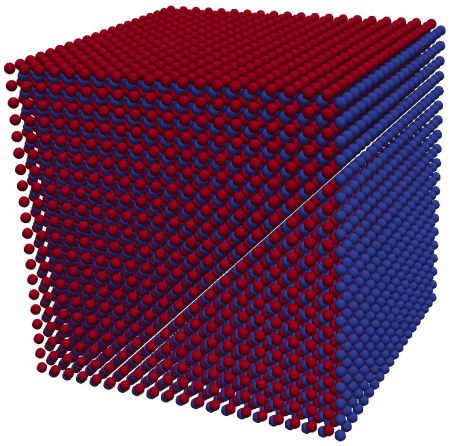


Figure 5.3.: Example of a 3D cubic grid configuration with 16 000 particles.

In the next experiments, cubic grids with $18.0 \cdot 10^6$ particles are generated in a domain equipped with a mesh of 113, 81 or 56 linked cells per dimension. To maintain the uniformity of the domain, a thermostat at very low temperatures (below 1 K) is applied to the system, such that the particle movement is minimized and particles remain in their initial cell throughout the run. The simulation is executed for eleven timesteps and the molecule update rate per second (MUPS) is measured from the start of the second iteration to the end of the last iteration and is computed according to Equation 4.2.

5.2.2. Comparison of the scalability

As the previous MolSim program does not support any vectorisation, the following graphs in Figure 5.4, Figure 5.5 and Figure 5.6 depict the performance of SoA based simulations without additional vectorisation for the three compared cut-off radii over the number of parallel OpenMP threads. However, MarDyn and MolSim compiled with the remaining autovectorisation while in AutoPas any kind of vectorisation has been disabled. Instead of a double precision, ls1-mardyn used a single precision in the world record, which considerably reduces the memory requirements and the computational effort of vectorised floating point operation. Moreover, the particle data is not stored as single particles but in the reduced memory mode (RMM), in which the SoA-structure is maintained throughout the program. An extraction of the particle data before the execution of the pairwise functor is therefore not required and the memory requirement of a particle is divided by 5.4 [ST18]. To get a better comparison, two simulations with ls1-mardyn are provided. The first, referred in the graph as MarDyn, is identical to the world record and stands as a reference. The second, MarDynRed, has a reduced performance as it is compiled without the RMM-mode and with a double precision, but it matches the parameters of AutoPas better.

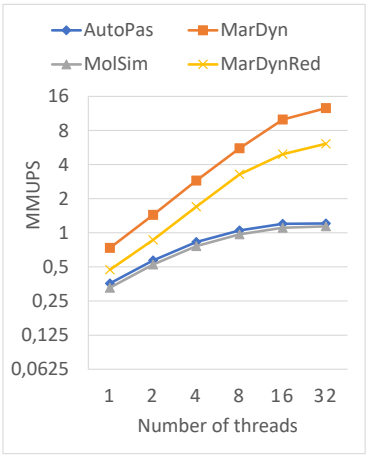


Figure 5.4.: $\frac{r_C}{\sigma} = 2.5$

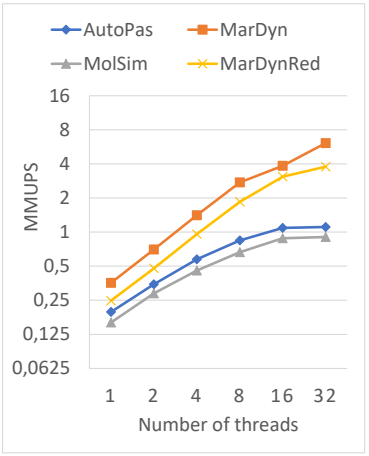


Figure 5.5.: $\frac{r_C}{\sigma} = 3.5$

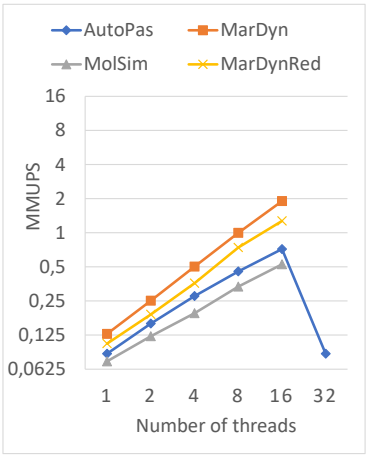


Figure 5.6.: $\frac{r_C}{\sigma} = 5.0$

Comparison between the four molecular dynamics tools in the SoA mode without vectorisation with a cubic grid of $18 \cdot 10^6$ *particles for different numbers of parallel OpenMP threads.*

In the last scenario, there are 56 cells per dimension, The sliced traversal is therefore only applicable up to 28 threads. ls1-mardyn and MolSim are not executable for a higher number of threads and AutoPas falls back to a serial execution as can be seen in Figure 5.6 for 32 threads.
The integration of the AutoPas library increased the performance in average by 8% for $r_C/\sigma = 2.5$, by 24% for $r_C/\sigma = 3.5$ and by 32% for$r_C/\sigma = 5.0$. The higher increase for large cells is due to a better implementation of the sliced traversal, since in the MolSim program, the synchronisation cells are not locked by the neighbouring thread as presented in 3.2.3, but are processed in a second step after the termination of the main parallel region. The higher number of required operations for larger cells explains the lower relative performance. Additionally, some some parts of the code which were previously serial, such as the application of periodic boundary conditions have been parallelised.
However, the performance of AutoPas remains well below the performance of the reduced MarDyn. Especially for higher parallelisation levels, the gap between both programs broadens. Indeed, the MUPS in MarDyn almost double for a duplication of the number of threads. On the other hand, AutoPas seems to converge towards a maximum performance for $r_C/\sigma = 2.5$ and $r_C/\sigma = 3.5$. Only cells with an edge length of $r_C/\sigma = 5.0$ induce a quadratic scaling as expected in the ideal growth though its gradient is smaller than in MarDyn.

5.2.3. Comparison of the vectorisation

In a second step, the impact of vectorisation on MarDyn and AutoPas is compared. The floating point operations per second (in GFLOPS) for the three vector instructions are given in the following graphs.

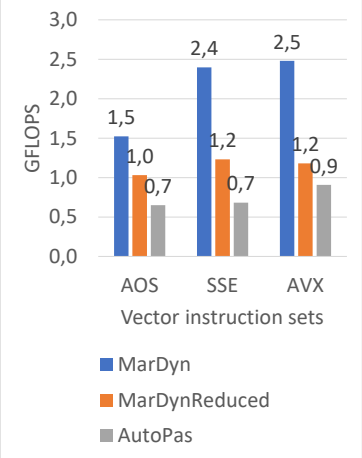


Figure 5.7.: $\frac{r_C}{\sigma} = 2.5$

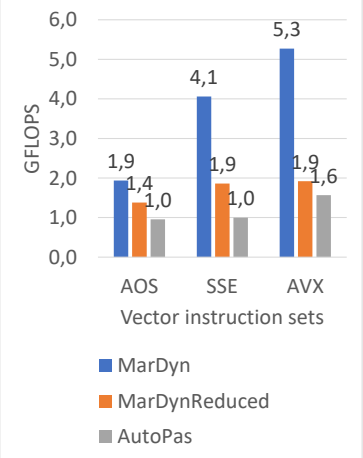


Figure 5.8.: $\frac{r_C}{\sigma} = 3.5$

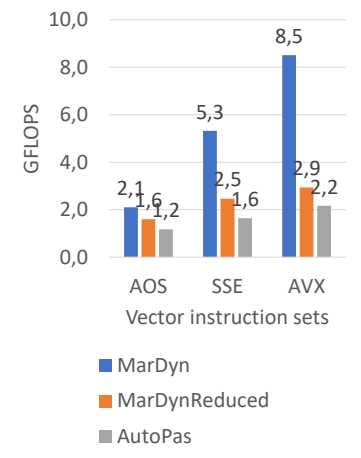


Figure 5.9.: $\frac{r_C}{\sigma} = 5.0$

Serial execution of the world record scenario of a cubic grid with $18 \cdot 10^6$ particles for different vector instruction sets and cell sizes.

The simulation with the reduced version MarDynRed is again better than AutoPas, and MarDyn as in the world record yields the best results with respect to the overall performance and the performance gain through vectorisation. The measured computational speeds are identical to the values in Figure 3(a) in [ST18]. The use of the AVX instruction set increases the computational speed by a factor 1.7 for $r_C/\sigma = 2.5$, a factor 2.7 for $r_C/\sigma = 3.5$ and a factor 4.0 for $r_C/\sigma = 5.0$. On the other hand, the reduced MarDyn and AutoPas experience a similar performance gain with an AVX vectorisation, which is however much smaller than in MarDyn as it only increases by a factor 1.2 for $r_C/\sigma = 2.5$, a factor 1.4 for $r_C/\sigma = 3.5$ and a factor 1.8 for $r_C/\sigma = 5.0$. This gain difference is caused by the single precision in MarDyn as the four-byte data type can be more efficiently vectorised than the less flexible eight-byte double type which is used in MarDynRed and in AutoPas. For all three simulation tools, the GFLOPS performance is better for large cut-off radii. It explains why the MUPS decrease between two distinct cut-off radii in Figure 5.4 is lower than expected. Indeed, the number of force computations increases by $\left(\frac{3.5}{2.5}\right)^3 = 2.7$ respectively $\left(\frac{5.0}{3.5}\right)^3 = 2.9$ even though the largest performance drop, experienced by MarDyn, yields $\frac{MUPS_{2.5}}{MUPS_{3.5}} = 2.1$ respectively $\frac{MUPS_{3.5}}{MUPS_{5.0}} = 2.7$
A more detailed analysis of the AutoPas vectorisation is provided in 5.3.

5.3. Vectorisation

5.3.1. Explanation of the vectorisation in AutoPas

As already presented in 3.2.2, pairwise interactions can either be computed using arrays of structures (AoS) or structures of arrays (SoA). The latter offers additionally the possibility of vectorising the force calculations. In general, vectorisation makes it possible to perform an operation between the elements of two vectors at the same time for all elements of the vectors. SIMD exploits data-level parallelism by subdividing data structures like arrays or vectors in registers holding multiple fields which depend on the compiler instructions. The

SSE vector instruction offers registers of 128-bits, which can then handle two double values, and AVX works with 256-bits registers which can contain four double values. Natively, SuperMUC-1-tn uses the AVX instruction sets.

5.3.2. Experiments with different vector instruction sets

To compare the performances of different vector instruction sets, a standard simulation environment is provided with a cubic grid of $2 - 1 \cdot 10^6$ single-site particles in a linked cells container which is processed by a 16-threads C08 cell traversal. Three different cut-off radii r_C and cell sizes are compared for a particle density, being $r_C/\sigma = 2.5$, $r_C/\sigma = 3.5$ and $r_C/\sigma = 5.0$. The explicit compilation with SSE, AVX and the native architecture, which should yield similar results than the AVX vector instruction, is compared to the SoA layout without any vectorisation and to the AoS layout. The performance is measured in millions of molecule updates per second (MMUPS) and the results are shown in Figure 5.10.

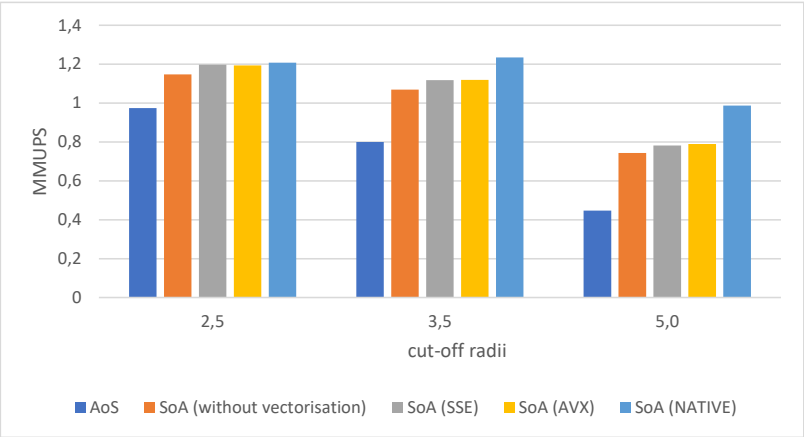


Figure 5.10.: Performance of the simulation with the AutoPas library for different vector instruction sets and cell sizes
Simulation run with $2.1 \cdot 10^6$ particles and a 16-thread C08 parallelisation

In all cases, the worst performance is reached with AoS. A large number of function calls is probably the reason for these bad results. Effectively, the AoS-functor is called at each calculation of a pairwise interaction. Inside the functor method, the particle class is called 16 times to fetch the current physical properties of the two particles such as the position as well as to add the computed force. On the other hand, the SoA-functor processes entire cells at once and is, therefore, only called once per cell and once per pair of adjacent cells in each iteration. The same holds true for the instantiations of the particle class, which are each called to load the SoA-arrays from the particle data before the execution of the SoA-functor and once after its execution to extract the forces from the arrays back to the particle data. As each cell is considered fourteen times in one iteration in three dimensions, if eight properties are loaded into the arrays and three extracted back, each particle is called 154 times independently of the chosen cell size. This constant number of calls is related to the decrease of the performance ratio of AoS over SoA for increasing cell sizes, which

goes from 85% to 60%, since the number of AoS-functor calls and consequent particle calls increases in contrast to the SoA-method. Another important factor which explains the superior performance of SoA is its cache-friendliness. All values of a given property are included in the same array and thus occupy the same cache line. On the other hand, in the AoS data layout, the properties are stored together with the instantiations of the particle class, which can be located anywhere in the memory. Therefore, the memory access is much cheaper if the SoA method is used instead of AoS.
SSE and AVX vector instructions yield very similar results which are, as expected, an improvement over SoA calculations without any vectorisation. The speed-up is higher for larger cells, reaching 4.4% for $r_C/\sigma = 2.5$, 4.6% for $r_C/\sigma = 3.5$ and 6.0% for$r_C/\sigma = 5.0$ and is due to larger vectorisation benefits if the underlying data arrays contain more elements. In addition to the AVX instruction set, a native vectorisation includes tuning options. They increase the efficiency of the AVX vectorisation but require large vectors to effectively apply the tuned vectorisation. Therefore, in small cells with few vectorisable computations, AVX and native vector instruction set yield similar performances while in larger cells, the performance could be increased by up to one fourth.

5.4. Scalability

5.4.1. Multithreading with OpenMP on the CoolMUC3

The Knight's Landing architecture in the CoolMUC3 makes running simulations with up to 256 parallel threads on one node possible. On this basis, the next section is dedicated to the analysis of the behaviour of the program at large numbers of parallel threads. To that effect, the worldrecord simulation of Section 5.2 is repeated on the CoolMUC3, however with a C08 traversal instead of the sliced traversal to avoid the fallback to a serial execution after reaching the maximum thread number. Next to the actual performance in Figure 5.11, the speedup of the simulation is represented in Figure 5.12 to illustrate the relative benefits of the parallelisation. The speedup $S(p)$ of a given configuration is the ratio of the performance with p parallel threads over the performance of a serial execution.

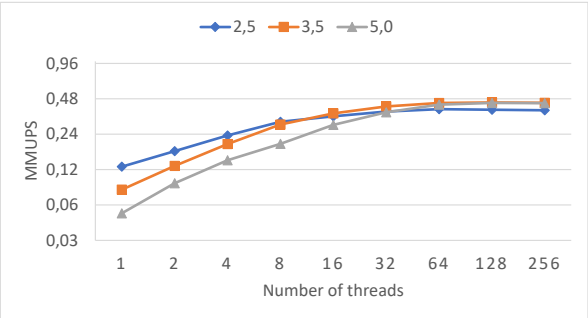


Figure 5.11.: Strongscaling on CoolMUC-3

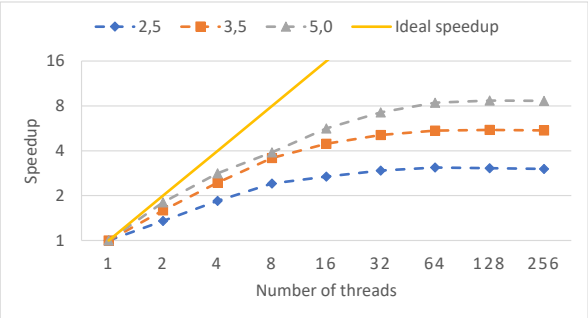


Figure 5.12.: Speedup on the CoolMUC-3

Performance of the MD program with the integrated AutoPas library on the world record scenario with $18 \cdot 10^6$ particles run on the CoolMUC-3 cluster with a native vectorisation for different numbers of threads and cut-off radii.

With a higher level of parallelisation, the overall performance increases and reaches its peak for all three cell sizes at 0.45 MMUPS if 64 parallel tasks are executed. Beyond this point, all cores in the node are used and hyper-threading begins, which induces a performance drop for the last two measure points.
However, the performance increase depends strongly on the cell size. A simulation with a cut-off radius of 5.0 reaches a speedup of 8.7, while a radius of 3.5 subjects at most a speedup of 5.5 and the smallest radius can only expect an execution to be up to three times as fast. Despite the lower number of required force computations, the performance of a system with $r_C/\sigma = 2.5$ is slower than a system with $r_C/\sigma = 3.5$ after 16 threads and is even outpaced by the simulation with the fewest cells at 64 threads.
One explanation of these different speedups is the more efficient vectorisation of larger cells as discussed is Section 5.3. The main reason is nevertheless the high spinning time of the C08 thread assignation. The higher cell number at small cut-off radii causes the number of tasks to increase by a factor eight. Moreover, if the linked cells are smaller, each task has to perform fewer force computations as in larger cells, and the relative spin time to assign a thread is larger.
One possible solution would be to adapt the number of C08 tasks per assigned thread in the dynamic scheduling to the total number of cells. If there are l linked cells in the domain and p OpenMP threads, the maximum number of C08 tasks each thread could get at once to exploit the parallel resources most efficiently and to reduce the spin time is $\frac{l}{8p}$. However, this reduces the adaptiveness of the C08 traversal and may produce additional overhead in the case of unbalanced particle distributions as it occurs in the sliced traversal. Nevertheless, the overall scalability of this molecular dynamics simulation tool is not optimal yet. The shape of the speedup curves does not match the ideal speedup of $S(p) = p$ for p threads and the MUPS seems to converge towards a maximum that does not depend on the cell size.
In contrast, the performance increase of ls1-mardyn for the same scenario approximately resembles the ideal speed up curve as it can be seen in Figure 5.13. The only deviations for all test cases from the ideal behaviour occur at 128 and 256 threads because of hyper-threading.

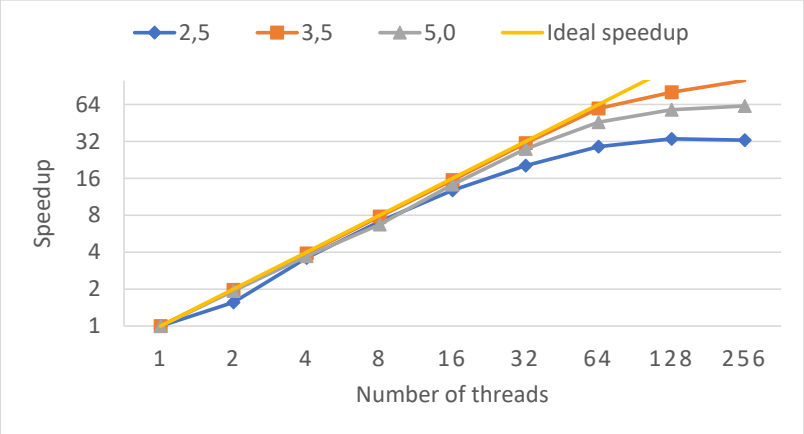


Figure 5.13.: Speedup of ls1-mardyn on the CoolMUC-3
Performance of ls1-MarDyn on the world record scenario with $18 \cdot 10^6$ particles run on the CoolMUC-3 cluster with a native vectorisation for different numbers of threads and cut-off radii

5.4.2. Analysis of the OpenMP parallelisation with VTune

The software performance analysis tool VTune Amplifier has been used to investigate why the AutoPas program is not satisfyingly scalable for higher number of threads, as can be seen in Figure 5.4 or Figure 5.12. It evaluates the execution time of each function and compares the CPU-usage of each parallel thread. Figure 5.14 shows the CPU usage over time of the AutoPas program for a world record simulation with two parallel threads. The worker thread is only started at the end of the preprocessing, which is marked in red. The MUPS measurement only begins after the completion of the first time iteration, marked in orange, and excludes postprocessing at the end of the execution of the program.

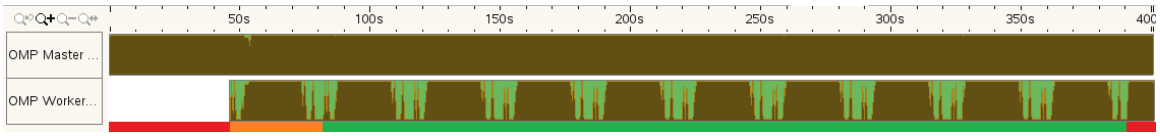


Figure 5.14.: CPU-usage per thread for the total execution time of the AutoPas program Simulation run on the SuperMUC-1-tn with the world record scenario with $18 \cdot 10^6$ particles with a native vectorisation and a cut-off radius of $r_C/\sigma = 3.5$ for two parallel threads.

The large, dark-green section at each iteration in the worker thread corresponds to the pairwise iterator which include the force calculations. The two small peaks right before and after the cell traversal are produced by the SoA Extractor and Loader. The three smaller peaks correspond to the partial integrations, the single functors and the application of periodic boundaries.
After the pairwise iterator, *ApplyPBC* from Subsection 4.3.2 is the most time-consuming functions as it requires 25% of the execution time of the main loop in the program. One third of the time in this function is used to particles which are near the border, as the coordinates of every particle in the domain has to be checked because the AutoPas library lacks an efficient region iterator..

In Figure 5.15, the execution of a time iteration with two threads is compared to an execution on 16 threads. For the sake of convenience, only one of 15 worker threads is shown in the second graph.

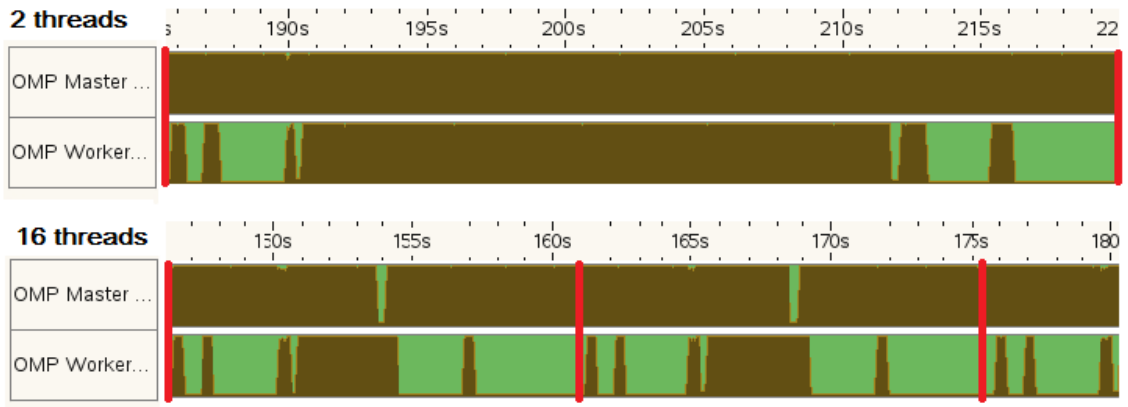


Figure 5.15.: CPU-usage per thread for a time iteration of the AutoPas program
Simulation run on the SuperMUC-1-tn with the world record scenario with $18 \cdot 10^6$ particles with a native vectorisation and a cut-off radius of $r_C/\sigma = 3.5$ for two and sixteen parallel threads.

As expected, the second graph shows a faster simulation. The red vertical bars separate two iterations and during the execution of one iteration with two threads, almost two and a half iterations could be performed on 16 threads. The most notable speed increase occurred in the sliced cell traversal, but the smaller parallel regions could also be executed faster. The execution time of the serial parts of the code remained constant at 148s for the whole program and at 96s for the performance measurement section, and their share of the total execution time increased accordingly. As a matter of fact, the section in which the MUPS measurement is performed comprises a share of 71% partial code of the execution time for 16 threads. With the assumption of a perfectly scalable code on an infinite number of cores, the theoretical maximal MUPS would be $1.69 \cdot 10^6$ MUPS, which is one third above the observed stagnation point of $1.20 \cdot 10^6$ MUPS in Figure 5.4. As the serial parts of the code do not depend on the cell size or the cut-off radius, the maximal performance for the three scenarios in Figure 5.11 is similar.
In a next step, most of the serial parts of the main loop have been removed to obtain a reduced AutoPas program which only comprises the pairwise particle iteration and the partial integrations. The overall performance is compared to the original AutoPas program in Figure 5.16 and the relative speedup to the serial execution is given in Figure 5.17.

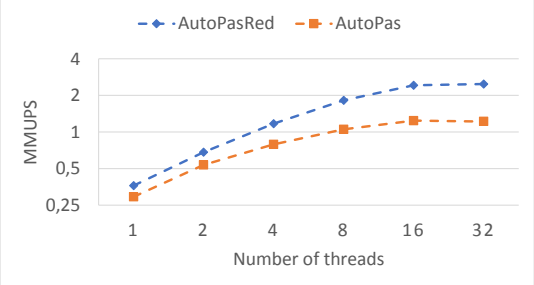


Figure 5.16.: MUPS of AutoPas and the reduced AutoPas.

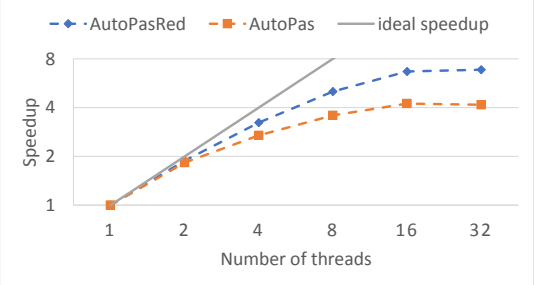


Figure 5.17.: Speedup of AutoPas and the reduced AutoPas.

Simulation run on the SuperMUC-1-tn with the world record scenario with $18 \cdot 10^6$ *particles with a native vectorisation and a cut-off radius of* $r_C/\sigma = 3.5$ *for two and sixteen parallel threads.*

The removal of the serial parts increased the maximal performance by a factor two. The reduced AutoPas has a higher scalability, even though the speedup also converges towards a maximum and does not reach the ideal speedup for higher number of threads as is, for instance, the case in ls1-mardyn.
Finally, the average CPU-usage per thread of the pairwise cell traversals as calculated by VTune Amplifier is presented in Figure 5.18.

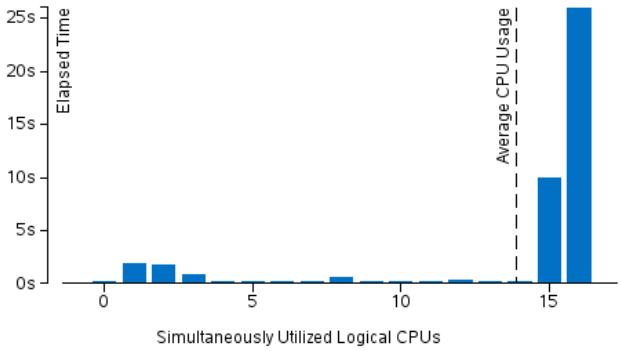


Figure 5.18.: CPU-usage per thread of the pairwise iterator in the AutoPas program Simulation run on the SuperMUC-1-tn with the world record scenario with $18 \cdot 10^6$ particles with a native vectorisation and a cut-off radius of $r_C/\sigma = 3.5$ for sixteen parallel threads.

On average, the sliced traversal uses 14.3 parallel threads instead of the available 16 threads. During one third of the execution time, one thread did not perform any computations. This is partially due to a load imbalance in the last thread which gets less cells from the sliced traversal as the number of linked cells per dimension is not divisible by 16 without remainder. Moreover, the existence of significant regions with less than four parallel threads degrades the performance.

5.4.3. Large systems

In this second part, the consistency of the performance of varying system sizes is evaluated. The particle density and the cell size $r_C/\sigma = 3.5$ remain constant throughout the simulations. Between two simulations, the domain length is doubled in each dimension. Thus, the total number of represented particles increases by a factor eight. All domains with 2^n cells per dimension and $n \leq 6$ are simulated, whereas the first simulation has one cell with 16 particles. As the available node memory does not allow particles in 2^7 cells, the performances for 80 and 96 cells per dimension, which correspond to a respective total of $17 \cdot 10^6$ and $29 \cdot 10^6$ particles, are provided.
The performances of the simulations is measured in millions of molecule updates per second (MMUPS) and is represented in Figure 5.19. To obtain the best possible simulation results on the SuperMUC-1-tn, a C08 cell traversal over 16 threads with a native vectorisation is applied to the system.

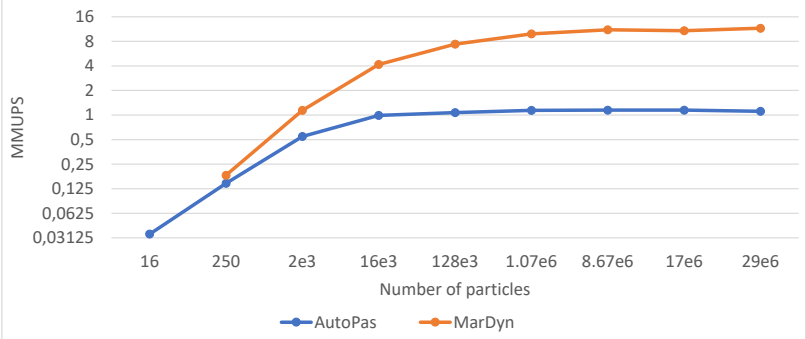


Figure 5.19.: Strongscaling of the system size
The simulation is run on SuperMUC-1-tn with a native vectorisation and 16 parallel threads. The domain is filled with a cubic grid in linked cells with a constant edge length of $r_C/\sigma = 3.5$. MarDyn is run in the RMM-mode with a single precision, which is how it was run in the world record.

For small domains up to 15 000 particles, the execution of constant parts of the program and too few cells to make use of all available threads degrade the performance. For larger systems, the molecule update rate reaches and remains at its maximum at around $1.15 \cdot 10^6$ updates per second. A similar behaviour can be observed with ls1-mardyn, even though the maximal reached molecule update rate is ten times higher as in AutoPas.

5.5. Load balancing

5.5.1. Description of the load balancing scenarios

Until now, only simulations with perfectly uniformly loaded domain have been considered. However, this is not always the case in reality. Therefore, this section analyses the behaviour of the program for unequally distributed particles.

Unbalanced load distributions mostly affect the parallel cell traversals since two threads may handle cells with different numbers of particles and may cause additional overhead at thread barriers due to unequal execution times. The main idea here is to investigate how the two cell traversals, C08 and sliced, adapt to different particle distributions.
Two scenarios are proposed on a same cuboidal domain. The first fills up the whole domain with a cubic grid containing $4.24 \cdot 10^6$ particles, the second one fills only one half of the domain with particles at the same density, thus there are in total $2.12 \cdot 10^6$ particles, and leaves the other half empty. Both traversals are then applied to test scenarios with, again, three cut-off radii of $r_C/\sigma = 2.5$, $r_C/\sigma = 3.5$ and $r_C/\sigma = 5.0$.

5.5.2. Scalability of the C08 and the sliced traversal

The three following graphs compare the ratio between the performances of the C08 traversal and the sliced traversal on the two previous scenarios for an increasing number of threads. A ratio inferior to one means a better performance of the C08 traversal.

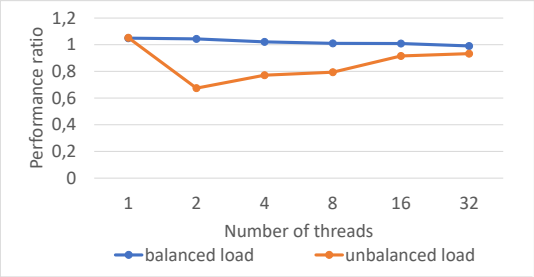


Figure 5.20.: $\frac{r_C}{\sigma} = 2.5$

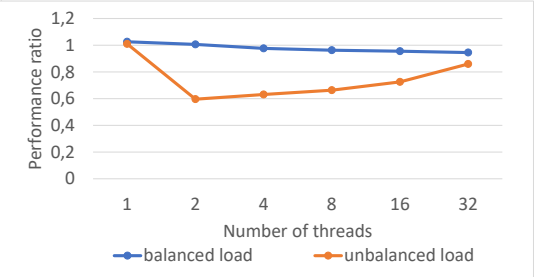


Figure 5.21.: $\frac{r_C}{\sigma} = 3.5$

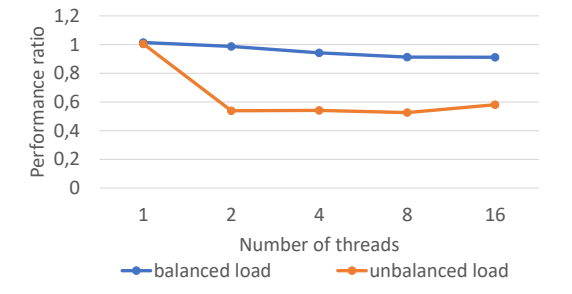


Figure 5.22.: $\frac{r_C}{\sigma} = 5.0$

Serial execution of the world record scenario of a cubic grid with $18 \cdot 10^6$ particles for different vector instruction sets and cell sizes.

In the balanced scenario, bot traversals yield similar results as the ratio goes to one, with a slightly better performance of the sliced traversal for a small number of threads and a slightly better performance of the C08 traversal for many threads.
In the unbalanced scenario, the C08 traversal leads to similar performances as in the balanced case, while the sliced traversal loses performance. Indeed, the latter splits the domain into equally sized chunks of cells and starts one thread on each chunk. In the unbalanced scenario,

half of the threads exclusively handle empty cells and have to wait for the end of the execution of the other threads, and therefore cause much more overhead time at thread barriers. On the other hand, the C08 traversal follows a dynamic scheduling, so any task that has finished its force computations gets a new task assigned, and no additional overhead occurs after the processing of an empty cell.
The worst relative performance of the sliced traversal is reached for two threads. While the C08 traversal almost doubles its performance, the sliced traversal has the same performance as in the serial case, as all particles are handled by one single thread. For higher numbers of threads, the difference between both methods also decreases, since the sliced traversal simultaneously processes non-empty cells and the speedup of the C08-traversal decreases.
Finally, the performance ratio is more in favour of C08 for larger linked cells. Indeed, for the minimum at two threads, the sliced traversal reaches 67% of the C08 performance for $r_C/\sigma = 2.5$, the ratio then decreases to 60% for $r_C/\sigma = 3.5$ and to 54% for $r_C/\sigma = 5.0$. A higher number of particles per cell increases the required computational effort to process full cells and the overhead in empty cells increases accordingly. Therefore, the performance of the sliced traversal deteriorates even more for a greater cut-off radius.
A similar behaviour of both cell traversals can be observed in ls1-mardyn. Figure 4 and 5 in [ST18] compare different cell traversals, including the sliced and C08 traversal, for two scenarios respectively with a balanced and an unbalanced load. In the first case, both traversals yield equally high MUPS, even though for eight and 16 threads the relative performance of the sliced traversal is better than in AutoPas as it reaches 1.15 times the C08 performance. The decrease of the MUPS with the sliced traversal to less than 70% of the C08 traversal also occurs in the scenario with an unbalanced load. However, it is only measurable for more than four threads and not for two threads as in AutoPas because of a different initial particle configuration with a dense central part, which is equally distributed on two threads in the sliced traversal.
To conclude, the C08 traversal is, in general, to be preferred over the sliced traversal. For a perfectly balanced particle distribution in the domain, both methods lead to a similar performance, and for unevenly distributed systems, the C08 traversal yields an unambiguously better performance.

5.6. Multi-site molecules

5.6.1. Description of the simulated molecules

The number of pairwise interactions increases and the overall performance decreases proportionally to the number of sites per molecule. Indeed, if two molecules are represented, the forces acting between all sites and the resultant torque on each molecule are computed. In the next section, the performances of four molecules are compared, and a single site atom is added to the simulation as a reference element for the performance.

- Argon (Ar) was chosen the reference particle. Its chemical stability and make it very popular in molecular dynamics simulations. The Lennard-Jones parameters of argon are taken from [GZ07]. As it is not a molecule, no partial charges are considered in argon and does not form an electrostatic potential.

- Carbon monoxide (CO) is composed of one carbon and one oxygen atom. The difference in electronegativity between both elements leads this molecule to a polarise light, which results in electrostatic forces between two molecules. As depicted in Figure 5.24 , it has a strong triple bond between both constituting atoms [Bou92].
- The most common example of a three-site molecule is water (H_2O). As the two $H-O$ bonds are not organised in a parallel configuration, but form a 104.45° angle as can be seen in Figure 5.25, the water molecule is polarised with a negative partial charge on the oxygen. This polarisation causes inter-molecular Coulomb interactions and the orientation of water molecules in their liquid phase to form hydrogen bonds. The Lennard-Jones parameters for water are provided in [ZZS10].
- Ammonia is a compound composed of one nitrogen and three hydrogen atoms, which are, similarly to water, not located on a plane, but form a pyramid and, therefore, bear partial charges on its different sites. The angle between two $N-H$ bonds is 107.4° [MMM11] and therefore lies between the angles occurring in water and methane.
- Finally, methane is the most simple configuration for an alkane, a chain of hydrogenated single-bond carbon atoms. Its special composition of a central carbon with four neighbouring hydrogen atoms in a tetrahedral structure as can be seen in Figure 5.27 prevents any polarisation since the sum of all local dipoles at the $C-H$ bonds is zero [MMM11]. Therefore, no Coulomb forces appear between two adjacent methane molecules.

Figure 5.23.: Argon (Ar)

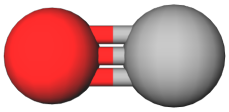


Figure 5.24.: Carbon Monoxide (CO)

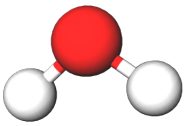


Figure 5.25.: Water (H_2O)

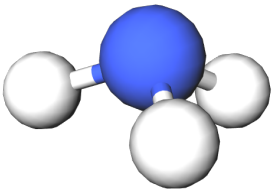


Figure 5.26.: Ammonia (NH_3)

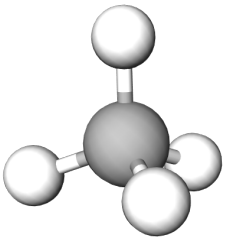


Figure 5.27.: Methane (CH_4)

These molecules have been chosen for their similar chemical properties regarding the Lennard-Jones potential and for their similar dimensions. Indeed, for all constituting elements of these molecules, the values for σ range from 2.81 to 3.35 Å; and the total masses of the molecules lie between 16.0 and 28.0 u. Therefore, the same simulation parameters can be chosen to compare the performance of the simulation with different particles.
A cubic mesh of 100 particles per dimension and mesh length of 4.0 Å; is placed in a domain with periodic boundaries and subdivided into 15 625 linked cells. The ratio $\frac{r_{cutoff}}{\sigma}$ is 4.0 and 64 particles lie within one cell. The domain is traversed by a C08 traversal with 16 OpenMP threads. A Lennard-Jones functor is applied to all five particles and is computed using a native vectorisation on a SuperMUC phase 1 thin node. Additionally, for the three polarised molecules carbon monoxide, water and methane, a Coulomb functor and the combination of Coulomb and Lennard-Jones functors, which best describes actual molecular interactions, are applied.

5.6.2. Performance of simulations with molecules

The following graph compares the molecule update rates per second (MMUPS) of the simulations from Subsection 5.6.1.

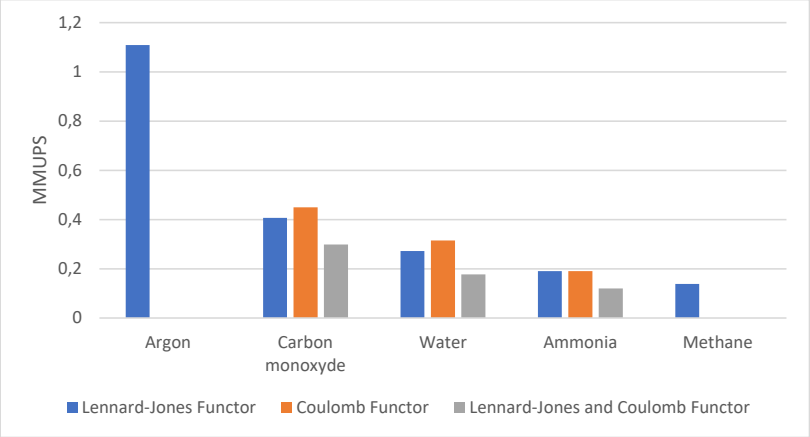


Figure 5.28.: Performance of the Lennard-Jones and the Coulomb functor with different molecules

First of all, the performance drops for an increasing number of sites per particle. This is explained by the higher number of required force calculations in the pairwise functor. Each site in the first molecule interacts with each site of the second molecule, thus, if the first molecule has n sites and the second m sites, the total number of force computations is given by $N_{forces} = n \cdot m$. For two molecules with the same number of sites n, the addition of one

site leads to a performance drop by a factor $\left(\frac{n}{n+1}\right)^2$ in the pairwise functor.
The large drop between argon and carbon monoxide is not only caused by the four-times higher number of site-site interactions, but also to a higher complexity of processing the molecules. As discussed in Subsection 2.2.2, the resultant force does not necessarily act on the mass centre of the molecule, and torques may appear. Consequently, the angular velocity and the orientation of the molecule have to be calculated at each iteration. Moreover, the inertia tensor of carbon monoxide has rank 2, so the angular momentum has to be calculated in the body frame, which is less efficient than in the general case, according to Subsection 4.2.1, this applies to the three other molecules.
The electrostatic potential yields slightly better results than the Lennard-Jones potential for the polarised molecules. Indeed, the computation of Lennard-Jones forces requires 13 additions and 15 multiplications while the computation of Coulomb forces only require 11 additions, 9 multiplications, and one square root operation.
The combination of the two functors leads to a better performance than expected if the whole computation time was used to compute the pairwise interaction, which is calculated as $MUPS_{comb;theor} = \frac{1}{\frac{1}{MUPS_{LJ}}+\frac{1}{MUPS_{Coulomb}}}$. For instance, in the case of water, the actual performance $MUPS_{Coulomb} = 1.78 \cdot 10^5$ is 22% higher than the theoretical performance of $MUPS_{comb;theor} = 1.46 \cdot 10^5$. This difference is due to a constant execution time at each iteration in the computation of the partial integrations and in the application of periodic boundary conditions, which does not depend on the number of pairwise functors. From the measured performances, it is possible to deduce the share corresponding to the constant, off-functor, part of the total performance. Let M_{LJ}, M_C and M_T be the measured MUPS of the respective simulations with the Lennard-Jones functor, the Coulomb functor and the combination of both. Moreover, let I_{LJ}, I_C and I_T be the share corresponding to the pairwise iteration(s) and C the share of the constant execution from the overall performance. Since the performance is measured in the inverse time dimension, the Equation 5.1 can be stated.

$$\frac{1}{I_T} = \frac{1}{I_{LJ}} + \frac{1}{I_C} \qquad \textit{and} \qquad \frac{1}{M_*} = \frac{1}{C} + \frac{1}{I_*} \tag{5.1}$$

Therefore, we have:

$$\frac{1}{M_T} = \frac{1}{C} + \frac{1}{I_T} \tag{5.2}$$

$$\frac{1}{M_T} = \frac{1}{C} + \frac{1}{I_{LJ}} + \frac{1}{I_C} \tag{5.3}$$

$$\frac{1}{M_T} = \frac{1}{C} + \frac{1}{M_{LJ}} - \frac{1}{C} + \frac{1}{M_C} - \frac{1}{C} \tag{5.4}$$

$$\frac{1}{C} = \frac{1}{M_{LJ}} + \frac{1}{M_C} - \frac{1}{M_T} \tag{5.5}$$

The share corresponding to the constant part of the total execution time in the main loop can be calculated as M_*/C and is, for water, equal to 24% if the two functors are applied. As the partial integrations and the application of the periodic boundary conditions do not depend on the structure or the size of a molecule, simulations with carbon monoxide, water or ammonia return similar values for C, varying between $1.22 \cdot 10^{-6}$ and $2.20 \cdot 10^6$ seconds per molecule per iteration ($MUPS^{-1}$).

6. Conclusion

The functions of the AutoPas library have been implemented in a molecular dynamics simulation tool. An analysis of the performance and of the scalability at node-level has been performed and compared to the highly scalable program ls1-mardyn.
Each particle contains data about the position, the velocity and the occurring forces in the current and the previous time step. To represent rigid molecules, the orientation, the current angular velocity and the applied torques are required in addition to the layout of the constituting atoms. The partial time integrations follow the Verlet-Störmer approach. The traversal of nearby particles pairs can either only be achieved through either direct sums or linked cells as the AutoPas class does not support Verlet neigbours' lists yet. To exploit the available hardware resources on a node, which consist of 16 cores on the SuperMUC-1-tn and 64 cores on the CoolMUC-3, an OpenMP parallelisation of the partial time integrations, of the application of boundary conditions and of the pairwise linked cells traversal have been provided. The latter is performed by autopas either through a C08 scheme or a sliced traversal. Periodic boundary conditions are best represented by halo particles placed in halo cells behind the domain delimitations which are integrated in the cell traversals. The computation of the angular momentum of a molecule from the applied torque shall be performed in the initial frame or in the body frame for particles with less than three rotational degrees of freedom, so the computation of the inertia tensor is only required once per molecule type at the beginning of the execution of the program. Lastly, pairwise functors used to compute the intermolecular forces from the Lennard Jones, the electrostatic or the gravitational potential support both the AoS and SoA data layout. SoA notably provides the possibility of an efficient vectorisation using SIMD instructions.
The majority of the node-level experiments have been performed on thin nodes on SuperMUC phase 1 with a uniform particle distribution over domain with different linked cell sizes and cut-off radii. The program time performance scales linearly with the number of assigned parallel threads up to eight threads. For more tasks, the performance increase slows up to stagnation. The use of structures of arrays instead of arrays of structures reduces the execution time of the main loop by up to 40% and the use of the native AVX vector instruction set on SuperMUC-1-tn increases performance by an additional 25%. For systems with a uniform particle distribution, the C08 and the sliced traversal yield similar time performances, but for an unbalanced load, the C08 traversal is much more efficient. For sufficiently large systems, the execution time per particle is not affected by the total system size. Finally, the use of molecules instead of atoms strongly increases the execution time by particle.
The developed simulation tool to test the AutoPas library might be improved with a better OpenMP parallelisation of the sections of the code, that do not depend directly on the AutoPas library, such as the partial integrations or the application of boundary conditions. Moreover, this program can be extended to support and to evaluate future features of the AutoPas library such as Verlet neighbour's list, the Reduced Memory Mode (RMM) or a multi-node parallelisation with the MPI protocol.

Appendix

A. Unit system

To get meaningful results, it is necessary to use coherent units. Since molecular dynamics happen at nanoscale, SI-units will yield unwieldy results with very high exponents and lead to numerical errors due to the floating point number representation. In this sense, the following base units shown in Table A.2 will be used throughout the thesis.

Name	Used unit	SI-Unit	conversion
distance	Angström $[A]$	meter $[m]$	$1\ A = 1.000 \cdot 10^{-10} m$
time	picoseconds $[ps]$	seconds $[s]$	$1\ ps = 1.000 \cdot 10^{-12} s$
mass	atomic mass $[u]$	kilograms $[kg]$	$1\ u = 1.661 \cdot 10^{-27} kg$
charge	elementary charge $[e]$	Coulomb $[C]$	$1\ e = 1.602 \cdot 10^{-19} J$
temperature	Kelvin $[K]$	Kelvin $[K]$	

Table A.2.: Used base units

Furthermore, several other units can be derived from these base units (Table A.4).

Name	Used unit	SI-Unit	conversion
velocity	$A \cdot ps^{-1}$	$m \cdot s^{-1}$	$1\ Aps^{-1} = 100m \cdot s^{-1}$
Force	$u \cdot A \cdot ps^{-2}$	N (Newton)	$1\ uAps^{-2} = 1.661 \cdot 10^{-13}N$
Energy	$u \cdot A^2 \cdot ps^{-2}$	J (Joule)	$1\ uA^2ps^{-2} = 1.661 \cdot 10^{-23}J$

Table A.4.: Derived units

Physical constants shall be expressed with these units:

- Coulomb's constant (cf. 2.6): $k_C = 1.389 \cdot 10^5\ uA^3ps^{-2}e^{-2}$
- Boltzmann constant (cf. 2.13): $k_B = 0.8314\ uA^2ps^{-2}K^{-1}$

B. XML files used as input

B.1. Global simulation settings

This section gives an overview of all options that can be used to run the AutoPas simulation tool. Each item in Table B.1 corresponds to an element in the XML-file for the global simulation settings. Some options will require complex types, which are defined in a separate table below. The column "Number" specifies how often the element can be declared in the XML-file.

Name	Type	Number	Description	Options
inputfiles	*string*	any	Name of the particle input files	
outputname	*string*	1	Base name of the output files	
endfile	*string*	1, optional	Save the final status of the simulation in a new XML-file with the specified name	
frequency	*int*	1	Defines at which frequency a VTK-file shall created	0 disables the output writer
profileFile	*string*	1, optional	Plot y-velocity profile in a CSV-file with the specified name	
profile BucketsX	*int*	1, optional	Number of divisions in X-direction to profile the y-velocity	
delta_t	*float*	1	Time step between two iterations	
end_t	*float*	1	End time of the simulation	
b_factor	*float*	1	Factor b in the Maxwell-Boltzmann distribution in Subsection 2.1.5	0 disables brownian motion
g_grav_x	*float*	1	Gravitation field in x-direction	
g_grav_y			Gravitation field in y-direction	
g_grav_z			Gravitation field in z-direction	
domainX	*float*	1	Domain size in x-direction	
domainY			Domain size in y-direction	
domainZ			Domain size in z-direction	

r_cutoff	*float*	1	Cut-off radius for the force calculations. Linked cells will automatically have it at least as edge length	
Container Type	*int*	1, optional	Method for the pairwise particle traversal default is 1	0 - direct sum 1 - linked cells 2 - Verlet lists
Force Computation Method	*int*	any, optional	Chose potential to calculate intermolecular forces default is 1	0 - no interactions 1 - Lennard-Jones 2 - Gravity 3 - LJ (molecules) 4 - Coulomb (molecules)
Periodic Computation Type	*int*	1, optional	Define the method to compute the forces at periodic boundaries as defined in Subsection 4.3.2 default is 2	0 - *ComputePBC* 1 - *computeNeighboursPBC* 2 - *ApplyPBC*
Traversal	*int*	1, optional	Define the linked cells traversal default is 1	0 - sequential 1 - C08 2 - sliced
Vectorisation	*boolean*	1, optional	If true, uses SoA, if false, uses AoS default is true	
bc_left	*int*	1	Set the boundary condition at the negative X-axis	0 - outflow 1 - periodic 2 - reflective
bc_upper	*int*	1	Set the boundary condition at the positive Z-axis	
bc_right	*int*	1	Set the boundary condition at the positive X-axis	
bc_lower	*int*	1	Set the boundary condition at the negative Z-axis	
bc_front	*int*	1	Set the boundary condition at the negative Y-axis	
bc_back	*int*	1	Set the boundary condition at the positive Y-axis	
thermostat	defined in Table B.3	1, optional	Define whether a thermostat should be applied to the system or not	

Table B.1.: Global settings

Name	Type	Number	Description	Options
initial	*float*	1	Sets the initial temperature	
timestep	*int*	1	Number of timesteps after which the thermostat shall be applied	0 disables the thermostat
heating	defined in Table B.5	1, optional	Define whether the system shall be cooled or heated over time	
ignoreY	*boolean*	1, optional	Does not consider the y-component of velocities	

Table B.3.: Thermostat

Name	Type	Number	Description	Options
target	*float*	1	Sets the target temperature	
temperature_step	*float*	1	Defines the temperature increment at each update	
timestep	*int*	1	Number of timesteps after which the temperature shall be updated	

Table B.5.: Heating

B.2. Particle input

The particle data is specified in a separate XML file. The constituting atoms of molecules are defined in an own particle type. The XML-file contains at least one definition of a particle type as given in Table B.7 and any number of molecule types as given in Table B.9. Moreover, at least one particle generator must be specified. It can either be a single particle (Table B.13), a cuboid of particles (Table B.15) a two-dimensional membrane with a harmonic potential between two adjacent particles (Table B.17) or a sphere of particles (Table B.19). Single-site atoms and multi-sites molecules can be mixed in one simulation.

Name	Type	Number	Description
id	*int*	1	unique id of the type
mass	*float*	1	mass (in u) of the particle
epsilon	*float*	1	epsilon (in u$Å^2ps^{-2}$) of the particle
sigma	*float*	1	sigma (in Å) of the particle
charge	*float*	1	charge (in e) of the particle
RtruncLJ	*float*	1, optional	Only to be specified if the cut-off radius for this particle differs from the global cut-off radius defined in Table B.1
fixed	*boolean*	1, optional	True if the particle cannot move. Default setting is false.

Table B.7.: Particle type

Name	Type	Number	Description
type	*int*	1	unique id of the type
usePA	*int*	1, optional	If true, use the body frame to compute the angular momentum. Else, the computations are performed in the initial frame. If it is not specified, the program choses on its own the best variant
sites	defined in Table B.11	at least 1	Define the constituting sites of the molecule

Table B.9.: Molecule type

Name	Type	Number	Description
typepart	*int*	1	id of the atom type, as defined in Table B.7
coord	defined in Table B.21	1	coordinate of the site with respect to the centre of mass of the molecule
partial _charge	*float*	1, optional	partial charge of the atom in the case of polarised molecule

Table B.11.: Sites

Name	Type	Number	Description
type	*int*	1	ID of the particle, it can either be the ID for an atom (Table B.7) or a molecule (Table B.9)
coord	defined in Table B.21	1	coordinate of particle
velocity	defined in Table B.21	1	initial velocity
force	defined in Table B.21	,1 optional	optional force acting on the particle throughout the simulation

Table B.13.: Single particle input

Name	Type	Number	Description
type	*int*	1	ID of the particle, it can either be the ID for an atom (Table B.7) or a molecule (Table B.9)
coord	defined in Table B.21	1	coordinate of the lower-left-front particle of the cuboid
dimension	defined in Table B.21	1	Number of particles in each dimension of the cuboid
mesh	*float*	1	mesh size of the cuboid
velocity	defined in Table B.21	1	initial velocity
force	defined in Table B.21	1, optional	optional force acting on the particle throughout the simulation
coord_force	defined in Table B.21	any, optional	mesh coordinates of the particles, on which the optional force should act
t_end_force	*float*	1	time after which the optional force should cease

Table B.15.: Cuboid input

Name	Type	Number	Description
radius	*float*	1	radius of the sphere
type	*int*	1	ID of the particle, it can either be the ID for an atom (Table B.7) or a molecule (Table B.9)
coord	defined in Table B.21	1	coordinate of the centre of the sphere
dimension	defined in Table B.21	1	Number of particles in each dimension of the cuboid
mesh	*float*	1	mesh size of the cuboid
velocity	defined in Table B.21	1	initial velocity
force	defined in Table B.21	1, optional	optional force acting on the particle throughout the simulation
coord_force	defined in Table B.21	any, optional	mesh coordinates of the particles, on which the optional force should act
t_end_force	*float*	1	time after which the optional force should cease

Table B.17.: Membrane input

Name	Type	Number	Description
stiffness	*float*	1	Stiffness of the cuboid
r_zero	*float*	1	average bond length
type	*int*	1	ID of the particle, it can either be the ID for an atom (Table B.7) or a molecule (Table B.9)
coord	defined in Table B.21	1	coordinate of the lower-left-front particle of the cuboid
dimension	defined in Table B.21	1	Number of particles in each dimension of the cuboid
mesh	*float*	1	mesh size of the cuboid
velocity	defined in Table B.21	1	initial velocity
force	defined in Table B.21	1, optional	optional force acting on the particle throughout the simulation
coord_force	defined in Table B.21	any, optional	mesh coordinates of the particles, on which the optional force should act
t_end_force	*float*	1	time after which the optional force should cease

Table B.19.: Sphere input

Name	Type	Number	Description
x	*int/float*	1	x - component
y	*int/float*	1	y - component
z	*int/float*	1	z - component

Table B.21.: Vector of integers or floats

List of Figures

List of Tables

Bibliography

[BBB+14] S. Becker, M. Bernreuther, M. Buchholz, W. Eckhardt, A. Heinecke, S. Werth, H.J. Bungartz, C. W. Glass, H. Hasse, J. Vrabec, and C Horsch, M. Niethammer. ls1-mardyn: the massively parallel molecular dynamics code for large systems. August 2014.

[Ber98] D. Berthelot. Sur les mélanges des gaz. *Comptes rendus hebdomadaires des séances de l'Académie des Sciences*, 126:1703–1855, 1898.

[Bou92] J.-P. Bouanich. Site-site Lennard-Jones potential parameters for N_2, O_2, H_2, CO and CO_2. 47, No, 4:243–250, 1992.

[Bro28] R. Brown. 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; and the general existence of active molecules in organic and inorganic bodies. *Philosophical Magazine*, pages 466–473, 1828.

[Cla04] D. Clark. *General Chemmistry - Pearls of wisdom.* Jones & Bartlett Learning, Sudburry, Massachussets, USA, 2004.

[Cou85] C.-A. Coulomb. Premier mémoire sur l'électricité et le magnétisme. *Histoire de l'Académie Royale des Sciences*, pages 569–577, 1785.

[Dys67] F. J. Dyson. Ground-State Energy of a Finite System of Charged Particles. *Journal of Mathematical Physics*, 8:1538–1545, August 1967.

[Eck14] W. Eckhardt. *Efficient HPC Implementations for Large-Scale Molecular Simulation in Process Engineering.* Phd, Institut für Informatik 5, Technische Universität München, Garching, June 2014.

[EL30] R. Eisenschitz and F. London. Über das Verhältnis der van der Waalsschen Kräfte zu den homöopolaren Bindungskräften. *Zeitschrift fur Physik*, 60:491–527, July 1930.

[GKZ07a] R. Griebel, S. Knapek, and G. Zumbusch. Numerical Simulation in Molecular Dynamics. Numerics, Algorithms, Parallelization, Applications. *Texts in Computational Science and Engineering*, 5:88, 2007.

[GKZ07b] R. Griebel, S. Knapek, and G. Zumbusch. Numerical Simulation in Molecular Dynamics. Numerics, Algorithms, Parallelization, Applications. *Texts in Computational Science and Engineering*, 5:427–428, 2007.

[GKZ07c] R. Griebel, S. Knapek, and G. Zumbusch. Numerical Simulation in Molecular Dynamics. Numerics, Algorithms, Parallelization, Applications. *Texts in Computational Science and Engineering*, 5:221–226, 2007.

[GZ07] S. Griebel, R.and Knapek and G. Zumbusch. Numerical Simulation in Molecular Dynamics. Numerics, Algorithms, Parallelization, Applications. *Texts in Computational Science and Engineering*, 5:96, 2007.

[Hoo78] R. Hooke. *De Potentia Restitutiva, or of Spring. Explaining the Power of Springing Bodies.* London, Kingdom of England, 1678.

[Jon24] J. E. Jones. On the Determination of Molecular Fields. II. From the Equation of State of a Gas. *Proceedings of the Royal Society of London Series A*, 106:463–477, October 1924.

[KML13] M. Kunowsky, J. Marco-Lozar, and A. Linares-Solano. Material demands for storage technologies in a hydrogen economy. 2013:10, March 2013.

[Lor81] H. A. Lorentz. Ueber die Anwendung des Satzes vom Virial in der kinetischen Theorie der Gase. *Annalen der Physik*, 248:127–136, 1881.

[MMM11] F. Mesli, R. Mahloub, and M. Mahloub. Molecular dynamics comparative study of methane nitrogen and methane-nitrogen.ethane systems. *Arabian Journal of Chemistry*, 4:211, 2011.

[New66] I. Newton. *Principia Mathematica Philosophiae Naturalis.* London, Kingdom of England, 1666.

[NWA13] L. Nikunen, V. Weinberg, and N. Anastapoulos. Best practice guide supermuc v1.0. page 13, May 2013.

[Smi61] Oliver K. Smith. Eigenvalues of a Symmetric 3 &Times; 3 Matrix. *Commun. ACM*, 4(4):168–, April 1961.

[ST18] S. Seckler and N. Tchipev et al. TweTriS: Twenty-trillion-atom simulation. *submitted*, July 2018.

[Syl52] J. J. Sylvester. XIX. A demonstration of the theorem that every homogeneous quadratic polynomial is reducible by real orthogonal substitutions to the form of a sum of positive and negative squares. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 4(23):138–142, 1852.

[Ver67] L. Verlet. Computer "Experiments" on classical fluids. I. Thermodynamical properties of Lennard-Jones particles. pages 98–103, 1967.

[ZZS10] Y. Zheng, A. Zaoui, and I. Shahrour. Evolution of the interlayer space of hydrated montmorillonite as a function of temperature. 95:1494, 09 2010.