TUM School of Computation, Information and Technology
Technische Universität München

AutoPas:
Automated Dynamic Algorithm Selection for HPC Particle Simulations

Fabio Alexander Gratl-Gaßner

Vollständiger Abdruck der von der TUM School of Computation, Information and Technology der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitz:

Prof. Dr. Hans Michael Gerndt

Prüfende der Dissertation:

1. Prof. Dr. Hans-Joachim Bungartz
2. Prof. Dr. Philipp Neumann

Die Dissertation wurde am 20.12.2024 bei der Technischen Universität München eingereicht und durch die TUM School of Computation, Information and Technology am 10.02.2025 angenommen.

Acknowledgements

Here is the place to express my gratitude for all the people I have met along the way of this exciting and beautiful journey that comes to a close with this thesis.

First and foremost, Prof. Hans-Joachim Bungartz, for providing this opportunity and the continued support. Together with Prof. Philipp Neumann, they provided just the right mix of strategic guidance and technical discussions, so thanks to both.

Next, I want to thank my former colleague, Nikola Tchipev, who brought me closer to the chair and set me on track for molecular dynamics with a guided research and Master thesis project, thus kickstarting this whole journey. Steffen Seckler was a significant influence, especially during my first years at the chair. Together, we explored the intricacies of `C++`, software development, and the Alli-Hans Arena, and I was able to learn a great deal from him.

But it is also important to extend my thanks to the new generation of Ph. D. candidates who will pick up this work and endure our code legacy. Sam Newcome, Markus Mühlhäußer, Manish Mishra, and Jonas Schuhmacher were fun to work with, bringing new perspectives to the AutoPas project, and a pleasure to share an office with. And also the rest of SCCS for a delightful atmosphere to be productive in, enjoy cake, and have fun together even beyond the working hours.

Moreover, I have to thank the long list of students who hopefully enjoyed working with me as much as I did with them in the scope of one or more seminar papers, Bachelor, or Master theses: Akash Mundra, Albert Noswitz, Alexander Haberl, Benjamin Decker, Christian Menges, Deniz Candas, Fritz Hofmeier, Jacky Körner, Jakob Andreas Englhauser, Jan Hampe, Jan Nguyen, Jeremy Harisch, Joachim Marin, Johannes Kroll, Johannes Spies, Jonas Schuhmacher, Julian Mark Pelloth, Julian Spahl, Leonhard Laumeyer, Ludwig Gärtner, Marco Papula, Maximilian Geitner, Mustafa Fatih Baysan, Nanxing Nick Deng, Nicola Fottner, Oliver Bösing, Raffael Düll, Raphael Penz, Sabrina Krallmann, Sascha Sauermann, Timur Eke, Tina Vladimirova, Tobias Alexander Humig, Twain Mark Thomas Henkel, Vincent Fischer, and Wolf Thieme.

Of course, I also want to thank my friends and family for their support and distractions over the years. Especially, I thank Fine for always patiently enduring discussions, not only listening but also providing insightful feedback, thoughts, and ideas. The combination of technical and emotional assistance was and will continue to be invaluable.

Abstract

Particle simulations come in many forms and types. For instance, Discrete Element Method simulations modeling powder transport, Smooth Particle Hydrodynamics to simulate materials behavior during high-velocity impacts, or Molecular Dynamics simulating nanobubbles and droplets. They can comprise up to trillions of particles and, due to the necessity for small time steps, need millions of iterations to evaluate the evolution of a scenario. This makes particle simulations an obvious application for High Performance Computing and the need to employ supercomputers. These simulation methods lead to application scenarios with very different computational profiles and structures. Even within the scope of Molecular Dynamics, the structure of a protein folding simulation is vastly different from that of the explosion of a liquid film. Especially in the latter example, this structure can change significantly throughout a simulation.

Fortunately, the literature offers an assortment of algorithms with different characteristics to tackle this heterogeneous field of problems. Unfortunately, identifying the optimal algorithm for a given problem is not trivial. This is true for application experts with limited education in computer science, but also for algorithm experts. The performance of an algorithm depends on many factors, such as the simulation method, the scenario setup, and the hardware on which it is executed. Therefore, a solution that finds at least a near-optimal algorithm for a specific scenario would go a long way to improve algorithmic simulation efficiency.

This thesis presents an implementation for this approach with the new library project AutoPas, offering a novel solution to this problem. It combines efficient algorithms for short-range particle simulations with automated dynamic algorithm selection to serve as the core data structure and performance driver for the particle interaction of a simulator. For this, a unified interface is developed, allowing interaction with the library without the need to know what algorithm it uses internally. Nevertheless, these abstractions must not come at the cost of performance. Thus, AutoPas still enables targeted optimizations, e.g., with SIMD instructions.

Furthermore, the thesis shows how AutoPas smoothly integrates into four different simulator codes. Using their own benchmarks, we demonstrate how AutoPas can enhance their flexibility and performance, resulting in speedups of up to 1.6.

Contents

List of Figures

List of Tables

Acronyms

f_r	Rebuild Frequency.
r_c	Cutoff Radius.
r_i	Interaction Length.
r_s	Verlet Skin.
s	Verlet Skin Factor.
t_r	Rebuild Interval.
AI	Artificial Intelligence.
ALL	A Loadbalancing Library.
AoS	Array of Structs.
AoSoA	Array of Structures of Arrays.
API	Application Programming Interface.
AVX	Advanced Vector Extensions.
AVX-512	Advanced Vector Extensions 512 Bit Extensions.
CFD	Computational Fluid Dynamics.
CPI	Cycles Per Instruction.
CPU	Central Processing Unit.
CRTP	Curiously Recurring Template Pattern.
CSF	Cell Size Factor.
DEM	Discrete Element Method.
DSL	Domain Specific Language.
ESA	European Space Agency.
FLOP	FLoating Point Operation.
FMM	Fast Multipole Method.
GPU	Graphics Processing Unit.
HPC	High Performance Computing.
L2L	Local to Local.
L2P	Local to Particle.
LEO	Low Earch Orbit.

lowess	Locally Weighted Scatterplot Smoothing.
LRZ	Leibniz-Rechenzentrum.
M2L	Multipole to Local.
M2M	Multipole to Multipole.
MD	Molecular Dynamics.
MPI	Message Passing Interface.
MSR	Machine State Register.
NASA	National Aeronautics and Space Administration.
Newton3	optimizations exploiting force symmetires using Newton's third law of motion.
ODE	Ordinary Differential Equation.
P2M	Particle to Multipole.
P2P	Particle to Particle.
RAPL	Running Average Power Limit.
SAT	Boolean Satisfyability Problem.
SCCS	Scientific Computing in Computer Science.
SIMD	Single Instruction Multiple Data.
SoA	Structure of Arrays.
SPH	Smooth Particle Hydrodynamics.
SSE	Streaming SIMD Extensions.
SVE	Scalable Vector Extension.
t-SNE	t-Distributed Stochastic Neighbor Embedding.
VTK	Visualization Toolkit.

1 Introduction

Numerical simulations are essential for a wide range of fields of research as well as industry. Fundamentally, such simulations are computational processes that evaluate mathematical models, often built on differential equations, to predict the state or behavior of physical systems or processes. Their purposes range from creating such predictions under various or uncertain conditions to gaining insights into complex phenomena that would otherwise be difficult or impossible to study or replicate to helping optimize and design processes by reducing the dependency on physical models. There exists a wide range of simulation techniques like finite elements, volumes, or differences [Ame14], and also mesh-free techniques like particle simulations. The latter, which shall be the simulation technique discussed in this thesis, again can be subdivided into a multitude of methods that can be applied to an even greater variety of problems that will be touched upon in Subsection 2.1.2.

Particle simulations are notoriously computationally expensive but highly relevant and popular, which justifies using the world's largest supercomputers to tackle the problems they are solving. This becomes clear when looking at the Gordon Bell Prize, one, if not the most important award in the High Performance Computing (HPC) world, which "is awarded each year to recognize outstanding achievement in high-performance computing"[1]. Over the last ten years, three of the winning groups worked on particle simulations, or more specifically on Molecular Dynamics (MD).

While there exist some popular simulator codes, there is always a need and drive for customized solutions. On GitHub alone, there are over 1 000 projects that match the search term "particle simulation" which were actively worked on in 2024[2]. Therefore, people are forced to re-implement core logic, conduct optimizations, and make fundamental design decisions repeatedly, which is only desirable in an educational environment. This is exacerbated because simulators for specialized requirements are often written by application engineers who are often not trained to judge the most suitable architecture or algorithms.

With this thesis, we present a novel approach: We implemented the library AutoPas, consisting of algorithms and data structures that usually sit at the core of a particle simulation to be the foundation of such simulations. The library follows a black box approach so the user does not have to worry or even know about the specifics of the employed algorithm. Then, our library applies automated dynamic algorithm selection to tune its internal configuration at runtime to the current state of the simulation at hand to achieve an optimal time to solution.

[1] `https://awards.acm.org/bell` Accessed: 20.12.2024

[2] `https://github.com/search?q=pushed%3A2024-01-01..2024-12-31+particle+simulation&type=repositories` Accessed: 20.12.2024

1.1 Focus and Objectives

We are interested in the optimal performance of general short-ranged particle simulations. Or to put it in another way, any application of particle interaction algorithms. Hence, as so often, the *No Free Lunch* theorem applies [WM97, Wol02]. It states that when we apply all solution algorithms to all problem instances, their average performance will be the same. Thus, this motivates us to look for and exploit *performance complementarity*. That is, applying only the optimal algorithm for a given problem instance exploits each one's strengths while mitigating others' weaknesses. The problem is how to decide which algorithm will perform best. This leads us to the main research questions of this thesis:

1. ***Is there a feasible Application Programming Interface (API) interface for all short-ranged particle interaction algorithms?***

   As a prerequisite to a qualitative assessment of different algorithms, a software solution has to be created that implements these algorithms in the same software environment for maximal comparability, as well as the ability to switch between them at runtime. To the best of our knowledge, no framework, library, or toolkit exists that implements a wide range of algorithms for particle interactions, like those presented in Subsection 2.1.3, and has the ability to freely switch between them while targeting HPC applications. Looking closer at this problem, this immediately poses many more questions such as: How should underlying data structures be organized for convertibility? Is it feasible for different algorithms to coexist and interact in a heterogeneous simulation? Does this lead to any synchronization requirements, e.g., when considering the rebuilding of lists?

2. ***Can an automated dynamic selection of the short-ranged particle interaction algorithm bring advantages for particle simulations?***

   The important keyword here is *dynamic*, since, assuming there is no global optimal algorithm and the simulation is not perfectly oscillating between two or more optima, static algorithm selection must yield advantages. Evaluating and adapting the interaction algorithm during the simulation on the fly is not free, so it is interesting to investigate this overhead against the potential gain in time to solution. This question targets more the theoretical perspective purposefully does not specify the type of particle simulation since we later want to see if this applies to arbitrary short-range particle simulations. Previous work by us and affiliates suggests potential in the idea which led to this question [TSH+18, GST+19, Tch20, Sec21].

3. ***Is this approach practical and delivers performance while general enough to extend beyond its application in MD?***

   To investigate the questions 1 and 2, we must create a software library that implements the underlying ideas. This must then be integrated into several HPC applications. Using this, we want to evaluate the usability of our API and investigate which kind of applications this approach is feasible for and which can

actually benefit from it. Due to the specific academic environment in which this thesis came to be we are initially mostly interested in MD simulations for process engineering [NSS+23]. However, it is important for us to show that this library is not tailored to MD but a performant abstraction layer that can serve as the base for arbitrary particle simulations.

The focus and core contribution of this work are discussing the pros and cons of algorithms for short range particle interactions, introducing the automated algorithm selection approach to the simulator domain, and finally the conceptualisation and implementation of the library AutoPas.

1.2 Structure of this Thesis

The thesis structure is aligned with the above-described focus and research questions. After the Introduction, Chapter 2 will lay out the theoretical foundations required to approach the topic. This chapter is divided into two major parts. First, particle simulations are discussed, from their general formulation to actual applications and how to implement them efficiently. Second, the algorithm selection problem is presented, defined, and characterized.

With the theory now familiar, Chapter 3 presents the library AutoPas. Its software architecture, interfaces, algorithm design, and hardware optimizations are shown. Then, it is shown how the theoretical concepts from the automated algorithm selection are implemented to the dynamic auto-tuning mechanic of AutoPas.

This implementation is then put to the test in Chapter 4, integrated into four simulators, two established ones and two that were developed as a side effect of this thesis. The performance of AutoPas, its implementations, algorithms, and tuning behavior in real simulation scenarios is tested, analyzed, and discussed.

Each of these chapters concludes with an interim summary to take a step back to summarize and reflect on the presented insights.

Finally, Chapter 5 summarizes everything that was brought forth, relates it to the research questions posed in Chapter 1, and formulates detailed answers.

2 Background

This thesis brings together two fields of research, whose theoretical pillars shall be explored in this chapter.

Section 2.1 gives a description of general particle simulations. We start with their fundamental principles, major categorization, and intricacies of their boundary treatments. With this, three standard particle simulation methods are explained that build on these fundamentals, their areas of application, and their similarities and differences. Next, the theoretical formulations of the algorithms for interaction partner identification are presented, which is the core of any particle simulation. Their advantages and disadvantages are highlighted through mathematical models, and the data structures they build are discussed. Continuing, we pick out important algorithmic parameters and design choices and analyze them based on benchmark implementations. The analysis of theoretical algorithms is concluded by presenting the concepts, pitfalls, and potentials of shared memory as well as instruction-level parallelism.

The second field is the algorithm selection problem presented in Section 2.2. First, the problem itself is defined from a mathematical point of view. It is then extended to its automation and approaches to tackle its computational complexity sketched. Finally, related problems are discussed, and their differences are pointed out or explained as to why they can be considered special cases of the initial problem.

2.1 Particle Simulations

Particle simulations, in general, are a vast field with numerous applications, far too many to exhaustively cover in this thesis. Some examples comprise astrophysics [Spr10], high energy events in material sciences [SS04, Lee06, YOYS14], process engineering [VKFH06, SNK+13, WHH17], transport systems of powder [SS22] or fluids [WZFD22], or space debris simulation [GGBI22]. The following section shall serve as an introduction to the topic, starting out in Subsection 2.1.1 with the governing principles of any particle simulation. Using these, Subsection 2.1.2 discusses simulation methods in general and their applications. From there, we come back to one of the core topics of the thesis, namely the underlying efficient algorithms employed to make these kinds of simulations possible, which will be presented in Subsection 2.1.3 and discussed in Subsection 2.1.4. Even though the focus of this thesis lies on simulations for short-range particle interactions, the concepts and how to efficiently compute long-range interactions is discussed briefly.

2.1.1 Fundamentals

The fundamental mechanisms of any particle simulation are the evaluation of particle interactions and their position propagation.

In general, all interactions that do not involve relativistic velocities and which occur in the so-called middle realm of physics, which covers the ranges from about 10^{-9} to 10^{22} cm fall into two categories: Short- and long-range interactions [Mic79]. The former decay quickly with the distance of the interacting particles and thus can be cut off beyond some threshold. The latter can never be disregarded because of their slow decay, no matter the interaction distance. In literature, there are several definitions for the distinction between short- and long-range interactions of varying complexity, of which three shall be presented. For example, a potential $U(r)$ is considered long-range if

- it decays slower in r than $1/r^d$ where d is the dimension of the problem [GKZ07], or
- its range $R = \hbar c/|U(r)|$, is infinite, where $\hbar$ is the Planck constant and c the speed of light [Mic79], or
- the integral $\int U(r)\mathrm{d}r$ does not converge [Kab12].

2.1.1.1 Short-range Interactions

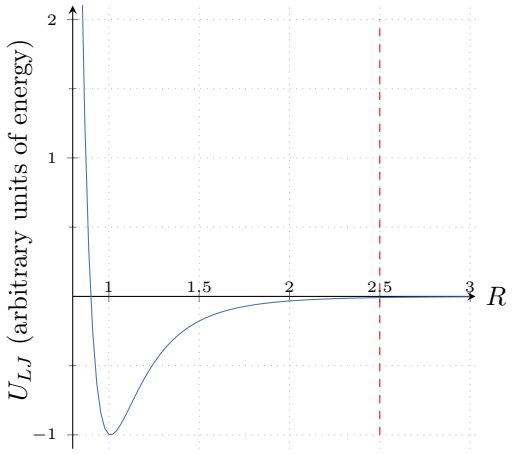
Short-range interactions are the most critical part of a particle simulation because they are present in every type of simulation and typically take up significant parts of the computational load due to their tricky computational complexity. Naively, all particles interact with all other particles in the simulation, leading to a computational complexity of $O(N^2)$ with N being the total number of particles. However, as by the definition of short-range interactions, interactions beyond a certain distance can be neglected. This distance is called the Cutoff Radius (r_c). Therefore, the complexity is reduced to $O(N \cdot N_{r_c})$, where N_{r_c} is the number of particles within r_c around one particle. Since $N_{r_c} \ll N$ the complexity of short-range algorithms is typically given as $O(N)$ [RBMC96, GKZ07]. However, it must be noted that for a fixed domain size and r_c, increasing N would also increase N_{r_c}. Thus, this factor can not be wholly disregarded as constant.

A widely used example for a short-range interaction is the Lennard-Jones 12-6 potential given by Equation 2.1 [LJ24, LJ31].

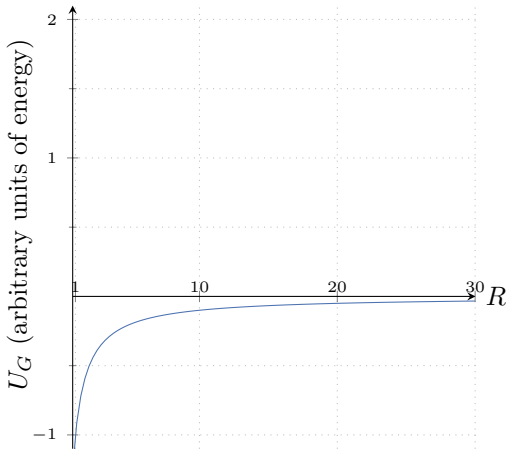
$$U_{LJ}(r) = 4\epsilon \left(\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right) \tag{2.1}$$

Where r is the distance between the two particles, and σ and ϵ are size and energy parameters from the molecular model. The potential is a model for the combination of the major intermolecular forces: The Pauli repulsion or Pauli exclusion principle [Pau25], represented by the minuend, and the attractive part of van der Waals forces, especially the London dispersion [Lon30], represented by the subtrahend.

Due to its high negative exponents, the potential converges quickly towards zero, as seen in Figure 2.1a. Also seen in this figure is a typical value for r_c at 2.5.



(a) Lennard-Jones potential with $\epsilon = 1$, $\sigma = 0.9$, and r_c=2.5 (red line)



(b) Gravity potential with $G = 1$ and $m = 1$

2.1.1.2 Long-range Interactions

In many particle simulations, especially MD, the question often is whether there is a long-range interaction, but whether it is important for the behavior that shall be observed. If it is relevant, long-range interactions are challenging in a different way than short-range ones because, for every particle, the influence of every other particle has to be considered. For this not to lead to the naive complexity of $O(N^2)$, algorithms that exploit tree-like structures can be applied to bring the complexity down to $O(N \log N)$ or even $O(N)$. It needs to be noted that these algorithms lead to approximations with arbitrary precision, usually dictated by the degree of chosen underlying polynomes [Tch20]. Usually, all long-range algorithms include at least some steps of a short-range algorithm for particles close to each other. This fall-back comes from the fact that for short distances, potentials are always more intense, as all known physical potentials $U(r)$ adhere to $\lim_{r \to \text{inf}} U(r) = 0$ and thus should not be approximated over short distances.

The most prominent example of a long-range interaction is the gravity potential given in Equation 2.2 [New87].

$$U_G(r) = -\frac{G m_1 m_2}{r} \tag{2.2}$$

Here, G is the gravitational constant, m_i the point masses of the involved bodies, and r their separation. Since this potential only decays with $1/r^1$, it meets any of the definitions for long-range interactions listed in Subsection 2.1.1. The stark contrast to a short-range interaction can also be seen from the speed of decay of the potentials in the comparative plots Figure 2.1b and Figure 2.1a.

2.1.1.3 Newton's Third Law of Motion

All non-fictitious forces in the aforementioned middle-realm of physics obey Newton's third law of motion. It states that for every force, there exists a force of equal magnitude

in the opposite direction [New87]. For interactions between two particles i and j, which are based on such forces, this means:

$$F_{ij} = -F_{ji} \tag{2.3}$$

Where F_{ij} is the force exerted from i on j and F_{ji} the other way around. This relationship offers a great source for optimization in particle simulations, as this means that for every pair of particles, the force magnitude only has to be computed once, thus cutting the computations necessary in half. This technique is also referred to as optimizations exploiting force symmetires using Newton's third law of motion (Newton3).

2.1.1.4 Particle Propagation

The purpose of particle simulations, particularly in contrast to mesh-based methods, is to simulate the movement of particles. This movement is typically enabled by assigning each particle a velocity attribute, which can be affected by global force fields or interactions between particles. The computation of movement is then carried out by solving Newton's equations of motion, which are treated as ordinary differential equations:

$$\vec{a} = \dot{\vec{v}} = \ddot{\vec{x}} = \frac{1}{m}\vec{F} \tag{2.4}$$

Here, $\vec{a}$ is the acceleration, $\vec{v}$ the velocity, $\vec{x}$ the position, m the mass of a particle, and $\vec{F}$ the force it is experiencing. Since this has to be solved for every particle, we get a system of $D \cdot N$ Ordinary Differential Equations (ODEs) of second order, with N being the number of particles, and D being the number of dimensions. This can be reformulated into a system of $2 \cdot D \cdot N$ ODEs of the first order:

$$\dot{\vec{r}} = \vec{v} \tag{2.5}$$

$$\dot{\vec{v}} = \frac{1}{m}\vec{F} \tag{2.6}$$

Several different numerical schemes are conceivable to solve these, offering various advantages to consider depending on the application. Typically, the schemes revolve around the Runge-Kutta approach [Run01, Kut01], which offers excellent precision but is, as most implicit schemes, rather compute expensive, or (Velocity-)Störmer-Verlet [Stö07, Ver67], which is cheaper to evaluate and symplectic. This means that it preserves the Hamiltonian properties of a system, or more intuitively in the context of particles, it will preserve the orbit of a particle without losing or gaining energy. Using cheap symplectic integrators leads to a trade-off where there might be errors in the particle's position. However, the overall statistical properties of the system are physically correct, which is fine for most applications. For applications where the (numerically) exact particle positions are relevant, specialized integrators can be employed [KMT94, BI21].

Leap-Frog A widely used time integration scheme based on the Störmer-Verlet approach, and mathematically equivalent, is the Leap-Frog method [VKBP02, HLW03, GKZ07].

$$\vec{v}_{t+\frac{\Delta t}{2}} = \vec{v}_{t-\frac{\Delta t}{2}} + \Delta t \vec{a}_t, \tag{2.7}$$

$$\vec{r}_{t+\Delta t} = \vec{r}_t + \Delta t \vec{v}_{t+\frac{\Delta t}{2}} \tag{2.8}$$

The advantage of this method, as shown in Equation 2.7 over the classic Velocity-Störmer-Verlet, is that it does not require the storage of values of more than one time step. If higher precision of the results is needed, the order of the method can be raised significantly by evaluating more intermediate time steps [Yos90].

Time step Whenever numerical (time) integration is applied, the question of the (time)step width is crucial for the method's accuracy. Especially in particle simulations, this is highly critical, as the integrators are of low order and lead to accumulating errors [Kim14, Kim15]. A theoretical upper bound can be derived using Nyquist-Shannon sampling theorem [Nyq28], which states that to fully capture a signal, it must be sampled at a frequency at least twice as high as its highest frequency component. In the case of MD, specifically biochemistry, the fastest motions come from vibrations of hydrogen bonds with a period of about ten femtoseconds (fs) [ADPK23]. Thus, the upper limit is five fs. However, it is general practice to go even lower, with GROMACS[1] and LAMMPS[2] using one fs as defaults.

2.1.1.5 Boundary Treatment

Any particle simulation, occurs in a defined region called the simulation domain. Since particles might wander towards the boundaries of these domains or even cross them, it is necessary to consider what should happen in these cases. Of course, the desired behavior is highly dependent on the actual type or scenario of the simulation, and even combinations can be relevant. These so-called boundary treatments or boundary behaviors have to consider two things. What happens to the interaction calculation of particles near the boundary, and what happens if a particle crosses the boundary. Thus, boundary conditions can be classified into a few categories that can be summarized in two groups:

Relocating Boundaries This group of boundary conditions adds or removes particles from the simulation. Particles might also be removed and reinserted in a different location. The crucial thing is that these boundary conditions affect any spatially-aware data structure that stores the particles.

Outflow Also called open or absorbing boundary conditions, particles are deleted as soon as they cross the threshold [GKZ07]. They are useful to simulate outlets or are a convenient default for simulations where particles never should leave the simulation domain [GGS22].

[1] `https://manual.gromacs.org/documentation/current/user-guide/mdp-options.html` Accessed: 20.12.2024

[2] `https://www.afs.enea.it/software/lammps/doc19/html/timestep.html` Accessed: 20.12.2024

Inflow A boundary of a simulation can (conditionally) create new particles into the domain, effectively creating a stream of inflowing particles. These are often coupled with outflow boundaries to maintain the amount of particles in the system [HV19].

Periodic If the simulation aims to capture the interior of a vast system, periodic boundaries can be employed. They mimic an infinitely large system by replicating part of the domain that is directly on the opposite side of the domain, creating a wraparound effect like the surface of a torus. In other words, in 1D, anything on the left end of the domain interacts with the right end of the domain. This does not only mean interactions have to happen across the wrapped boundaries, but also particles that cross them have to be relocated to the other side of the domain [RBMC96]. Often, especially in large simulations that are computed on distributed compute resources, this is implemented with a so-called halo layer. In this, copies of particles from the other side of the system are stored and used as interaction partners. These particles are here referred to as halo particles, or sometimes in literature as ghost particles.

Interacting Boundaries These boundary conditions do not simply relocate particles but interact with them, e.g., via a potential to smoothly enforce a boundary condition. Typically, when used independently, they do not intend for particles to cross the domain boundaries and act from a behavioral perspective as reflective boundaries.

Dirichlet Also called fixed boundary conditions, variables like, e.g., velocity, are set to a constant value at the domain's boundary. For particle simulations, this could be a set of particles on the boundary whose velocity is always kept at zero, such that they act as a wall for other particles to not leave the domain.

Neumann A more sophisticated way to implement reflective boundaries is akin to Neumann boundary conditions, which specify the derivative of the evaluated function, e.g., the force enacted on a particle. This can be achieved by placing mirror particles to simulate a repulsive wall [GKZ07].

2.1.2 Simulation Methods

Many different kinds of simulations can be conceived using the toolset established in Subsection 2.1.1. In this thesis, we do not aim to simulate one specific physical process but look at particle simulations in general from a more abstract perspective. We appreciate the various methods of applying the particle simulation technique, as, for example, listed in the introduction of Section 2.1, but from the perspective of the research questions this thesis addresses, they all boil down to computing short-range pairwise interactions between particles and propagating them. Nevertheless, to achieve a better understanding of context, different short-range particle simulation methods shall be discussed. To avoid a too extensive detour from the main topic of this thesis, only a few methods are briefly explained, which serve as examples to give an idea about the principles, capabilities, limitations, and applications of particle simulations. It should illustrate variety but also what these simulation methods have in common.

2.1.2.1 Molecular Dynamics

MD is the simulation technique to study the physical behavior of large numbers of individual atoms or multi-site molecules. Over a given timeframe, the potentials between particles are evaluated, and from those, their movements are computed using Newton's equations of motion, as discussed in Subsection 2.1.1.4. The evaluated forces can be short- or long-range and usually originate either from classical interatomic potentials, like the Lennard-Jones potential discussed in Subsection 2.1.1.1 or the Coulomb potential [Cou85], or quantum mechanics, e.g. looking at the degrees of freedom of each electron to mitigate the Born-Oppenheimer approximation [BO00, KMV22]. Observing the particles' trajectories over time provides insights into the dynamic behavior and evolution of the overall system.

The areas of application for MD are broad and spread over different fields of science. Examples include, neither in an exhausting list nor any particular order, drug development in medicine [DM11, KBC20], analysis of protein folding in molecular biochemistry [SKL07, FHLS10], but also studies of material properties in thermo dynamics [NSS+23], or microscopic processes when a laser is fired on metal [WZ14].

However, MD also comes with limitations, the most severe being limitations in scale and time. For example, we can look at the theoretical, minimal computational costs to simulate every molecule of a drop of water for one second. To derive the number of particles involved, we calculate the molar mass, $\mathtt{m}$, of water, which is the sum of two hydrogen and one oxygen atom:

$$\begin{aligned} \mathtt{m}_{Water} &= 2\mathtt{m}_H + \mathtt{m}_O && (2.9)\\ &= 2 \cdot 1.008\, g/mol + 15.999\, g/mol && (2.10)\\ &= 18.015\, g/mol && (2.11) \end{aligned}$$

Next, we compute the number of moles, n per drop. For the mass of the drop, we take the widely used metric drop [Jac21], $m_{drop} = 0.05\, g$:

$$\begin{aligned} n_{Drop} &= \frac{m_{drop}}{\mathtt{m}_{Water}} && (2.12)\\ &= \frac{0.05\, g}{18.015\, g/mol} && (2.13)\\ &= 0.002775465\, mol && (2.14) \end{aligned}$$

Using this result and Avogadro's constant N_A [Mil13], we arrive at the number of H_2O molecules in a drop of water N_{Drop}.

$$\begin{aligned} N_{Drop} &= N_A \cdot n_{Drop} && (2.15)\\ &= 6.02214076 \cdot 10^{23}\, 1/mol \cdot 0.002775465\, mol && (2.16)\\ &= 1.67142409 \cdot 10^{21} && (2.17)\\ &\approx 1.67 \cdot 10^{21} && (2.18) \end{aligned}$$

Now, if even individual atoms should be simulated, this number has to be multiplied by three for the three atoms in each molecule, so we arrive at about $5 \cdot 10^{21}$ particles. For comparison, the largest particle simulation known at the time of writing this thesis consisted of $3 \cdot 10^{13}$ particles [CLZ+21]. The primary limitation for size is the available amount of main memory in a computer system. Assuming that a minimal particle consists of the three vectors for position, force, and velocity, each with three double precision floating point values plus one ID, all being 64 bit numbers, this particle has a memory footprint of 640 bit or 80 byte. To store just the particles in the drop of water, $4 \cdot 10^{23}$ Byte, which is $4 \cdot 10^{8}$ Petabyte or 400 Zettabyte, would be required. The currently largest supercomputer Frontier[3] has shy of 10 Petabyte of combined main and Graphics Processing Unit (GPU) memory[4]. So, we would need more than 40 million instances of this machine to accommodate just the memory for the minimal particle model of one drop of water.

2.1.2.2 Discrete Element Method

At first glance, Discrete Element Method (DEM) is similar to MD, as interacting particles are modeled and moved via Newton's equations of motion. The central modeling aspect that distinguishes DEM from (most) other particle methods is that particles are not viewed as points but objects with volume, geometry, and thus rotational degrees-of-freedom, necessitating the use of the Newton-Euler equations for torque and rotation [Eul65]. This focus on contact interactions means that the method is usually used with short-range only, although applications involving long-range interactions, usually electrostatic forces, have been published [PWE+15, LMLY10].

Primarily, contact interactions are considered and evaluated based on the Hertzian Law [Her81], which models the contact force of two colliding deforming, soft particles from the point when their volumes overlap [WA20]:

$$F_{Hertz} = \frac{4}{3} \underbrace{\frac{1}{\frac{1-\nu_i^2}{E_i} + \frac{1-\nu_j^2}{E_j}}}_{\text{Young's modulus}} \sqrt{\underbrace{\frac{1}{\frac{1}{R_i} + \frac{1}{R_j}}}_{\text{Effective radius}}} (\delta_i + \delta_j)^{\frac{3}{2}} \tag{2.19}$$

Here, δ_i, δ_j ara the distance from the particles' centers to the contact surface, R_i, R_j the particles' radii, E_i, E_j their elastic moduli, and ν_i, ν_j their Poisson ratios. The latter two are material properties related to their deformability. Depending on the application, this model has to be extended to include further attractive F_a or dissipative forces F_d so that the overall contact force is composed as shown in Equation 2.20.

$$F = F_{Hertz} - F_a + F_d \tag{2.20}$$

[3] `https://www.top500.org/system/180047` Accessed: 20.12.2024

[4] `https://docs.olcf.ornl.gov/systems/frontier_user_guide.html#frontier-compute-nodes` Accessed: 20.12.2024

This allows for arbitrarily complex contact models involving drag, shear, lift, or even heat generation and conduction [Lud08]. Therefore, the DEM lends itself naturally to engineering applications like the simulation of granular flows [ZS13, GZ14, EEZS+21], powder [SLBT04, PYJ+17, TSYN21], or rock mechanics [JH02, MCG10, HSD12]. These applications often involve coupling to other simulation techniques like computational fluid dynamics, discrete fracture networks, or the finite element method to mitigate the same difficulties that MD, namely limits in the number of particles and time steps due to computational effort.

2.1.2.3 Smooth Particle Hydrodynamics

The idea of Smooth Particle Hydrodynamics (SPH) is that the collective movement of a group of particles is similar to the flow of liquid and can be modeled via the Navier-Stokes equations. A system is represented by a cloud of particles parametrized to represent given materials. Gaussian-like smoothing kernels are applied on these particles, leading to smoothed densities, from which fluid densities can be calculated via equations of state. From these densities, pressure gradients can be derived, which are used to calculate the acceleration of the particles. This means that the movement of the particles represents the movement of the fluid, and higher resolution is automatically directed towards regions with higher pressure because this is where more particles are present [LL10, Yas17].

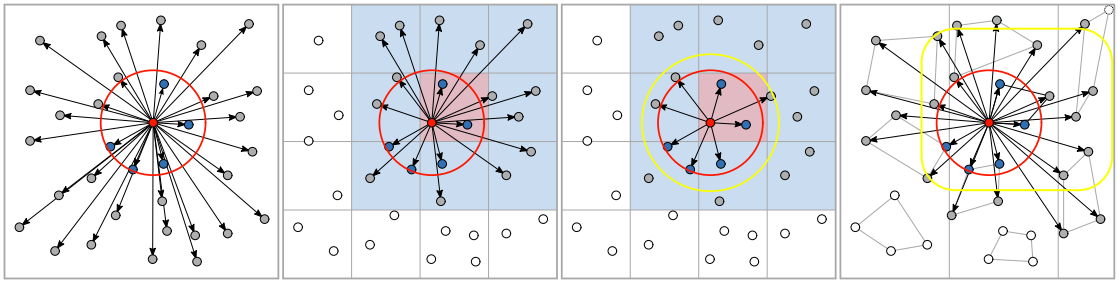
The method was developed initially for astrophysics [Luc77, GM77], where it is still in use for various scenarios [Spr10]. Other fields of application include studies of fluids in hydrodynamics [HLYK08, HA09], biomedical applications due to its ease of use with moving and deforming structures in fluid-structure interations [Tom17], or computational mechanics to study explosions [LLLZ03].

SPH is an alternative to mesh-based methods usually employed in Computational Fluid Dynamics (CFD) [TYLT23], over which it has some advantages like inherent mass conservation since the particles themselves model mass. A wide range of method variants has been proposed to enhance precision for, e.g., conservation of linear and angular momentum or discontinuities like shock waves[LL10, ZZW+22]. Furthermore, it is a complex problem to create a suitable mesh for complicated geometry or scenarios with deformable boundaries or crack analysis [HKB+21].

From the perspective of general particle simulation, one curiosity about SPH is that in contrast to other methods that simulate physical potentials, the smoothing potentials in SPH are not necessarily symmetric between particles, which means they do not adhere to Newton's third law of motion [BRP05].

2.1.3 Efficient Algorithm Archetypes for Interaction Partner Identification

As discussed in Subsection 2.1.1, to compute the effect of an interaction on one particle, all its interaction partners within r_c must be identified. The specific implementation of the short-range neighbor identification algorithm might differ in detail depending on the simulation method and software package. However, they all build upon one of a few main algorithm archetypes: Direct Sum, Linked Cells and Verlet Lists.



(a) Direct Sum **(b)** Linked Cells **(c)** Verlet Lists **(d)** Verlet Cluster Lists

Figure 2.2: Neighbor identification algorithms for short-range force calculations. The red circle symbolizes the Cutoff Radius r_c. Particle colors indicate the relation to the particle of interest:
Red: Particle for which interactions are calculated.
Blue: Particles inside the red-colored particle's cutoff radius.
Gray: Potentially in range of the red particle, therefore, the distance needs to be calculated. Arrows indicate when this is needed in every time step.
White: particles that are out of range and which are, thus, omitted.

Algorithm 1: Direct Sum

```
1 for p1 : particles do
2     for p2 : particles do
3         if p1 != p2 and distance(p1, p2) < cutoff then
4             interaction(p1, p2)
```

Algorithm 2: Linked Cells

```
1 for cell1 : cells do
2     for cell2 : neighborCells(cell1) do
3         for p1 : cell1.particles do
4             for p2 : cell2.particles do
5                 if p1 != p2 and distance(p1, p2) < cutoff then
6                     interaction(p1, p2)
```

Algorithm 3: Verlet Lists

```
1 for p1 : particles do
2     for p2 : p1.neighborList do
3         if distance(p1, p2) < cutoff then
4             interaction(p1, p2)
```

Algorithm 4: Verlet Cluster Lists

```
1 for cluster1 : clusters do
2     for cluster2 : cluster1.neighborList do
3         if distance(cluster1, cluster2) < cutoff then
4             for p1 : cluster1 do
5                 for p2 : cluster2 do
6                     if distance(p1, p2) < cutoff then
7                         interaction(p1, p2)
```

2.1.3.1 Direct Sum

The naive approach of looping over all particles with a double loop as sketched in Algorithm 1 is not necessarily an efficient algorithm for the pairwise neighbor identification as it scales with $O(N^2)$, quickly leading to too many distance checks as can be seen in Figure 2.2a. However, for low numbers of particles (less than 1 000), it is a reasonable choice due to its non-existing overhead from data structures or additional algorithm steps, as well as its great potential for vectorization.

2.1.3.2 Linked Cells

Also referred to as Cell Lists [RBMC96], the idea is to reduce the region that has to be searched by storing the particles in a data structure that retains (some) structural information of the particle distribution.

Logical Data Structure The domain is partitioned into a usually uniform, but not necessarily equilateral, grid of cells, each with a mesh size larger or equal to the cutoff radius. As illustrated in Figure 2.2b, particles are organized into these cells. The Linked Cells algorithm identifies the neighbors of the red particle by examining its cell (the red cell) and the surrounding cells (the blue cells). It then computes the pairwise distances to all particles within these colored cells, as indicated by the arrows, and only evaluates the interactions to those within the cutoff (blue particles).

Computational Complexity With this reduction to only a constant number of neighbor cells, the computational complexity for homogeneous scenarios is reduced to $O(N \cdot N_{cell})$, where N_{cell} is the average number of particles per cell. As explained above, this can be considered $O(N)$. For highly inhomogeneous scenarios, mainly when the bulk of particles is situated in only a few cells, the processing of individual cell pairs, which is in $O(N_{cell}^2)$ can become problematic.

Advantages from the position-aware cell structure The spatial cell data structure brings many distinct advantages. It has a very low memory overhead, as theoretically,

only the pointers to the start of each cell need to be stored. Thus, the memory consumption is independent of the number of particles. Storing all cells in a container with random access like `std::vector` allows accessing all neighbor cells of any cell, as done in Algorithm 2 line 2, in $O(1)$ since it boils down to a simple index calculation. This also means that as long as the cells are aligned with the coordinates and size of the bounding box of the domain, logical separation of halo and owned particles happens implicitly. Consequently, particles close in the simulation domain are also close in memory. That is an advantage because, as seen in Algorithm 2, particles are processed cell by cell, meaning accessing all particles within one cell is highly cache efficient. Also, assuming cells fit comfortably into the cache, all particles of *cell1* are kept in the cache until the cell is fully processed. This cache-friendly memory layout lends itself to Single Instruction Multiple Data (SIMD) vectorization efficiently [Fom11].

Disadvantages from the cell structure Due to its grid structure, Linked Cells also creates some overhead. The spatial layout, namely the sorting of particles into their correct cells, needs to be updated regularly. Though this can be done in $O(N)$ since calculating the correct cell and accessing it can both be achieved in $O(1)$ for each particle, care must be taken not to trigger significant memory reallocations. Another disadvantage immediately becomes visible when looking at Figure 2.2b and comparing the blue area against the red circle. Our region of interest, the cutoff region, is approximated by a rectangle in 2D or even a cuboid in 3D. Assuming particles are distributed uniformly, and cells have an edge length of r_c, this means that, as we can see in Equation 2.21, only less than 16% of particles for which distances are calculated are within the cutoff region. This ratio is also referred to as the hit rate:

$$h_{LC} = \frac{V_{r_c}}{V_{searched_{LC}}} = \frac{\frac{4}{3}\pi r_c^3}{(3r_c)^3} \approx 0.155 \tag{2.21}$$

The third disadvantage comes from treating cells as the primary key of the data structure, and thus more important than what is actually of interest, namely the particles. When iterating over all particles, all cells must be checked, regardless of whether they are empty or not. In highly inhomogeneous scenarios, this overhead can grow noticeable.

2.1.3.3 Verlet Lists

This algorithm archetype, also sometimes called neighbor lists, initially works without any spatial structuring of the simulation domain. For each particle, lists are created, which contain all neighbor particles within r_c. Then, during the calculation of the pairwise interactions, only all neighbor lists need to be processed, as illustrated in Algorithm 3. Since particles only move small distances between two-time steps, as explained in Subsection 2.1.1, these lists can be reused over several time steps by increasing their size by a so-called Verlet Skin Factor (s). This way, particles about to enter the cutoff sphere before the next rebuild of the lists are captured, as visualized in Figure 2.2c. The interval in time steps for which the lists are kept is here referred to as the Rebuild Interval (t_r) or its inverse, the Rebuild Frequency (f_r) [Ver67].

Rebuilding Neighbor Lists Since the Verlet Lists algorithm works by wrapping the crucial question of identifying the neighbors away into simply using lists, the question arises how to efficiently build these lists. A typical choice is to employ a Linked Cells algorithm for this, which acts as a bucket-sort for rebuilding the lists.

Computational Complexity To judge the computational complexity of Verlet Lists, the two major algorithm steps need to be evaluated: The pairwise force calculation and the rebuilding of the neighbor lists. For the former, a complexity of $O(N \cdot N_{list})$ where N_{list} is the average size of a neighbor list, is achieved because, for all N particles, N_{list} number of elements need to be evaluated. Here, a similar argument as for Linked Cells can be applied, so this step can be considered in $O(N)$. Since the rebuilding is done with the Linked Cells algorithm, which is in $O(N)$, the whole Verlet Lists algorithm is in $O(N)$.

Advantages from List Structure The immediate advantage of knowing which particles are close, even if the list contains some particles outside the cutoff sphere, is the very high hit rate due to the approximation of this sphere via a slightly wider sphere.

$$h^*_{VL} = \frac{V_{r_c}}{V_{searched_{VL}}} = \frac{\frac{4}{3}\pi r_c^3}{\frac{4}{3}\pi (r_c \cdot s)^3} = \frac{1}{s^3} \tag{2.22}$$

However, rebuilding the lists also has to be considered, which we can do by calculating the weighted average of a rebuilding and a non-rebuilding iteration. The number of iterations for which the lists are valid is given by Rebuild Interval (t_r).

$$h_{VL} = \frac{(h^*_{VL} \cdot t_r + h_{LC})}{t_r} = \frac{\left(\frac{1}{s^3} t_r + 0.155\right)}{t_r} \tag{2.23}$$

From this, we can visualize the advantage of Verlet Lists over Linked Cells for a fixed $t_r = 20$, as seen in Figure 2.3. In general, the skin s^* to have the same hit rate between the two algorithm archetypes is shown in Equation 2.24:

$$s^* \approx \frac{1.8616}{\sqrt[3]{\frac{t_r - 1}{t_r}}} \tag{2.24}$$

We can modify Linked Cells to also have a skin, which increases the cell size so cells do not have to be rebuilt every iteration. This changes Equation 2.21, such that the search volume is expanded by s.

$$h'_{LC} = \frac{V_{r_c}}{V_{searched_{LC'}}} = \frac{\frac{4}{3}\pi r_c^3}{(3 r_c s)^3} \approx \frac{1.55}{s^3} \tag{2.25}$$

Comparing against this version of Linked Cells the advantage of Verlet Lists becomes dependent only on t_r as can be seen from Equation 2.26:

$$\frac{h'_{LC}}{h'_{VL}} = \frac{\frac{1.55}{s^3}}{\frac{\frac{1}{s^3} t_r + \frac{1.55}{s^3}}{t_r}} = \frac{1.55 t_r}{t_r + 1.55} \tag{2.26}$$

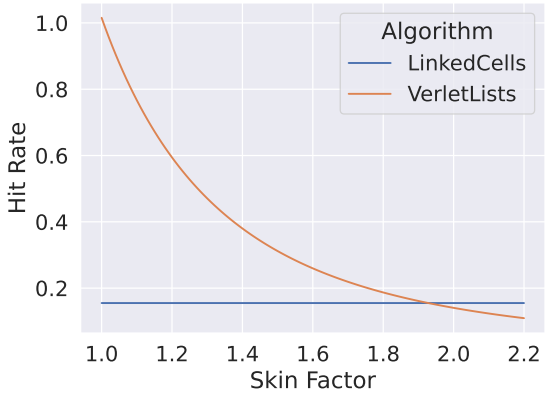


Figure 2.3: Hit rate of Linked Cells (no skin) and Verlet Lists for a fixed Rebuild Interval $t_r = 20$.

Disadvantages from List Structure When processing the neighbor lists, access to the neighbor particles is mostly random access into the particle storage. It can never be fully contiguous access, as this would require resorting of the particles for every neighbor list. This decreases vectorization performance as suboptimal gather and scatter instructions are needed (if available). Furthermore, shared memory parallelization with Newton3 comes with the challenge that there is no inherent knowledge of which neighbors might be shared by some particles, thus making them vulnerable to data races. One way to solve this is through the use of thread buffers[5]. This is discussed further in Subsection 2.1.5.

The most significant disadvantage of Verlet Lists is their high memory consumption, which lies in $O(N \cdot N_{list})$. While this is still in $O(N)$ as discussed above, the factor N_{list} becomes very noticeable here. For example, let us again consider the minimal particle from Subsection 2.1.2.1 consisting of three vectors: position, velocity, and force, as well as one id, each being 64 bit values. The particles in the neighbor list are either referred to by their ID or a pointer, usually both being a 64 bit value. Looking at a scenario with a density of $\rho^* = 0.8442$ and $r_c = 2.5$ (the numbers are from a LAMMPS benchmark[6]), the lists hold on average $\frac{4}{3}\pi r_c^3 * \rho^* \approx 55.25$ particles. This means the storage for the neighbor lists accounts for over $\frac{N55.25 \cdot 64\,\text{bit}}{N(10+55.25)64\,\text{bit}} \approx 84\%$ of the total memory consumption of the simulation.

2.1.3.4 Verlet Cluster Lists

A more complex approach that draws on the ideas of Linked Cells and Verlet Lists is sketched in Figure 2.2d. Here, the idea is to have a binning into 2D towers, which are sorted along the third dimension to preserve some geometric information. Then, subsequent particles in the same tower are grouped into so-called clusters. For these clusters, neighbor lists of clusters are created based on the assumption that particles

[5] `https://docs.lammps.org/Developer_par_openmp.html` Accessed: 20.12.2024
[6] `https://www.lammps.org/bench.html#lj` Accessed: 20.12.2024

that are near each other will have similar neighbor lists. Creating a clean Interaction Length (r_i) for each cluster is costly, as it involves the conjunction of multiple spheres. Instead, as seen in Figure 2.2d, a box-like over-approximation can be constructed. The actual pairwise interaction evaluation then follows Algorithm 4, which again is a mixture of Algorithm 2, with the traversal of a spatial structure, and Algorithm 3 with the list traversal.

Advantages of Clustering The clustering of nearby particles brings several advantages. Memory consumption compared to Verlet Lists can be cut by a factor inverse to the cluster size, $n_{cluster}$ resulting in $O\left(\frac{N}{n_{cluster}} N_{list}\right)$. Furthermore, although memory access of (neighbor) clusters is still random, accessing all particles within a cluster is contiguous in memory, thus opening up decent vectorization opportunities if the cluster size is chosen as a multiple of the vector register size of the hardware.

Disadvantages of Clustering The primary disadvantage is the fact that once two clusters are within r_i, all pairwise distances have to be calculated. This means that in the worst case, one particle of the neighbor cluster might trigger the evaluation of several unnecessary distance calculations, leading to an overall worse hit rate than Verlet Lists as particles from outside r_i will be considered which cannot be within r_c.

2.1.3.5 Fast Multipole Method

Since long-range interaction algorithms are neither the focus of AutoPas nor this thesis, a short overview of the Fast Multipole Method (FMM) shall serve as one example of what can be done on this frontier, to provide a perspective on the differences and challenges to short-range algorithms. Other notable examples include the Barnes-Hut algorithm [BH86], Ewald summation [Ewa21], Particle Mesh Ewald (PME) [DYP93], and the Particle-Particle/Particle-Mesh (P3M) [EHL80] algorithm. A much more extensive discussion of the FMM algorithm, its implementation, variants, and performance can be found in the literature [Kab12, Gra17b, Yok13, Tch20]. The brief idea is to reduce the computational complexity by grouping clusters of well-separated particles into pseudo particles for sources (multipole expansions) and targets (local expansions), interact those, and then obtain the influence on the actual particles from its contribution to its local expansion.

Expansions Conceptually, the combination of particles into multipole and local expansions is shown in Figure 2.4, as well as why they have an area of validity and need to be well separated. Mathematically, the multipole expansions are a sum of coefficients a_i from the particles' influence divided by the distance to the center of the pseudo particle:

$$\sum_{i=0}^{p} \frac{a_i}{r^{i+1}} \tag{2.27}$$

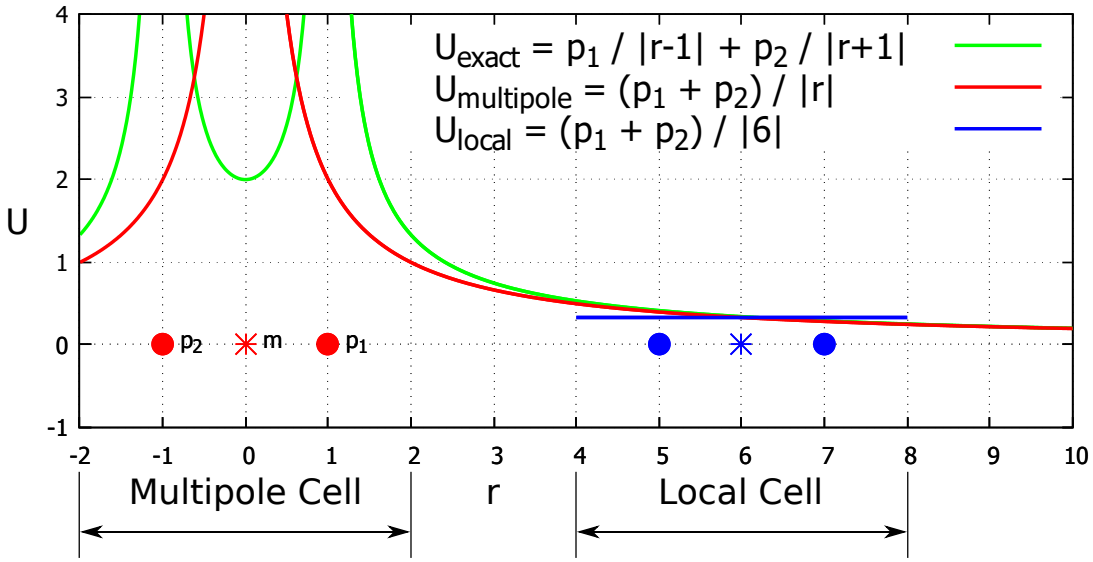


Figure 2.4: Simplified sketch of how multipole and local expansions are used to group particles and approximate their interactions over longer distances. After some distance, the potential of the multipole expansion (red) approximates the accumulated exact potential of p_1 and p_2 (green) well. It can be used to model the influence on the local expansion (blue). The induced error in the individual particle's resulting potential is the difference between the blue and the green line at the r positions of the particles. Source [Gra17b].

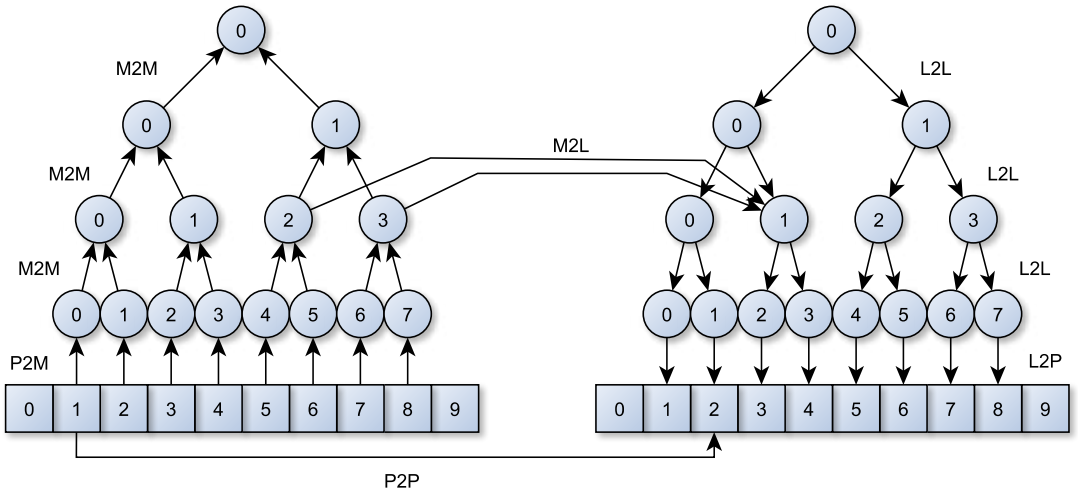


Figure 2.5: A graphical visualization of the algorithmic flow of the FMM. The left tree, building the multipoles with the P2M and M2M operations, represents the upward pass, the interactions from the left to the right tree via the M2L operations the horizontal pass, and the right tree with the L2L and L2P operations the downward pass. Short-range interactions via P2P operations are shown in the bottom layer. Most P2P and M2L operations are omitted for visibility. Source [Gra17b].

Here, p is referred to as the order of the expansion. Conversely, the local expansion follows the form:

$$\sum_{i=0}^{p} b_i r^i \tag{2.28}$$

With $p \to \infty$, the approximation error approaches zero, so in practice, the expansion order has to be chosen according to the precision needs of the application [Kab12]. However, since the computational complexity of calculating one expansion is typically in the range of $O(p^6)$ [Yok13] to $O(p^2 \log p)$ [EB96], the user is incentivized to keep p as low as possible. A number of possibilities have been proposed to formulate the contribution of particles to the expansions, such as solid harmonics, spherical harmonics, or cartesian Taylor, providing varying advantages in computational and memory complexity, as well as ease of implementation [Kab12, Yok13, Tch20].

Algorithm Description An intuitive flow of the FMM for 1D is sketched in Figure 2.5. The domain is divided into cells similarly as in the Linked Cells algorithm. This is visualized in Figure 2.5 with the square nodes with nodes 0 and 9 being halo cells. The cells at the bottom of the left tree represent the state of the particles at the start of the interaction evaluation and those at the bottom of the right tree at the end. In a first step towards the computation of the long-range interactions, also called the far field, multipole expansions are calculated to represent these cells in so-called Particle to

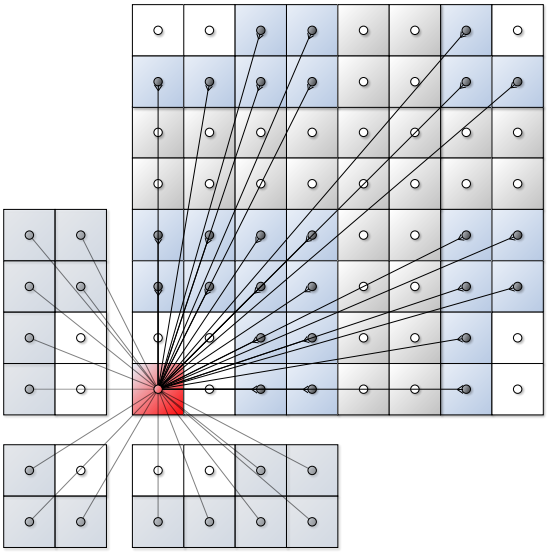


Figure 2.6: Visualization of one local (red) and all its multipole interaction partners on the same level (blue), including periodic boundary images (gray cell with gray point) during the M2L phase. Multipole expansions, which are white points in a gray cell, have already contributed their influence on a higher level, while the multipole expansions in white cells are too close to contribute on this level. Source [Gra17b].

Multipole (P2M) operations. These multipoles are then combined to a coarser layer of multipoles recursively with Multipole to Multipole (M2M) operations until one multipole represents the whole domain, creating a tree of layers, which is an octree in the 3D case. This whole procedure is also referred to as the upward pass. Within each of these layers, long-range interactions are applied in Multipole to Local (M2L), resulting in a tree of local expansions. Each local expansion contains the contributions of the multipole expansions from a specific distance band, as shown in Figure 2.6 with the blue areas. White areas are too close to the red target expansion to converge, while gray areas are so far away that they can be covered more efficiently with sufficient precision on higher levels. Finally, the tree of local expansions is reduced in the downward pass with Local to Local (L2L) operations, and the effect on the actual particles is eventually calculated with the Local to Particle (L2P) operations. Since the M2L operations can, by design, not cover interactions between particles close to each other, Particle to Particle (P2P) operations are evaluated between neighboring cells. This is also referred to as the near-field evaluation, which is basically the same as evaluating short-range interactions via the Linked Cells algorithm.

If the simulation features periodic boundaries, the number of M2L operations per level increases to include periodic images of well-separated neighbor expansions, as shown in Figure 2.6.

Computational Complexity The near field computations are analogous to the Linked Cells algorithm and are thus in $O(N)$. The far-field computation only depends on the number of particles during the P2M, and L2P steps, where sums over all particles covered by the respective expansion are calculated, which places these steps in $O(N)$ [GR87]. All remaining operations are independent of the number of particles and only depend linearly on the number of tree nodes M. As M is chosen independently of N and typically $M \ll N$, it can safely be assumed that $O(M) \in O(N)$. Hence, since all steps are in $O(N)$, this places the whole algorithm in $O(N)$.

2.1.4 Discussion of Archetypes

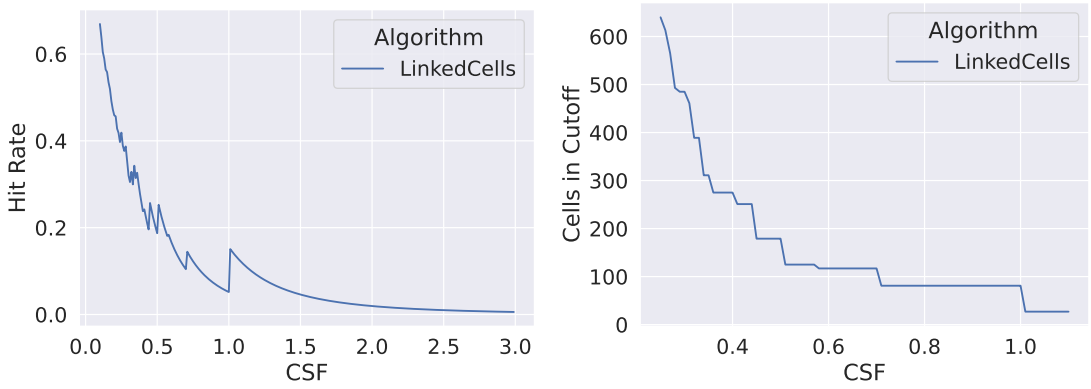
As already touched upon in Subsection 2.1.3, each algorithm archetype comes with pros and cons inherent from its core idea. Furthermore, there are algorithmic parameters and design choices, that play a significant role in the performance of one or several of these algorithms, which shall be discussed here.

Cell Size Factor As mentioned above, the choice of the side length of the cells, also called cell size, significantly impacts the algorithm's performance. If the cell size equals r_c, finding all neighbors for a particle in 3D can be limited precisely to the particle's cell and all 26 surrounding cells. We introduce the Cell Size Factor (CSF), which describes the side length of a cell as a multiple of r_c. If the cells are smaller than r_c, so CSF< 1, more neighboring cells need to be taken into account. On the one hand, this leads to a better approximation of the spherical cutoff region through smaller cell blocks, yielding a generally higher hit rate, except around the discontinuities, clearly visible in Figure 2.7a. Those always appear when the number of cells that must be considered changes. For example, as seen in Figure 2.7b, for CSF$= 1.0$, we have to consider 27 cells per cell and get a hit rate of approximately 15%. For CSF slightly smaller than 1, two cells per direction need to be considered, so 81 in total, with almost the same total volume. Hence, the hit rate first plummets down to 0.05% but then rises as the cells get smaller again and more cells need to be considered. In general, it is not trivial to calculate the number of cells for a given CSF as this is a 3D variant of Gauss's Circle Problem [LD17]. On the other hand, a CSF< 1 rapidly increases the number of cells, worsening particle data fragmentation into smaller cells and thus decreasing memory alignment as well as prefetching efficiency.

Cells with a CSF of 0.5 are also used by LAMMPS underneath their neighbor lists[7]. The effects of CSF< 1 have been discussed in [MR99] where the authors conclude "The optimum cell size might vary from machine to machine and implementation to implementation." Consequently, this suggests that for a given machine and implementation, there exists an optimal number of particles per cell that should be targeted by adjusting the CSF.

An immediate implication from this is that especially for very sparse scenarios, increasing the cell size beyond r_c, meaning CSF> 1 can be reasonable, leading to a worse cutoff

[7] `https://docs.lammps.org/Developer_par_neigh.html` Accessed: 20.12.2024



(a) Ratio of distance to force calculations

(b) Number of cells within r_c

Figure 2.7: Impact of CSF on hit rate and number of cells in r_c. Smaller cells increase the hit rate but quickly lead to an enormous amount of cells.

sphere approximation and thus lower hit rate, but a better vectorization performance thanks to memory alignment.

Verlet Skin and Rebuild Interval As discussed in Subsection 2.1.3.3, the main advantage of Verlet Lists comes from its spherical lists being a better approximation of the spherical cutoff and the fact that lists can be reused for several iterations. These advantages are governed by the parameters Cutoff Radius (r_c) and Rebuild Interval (t_r). Hence, the choice of these is crucial. The idea of the skin is to capture all particles that can potentially enter a particle's r_c range until the next rebuild interval. For this, the absolute Verlet Skin (r_s) has to be as wide as the sum of the distances the two fastest approaching particles cover during one t_r. Since it is disproportionately expensive to track the distance every particle travels towards its neighbors, a reasonable upper bound for a static t_r is to choose r_s as in Equation 2.29.

$$r_s = 2|\vec{d}_{max}|t_r \tag{2.29}$$

Here, $\vec{d}_{max}$ is the longest movement vector applied to any particle in the system, making $|\vec{d}_{max}|$ the maximum distance a particle in the system covers in one-time step. This upper bound models the following worst case: At the time of a list rebuild there are two particles whose distance to each other is just above $r_c + r_s$. These two particles move straight towards each other at the same speed $\vec{d}$ and are the fastest particles in the simulation. This means, that they are within r_c after $r_s/2\vec{d}$ time steps, from which the bound in Equation 2.29 follows.

An easy way to obtain a dynamic and even better fitting t_r is to track the distance every particle moves from its position at the last list rebuild. As soon as one particle covers $r_s/2$, we cannot cheaply guarantee that there is no other particle that has moved the same distance towards this particle. Hence, the lists should be rebuilt.

From Equation 2.29, we see that the choice of r_s directly impacts t_r or the other way around. While we want to minimize r_s and maximize t_r, a smaller r_s requires a lower t_r and vice versa. As a reminder, the relation of r_s and s is:

$$r_c + r_s = r_c \cdot s \tag{2.30}$$

With this, we can for a given s, compute the Rebuild Frequency (f_r) as the number of iterations it takes for the fastest particle in the system to cover half of the yet-to-be-determined skin distance:

$$f_r = \frac{1}{t_r} \tag{2.31}$$

$$= \frac{1}{\left\lceil \frac{r_s}{2|\vec{d}_{max}|} \right\rceil} \tag{2.32}$$

$$= \frac{1}{\left\lceil \frac{r_c(s-1)}{2|\vec{d}_{max}|} \right\rceil} \tag{2.33}$$

To get an idea of the impact skin and rebuild frequency have on the relative performance of Verlet Lists to Linked Cells, we consider the total number of neighbor distance evaluations per iteration n_{VL} and n_{LC} as shown in Equation 2.34. This can be formulated as the sum of calculations that are done less than Linked Cells due to the better spatial approximation, plus the number of evaluations needed for rebuilding the lists, averaged over all iterations:

$$n_{VL} = n_{LC} \left(\frac{\frac{4}{3}\pi \left(r_c s\right)^3}{3{r_c}^3} + RebuildFrequency(f_r) \right) \tag{2.34}$$

Now, with Equation 2.31 and Equation 2.34, we can formulate a theoretical speedup of Verlet Lists over Linked Cells purely based on the number of distance evaluations.

$$S_{VL} = \frac{n_{LC}}{n_{VL}} = \frac{1}{\frac{\frac{4}{3}\pi s^3}{3^3} + f_r} \tag{2.35}$$

Since this function depends on f_r, it directly depends on the speed of the fastest particle in the system. The heatmap in Figure 2.8b visualizes the speedup of Verlet Lists over Linked Cells for any combination of s and t_r. Warmer colors in the bottom right corner visualize the valid intuition that higher rebuild intervals and lower skin factors increase the performance of Verlet Lists.

Given a particle speed, an optimal skin factor can be computed by minimizing Equation 2.34. This relation is plotted with the blue line in Figure 2.8a. In order to make the reasoning more generally applicable, particle speed is considered relative to r_c. Equation 2.34 features a `ceil` operator, which makes it nontrivial to identify the minimum and which also leads to the spikes we see. These spikes perfectly align with the number of necessary iterations between two rebuilds, shown in green. Individual pattern breaking

spikes are artifacts of the optimization process to calculate the optimal skin factor. From this skin factor, Equation 2.35 can be used to calculate how many distance calculations Verlet Lists has to perform relative to Linked Cells, which is the inverse of the speedup.

Overall, three observations can be drawn from this graph. First, as long as particles are so fast that they cross the cutoff in fewer than five iterations, Linked Cells have the advantage in the number of calculations. Second, the optimal skin barely exceeds 1.6, and considering that this is in a region where Linked Cells should be faster anyways, the largest s where Verlet Lists are more efficient is about 1.5. Third, with more iterations necessary to cover the cutoff distance, the skin factor slowly converges to one. This is slightly deceiving because more iterations in this context can either mean that particles are slower, or that r_c is larger. With a larger r_c, a larger s has a higher impact as the increase in volume of the respective sphere is cubic in $r_c + s$.

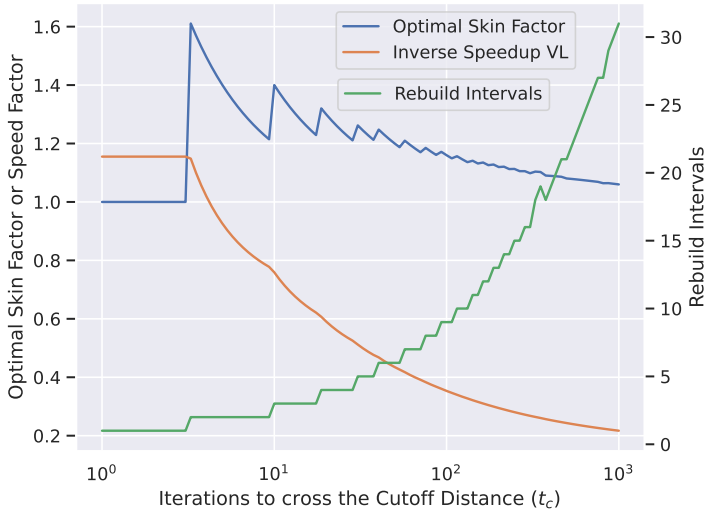
The data of the optimal skin from Figure 2.8a is also plotted as the cyan line in Figure 2.8b. Here, since there are multiple skin values for one value of t_r, an area is shown with the graph representing its middle. This line shows the actually achievable optimum within the technically possible ranges.

This discussion only considers the relative amount of computations of Verlet Lists and Linked Cells. The actual speed of the two algorithms also significantly depends on other, more hardware-related factors, like the ability to vectorize the algorithm, or how well a processor can prefetch memory and pipeline instructions, which will be touched upon in Subsection 2.1.6.

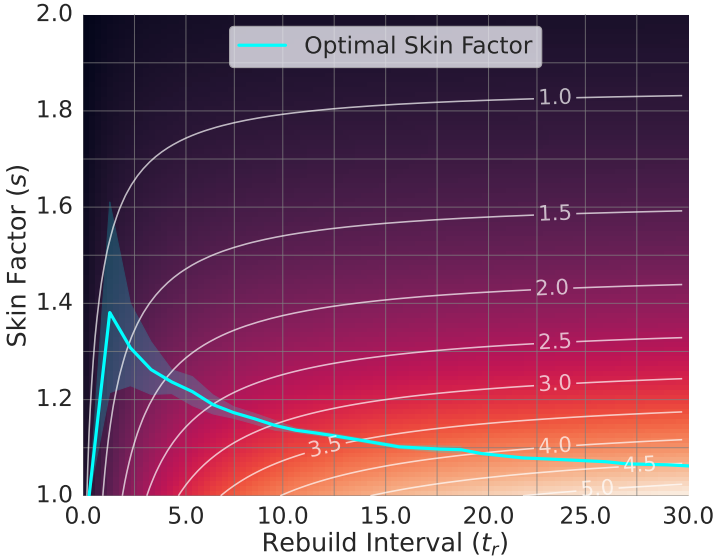
Linked Cells optimizations The simplicity of the Linked Cells algorithm lends itself to a list of tweaks and optimizations that address its primary sources of overhead, namely the low hit rate and computational cost, as well as memory interactions during the sorting of particles into cells. Some of these optimizations, which have been implemented in the context of this thesis, shall be briefly discussed here.

Skin for Cells Similar to the neighbor lists in Verlet Lists, the cell structure does not have to be updated in every iteration as long as it can be guaranteed that all neighbors of a given particle can always be found. Thus, if the side lengths of every cell are increased by the same skin length r_s as in the Verlet Lists algorithm, the search volume is increased, and Linked Cells can follow the same update rhythm as Verlet Lists. This, however, comes at the cost of a decreased hit rate. Hence, this optimization is only beneficial for scenarios with very slowly moving particles where a small skin allows for a high Rebuild Interval. In those scenarios, where particles barely move, resorting is still not for free because, after each move during a rebuilding iteration, it has to be checked whether each moved particle is still in its correct cell.

Sorted Cell Interactions A highly efficient way to increase the hit rate is to introduce more geometric information into the algorithm. For an interaction between two cells, the positions of all particles are then projected onto the straight line that goes through the center of both cells. For each particle, the loop over all other

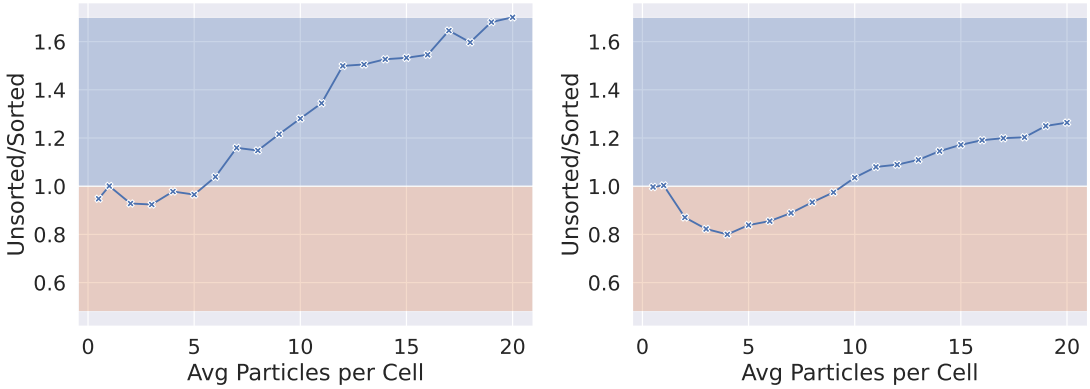


(a) For a given particle speed t_c, relative to r_c, the optimal s (blue) and resulting t_r (green) are shown. The orange line shows how many distance calculations are necessary for Verlet Lists relative to Linked Cells.
From $t_c > 4$ this factor is less than 1, which means that Verlet Lists have an advantage over Linked Cells.



(b) The color gradient shows the theoretical speedup of Verlet Lists over Linked Cells based on the number of distance calculations, disregarding the validity of the s. The cyan line shows the optimal skin value for a given t_r based on the data from Figure 2.8a. As there are multiple skin values for one t_r (see horizontal green lines in Figure 2.8a), depending on the particle speed, the corresponding area is shown.
The plot shows the limits of the speedup for combinations of s and t_r.

Figure 2.8: Analysis of optimal skin factors and its relation to t_r and particle speeds.



(a) Workstation
Intel® Core™ i7-10700 @ 2.90GHz

(b) HSUper
Intel® Xeon® Platinum 8360Y @ 2.40GHz

Figure 2.9: Impact of sorting the particles during the cell-cell interaction on different machines. Shown is the relative performance, where a value above one indicates an advantage for sorted and below one for unsorted. Higher particle densities benefit from sorting. For details about the setup see Section A.1.2.

particles then follows this sorting and can be aborted as soon as the first particle is encountered that is farther away than r_c. The authors of this idea reported a speedup of about 28% over the unsorted Linked Cells algorithm for the pair-wise interaction [Gon07]. However, since the sorting has to be done for every cell 27 times, or 14 times if Newton3 is leveraged, a non-insignificant overhead is introduced. Since higher numbers of particles are likely to benefit more as the cell-to-cell interaction is in $O(N^2)$, this begs the question of whether there is a range of number of particles for which sorting is not beneficial and should thus be avoided. This behavior was tested on different machines and the results are shown in Figure 2.9. While the expected trend that more particles per cell give an advantage to sorting can be seen clearly in both plots, the impact of sorting, as well as the break-even point, differ significantly between the machines. On the workstation seen in Figure 2.9a, the worst slowdown of sorting too sparse cells is only about 10%, and the highest speedup at 20 particles per cell is almost 1.7, with the break-even between 5 and 6 particles per cell. The most significant slowdown on HSUper is worse with 20%, and also, the highest speedup is lower at just over 1.26. Most interestingly, the break-even lies between 9 and 10 Particles per cell, so choosing a fixed sorting threshold at six particles per cell, which would be optimal for the workstation, would result in a worst-case slowdown in HSUper of 15%, highlighting the importance that this parameter should be tuned machine dependent.

Cells as Reference Data Structure One of the most central design decisions about the Linked Cells algorithm is the choice of its underlying data structure to store the particles. In general, two approaches are possible: Either to store the particles in

the cells, then the data structure is similar to a hash map, where the cells are the buckets, or to store all particles in one large continuous vector and have the cells only refer to locations within this. The advantage of storing particles within cells is that spatially near particles are always guaranteed to be also near in memory. Particularly, particles within one cell need to be accessed directly sequentially during the calculation of interactions, making this favorable for vectorization. When particles are stored in a central memory location, it is neither guaranteed that particles are sorted according to their cells nor, even if they are, that cell ranges coincide with cache line boundaries. In [DCGGM11], the impact of sorting particles according to different orderings was studied. The authors conclude that reordering particles to optimize memory access patterns can improve runtime by up to 20% for their case studies. Experiments on our implementation do not show the same severity as shown in Figure 2.10. Here, the differences in iteration speed between three versions of Linked Cells are shown. The overall data is very noisy, but looking at the median, 25th, and 75th percentile, we can see that the sorted reference-based version is about 5% faster than the others, which are very similar to each other.

2.1.5 Shared Memory Parallelism

Everything discussed so far only considers sequential execution. In light of the design of current Central Processing Units (CPUs) featuring up to 128 cores[8] leveraging parallelism is essential to make use of the full potential the hardware has to offer. Thus, this and the following sections will focus on shared memory- and instruction-level parallelism.

When talking about parallelism, it is important to have a clear understanding of a few important expressions:

Task Some unit of work, or set of instructions that are bundled and can be executed (optimally) independently. For example individual iterations of a loop could each be a task.

Race condition An error due to an unintended ordering of operations from multiple threads is called a race condition.

The classic example of this is one thread initializing a resource (e.g., a file). Next, a second thread alters the resource (e.g., deletes the file), and then the first thread wants to use the resource (e.g., write to the file), assuming it is still in the same state it has seen it in previously. This is also called a time-of-check to time-of-use (TOCTOU) bug [WP05].

Data race When multiple threads access the same memory location simultaneously and at least one of them is writing, the resulting error is called a data race.

For this, at least one of the threads must write on said memory. If both are writing, the final value in memory might end up even different from both written values

[8] `https://www.cpu-world.com/CPUs/Zen/AMD-EPYC%209754.html` Accessed: 20.12.2024

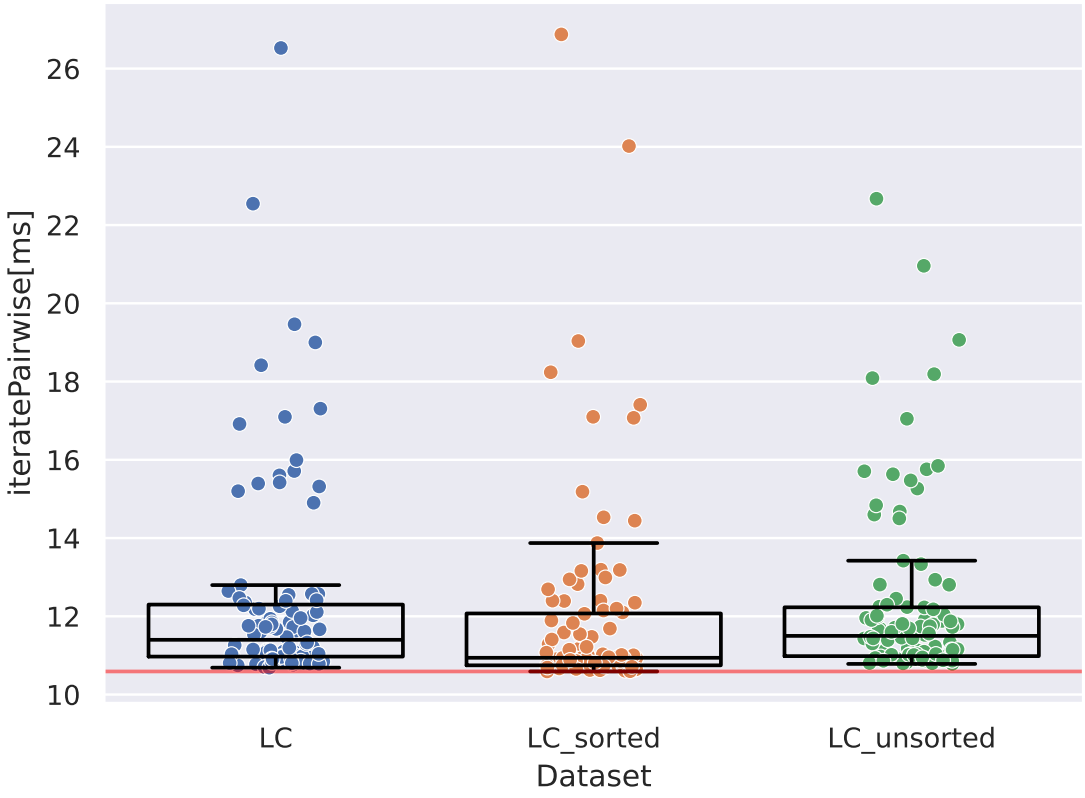


Figure 2.10: Boxplot comparison of time per iteration for Linked Cells with particles stored in cells (blue), Linked Cells with references in cells with sorting of the central data storage (orange), and without sorting (green).
Shown is the time for 100 calls to `iteratePairwise()`, on the AoS format, without sorting of a static simulation of 10^5 randomly distributed particles in a 100^3 box with r_c= 2.5 and r_c= 0, applying lc_{c08} with 16 threads. The red line marks the fastest iteration for visibility.
Sorting particles seems to be marginally faster on average, but not significantly.
For details about the setup see Section A.1.2.

due to the writing procedure being torn apart [SI09]. A typical example of this is two bank transactions happening simultaneously. For each transaction, a process reads the current value, updates it, and stores the new value. If two processes read the initial value, update it, and then store it, the stored value that comes in first will be overwritten, and the update from it is lost.

Embarrassingly parallel A set of tasks that can be executed in parallel and never cause any race conditions, or data races is considered embarrassingly parallel.

Degree of Parallelism For an algorithm, this is the fraction of the total work that can be executed at the same time. This in combination with the problem size effectively determines the maximum number of parallel workers that can effectively be employed for a given scenario. The highest possible degree of parallelism are embarrassingly parallel problems.

Load Balancing Naively spreading a problem over multiple worker threads can result in some threads ending up with more work than others. This leads to all threads having to wait for the one with the most work which is an undesirable inefficiency. The act of avoiding this is called load balancing. Two general approaches exist:

Static load balancing estimates the amount of work per task before execution and with this assign tasks to threads in advance.

Dynamic load balancing assigns tasks to threads at runtime, trying to keep all equally busy.

The tradeoff between the approaches is the quality of the balancing vs overhead costs. Cost estimate models tend to be rather complicated and of varying precision [Sec21], while assigning or shifting work at runtime requires either a central entity storing and distributing tasks in a synchronized fashion or individual workers synchronizing, exchanging load information and rebalancing their work.

Granularity The size of parallelizable tasks as the fraction of the total amount of work is referred to as the granularity of the parallelization. Granularity has a close relation to the degree of parallelism as finer grained strategies offer higher degree of parallelism by subdividing the problem into more smaller tasks. This also makes load balancing easier, because if there is an imbalance between two threads it can be rebalanced more precisely by shifting smaller tasks. On the other hand, this increases the overhead from scheduling, as there are more tasks to distribute.

As discussed previously in Subsection 2.1.1.1, computing the particles' interactions is the most computation-intensive part of a particle simulation. Hence, parallelizing this is of utmost importance. Particle simulations provide a potentially very high degree of parallelization since all particles can be processed simultaneously. While this is always true for steps like the velocity and position updates during the time integration, care has to be taken when exploiting Newton3, described in Subsection 2.1.1.3. With this, the force values of both particles i and j are updated during the interaction computation, which will lead to a problem if some interaction between particles j and k is evaluated

simultaneously, also updating both involved particles, resulting in a data race in the force vector of j. To avoid this, several techniques are possible:

Buffers Instead of writing the computed force to the same memory, each thread writes to an exclusive buffer location. All buffer entries must be reduced to the end result at the end of the parallel region, usually summing them up.

The advantage of this technique is that it makes the initial calculation again embarrassingly parallel, but at a memory cost of $O(N * t) \in O(N)$, where N is the number of particles and t the number of threads that have to access each particle. Note that,depending on the algorithm, t is not necessarily equal to the total number of threads. Another disadvantage is that the reduction step requires an extra pass over all particles again because, during the force computation, it is not trivially possible to know when all potential interaction partners are processed. However, it can be combined with the pass-of-the-time integration, which is also straightforward to parallelize.

Locks The shared memory locations can be granted exclusive access via locks, making data races impossible. Placing a lock around every particle's force vector is excessive and detrimental to performance since using them can induce significant CPU stalls. Each access to a lock has to be an atomic operation, the state of the lock has to be kept coherent among all involved threads, and if a thread hits a locked lock, putting it to sleep induces a potentially expensive system call, depending on the architecture. Therefore, locks should only be employed in limited numbers, and the probability of a thread hitting them can be kept reasonably low.

An example would be dividing the domain into sections for each thread and only placing locks at the interfaces, thus preventing threads from working on shared data simultaneously while using fewer locks. This can be improved even further if the threads schedule the processing of their domains so that it is unlikely that two threads work on the same interface region at the same time. If all of these goals can be achieved, the overhead from locks laid out previously becomes insignificant while still guaranteeing computational correctness.

Synchronization The third approach tries to find as much race-condition-free parallelism as possible and process these code regions one after another. For instance, this can mean splitting the domain into several parts, each of which can be individually processed embarrassingly parallel. Then, after each of these parts, a so-called barrier is placed, where the threads wait until all of them have reached the barrier and they synchronize.

Technically this might not be very different from locks but conceptually locks focus more on keeping a concurrent algorithm race-condition-free, while synchronization keeps a parallel algorithm in lock step.

An example of this is so-called domain coloring, where each of the aforementioned parts is one color. Each color partitions the domain into many small blocks, with each block representing a task that is scheduled as a whole. The goal is to never

have two blocks of the same color touching each other, to minimize the number of colors used, and to optimize the volume of the blocks. If they are too small, too much overhead is generated from dynamically scheduling the tasks. If they are too large, load imbalances can occur, or the degree of parallelism declines.

The advantage of this method is that, if employed correctly on a fitting problem, fewer synchronization steps are necessary than locks. Within each color, maximum efficiency is achieved since there are no constraints on the execution of tasks and potential for dynamic scheduling. On the negative side, each color barrier introduces measurable wait time for most threads [Gra17a].

In the following, the potential for parallelism of each of the short-range algorithm archetypes presented in Subsection 2.1.3 will be discussed. An extensive list of implementations of these approaches and further considerations can be found in our previously published work [GSBN22].

Direct Sum When using Newton3, Algorithm 1, can only be parallelized with locks on every particle, which is highly inefficient as discussed above. Another option is to not use Newton3, turning the problem into an embarrassingly parallel one, however, this leaves behind a significant optimization. A third option would be using buffers for every particle for every thread, which is highly memory intensive. The reason for the difficulties in parallelizing this algorithm is that for no particle, anything beyond their position is known. Hence, it would be impossible to determine cheaply if a particle is currently involved in the interaction evaluation of another thread.

Linked Cells With the Linked Cells method, particles are stored in a (3D) grid, which provides spatial information about groups of particles. This information allows efficient use of all of the above techniques if parallelization is applied at the cell level, which means the outermost loop in Algorithm 2.

If buffers are used, their number can be limited since each cell will only ever interact with $^{26}/_{2}$ other cells and itself due to Newton3. Thus, only 14 buffers would be needed, which is still a significant memory investment. This can be brought down further by scheduling techniques that guarantee how many threads will access a cell during a full iteration [Tch20].

In principle, placing a lock on each cell seems possible and significantly more efficient than on each particle. Previous work has shown that this approach can be inefficient due to the overhead of the locks themselves [Gra17a]. An alternative approach is to define and lock large regions of cells. For example, slicing a domain along its longest dimension into one slice per thread, thus only needing as many locks as threads. This method satisfies all the desired properties sketched above and can deliver competitive performance [TSH+18].

Verlet Lists The list structure, as presented in Algorithm 3, suffers from the same limitations to parallelism as the Direct Sum method. However, since Verlet Lists typically use Linked Cells as the underlying storage structure, the grids' information can

be used to create parallel algorithms. The problem remains that the neighbor lists usually provide no information about which cell a neighbor is stored in, thus rendering some cell-based approaches unfeasible. One possible solution would be to create several neighbor lists for each particle, one for every neighboring cell. Nevertheless, this creates additional overhead because several disconnected lists have to be processed and is thus only beneficial if particle number density is very high or r_c is wide so that lists are of sufficient length [GSBN22].

Verlet Cluster Lists Having an underlying grid structure, very similar possibilities and limitations exist for Verlet Cluster Lists as for Verlet Lists with underlying Linked Cells. The primary difference is that this grid is only 2D instead of 3D, thus offering a lower degree of parallelism.

2.1.6 Instruction-Level Parallelism

Modern processors possess dedicated instructions to simultaneously apply the same operation to several values. These are called SIMD instructions. Several instruction sets with different register lengths exist, like Streaming SIMD Extensions (SSE) (128 bit), Advanced Vector Extensions (AVX) (256 bit), Advanced Vector Extensions 512 Bit Extensions (AVX-512) (512 bit). These registers can store 32 or 64-bit values, dividing them into lanes. For example, using AVX with 64-bit values, such as IEEE double precision floating point numbers [C/M19] results in four lanes. This form of parallel processing of data is also called instruction-level parallelism or vectorization, even though the latter can also refer to the use of vector processors [HF84], which shall not be discussed here. In contrast to parallelization, vectorization is applied to the innermost loops since SIMD performs best on cache line aligned and consecutive data due to how data is loaded from memory into caches and then registers. So, the potential for vectorization of an algorithm can be judged primarily by the amount of independent processing of consecutive memory. For this analysis, we assume that accessing the same data fields from consecutive particles is consecutive data access since this is only a detail about how this data is stored. This will be further discussed in paragraph 3.1.1.3. Additionally, for all statistical considerations, we assume a homogeneous particle distribution. While in reality, this often does not hold for the entire simulation, locally over the volume of a few cells of Linked Cells, the assumption is fulfill. Either way, the modeling based on homogeneous distributions provides a good baseline for an average case and shows general trends and differences between algorithms.

Direct Sum At first glance, Direct Sum has enormous potential for vectorization because, for every particle, all other particles are accessed consecutively, as can be seen in the inner loop of Algorithm 1. However, since every particle pair is checked for a potential interaction, the probability of two particles p_i and p_j actually being within

range r_c is rather small, specifically:

$$P[|\vec{p_i} - \vec{p_j}| < r_c] \coloneqq P[r_c] = \frac{\frac{4}{3}\pi {r_c}^3}{r_{domain}^3} \tag{2.36}$$

Here, r_{domain} is the side length of the domain, which is, without loss of generality, assumed to be cubic with periodic boundary conditions. This length can also be expressed as a multiple n_{domain} of r_c, which simplifies the expression:

$$P[r_c] = \frac{\frac{4}{3}\pi {r_c}^3}{(n_{domain} r_c)^3} = \frac{\frac{4}{3}\pi}{n_{domain}^3} \approx \frac{4.19}{n_{domain}^3} \tag{2.37}$$

Usually, a domain is significantly larger than the cutoff used, but even if $n_{domain} = 3$, the probability for two arbitrary particles to be in range is already less than 16%, or 3% for $n_{domain} = 5$.

In vectorized code, no actual branching for individual lanes is possible. Instead, masking is employed. This means that lane values are set to zero depending on some condition, but the linear program execution continues. Thus, only if all four distances are greater than r_c the body of the `if` in Algorithm 1 can actually be skipped. In all other cases, the body needs to be evaluated, even if this means that some lanes carry out useless computations and are thus wasted.

Let us assume particles are distributed uniformly in the domain. Since Direct Sum does not sort particles in any way, sooner or later, particle positions in memory do not correlate with their position in the domain, even when it was initialized as a grid, due to the chaotic behavior of particle dynamics.

The probability that one inner loop actually needs to evaluate the interaction is the same as the probability that there is at least one hit h, i.e. a particle within the cutoff range. This can be expressed as the complementary probability that all particles are too far away from the particle of the outer loop:

$$P[\textit{interaction calculation}] = P[h \geq 1] = 1 - (1 - P[r_c])^{l_{vec}} \tag{2.38}$$

Here, l_{vec} is the number of parallel lanes supported by the SIMD instructions.

The shape of the probability function is shown in Figure 2.11 for low values of n_{domain} and values for l_{vec} up to instruction sets like AVX-512. However, the actual runtime depends on many more factors, for example, on how much branching is done, the cost in instructions of the interaction relative to the distance calculation, switching between serial and vectorized instructions, the interleaving of instructions by the out-of-order processor that is allowed by the implementation, cache prefetching, compiler optimizations, and so on. If there is no cutoff condition, it is reasonable to expect a speedup of near 4 from AVX, which has a $l_{vec} = 4$. With the cutoff condition however, that speedup is significantly diminished by the probability $P[h \geq 1]$ and thus $P[r_c]$ which makes n_{domain} one of the most crucial parameters as long as there are sufficient particles to fill the domain so that the statistical assumptions from above can be applied. For $n_{domain} = 3$, we measured speedups for the Lennard-Jones potential of around 2.5, but already at $n_{domain} = 5$, this drops down to less than 1.1.

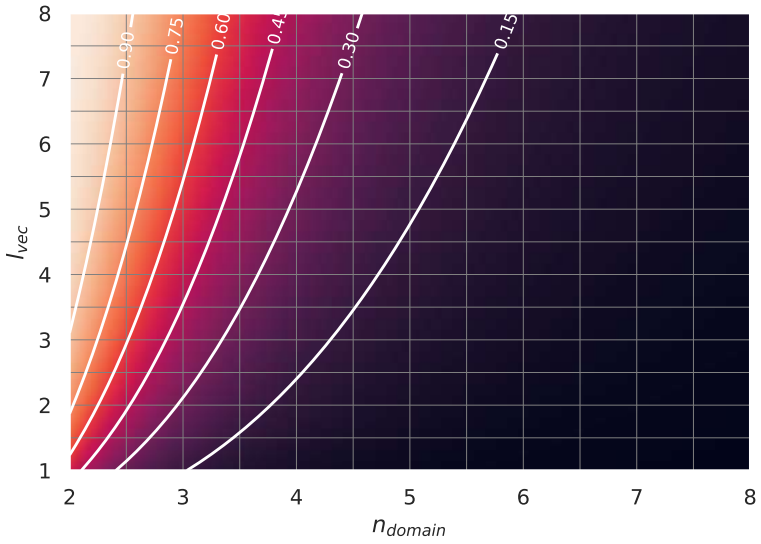


Figure 2.11: Visualization of Equation 2.38, the probability for a vectorized inner loop to contain at least one interaction.

Linked Cells The potential for vectorization for Linked Cells is closely related to that of Direct Sum. Looking at Algorithm 2, when we apply instruction-level parallelism to the innermost loop, lines 3–6 are precisely the same as in Direct Sum. The difference is that the particles of the innermost loop are in $^{26}/_{27}$ cases from another cell, which decreases $P[r_c]$ significantly. To calculate the probability, the following integral has to be solved:

$$\int_{A_{1z}}^{B_{1z}} \int_{A_{1y}}^{B_{1y}} \int_{A_{1x}}^{B_{1x}} \int_{A_{2z}}^{B_{2z}} \int_{A_{2y}}^{B_{2y}} \int_{A_{2z}}^{B_{2z}} \frac{1}{(B_1 - A_1)^3} \frac{1}{(B_2 - A_2)^3}$$

$$\cdot I_{(0,r_c)}(\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}) dx_2 dy_2 dz_2 dx_1 dy_1 dz_1 \tag{2.39}$$

Here, A_1, A_2 and B_1, B_2 are the boundaries of the two cells as 3D coordinates, $I_{(0,r_c)}(d)$ is the indicator function that some distance d is within $(0, r_c)$, and $x_1, x_2, y_1, y_2, z_1, z_2$ are the 3D coordinates of the particles p_1, p_2. Solving this is very involved but can also easily be derived from simulations. As an example, for cells with a side length of r_c, the hit rates for the four cases of cells are: Within the cell 90%, two cells with a shared face, 33%, a shared edge 9%, and only a shared corner 2%.

Within each cell pair interaction, similar considerations as for Direct Sum apply. The major difference, however, is that cells usually only hold a small number of particles, which leads to shorter sequences of continuous memory accesses.

Verlet Lists Even though the classic Verlet Lists algorithm Algorithm 3 looks very similar to Algorithm 1 for Direct Sum, the crucial difference in the head of the inner

loop complicates vectorization severely. Here, the loop does not traverse a data structure where particles are actually stored but a list of references. As a result, the memory access pattern is potentially arbitrary. Even sorting particles in storage according to the list at hand would not solve the problem because each particle has its own list thus, the particles would have to be resorted constantly. This makes gather and scatter load and store operations necessary which are not widely efficiently supported by hardware. Therefore, modeling the vectorized performance of Verlet Lists is primarily a matter of modeling the memory interactions of a given system.

Verlet Cluster Lists The cluster list algorithm Algorithm 4 aims to overcome the crucial weakness of Verlet Lists. As in Algorithm 3, the loop in line 2 traverses, not the primary storage data structure but a list of references. However, the trick in this algorithm is to choose the cluster size according to the number of available lanes l_{vec}, and the innermost loop in line 5 does traverse the actual storage. Due to this, the innermost loop can be vectorized in an efficient manner [PH13]. The problem is that this loop is very short. Thus, the vectorization should not end there, but at least the whole processing of the second innermost loop in line 4 should still only involve SIMD instructions. Otherwise, frequently switching between scalar and SIMD instructions will lead to significant performance degradation because the processor has to switch between two dedicated pipelines and sets of registers.

2.2 The Algorithm Selection Problem

Looking back at Section 2.1, we can see that there is a wide range of algorithms and even more variations to evaluate interactions in a particle simulation. For any application, it is, even for experts, let alone domain scientists, difficult to judge which configuration is optimal as their performance characteristics can vary widely, as will be shown in Chapter 4. This is because, as is the case in many other optimization problems, there is no silver bullet that is (near-)optimal in every situation [YWLC04]. The problem becomes even more complex in real-world applications, as the optimum can shift throughout a simulation, as demonstrated in [GST+19].

2.2.1 Problem Definition

Whenever one needs to solve any given problem, for example, finding the solution to $1 + x = 43$, this problem can be thought of as an instantiation from some problem space $\mathbb{P}$, in this case, linear equations. Any problem has $0, 1, \ldots, \infty$ number of solutions, represented in the problem-specific solution space $\mathbb{S}$, here $|\mathbb{S}| = 1$.

To find any such solution, algorithms are applied. Depending on the problem, there could be any[9] number of algorithms which make up the algorithm space $\mathbb{A}$, also called

[9] While the theory presented here does not include an upper bound on the number of algorithms, the problems in Chapter 4 have dozens to low hundrets of algorithms. It is unclear if negative scaling effects would occur if $|\mathbb{A}|$ would be in the thousands.

algorithm portfolio. These algorithms may yield results of varying quality. Nevertheless, any algorithm $a \in \mathbb{A}$, that is applicable to $x \in \mathbb{P}$ will find a solution $s \in \mathbb{S}$, except for so-called Las Vegas algorithms, which will not be considered here [Bab79]:

$$a \colon \mathbb{P} \to \mathbb{S}; \quad (x) \mapsto s \tag{2.40}$$

Since for many problems $|\mathbb{A}| > 1$, immediately the question arises which algorithm to pick to get the most satisfying solution in the most desirable way. This is the algorithm selection problem, which was first described by John Rice in 1976 using similar reasoning [Ric76].

In its most general form, algorithm selection is formally the problem of optimizing a function f that maps an Algorithm $a \in \mathbb{A}$ and a problem instance $x \in \mathbb{P}$ to some performance $p \in \mathbb{R}$:

$$f \colon \mathbb{A} \times \mathbb{P} \to \mathbb{R}^n; \quad (a, x) \mapsto p \tag{2.41}$$

Performance can take many forms, e.g., time to solution, accuracy, memory requirements, or even a combination, and is subject to the broader problem context. Conceptually, performance can thus be in $\mathbb{R}^n$. However, for the sake of comparing performances, it is helpful to apply some norm to bring this into $\mathbb{R}$.

The general optimization problem is thus formulated to find a a_{opt} such that:

$$f(a_{opt}, x) \geq f(a, x) \quad \forall x \in \mathbb{P}, \forall a \in \mathbb{A} \tag{2.42}$$

where "$\geq$"denotes a "better-than" or "higher utility" relation that depends on the performance metrics. E.g., a higher accuracy or a lower time to solution.

No Silver Bullet The inequality in Equation 2.42 would imply that there is an algorithm a_{opt} which is superior to all other algorithms in $\mathbb{A}$ in every problem instance. Unfortunately, this is rarely the case and would limit the need for automating the algorithm selection problem. In practice, we see that the algorithms' performance highly depends on the problem, and no global optimum can be assumed.

Thus, one option is to restrict the problem space to $\mathbb{P}^* \subset \mathbb{P}$ and only look for a local solution:

$$f(a_{opt}, x) \geq f(a, x) \quad \forall x \in \mathbb{P}^*, \forall a \in \mathbb{A} \tag{2.43}$$

Another approach is to introduce an averaging function m, which relaxes the requirements with respect to the generality of each individual problem instance.

$$m : \mathbb{R} \times \cdots \times \mathbb{R} \to \mathbb{R}; \quad (p_0, \ldots, p_{|\mathbb{P}|}) \mapsto \overline{p} \tag{2.44}$$

Now we only look for the algorithm that is best on average using m:

$$m\left(f(a_{opt}, x_0), \ldots, f(a_{opt}, x_{|\mathbb{P}|})\right) \geq m\left(f(a, x_0), \ldots, f(a, x_{|\mathbb{P}|})\right) \quad \forall a \in \mathbb{A} \tag{2.45}$$

2.2.2 Automated Algorithm Selection

Building on top of the previous definitions, the automated algorithm selection problem is the algorithmic automation of finding a solution to the algorithm selection problem.

To understand how to tackle this, some characteristics of the optimized function f, from Equation 2.40, need to be understood.

Continuity Typically, f is a mapping of categorical values of $\mathbb{A}$ which might even lack an ordering. Thus, no reasonable guarantees about continuity can be given. This makes it harder to make assumptions about how p will change for different a.

Bijectivity f is neither surjective nor injective, meaning that we neither know which exact performances are reachable with $\mathbb{A}$, nor that for a given problem, each algorithm produces a unique performance. This implies that the solution to the optimization problem might not be unique.

Complexity The worst-case runtime complexity of solving the algorithm selection problem as in Equation 2.42, is $O(|\mathbb{A}| \cdot |\mathbb{P}|)$, since we would have to evaluate every algorithm for every problem instance to establish the relations. The same also holds for the relaxed optimum in Equation 2.45 since still, all performance values need to be known.

In the context of particle simulations, this is a significant problem because even if we only consider one problem instance, so $|\mathbb{P}^*| = 1$, testing the whole of algorithm space is inefficient because evaluating f from Equation 2.42 means evaluating the particle interactions, which is an expensive operation.

The key takeaway here is that the only ways to reduce the theoretical complexity for finding a reasonable solution to the algorithm selection problem are:

- Reducing $|\mathbb{P}|$ by only looking at a restricted subset $\mathbb{P}^*$.
- Bringing structure to the search spaces $\mathbb{A}$ and $\mathbb{P}$, so that by evaluating $f(a_0, x)$ conclusions about the runtime of e.g. $f(a_1, x)$ could be drawn.
- Limit the number of evaluations of f through prior (expert) knowledge. This could either mean avoiding applying algorithms to problems that are generally known to be a bad combination (expert knowledge) or, in the context of a simulation, building a knowledge base on the fly and drawing conclusions from this (prior knowledge).

2.2.3 Closely Related Problem Variants and Applications

Typically, the application area of most algorithm selection problems is Boolean Satisfyability Problem (SAT), where it has to be decided if there exists a true / false configuration for the variables to make the expression true [BHvM09]. Another is finding optimal scheduling algorithm for dynamic loop scheduling from a given portfolio [Cio08, SMS+14]. Further applications are the Travelling Salesman problem, where the goal is to find the shortest route that covers all nodes in a graph [DFJ54], or Artificial

Intelligence (AI) planning, which is about finding a valid set of actions, also called plan, that leads from a start to a goal state, or deciding that no such plan exists [GNT04].

All these applications have two things in common: They are at least NP-hard, and they all are conceptually very different to the optimization problem discussed in this thesis, the identification of an algorithm configuration for a simulation.

There exist a number of variants of the above described algorithm selection problem, all of which are closely related but sufficiently different to require dedicated treatment. This categorization follows loosely tha previously proposed summary of the field [KHNT19].

Per-instance Algorithm Selection This describes the problem of finding the optimal algorithm to exactly one problem instance $x \in \mathbb{P}$. The classic example for this is in SAT, to find a valid configuration for one given expression. This is the form of the problem that was considered originally by Rice [Ric76].

Per-Set Algorithm Selection Here, one algorithm has to be found for a given set of problems. This is the problem stated in Equation 2.45 and ist often solved by an exhaustive search of all algorithms in the representative subset [KHNT19]. An application example is AI planning, where an AI architecture for a specific problem domain $\mathbb{P}^*$, which comprises of a set of instances, is searched.

Algorithm Configuration In contrast to algorithm selection, algorithm configuration looks for the optimal values for configuration parameters of an algorithm. Usually, this means that the space of options is significantly larger and might even include continuous options. This is very similar to hyper-parameter tuning, which is often seen in e.g. machine learning. Considering that algorithms can be configured with further arguments $h \in \mathbb{H}^m$, where m is the number of hyper-parameters, like, for example, block sizes in matrix decompositions, number of layers in machine learning models, choice of pivot elements in search algorithms, Equation 2.40 can be extended to:

$$a \colon \mathbb{P} \times \mathbb{H}^m \to \mathbb{S}; \quad (x, h) \mapsto s \tag{2.46}$$

Algorithm configuration aims to find $h_{opt} \in \mathbb{H}^m$ to achieve the best possible solution $s_{opt} \in \mathbb{S}$. The main difference to classic hyper-parameter tuning is, that hyper-parameter tuning typically focuses on optimizing an algorithm for a specific and narrow application context, while algorithm configuration might be applied to a more heterogeneous $\mathbb{P}$ which in turn can require trade offs in individual peak performance to improve the average performance. Thus, algorithm configuration also exists as per-instance and per-set configuration, similar to algorithm selection. Further applications, which are not hyper-parameter tuning are set covering optimization, which asks for a minimal set of given subsets to fully cover the overarching set, or mixed integer programming problems, which are optimization problems of linear functions that are constrained by linear inequalities and constrains on the variables [KMST10].

Algorithm Schedules If multiple plausible algorithms can be chosen, there is the question in which order to try them, especially if there is a limited compute budget. Therefore, this is often combined with other algorithm selection techniques to test a few of the most promising candidates [KMS+11, VCG+15, LHH15, LHHS15]. This can also be used effectively as per-set selector [Rou12]

Parallel Algorithm Portfolios Similar to algorithm schedules, here all candidates are applied concurrently, and as soon as any algorithm comes to a solution, all others are aborted. If compute resources are limited, this approach has to consider the impact on algorithms when the resources are shared, effectively limiting what each algorithm has available.

Although the clearly defined, these problems are not necessarily completely distinct. Some applications like SATzilla [XHHLB08], or AutoFolio [LHHS15] tackle the actual problem they try to solve by applying a layerd approach solving e.g. a per-set selection, to feed a parallel algorithm portfolio, which is used to determine the solution for an given instance selection.

Generally though, in contrast to the work presented in this thesis, most research seems to be focused on offline or static algorithm selection. Online or dynamic selection seems to gain traction and is mostly covered by reinforcement learning approaches, see [ACMR06, GS10, DBK+16]. To add to this, to the best of our knowledge, there seems to be no research or attempts to apply any form of algorithm selection to particle simulations, hence, no more closely related work can be discussed.

2.3 Interim Summary

In this chapter, Subsection 2.1.1 introduced the fundamentals of particle simulations, especially short-range interactions and standard simulation features like particle propagation and boundary treatment. Using these fundamentals, Subsection 2.1.2 laid out three categories of applications, MD, DEM, and SPH, which physical processes they typically model and what they have in common.

With this, our scope of application of particle simulations was defined, and the following sections discussed efficient core algorithms and techniques for them. Subsection 2.1.3 introduced a zoo of algorithm archetypes, like Linked Cells and Verlet Lists, that can be used to efficiently implement the particle interaction, which is usually the most compute-intensive part of such simulations and discussed their pros and cons. Subsequently, in Subsection 2.1.4, parameters and variations of these algorithms were discussed, and their effects on performance were demonstrated. It was demonstrated that models for relative algorithm performance can be constructed, for example, using the skin factor. However, they do not capture the complete relative performance characteristics, and their actual performance also heavily depends on implementation details, which made performance predictions hard.

In the last parts of this section, Subsection 2.1.5 and Subsection 2.1.6 introduced shared memory and instruction-level parallelism. Here, their respective fundamentals

were defined, the algorithms' potentials were evaluated, and theoretical limitations and practical difficulties were highlighted. Here again, it was shown, among others at the example of the vectorization potential of the different algorithms, that the actual impact of vectorization depended both on implementation choices as well as the structure of the scenario, thus complicating direct performance predictions.

With this chapter, it was established that a wide range of possible algorithms, implementations, and optimization parameters and choices exist to compute the interaction step in particle simulations.

This led to the algorithm selection problem, which was formally defined in Subsection 2.2.1. In the subsequent Subsection 2.2.2, the automation of solving this problem was discussed along with its properties and ways to approach it. Finally, Subsection 2.2.3 drew connections to similar algorithm selection problems as the one that is tackled by this thesis. The insight from this part was that solving the algorithm selection problem is in linear time complexity relative to the number of algorithms available. However, since individual evaluations are costly, techniques have to be found to identify a minimal number of algorithms potentially suitable for the problem at hand and only focus on them.

To summarize, this chapter laid out the theoretical foundations as well as practical considerations and approaches of the two main pillars of this thesis: particle simulations and the automated algorithm selection.

3 AutoPas

After Chapter 2 all necessary fundamental concepts are established and we can now move on to their implementation. This chapter is a comprehensive presentation of the AutoPas library. Figure 3.1 shows its logo and icon. First, the software itself is discussed from a user's and developer's perspective including a deep dive into its implementation and optimizations. Next, we present the internal auto-tuning procedure, how it interacts with the simulation workflow and implemented tuning strategies. Finally, we widen our perspective to other significant or related MD software packages with the aim to draw comparisons to our library and highlight the strong attributes of their work and the uniqueness of AutoPas, and how it fits into the bigger picture of particle simulations.

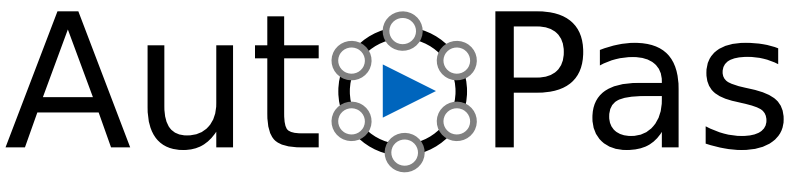


Figure 3.1: The large AutoPas logo. If space is limited or only an icon is needed, the circular symbol, which here takes the place of the o, can be used alone.

3.1 The Library

The library project presented in this thesis has the purpose to significantly contribute to the answer to the research questions that were formulated in **??**. To achieve this, it was designed with the following goals in mind:

Usability The user-side API should be kept simple with only a minimal number of necessary interaction points. This means, that most setup, update, or cleanup logic should be triggered as automatically as possible. Examples for this would be the maintenance of Verlet Lists, or the application of the correct interaction kernel depending on the current data layout.

Customizability The library should be highly configurable so that it can be tailored to very specific experiments or needs. This is achieved on the one hand by a wide range of internal settings that can be configured by the user, as well as the fact that they provide their own particle and force model.

High customizability can quickly get in the way of good usability since a more complex interface is typically more complicated to use. Therefore, to counteract

this, all configurable options have to have sane defaults so that the library follows the philosophy that the user only has to touch / configure what they actively want to change. Anything they do not touch should generally have reasonable defaults or tune itself.

Performance / Scalability Since the main field of application of AutoPas is HPC optimal code efficiency, good performance as well as high scalability are core requirements. As the library is only concerned with node-level parallelism, distributed memory parallelism with AutoPas will be discussed in Subsection 3.1.1.4, scalability refers to shared memory parallelization with OpenMP.

Modularity AutoPas implements various data containers and parallelization strategies that need to be switched and recombined at runtime. Hence, high modularity is desirable, so that components act as independently as possible. The library achieves this by high cohesion and low coupling of classes, making extensive use of free functions. Most importantly is the use of the strategy software pattern for all exchangeable components like data structures or parallelization strategies.

As a positive side effect, this greatly facilitates bringing in developers that work only for a short time on the project, like students with thesis projects, since they only need to familiarize themselves with a very small and self contained section of the code.

Maintainability We want to ensure that AutoPas will be used and continued to develop beyond the scope of this thesis. For this, the code must be straight forward to understand, maintainable, and easy to extend. One aspect of achieving this is through maximal reuse of implemented logic while avoiding wide spread coupling of components. So called code clones, sections of duplicated code with only minor modifications, can often lead to copy paste errors, an increase in lines of code and thus should be avoided [Kos07]. An exception for this guideline applies when avoiding the clone significantly increases the code complexity, thus decreasing its maintainability. Figure 3.2 shows that less than 7% of the AutoPas library contains code clones. Compared to the usual 7–23% reported in general software projects [Kos07] we consider this to be an excellent value.

Another side effect of low code duplication is that, any optimization immediately is available to all relevant algorithms. We achieve this for example a deep class inheritance hierarchy, which can be especially seen in the implementation of the traversals as seen in Figure 3.7. This reduces the overall size of the code, making it easier to grasp for new developers and implicitly points out common behavior of different algorithms. Another aspect is the centralization of common logic steps in a logic layer between the main interface and the actual data containers, which manages and sanitizes all actions and queries.

A second example for avoiding code duplications is the implementation of iterators, where general iteration management is centralized in the `ContainerIterator` class and data structure specific access logic is implemented in the containers. Generally

Path	Files	Source Lines of Code	Method Length Assessment	Nesting Depth Assessment	Clone Coverage
Summary	538	51.4k			14.5%
applicationLibrary	43	8k			26.8%
cmake/tests	1	4			0.0%
examples	62	7.5k			6.9%
src/autopas	266	24.1k			6.5%
tests	157	10.9k			26.3%
tools	9	837			0.0%

Figure 3.2: Overview of high level metrics of the AutoPas repository analyzed with Teamscale[1]. Only, the content of `src` is the actual AutoPas library which has an excellent clone coverage rate of less than 7%. The same can be said of the examples which include md-flexible which is presented in Section 4.1. `applicationsLibrary` is a collection of functors and particle models which explains the high clone coverage since they usually share several properties. Also `tests` usually contain code clones for test setup reasons.

through the high use of interface polymorphism in the logic layers of the library the compiler guides new developers towards implementing all relevant functionality. This in combination with descriptive naming and extensive documentation of the interfaces ensures that developers of all skill and experience levels can contribute new algorithm modules to AutoPas.

Testing Finally, a vast infrastructure of more than 16 000 unit and integration tests supports the library. Especially helpful are automatically generated tests for all algorithm options checking common behavior and apply cross validation to ensures code correctness. These tests are immediately available for any newly implemented containers or traversals.

Many codes often go for a tight coupling between their modules to achieve optimal performance. However, this contradicts our design goals of modularity and maintainability. Therefore, large parts of Subsection 3.1.2 will be spent explaining how and where we employ abstraction to facilitate modularity and minimize code duplication to increase maintainability and where we focus on specialized performance optimizations.

Starting with a high-level perspective, we will first discuss the user's perspective on AutoPas, which will cover what users need to provide the library with and what they get out of it. Then, we will switch the focus to the internal developer's perspective, which delves into the software engineering aspects, optimizations, and lessons learned along the way during the development process. Lastly, low-level hardware-aware optimizations of the interaction kernels that were implemented will be discussed.

3.1.1 Design, Structural Overview and Usage

AutoPas employs a very abstract model of particle simulations that separates the simulation loop into three major phases as shown in Figure 3.3a:

Interactions Here, the particle-particle interactions, introduced in Subsection 2.1.1.1, are performed. The computational complexity of this phase is naively in $O(N^2)$ (for pairwise interactions) but can be brought down to $O(N)$ as discussed above. Nevertheless, this is usually the most computationally expensive phase. In the context of AutoPas, this is currently limited to short-range interactions.

Propagation In this phase, the positions of particles are updated. Usually, for example, in MD, the changes in the particle positions stem from their directed velocity, which is a result of the forces they experience due to the particle-particle potentials computed in the previous phase. However, this is not necessarily the case since the induced forces could also solely come from other sources, such as global force fields, for example, in the space debris simulation presented in Section 4.4. Nevertheless, this phase only requires touching every particle a constant number of times. It thus can be considered to be in $O(N)$, albeit potentially with some significant factors depending on the intricacy of the computation.

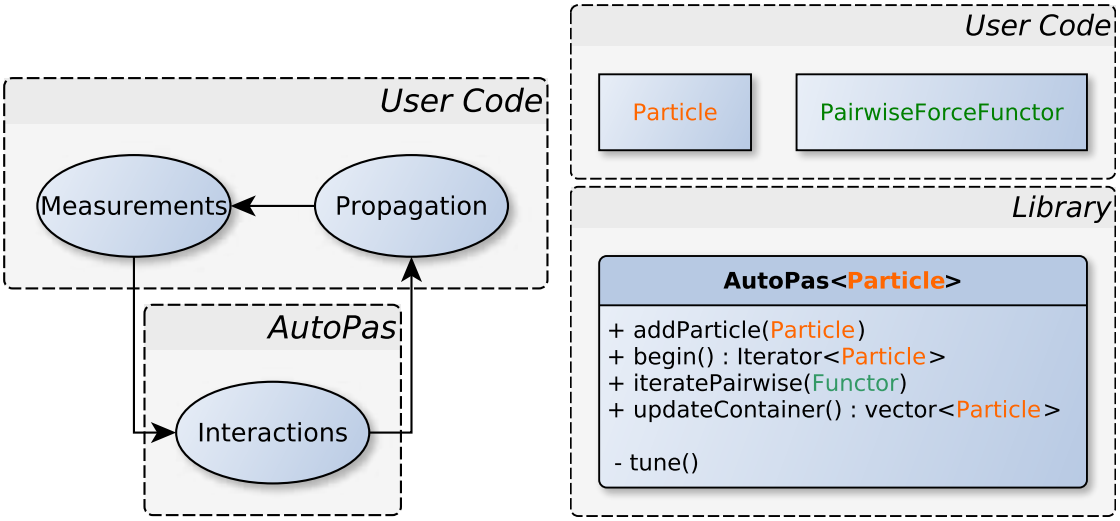
Measurements This is the phase where the application scientist retrieves value from the simulation. Typically, this involves reading velocities or masses in specific regions or sampling custom particle properties. For some metrics like the radial distribution function, it is necessary to know the pairwise distances. However, this is something that is already computed in the interactions phase and thus does not have to be redone. Thus, we also consider this phase to be generally in $O(N)$.

As sketched in Figure 3.3a, AutoPas only handles the interaction phase, and it is up to the user to implement their propagation and measurements. This decision was made because the interactions phase is usually the most challenging phase, and the latter two phases can be highly customized, making it difficult to find a common core for the most expensive operations in these phases. For example, in the propagation phase, any time integrator algorithm might be used, depending on the model and precision constraints of the simulation, and the measurement phase is, from the perspective of AutoPas, entirely arbitrary.

From the user's perspective, AutoPas first seems like a generic data container, maybe not too dissimilar to something like `std::vector`. One instantiates it with a template parametrization, the particle type, and one can pass objects of this type to the AutoPas object via functions like `AutoPas::addParticle()`. There are also functions to query the number of stored elements, `AutoPas::getNumberOfParticles()`, or iterate over all with an iterator object obtainable via `AutoPas::begin()`.

3.1.1.1 Software Architecture

The ISO/IEC/IEEE 42010:2022 standard defines software architecture as:



(a) AutoPas high-level abstraction of particle simulations split into three major phases.

(b) Minimal overview of the main interface of AutoPas, its primary interaction points, and what it expects a user to provide.

Figure 3.3: High-level user's perspective of how AutoPas fits in particle simulations and how a user is expected to interact with and employ it.

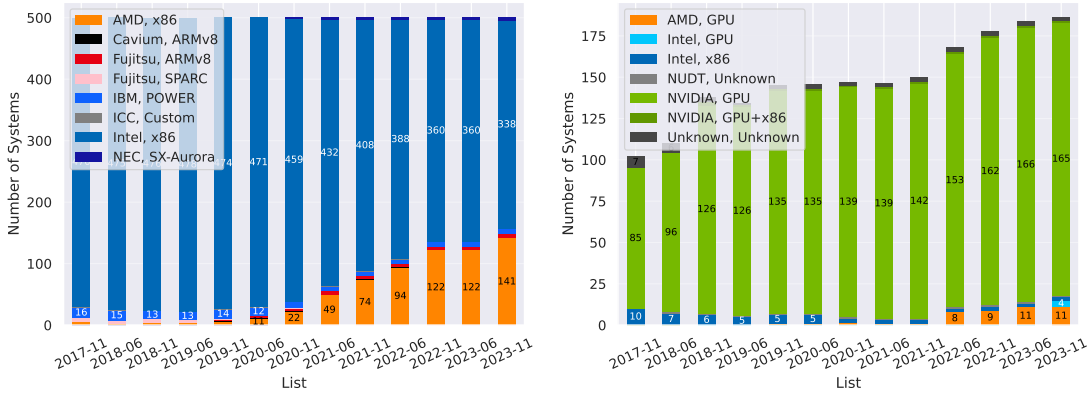
> "fundamental concepts or properties of an entity in its environment (3.13) and governing principles for the realization and evolution of this entity and its related life cycle processes". [ISO22]

In the spirit of this definition, we shall provide a non-formal short overview of the architecture of AutoPas that conveys the library's key ideas, concepts, and core structure.

Environment The environment that AutoPas is intended for is numerical HPC simulations of short-range particle systems as described at the start of Subsection 3.1.1. These simulations typically run on capable workstations or cluster computers, hence a high degree of parallelism can be expected to be available. As AutoPas only focuses on node-level performance, it only implements shared memory parallelism through OpenMP. However, since simulations for cluster computers also have to leverage distributed memory parallelization, e.g., via Message Passing Interface (MPI), the library must also work in this context. This will be discussed in Subsection 3.1.1.4.

In order to determine what kind of hardware AutoPas should target, we take a look at the TOP500 list of the largest supercomputers on the planet. For a long time now, 100% of the employed operating systems on all supercomputers have been some form of Linux. Thus, there is no reason to target any other platform.

We can see in Figure 3.4a that at the time when AutoPas was started at the beginning of 2018, the vast majority of systems, over 95%, were built on x86 processors, and since then, this has not changed significantly. The most significant changes in the domain of processors are that since around 2020, AMD is becoming increasingly relevant, and CPUs



(a) Processor types and vendors. Almost every system runs on x86 from Intel with a growing share supplied by AMD.

(b) Accelerator types and vendors. Since 2020 more and more systems use accelerators and almost all are GPUs from NVIDIA.

Figure 3.4: Developments of hardware distributions in the TOP500[2] from 2018 until 2023.

have more and more cores, which emphasizes the importance of supporting different types of CPUs with different shared memory parallelization behaviors.

As is shown in Figure 3.4b, accelerators have always been used in a notable part of the TOP500. Around the end of 2018, there was a jump in GPU systems, which stayed at a level for about three years. However, starting in 2022, we can observe a sharp and steady increase in systems using GPUs, mainly using NVIDIA CUDA. This is especially true when looking at the top 10 of the list, where at the end of 2023, nine systems use some type of GPU[3]. We acknowledge this trend and thus the shift in the target environment, which makes it highly important to also support GPU offloading in AutoPas in the future.

Core structure The main idea of AutoPas is to serve as a black box particle container that can internally switch its layout. Hence, the library is designed around a common particle container interface, and the classes implementing it, called particle containers, are primary components through which almost all particle-related logic flows. The particle containers are hidden behind two interface layers to facilitate interaction with the library and achieve the black box behavior. One layer is dedicated to managing internal logic, and another is designed to provide a stable interface for users, enhancing usability.

All core components and their interaction for the most common tasks like adding particles, applying a functor, and invoking the interaction pipeline are shown in Figure 3.5.

Particle A user-provided class that implements their particle model. This will be discussed in more detail in paragraph 3.1.1.2.

Functor A user-provided class that implements the user's force kernel. This will be discussed in more detail in paragraph 3.1.1.2.

[3] `https://www.top500.org/lists/top500/list/2023/11/` Accessed: 20.12.2024

AutoPas The main and top-level interface and primary point of interaction for the user. The user (usually) instantiates one object of this class to store all local particles and treats it like a container that features smart functions for particle interactions. This interface fulfills the role of a stable front-end for the user to work with that hides any changes in the internal structure. Since this class can be considered the public API of AutoPas, it is intended to experience as little change as possible for compatibility reasons.

LogicHandler The second interface layer is an internal class that handles high-level logic, sanity, and coherency checks. This class orchestrates the instantiation and management of all algorithmic options, called `Configuration`, and ensures the interactions with the particle container for particle insertion, deletion, and relocation are done correctly. It coordinates the containers' internal Verlet-like interface while providing the functionality of a black box Linked-Cells-like interface to the outside [SGH+20].

AutoTuner An independent internal class that handles the automated algorithm selection. It manages the search space and applies one or several tuning strategies to it to determine the optimal algorithm combination. This class provides the `LogicHandler` with `Configuration` objects to instantiate and apply.

ParticleContainer An internal category of classes that each implements a neighbor identification algorithm as a data structure, like, for example, Linked Cells or Verlet Lists. They store the real particle data and manage the actual particle interaction and interaction through iterators.

Traversal An internal class category that each implements a specific shared-memory parallelization for a specific `Container` class. They are ephemeral and always wrap a functor.

3.1.1.2 User-Provided Classes

Since AutoPas requires the user to provide their models in a certain way, this section will explain the structure and format of the expected classes, why they are built that way and discuss drawbacks, benefits, and potential future improvements.

Particle Class The user provides their particle model as a `C++` class that contains the properties of an individual particle in the classic object-oriented fashion. For interface compatibility, AutoPas provides a base class called `ParticleBase` that defines all necessary members, as well as utility methods such as getter, setter, and stream operators.

Listing 3.1 outlines the basic layout of the particle class that AutoPas expects and `ParticleBase` implements. From the library's perspective, all a particle needs is an ID to uniquely identify it, a position to file it correctly in the internal data structure, and an ownership state. The latter describes if a particle is owned by this instance of AutoPas (`OwnershipState::owned`), some other instance (`OwnershipState::halo`).

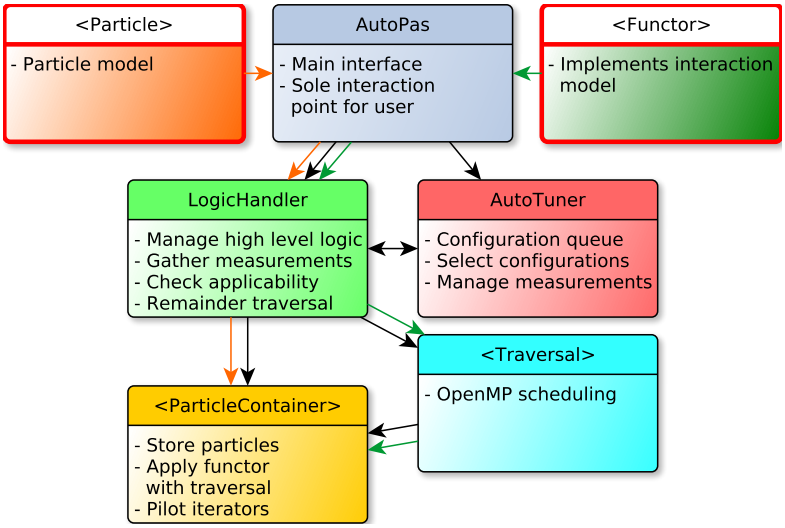


Figure 3.5: Overview of the main components of AutoPas. Boxes with a red border have to be supplied by the user. The orange path shows the logic flow of adding a particle, the green for applying a functor, and the black for the whole interaction pipeline.

There exists a third state (`OwnershipState::dummy`) which is neither of the previous and is primarily used by AutoPas to mark particles that only exists for bookkeeping or data structure reasons.

```
1  class ParticleBase {
2    protected:
3      size_t _id;
4      std::array<double, 3> _r;
5      autopas::OwnershipState _ownershipState{OwnershipState::owned};
6
7    public:
8      enum AttributeNames :
9              int { ptr, id, posX, posY, posZ, ownershipState };
10     using SoAArraysType = typename autopas::utils::SoAType<
11             ParticleBase*, size_t, double, double, double,
12             OwnershipState>::Type;
13
14     template <AttributeNames attribute>
15     constexpr typename std::tuple_element<
16             attribute, SoAArraysType>::type::value_type get() {
17         if constexpr (attribute == AttributeNames::ptr) {
18             return this;
19         } else if constexpr (attribute == AttributeNames::id) {
20             return _id;
21         } else ... // all other attributes
22     }
23     // setter analogous
```

```
24  };
```

Listing 3.1: Sketch of the base class that defines what properties AutoPas expects every particle to have.

The enumeration `AttributeNames`, the tuple type `SoAArraysType` combined with the template-based getter describe the particle's member variables and types and provide a way that AutoPas can use them to generate internal data structures. This will be further discussed in Subsection 3.1.2.5.

Whenever users implement their own particle class, they are free to inherit from `ParticleBase` or reimplement the whole interface because AutoPas expects the particle class as a template parameter and thus will always use the concrete type.

Functor Class Similar to the particle mode, the user also provides the force model as a `C++` class. This class must implement the force interaction for several data structures according to an interface in the AutoPas base-class `functor`. Again, as with the particle interface, the user does not have to inherit from the base class because it is passed to AutoPas as a template. However, the base class provides an automatism to convert between Array of Structs (AoS) and Structure of Arrays (SoA), which would have to be reimplemented. This automatism is explained in Subsection 3.1.2.5.

As outlined in Listing 3.2, the interaction is implemented both for AoS and SoA separately due to different data access patterns and a current lack of an efficient abstraction for this. To improve the efficiency for the data movement to and from SoA, the functor can and should communicate the minimal set of members of the particle class that are needed for the computation in `getNeededAttr()`, and those coming out of it in `getComputedAttr()`. Irrespective of the container currently in use only one AoS and SoA implementation are required. Currently, for SoA Verlet style containers use an interface that expects only one soa and a neighbor list, however this usually can be tied to the same core interaction implementation, as is demonstrated in the librarie's examples.

```
1  template <class Particle>
2  class LJFunctor : public Functor<Particle, LJFunctor<Particle> {
3    public:
4      // AoS part
5      void AoSFunctor(Particle &i, Particle &j, bool newton3) {
6          double dr = distance(i.getR(), j.getR());
7          if (dr > _cutoff) {
8              return;
9          }
10
11         double f = lennardJonesForce(dr, _sigma, _epsilon);
12         i.addF(f);
13         if (newton3) {
14             j.subF(f);
15         }
16     }
17     // SoA part
```

```
18      constexpr static auto getNeededAttr() {
19          return std::array<typename Particle::AttributeNames, 9>{
20              Particle::AttributeNames::id,
21              Particle::AttributeNames::posX, ... /*posY, posZ*/};
22      }
23      constexpr static auto getComputedAttr() {
24          return std::array<typename Particle::AttributeNames, 3>{
25              Particle::AttributeNames::forceX, ... /*forceY, forceZ*/};
26      }
27      void SoAFunctorPair(SoAView<SoAArraysType> soa1,
28                          SoAView<SoAArraysType> soa2, bool newton3) {
29          const auto *const __restrict x1ptr =
30              soa1.template begin<Particle::AttributeNames::posX>();
31          // force calculation similar to AoS Functor
32      }
33  };
```

Listing 3.2: Sketch of a functor class that defines and implements the force interaction for the Lennard-Jones potential. Shown here are the functions that implement the interactions specific to the available data layouts, as well as the functions that indicate which parts of the SoAs have to be loaded and retrieved.

3.1.1.3 Internal Algorithmic Options

Most available algorithmic options were already discussed and published in [GSBN22]. New algorithms are mostly variants of existing ones that have more complex load balancing mechanism, at the cost of more overhead. Examples for these are `sli_balanced`, `vlc_c08`, and Pairwise Verlet Lists. Hence, in this thesis, we will only briefly summarize all algorithms and present any new ones implemented since then.

Data Layouts In the context of AutoPas, data layout refers to how the particle data is arranged in memory. Two options are implemented

Array of Structs (AoS) Here, particles are stored as regular objects in an array-like standard container like `std::vector<Particle>`. This layout excels when individual full particles have to be accessed.

Structure of Arrays (SoA) In this layout, there is one array-like container (e.g. `std::vector<double>`) per particle property. All properties of a particle i are on the i-th position in each vector. The advantage of this layout is very efficient access to individual properties of consecutive particles.

Internally, AutoPas stores all particles in AoS format, mainly because it makes interacting with them more intuitive. When an algorithmic configuration requires the SoA format, this is converted on the fly in each iteration. The container reuses the memory for these SoA buffers. Therefore, this repeated conversion does not allocate new memory, thus keeping the operation within reasonable costs.

Containers Particle containers represent highly specialized implementations of the algorithms presented in Subsection 2.1.3. They maintain how the actual particle data is stored and provide efficient ways to identify the neighbors for the particle interactions. Since [GSBN22] lays out descriptions of the available options, only short intuitions will be listed here.

Direct Sum Two cells, one for all owned, one for all halo particles. This leads to minimal memory and data management complexity overhead.

Linked Cells A regular 3D grid of cells is implemented as a 1D vector of cells that store the particles. Due to this close relation of spatial and memory location, this container is very SIMD friendly.

Verlet Lists The Neighbor lists are vectors of pointers for each particle that are stored in one big map that sits on top of a Linked Cells data structure, which is used to create the lists. This trades SIMD friendliness with a significant increase in hit rate.

Var Verlet Lists A dynamic interface to easily implement different neighbor list styles. The exact behavior depends on the chosen neighbor list, which depends on the traversal choice. Hence, more discussion about this can be found in paragraph 3.1.1.3.

Verlet Lists Cells In this Verlet Lists implementation, neighbor lists are grouped by the cells of the underlying Linked Cells structure. This restores some spatial information, but only about the particle that owns the list and less about those on the list.

Pairwise Verlet Lists The container uses the exact implementation as Verlet Lists Cells but with one list per cell pair for each particle. This increases the spatial information but results in significantly smaller lists, creating overhead from jumping lists too often.

Verlet Cluster Lists Here, nearby particles are grouped into clusters, and only one neighbor list per cluster is created. Furthermore, these lists store references to clusters instead of particles, decreasing the memory consumption and unlocking some SIMD potential if the cluster size relates to the vector register size.

Optimization Options AutoPas implements a few optimization options that affect the details of the algorithms but do not change them fundamentally.

Cell Size Factor (CSF) This factor alters the size of the cells in any Linked Cells based container as a factor of r_i. The effects of this have already been discussed in paragraph 2.1.4. However, in any real implementation, cells cannot have completely arbitrary sizes since they still have to fill the domain, and AutoPas does not allow for cells to be cut. This means that if AutoPas is given a continuous range of CSF

values, only those for which the following expression results in an integer will be considered:

$$\frac{r_{domainX}}{r_i \cdot CSF} \tag{3.1}$$

where $r_{domainX}$ is the length of the domain in the X dimension. This also solves the problem of having to deal with continuous tuning parameters.

Newton3 With this option, the optimization using Newton's third law of motion, described in Subsection 2.1.1.3, can be toggled on or off. This might be interesting because it allows for more parallelism. Also, from a model correctness perspective, using Newton3 is not always valid, as was mentioned in Subsection 2.1.2.3.

Cell Sorting This tuning option is not part of the algorithmic configuration the AutoTuner selects, but part of the mechanic that handles the particle pair interactions between cells called the `CellFunctor`. The idea is to create a cheap ordering so that when we interact two cells A and B, we can, at some point, say that any further particles of B are farther away from the particle of A that we are currently looking at. Such an ordering is achieved by projecting all 3D particle positions on the 1D straight line connecting the two cells' centers. For interactions within one cell, the line is one of the cell diagonals. Particles are then processed in the order they appear on this line. The loop can be aborted as soon as the projected distances exceed r_c, skipping any further unnecessary distance calculations. This works because the difference in the projected 1D positions will always be smaller than the difference in the 3D positions. The implementation of this optimization in AutoPas was inspired by a publication by Pedro Gonnet [Gon07].

Adding the projection and sorting of particles to the interaction creates overhead that the optimization needs to overcome. Thus, there needs to be a certain number of particles involved in the interaction for this to pay off. Therefore, AutoPas only activates this feature if the sum of the number of particles in all involved cells is above a given threshold.

Traversals Parallelization strategies for the neighbor identification in each of the above-described containers are implemented in so-called traversals. Basically, they describe an ordering in which the container is processed and whose parts can safely be processed in parallel when making use of Newton3. There are a few design considerations when conceiving such a traversal:

Buffers vs Locks vs Synchronization The tradeoffs between these data race avoidance techniques have already been described in Subsection 2.1.5. At this point, traversals in AutoPas do not implement any buffer-based algorithms, mainly because of a concern for scaling memory consumption with the number of threads. All other traversals fall in either of the remaining two categories. In the context of AutoPas, synchronization strategies are implemented as so-called color (short c-based) traversals, while lock-based strategies are called slice-based.

Cache reuse When the whole simulation is too large to fit into the cache, loading parts of the data as rarely as possible from slower parts of the memory hierarchy is desirable. Therefore, traversals should process as many interactions as possible with the currently loaded data while at the same time not creating data races due to neighboring threads' overlapping regions of responsibility.

Predictability of access patterns A processor's ability to efficiently fill its caches is paramount to the performance of any program where the bottleneck is not the mere execution of operations. Thus, processors employ prefetching to try to guess ahead of time what data will be needed [CKP91]. Prefetching techniques are complex and work on many levels. However, they work best on code that follows predictable access patterns [LKV12]. For this reason, traversals employing dynamic load balance are at an inherent disadvantage because their patterns are less predictable due to the nature of the decision of which thread processes which part of the domain at runtime. To some degree, this can be mitigated by decreasing the granularity of tasks, e.g., by bundling up several smaller tasks, but this comes at the cost of load balancing precision.

Similar to containers [GSBN22], it provides extensive descriptions of the available options. Similar to containers, extensive descriptions of the available options are provided in the initial AutoPas release paper[GSBN22]. Thus, this thesis will only give short intuitions. For the current complete list and detailed description, see the official documentation[4].

Direct Sum Traversals For this container, only the `ds_sequential` exists due to a lack of importance of this container. It processes all particles on the domain sequentially because when Newton3 is active, data races can not be avoided.

Linked Cells Traversals A wide range of colored traversals exist for linked cells that mainly differ in their stencil, here called base-step, and task granularity. Named by the number of colors and ordered by their granularity, they are: `lc_c01`, `lc_c01_combined_SoA`, `lc_c18`, `lc_c08`, `lc_c04_HCP`, `lc_c04`, `lc_c04_combined_SoA`, `lc_sliced_c02`. Additionally, two slice-based traversals have been implemented. `lc_sliced_balanced` offers several heuristics, like squared number of particles per cell, to employ static load balancing, while `lc_sliced` trades this for eliminating any overhead. Both of them employ the `c08` base step and use a 1D slicing of the domain.

Verlet Lists Traversals Similar to Direct Sum, only the `vl_list_iteration` traversal for the simple Verlet Lists container exists. This, however, is due to the need for more structural information. It also does not support Newton3 but processes all lists in parallel.

[4] `https://autopas.github.io/doxygen_documentation/git-master/classautopas_1_1options_1_1TraversalOption.html` Accessed: 20.12.2024

Var Verlet Lists Traversals Also, only the `vvl_as_built` traversal is available for this container. It provides a cheap form of static load balancing by tracking which threads built which lists using the underlying Linked Cells data structure and then using the same thread mapping for the processing of these lists. This achieves a decent load balancing but is highly sensitive to hardware fluctuations.

Verlet Lists Cells Traversals Since the Verlet Lists Cells container associates neighbor lists with cells, all traversals that are available for Linked Cells are possible, as long as they only use the base cells lists, which eliminates those that depend on the `c08` base step. The implemented traversals are named after their Linked Cells counterparts: `vlc_c01`, `vlc_c18`, `vlc_sliced`, `vlc_sliced_balanced`, `vlc_sliced_c02`.

In order to create a `c08` like traversal, the neighbor lists have to be restructured. For each particle, they only must contain interactions with particles of one base step, and the lists have to be stored with the base cell of the corresponding base step. This implies that some lists are not stored with the cell in which any of its particles are and that there are several lists for each particle involved in several base steps. Such an implementation is provided in AutoPas under the name `vlc_c08`.

Pairwise Verlet Lists Traversals As the Pairwise Verlet Lists container is just the Verlet Lists Cells container with a different form of neighbor lists, the same traversals are available with the `vlp` prefix, plus `vlp_c08`, a Verlet equivalent of `lc_c08` as the pairwise lists offer enough structural information to make this base step possible.

Verlet Cluster Lists Traversals Traversals for this container work slightly differently since the internal tower structure is, from a scheduling perspective, equivalent to a 2D grid vs the 3D grid of Linked Cells based containers. This means that there is the `vcl_c06` traversal, which is the 2D equivalent of a `c18` based traversal, as well as `vcl_cluster_iteration`, which is the cluster-based equivalent of `vl_list_iteration`. Similar as for Verlet Lists Cells a (2D)`c08` based traversal is more complex to implement, since the neighbor lists do not contain information about the tower a cluster is in. Somewhat similar to the latter, but unique to this container, is `vcl_c01_balanced`, which is incompatible with the before mentioned traversal but, instead of a dynamic load balancing, employs a static load balancing by assigning towers to threads according to one of several heuristics, like length of their neighbor lists. Furthermore, there are 2D versions of all three sliced traversals.

3.1.1.4 Distributed Memory Parallelism Context

The scope of AutoPas is primarily limited to the node level. Nevertheless, since it is intended to work in the environment of large-scale HPC simulations that work with some form of distributed memory parallelization. Since AutoPas is designed with a black box interface, it does not particularly care which specific technology is used. However, due to its prevalence, it was evaluated in the context of several MPI applications.

Here, the idea is to instantiate AutoPas once per rank and let it manage all local particles. The local AutoPas object will then tune itself independently, which can lead to the interesting state that throughout a vast MPI based simulation, different regions or subdomains use different algorithms, leading to a heterogeneous algorithm landscape. Since AutoPas does not require any assumptions about its internal algorithmic configuration, the library is compatible with itself, meaning all AutoPas objects throughout the simulation can easily be used to exchange particles that move from one rank to another without having to consider things like algorithm changes, list or other data structure update cycles. Yet, as the exchange of particles or any boundary treatment is highly simulation and scenario-specific, this is not part of the library's functionality and must be implemented by the users themselves or, more precisely, the codes that AutoPas is embedded in. During it's update of the internal datastructure, AutoPas tells the user which particles have left the local domain by removing them and returning them. Incoming particles are accepted at any point of the simulation. However, if they are inserted while AutoPas conducts a (pairwise-)traversal it is undefined whether the inserted particle will be encountered in the further traversal or not. The same holds for inserting particles while using an iterator.

For further details about the implementation, interface considerations, and examples, see previous publications by Steffen Seckler [SGH+20, Sec21].

3.1.2 Software Engineering Aspects

Moving on from the user's perspective, this section discusses the developers' view from inside the project. The idea and purpose are to give insight into underlying design processes, decisions, and essential internal mechanics.

3.1.2.1 Black Box Container Interface

To achieve the goal of providing a black box interface, AutoPas needs to unify the interfaces to all internal data containers. Generally speaking, there are two kinds of interface styles. The crucial points revolve around particle insertion and the update policy of the internal container, meaning the rebuilding of lists and resorting particles that have moved into their correct cells:

Verlet Lists Style This interface follows the idea of Verlet Lists, allowing the addition and removal of particles only when rebuilding the neighbor lists. Inserting particles at any other point invalidates the lists, triggering a rebuild. Containers are only updated every few iterations, leading to particles that are outside the cell they are stored in by up to $\frac{r_s}{2}$.

Linked Cells Style In every iteration, the container updates to allow the addition of particles at any time. Particles will always be stored in the cell they belong to when looking at their position.

Strictly implementing a Linked Cells style interface for AutoPas is impossible as the continuous updating is incompatible with the internal Verlet Lists containers. A strict

Verlet Lists style interface is possible in two ways. Either an update method that behaves differently depending on the underlying container or changing the Linked Cells container, extending all cells by r_s as suggested in Subsection 2.1.3.3, and only updating it in intervals. Nevertheless, this would leave the user responsible for cautiously interacting with the AutoPas object and only inserting particles when appropriate, rendering especially MPI based simulations more error-prone.

For AutoPas, the decision was made to implement a hybrid style to maximize the library's usability, which is one of the primary design goals. From the user's perspective, the main interface feels like a Linked Cells style interface where particles can be added at any time, and `updateContainer` has to be called in every iteration. On the inside, however, a Verlet Lists style interface is implemented by the `LogicHandler`. This means that if particles are added to AutoPas at a point when they would invalidate the neighbor lists, instead of adding them directly to the container, they are added to buffers in the logic layer. Whenever particles are iterated or pairwise interactions are evaluated, the `LogicHandler` has to ensure that any particles from the buffers are correctly interacting with each other and the container. Upon a container update, anything from the buffers is inserted into the actual particle container.

The update function only updates the internal data structure and neighbor lists when necessary but will always return any particles that have left the domain. This means non-Verlet containers are also only rebuilt every few iterations, and their internal structure is extended by r_s.

Since data structures must not be changed between updates, no particle deletions are possible. We, however, still need to delete particles e.g. when they leave the container or are eliminated due to the simulation scenario. This is achieved by marking these particles as deleted by setting their `OwnershipState` to `dummy`, which excludes them from any interaction calculations. They are removed upon the next full container update.

3.1.2.2 Providing Usability for Frequent User-side Activites: Options

Each code that uses AutoPas and wants to provide their users some mechanism to configure the library has to use the option `enum`-like types provided by AutoPas, e.g. `ContainerOption` or `TraversalOption`. Usually they then need functionalities like converting these to strings, parsing from strings, or getting the set of all possible values. To improve the library's usability and enhance the user experience, AutoPas implements a mechanism that provides all of the above for all its option classes.

The pattern, shown in Figure 3.6, consists of a base class `Option`, and several derived classes for the actual options. The base class provides implementations of common functionalities like, string conversion, parsing, and returning sets of values. For it to be able to work seamlessly with the data types of the actual option, it implements the Curiously Recurring Template Pattern (CRTP) [Cop96], generally known as F-bounded polymorphism [CCH+89], where each derived class passes itself as a template parameter to the base class upon declaration. All derived classes define an enum for the possible values and a mapping of these values to strings, which is used by the base class for parsing and string conversion. Furthermore, the derived options implement a function

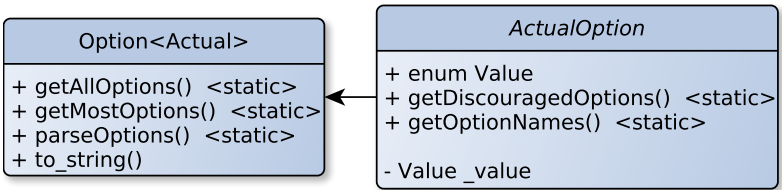


Figure 3.6: AutoPas option mechanism implementing CRTP.

`getDiscouragedOptions()`, which can be used to point out values that are possible, but unlikely do be desired by the general user. An example for this would be the `ContainerOption DirectSum` due to its $O(N^2)$ runtime complexity.

Another advantage of this setup is that implementations in the base class can be made abstract from the actual options and optimizations and enhancements can be used immediately by all derived classes. For example, the parsing was extended with the Needleman-Wunsch algorithm [NW70] to enable fuzzy parsing, which matches the input against the closest matches.

3.1.2.3 Merging Common Behavior: CellPairTraversals

To adhere to the design goal of high code maintainability, AutoPas has to identify overlaps and abstractions to implement common algorithm behavior only once without sacrificing performance. On the one hand, this can be done at the algorithmic level, where, for example, we see that Verlet Lists have to build on top of Linked Cells to efficiently rebuild the neighbor lists. Therefore, it makes sense to implement any Verlet Lists container on top of a Linked Cells container, using all optimizations implemented in the latter. On the other hand, this can also be done on the logical level, which means forming abstractions where behavior overlaps. An example of this is the abstraction by inheritance in the cell pair-based traversals of AutoPas, which is visualized in Figure 3.7. Here, going from left to right, the leftmost class `CellPairTraversal` defines the interface, the next layer the OpenMP parallelization style. The third layer handles how the colors or slices are distributed, and finally, on the far right, the remaining details are implemented, resulting in the actual traversals.

3.1.2.4 Abstracting Specialized Behavior: `ContainerIterator`

As performance is one of the most important design goals of AutoPas, it is necessary to identify where specialized behavior is necessary, e.g., on the specific data structure level, and make it accessible via an easy-to-use abstract interface. A prime example of this is the implementation of iterators, here called `ContainerIterator`. Iterators provide a way to access particles in the AutoPas object sequentially. There is no index-based random access. However, iterators can be restricted to given `OwnershipState`s and regions.

Iterators provide a quick and straightforward way for users to interact with particle data in AutoPas. For this, they need to be cheap to instantiate, must not have signifi-

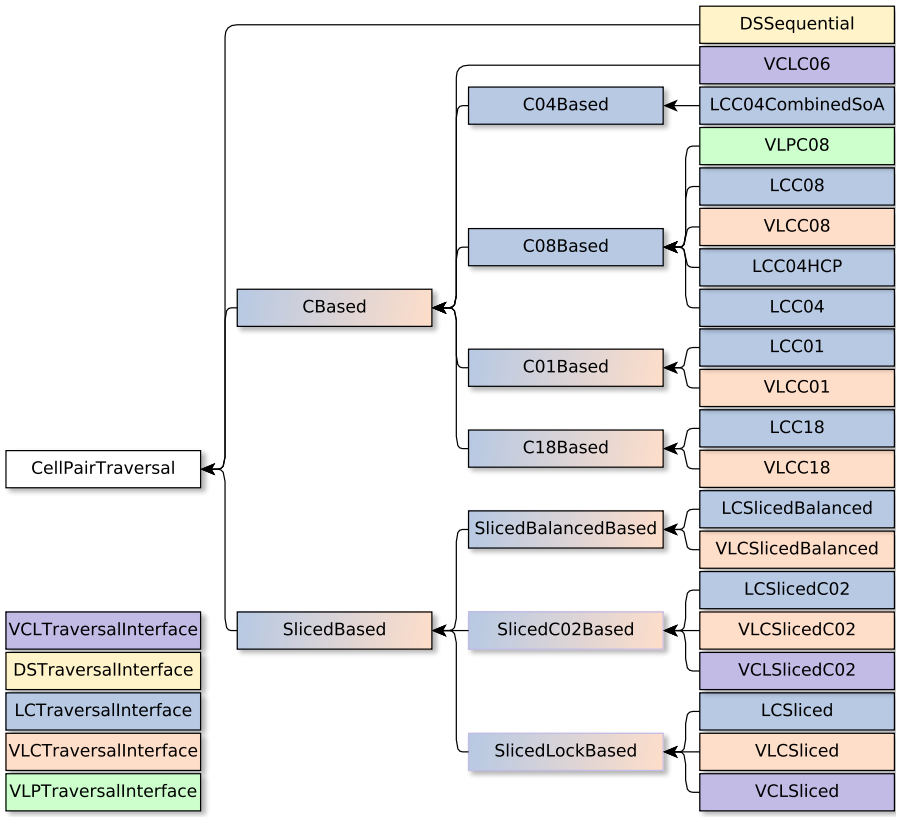


Figure 3.7: Inheritance relations of cell pair-based traversals. Arrows indicate an inheritance relationship, and colors indicate alignment with a container-related interface. All boxes on the far right represent traversals. Everything else is interfaces that gradually specify the behavior. Figure based on [GSBN22].

cantly more overhead than those of, e.g., `std::vector`, and especially the variants that are restricted to a specific region, should be highly optimized for the current particle container for maximal speed.

Following this principle, the user requests an iterator via the main interface's `begin` function, which invokes its counterpart in the logic handler. Multiple versions exist that all append the particle buffers and then forward the request to the container but set the appropriate template parameters, whether it is a region iterator, a const iterator, or both. This principle of abstract and unified simple interfaces hiding specialized behavior becomes even more apparent when looking at the increment operator of the `ContainerIterator`. An iterator has to solve the problem if given a certain particle to quickly find the next one and eventually coming across every particle in the container exactly once. In AutoPas this is particularly challenging since the internal data structures differ significantly and polymorphism by providing one iterator per container is too expensive.

Thus, from a conceptual standpoint, the `ContainerIterator` concept revolves around an iterator that maintains two indices for uniquely identifying a particle within any given container data structure. An index named `_vectorIndex` pinpoints the storage unit within the container, such as a cell, tower, buffer, or something similar. Complementing this, `_particleIndex`, identifies the particle within that unit. The trick is that these indices hold meaning solely in the context of the current container. What is guaranteed across all containers is that the index pair 0/0 will invariably point to the container's first particle. Subsequently, it is up to the container to inform the iterator about the next set of valid indices, a process exemplified in Figure 3.8.

From the user's perspective, they simply invoke the `operator++` on a `ContainerIterator` object. In the iterator, this then increments the `_particleIndex` intending to get the next particle from the storage unit, or if the end of it has been reached, getting the first from the next. Depending on whether the iterator is still processing the container or has already progressed to any additionally appended vector, it processes those or queries the container with the index pair. Here, a crucial maintainability and consistency optimization is that both the container and the iterator use the same free function to judge if a particle fits the requirements of the iterator. The specialized performance optimization in this process is that the container can use the `_vectorIndex` to directly access the correct storage unit, implementing a mechanism similar to random access as close as possible in these kinds of data structures. This solves the key problem of providing access to this and the next particle in $O(1)$. In the further process, the container identifies the next suitable particle, if there is any, and returns a pointer to it together with its corresponding index pair, which is stored in the iterator to identify its position in the container and where to go next. Additionally, the iterator also stores the pointer to the particle to facilitate subsequent access to it.

In order to avoid any code duplication between regular and region iterators, the latter is realized by placing additional location checks and vector index jumps in both the `ContainerIterator` and the container classes, which are enabled via a template flag. Since the region variant of the iterator has to store the region's bounds but the regular iterator does not, an optimization can be applied to avoid the unnecessary instantiation

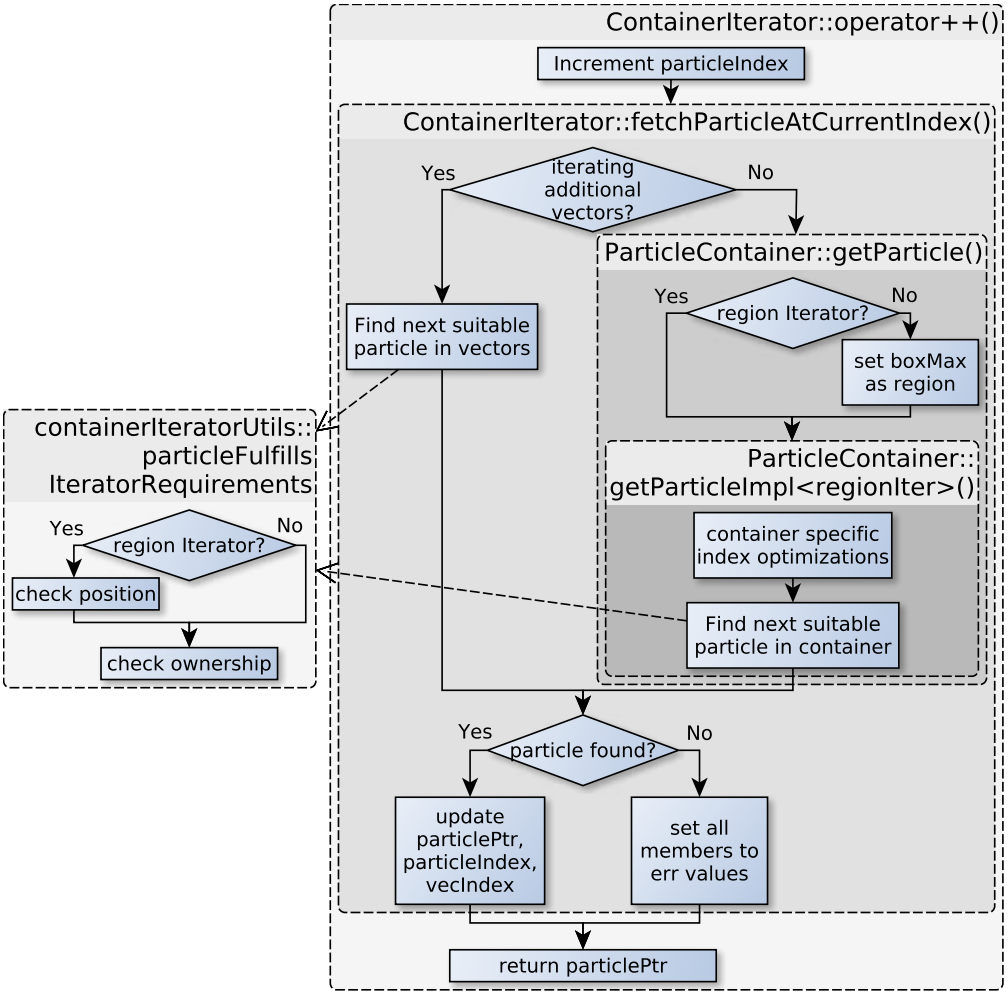


Figure 3.8: Logic flow and involved classes of the incrementation operation of the `ContainerIterator`. Black filled arrows indicate algorithmic forward flow. Dashed arrows represent calls to helper functions.

of the respective arrays. As seen in Listing 3.3, the type of the member variables depends on the region iterator template flag. According to this, no memory for the coordinates is required if the type is set to the empty struct. The annotation guarantees that no unnecessary dummy allocations take place.

```
1 template <class Particle, bool modifiable, bool regionIter>
2 class ContainerIterator {
3   // rest of class definition ...
4   struct Empty {};
5   using RegionCornerT =
6     std::conditional_t<regionIter, std::array<double, 3>, Empty>;
7   [[no_unique_address]] RegionCornerT _regionMin{};
8   [[no_unique_address]] RegionCornerT _regionMax{};
9 };
```

Listing 3.3: Implementation of the mechanism for optionally allocated memory for the region iterator boundaries. The member variables `_regionMin` and `_regionMax` are only needed if the iterator is a region iterator, which is defined via the template parameter `regionIter`. Thus, depending on the template flag, their type is either a 3D array or an empty struct. Additionally, they are tagged to inform the compiler that it does not have to provide unique addresses for these variables. This means that in the non-region iterator case, these members still exist but solely as symbols with no memory or overhead of their own.

Region and regular iterators also need dedicated constructors that mainly differ in passing region bounds. To prevent erroneous usage, static checks are in place that compare the constructor to the template flag.

Like the region variant, the const variants are basically the same code. Therefore, const iterators are implemented by casting the const onto the iterator in the `ParticleContainerInterface` layer.

To summarize, the combination of the above mechanisms implements four iterator variants with only one code instance while providing container-specific optimizations and safeguards against misuse.

3.1.2.5 Code Generation for User Types: Generated SoA

As discussed in paragraph 3.1.1.2, AutoPas leaves the definition of the particle class to the user. This class is then used for the AoS data layout. Nevertheless, the library also aims to provide an SoA layout and wants to avoid offloading the burden of implementing this to the potentially inexperienced user. For this, AutoPas provides a mechanism where the user adds some type information to his class from which then the whole SoA data structure, as well as the interaction with it, is generated via template metaprogramming. This is visualized in Figure 3.9.

As already shown in Listing 3.1, the user has to define an `enum AttributeNames` that has an entry for a pointer to itself and at least all class members that are used in the functor. `SoAType` employs variadic templates to store all corresponding data types. The information from these two constructs, in combination with the template-based getter

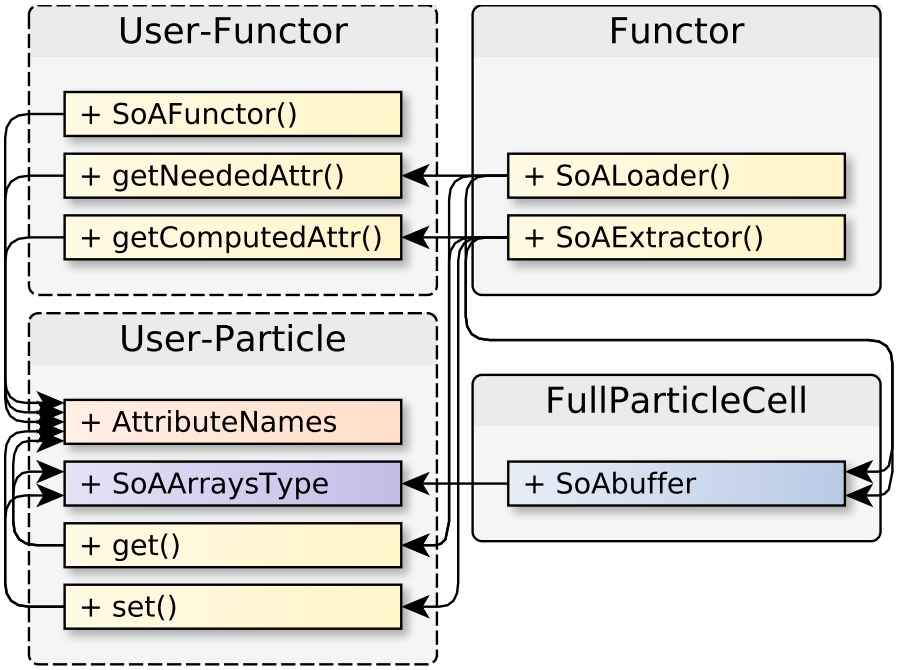


Figure 3.9: Sketch of the data and information flow of how AutoPas generates tailored SoA for arbitrary user particles and accesses them in the user-provided functor.
As explained in more detail in Subsection 3.1.2.5 the user describes their particle class members in their Particle's `AttributeNames` enum and their types in the `SoAArraysType` tuple. With these, the Particle also has to implement getters and setters that take an `AttributeName` as template argument and obtain the respective type information from the `SoAArraysType`. In their functor, the user declares which members of the particle class are needed and computed by the functor by putting the respective `AttributeNames` into the functor's functions. Using all the above, the `SoALoader` and `SoAExtractor` from the library's `Functor` base class can then instantiate the cells' `SoABuffer` and load the needed particle data into it, as well as extract it.
Figure based on [GSBN22].

and setter function, provide a rudimentary form of reflections for this class. Storage classes, like `FullParticleCell`, then access the type information, as shown in the lower half of Figure 3.9, and generate one array per type. For getting the AoS data in and out of this SoA structure, the base functor class provides so-called loader and extractor functions, which make use of the template-based getter and setter functions from the particle class. To limit the load and store operations to only the particle data that is needed, the loader and extractor functions are only called with the attribute `enums` defined in the `getNeededAttr()` and `getComputedAttr()` functions of the functor.

For placing the minor effort of implementing the extra type information on the user, This mechanism enables AutoPas to work with all types of particles in SoA form to sustain maximal efficiency.

3.1.2.6 Neighbor List Memory Management

As already discussed in paragraph 2.1.3.3, any form of Verlet Lists is very memory intensive, even so that the memory for neighbor lists can dwarf the memory needed for the particles. Therefore, looking at how this memory is managed is essential, especially concerning allocations, deallocations, and reallocations. Since the lengths and number of lists can not be known at compile time, neighbor lists are usually allocated on heap memory. These allocations can be very time-consuming since they involve complex algorithms to identify free memory blocks and might even trigger calls into the operating system [GM09]. Thus, the number of such allocations should be kept to a minimum, especially if the employed force potential is not very memory intensive like the Lennard-Jones 12–6 potential [Tch20], meaning that instead of doing several smaller allocations, only doing one big.

Verlet Lists Cells For Verlet Lists Cells, three different layers of allocations have to be considered:

List of lists per cell The vector holding a bundle of lists for each cell. This has to have a size equal to the number of cells.

Lists per cell The vectors holding all lists related to one cell. Depending on the traversal, this is either the number of particles in the cell or, in the case of a c08-based traversal for all particles in a base step. The latter has to be estimated. Each entry in these vectors is a pair consisting of a pointer to a particle and a neighbor list.

Neighbor lists The vectors holding the actual neighbor pointers. These vectors' lengths have to be estimated.

Looking at Figure 3.10, we can see the impact of optimizations for these three levels for a small benchmark simulation using `vlc_c08` over ten Verlet rebuild cycles.

The blue bar shows the behavior for no optimizations at all. This means the vector of lists per cell is cleared in every rebuild, leading to all other vectors being deallocated. They are then rebuilt with `push_back` operations without previous calls to `reserve`.

In the orange column, the optimization of not clearing the first layer is implemented, meaning the pairs in the lists per cell are reset to a `nullptr` and the neighbor list cleared. Note that none of this deallocates any memory. Hence, we see a drop of 83% in the accumulated size and 44% in the number of allocations. Interestingly, this optimization, on its own, only leads to a minor speedup of 1.6%, initially suggesting that the number and size of allocations do not matter too much.

Optimizations depicted by the green bar target the second layer and implement a preallocation for the number of lists stored per cell. As this benchmark employs `vlc_c08`, the allocation size can only be done with an estimate. The heuristic examines all interacting cell pairs in a base step, their relative positions, and the number of particles. These particle numbers are then added up with factors depending on the relative cell positions, to estimate the number of lists that need to be created in the base cell of the step. Since it is cheaper to overestimate than to dynamically reallocate, the factors in the implementation were chosen very generously, ranging from 1.2 to 8 times the memory needed per list with larger overestimates for smaller ones. The figure shows that this barely changes the number and size of allocations relative to the blue baseline. This is to be expected because, as in the green case, the primary structure is rebuilt every time, only very few allocations can be saved as the number of lists per cell is rarely very high. Nevertheless, the reduction of repeated reallocations triggered by `push_back`s in the loops that add lists yield a speedup of 12%.

The red bar stands for the impact of estimating and then preallocating the length of the neighbor lists, optimizing the third layer. Similar to the second layer optimization, this is a number that is very hard to estimate and highly fluctuates from list to list, even in homogeneous scenarios. In the implementation at hand, the estimate simply calculates how many particles would be in a sphere of length r_i if they were uniformly distributed throughout the domain and uses this value for all lists. Thus, the estimates here are even more generous, which leads to a significant increase in the total allocation size because even the shortest lists now have the same reservation size as the largest ones. On the flip side, this reduces the number of allocations significantly, in a similar order of magnitude to the primary layer optimizations. The speedup this gains is a little over 13%, slightly higher than the speedup achieved by the second layer optimization despite the egregious overallocation. This confirms the expectation that allocating large chunks of memory in one go is cheaper than triggering smaller allocations on the tested Intel x86 architectures. Should the allocation sizes grow too large, this gain will diminish. With the current implementation, this could especially be the case in a large and extremely inhomogeneous scenario because our estimate of the list length depends on a average particle distribution. Hence, for most of the domain lists would be significantly too large, leading to a lot of unnecessarily allocated memory that can potentially cause slow allocations. Also on architectures where memory allocations have a different runtime profile or that do not have as much memory available behavior can change.

Looking at the individual runtime improvements or speedups, it seems like the preallocations for the number (green) and lengths (red) of lists have a significantly higher impact on the time to solution than the re-usage of the primary structure (orange). However, when employing only the second and third-layer optimizations but not the

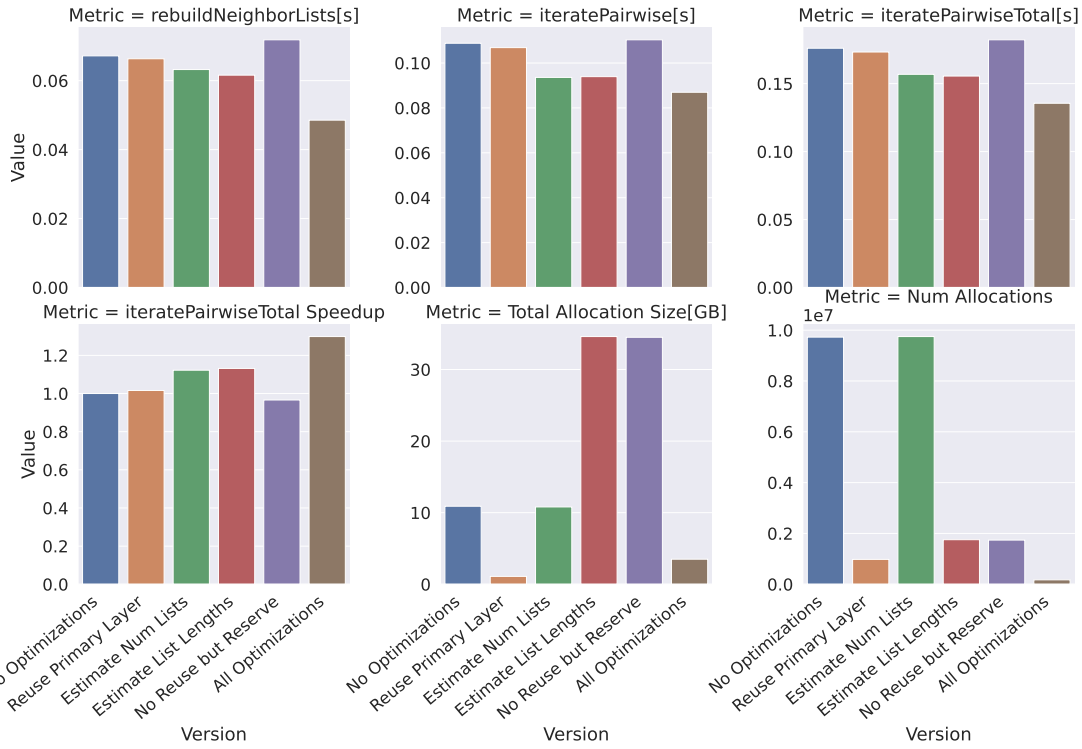


Figure 3.10: Impact of different memory optimizations on Verlet Cluster Lists. Average timings and accumulated memory allocations over ten rebuild cycles. Only the combination of all optimizations yields a significant speedup. At the time of the benchmark, about 6 million allocations came from outside of the list rebuilding. For details about the setup see Section A.1.2.

first, we actually get a slowdown, as shown by the purple bars. This is because the overall number is still very high due to the generous estimates, but they have to be done repeatedly because the lists are not reused.

Only when all three optimizations are applied together, a significant speedup of about 30% in this case can be observed, shown by the brown bar. The reason why the total size of allocations for all optimizations (brown) is higher than only optimizing the first layer (orange) is because the second and third-layer optimizations overestimate the memory needed, leading to fewer but bigger allocations.

Verlet Cluster Lists For some container implementations, retaining the list structure is not as straightforward, for example, because the lists are integrated into another data structure that is rebuilt regularly. To improve memory management efficiency for such containers, AutoPas implements a neighbor list buffer that acts as a memory pool that provides neighbor list objects that are never deallocated. This class allocates space for a given number of lists of a given length and never lets go of the memory. It associates

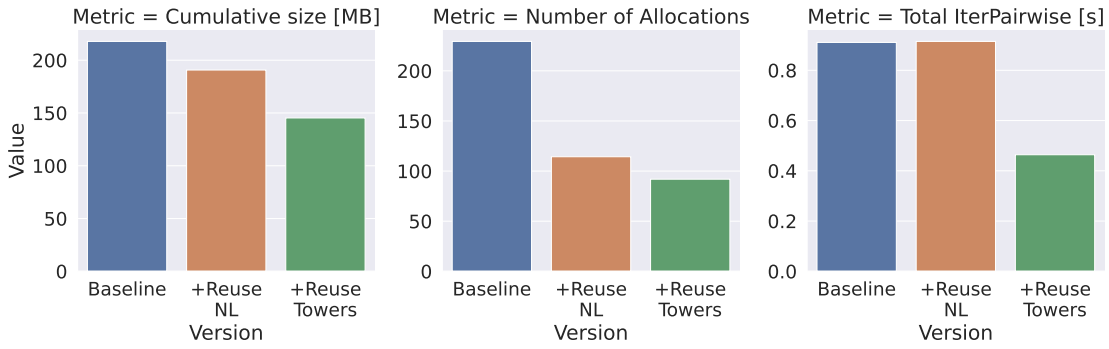


Figure 3.11: Impact of multiple memory optimizations on Verlet Cluster Lists. "+" means that the optimizations to the left are also included. Combining both optimizations yields a speedup of about 50%.
For details about the setup see Section A.1.2.

its lists via indices with a key, which is defined as a template argument, e.g., a pointer to the list's particle. A Verlet Lists style container can then use these lists by requesting new lists, which returns the index of one of the previously allocated lists. If more lists are required, they are added dynamically, and individual lists can be grown with a customizable growth factor. When the container no longer needs the lists, e.g., because it is about to rebuild them, the lists are not deleted or their content destroyed, but just the index mapping is flushed. Only when the same list is given out as a seemingly new list again its content is destroyed, not its memory reallocated. This keeps the overall memory movement to a minimum since even if initial estimates about the list sizes are off, enough space will be allocated and reused for the remainder of the simulation.

This neighbor lists buffer is used to optimize the memory management of the Verlet Cluster Lists container, and its impact is shown in Figure 3.11. Similar to the optimizations for Verlet Lists Cells, a major improvement in time to solution by about 50% is observed when optimizations across all layers are combined.

3.1.3 Hardware-aware optimizations

Even though AutoPas optimizes node-level performance, the user has to provide the force computation kernel in the form of a functor, themself, and thus its optimization. For MD AutoPas comes with several examples of hand-optimized implementations for the Lennard-Jones potential for different platforms. There are AVX and AVX-512 intrinsics-based functors for x86 processors and an Scalable Vector Extension (SVE) based for ARM architectures of level ARMv8.2-A or newer.

In Figure 3.12, the performance of the different functors relative to explicitly disabled vectorization and AoS layout. It can be seen that only switching to the SoA layout even slows down the simulation, most likely due to the overhead from switching data layout and no possibility to benefit from loading multiple particle data. Also, the vectorization

level does not significantly impact computations done in the AoS layout. This is expected since they use the same code, which offers no real vectorization potential. Thus for the rest of this analysis we focus on the performance of versions using SoA. We see that the hand-crafted version using AVX2 intrinsics achieves a speedup of just over three compared to the not vectorized AoS baseline. Even though four particles can be processed in parallel, a speedup of four is not achievable for two reasons: A speedup of four would necessitate the code to be 100% vectorized, and at least the logic to find the next cell is not. Secondly, as discussed in Subsection 2.1.6, the amount of unnecessary force calculations also significantly impacts the maximal achievable speedup by vectorization. Hence, we consider three a good speedup from AVX. The middle columns show the performance of code that was written without intrinsics but automatically vectorized by the compiler, in this case, Clang 17. Compared to the not vectorized version, it achieves a speedup of almost two but lags behind the hand-crafted version by almost a factor of two. This suggests that the compiler was only able to make use of half of the vector width or less vector instructions, even though the binary file clearly shows widespread usage of AVX instructions.

Since manually optimizing the interaction kernel, e.g., using intrinsics functions, can be tedious and potentially require some expert knowledge, attempts were made to facilitate this.

One approach is to create a Domain Specific Language (DSL), which allows us to express the functor in simple terms. The structured way of a DSL can then be used to generate `C++` code for all versions of the functor functions, which are subject to the optimizations employed by the compiler. This approach was implemented in the scope of a student project, showing promising results, especially in the AoS case, but performance for the SoA case is still suboptimal [Gä19]. Due to a lack of applications at the time of the student project, no further investigation was undertaken. However, there are no known definitive obstacles that would prohibit equivalent performance.

A second approach is to still implement the functor manually, however not in platform specific intrinsics but using platform independent SIMD wrappers e.g., Google Highway[5].

3.2 Dynamic Auto-Tuning

Now that the zoo of algorithmic choices and their optimizations have been discussed, this section will focus on the second pillar of the library. One of the main features of AutoPas is its ability to autonomously adapt its internal algorithm for the particle interaction to the current characteristics of the simulation to provide optimal time to solution. This process is called dynamic auto-tuning or tuning at runtime.

[5] `https://github.com/google/highway` Accessed: 20.12.2024

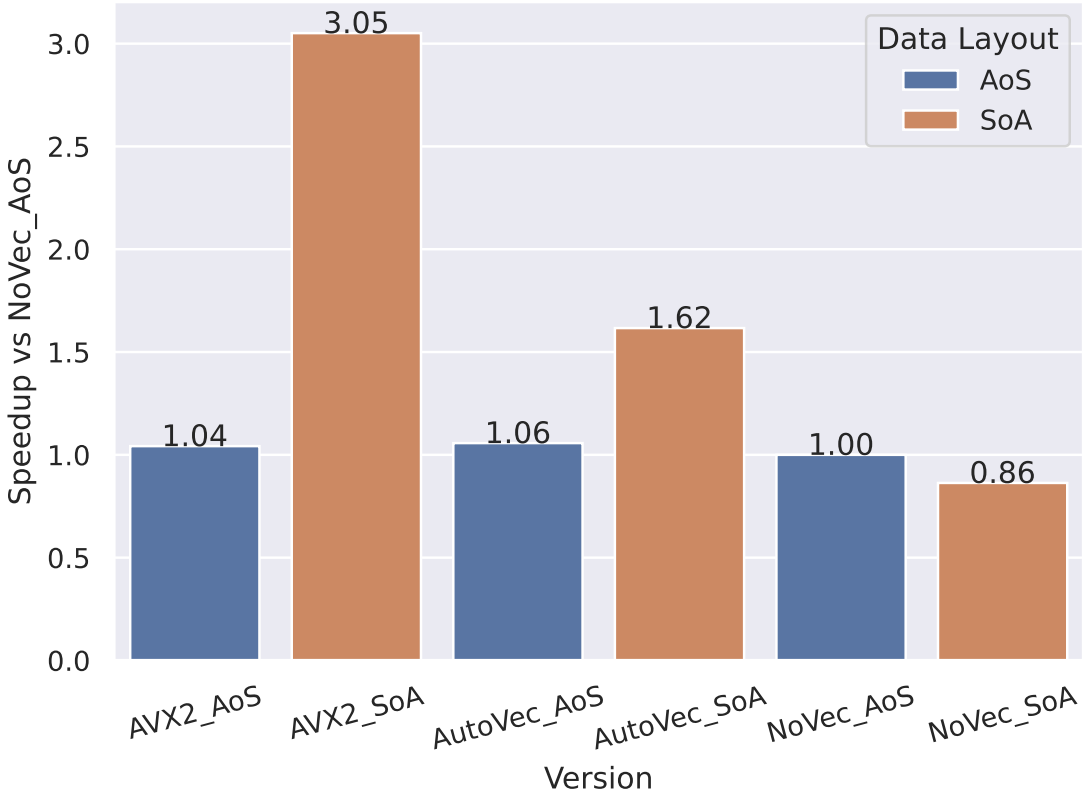


Figure 3.12: Speedup of `iteratePairwise()` from vectorization for the Linked Cells container. A handwritten intrininsics functor achieves significantly better performance than the auto vecorization by the Clang 17 compiler.
For details about the setup see Section A.1.2.

3.2.1 Translating Theory into the Implementation in AutoPas

Formally, dynamic auto-tuning here means automatically solving an algorithm selection problem, so we need to define the sets we described in Subsection 2.2.1 for the implementation in AutoPas.

Problem space $\mathbb{P}$ Particle simulations typically have a very narrow time step width, as mentioned in Subsection 2.1.1. Therefore, we assume that the positions of particles, and thus the state of the simulation relevant for the tuning, is the same in subsequent time steps. We can use this assumption to evaluate multiple algorithms on subsequent time steps and compare their performance directly.

Further, based on the knowledge that these systems evolve very slowly, we assume that the algorithm a_{opt}, which is chosen to be optimal, will be the optimum for several time steps. Currently, this number has to be supplied by the user. However, in theory, it should be possible to observe the phase state and decide when it has changed far enough that it warrants a reevaluation of a_{opt}.

With these assumptions, the size of the problem space, the number of different problems for which we need to find the optimal algorithm is reduced to:

$$|\mathbb{P}| = 1 \tag{3.2}$$

Algorithm space $\mathbb{A}$ For AutoPas, the content of a `Configuration` object defines the entire algorithm employed for executing the particle interactions. This means that the search space consists of the Cartesian product of all available options minus incompatible combinations. For example, the container parameter can be disregarded for the cross-product since each traversal is only compatible with precisely one container. Also, a few traversals are not compatible with Newton3. The number of available algorithm configurations equals the entire search space, thanks to Equation 3.2 and disregarding continuous options. At the time of writing this dissertation, this size is:

$$|\mathbb{A}| = 160 \tag{3.3}$$

See Section A.7 for a complete listing.

This size is already too large to be tested exhaustively, so adding more choices for existing options or even new options should be done carefully, as this quickly increases the size of the total search space. As discussed in Subsection 2.2.2 solution strategies have to focus on only evaluating a small subset of $\mathbb{A}$.

To further complicate matters, very little is known about the structure of $\mathbb{A}$. Usually, the choice of the container tends to have the most significant impact on runtime, but deciding a priori which container is the most optimal or even which containers will perform similarly is generally hard since this is impacted by many factors, as already discussed in Subsection 2.1.4.

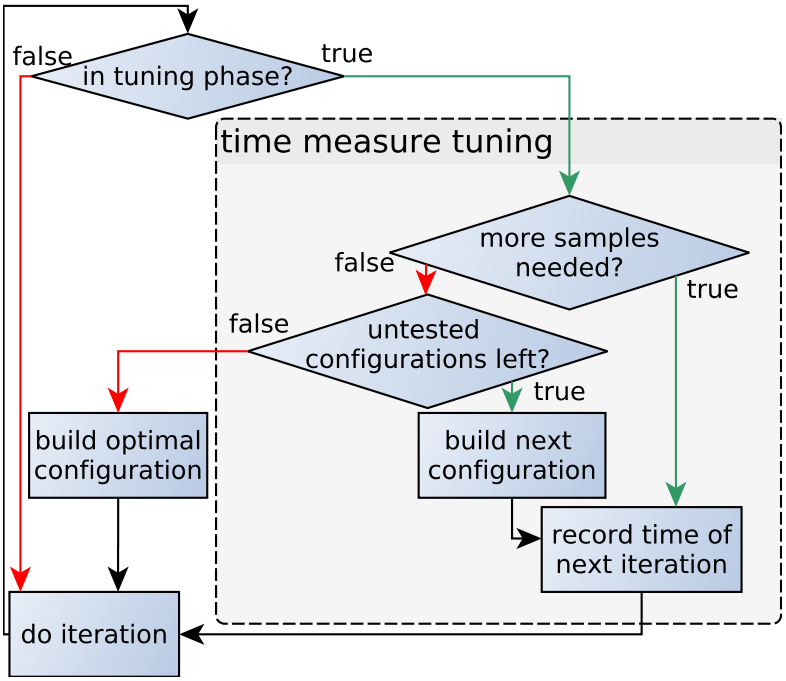


Figure 3.13: The high-level logic flow of the AutoPas tuning Loop.

3.2.2 Tuning Loop

While the previous section explained how the theoretical concepts are mapped to AutoPas, this section explains how these concepts are actually implemented.

As mentioned in Subsection 3.2.1, we assume that the simulation evolves slowly over the course of thousands of iterations. Thus, the general approach is to identify a currently optimal configuration, apply it to the simulation for a few thousand iterations, and then reevaluate. This back and forth is referred to as a tuning cycle consisting of the tuning and the non tuning phase.

The whole tuning logic can be viewed as a loop as sketched in Figure 3.13. If AutoPas is not in a tuning phase, it simply reuses the current configuration. In the other case, it samples all potentially interesting configurations, and measures their performance by computing the subsequent iterations with them. As soon as all candidate configurations have been evaluated, the optimum can be selected and the circle can be cut short again until the next tuning phase.

Looking more closely into the implementation, the heart of the implementation of the tuning logic is the `AutoTuner` class, which was already mentioned in Subsection 3.1.1.1. It is responsible for managing the available algorithm space, here called search space $\mathbb{A}$, any measurements taken, applying tuning strategies, and identifying the optimal configuration. Before diving into the optimization algorithms, the overarching algorithm of the tuning pipeline needs to be discussed. This is depicted in Figure 3.14.

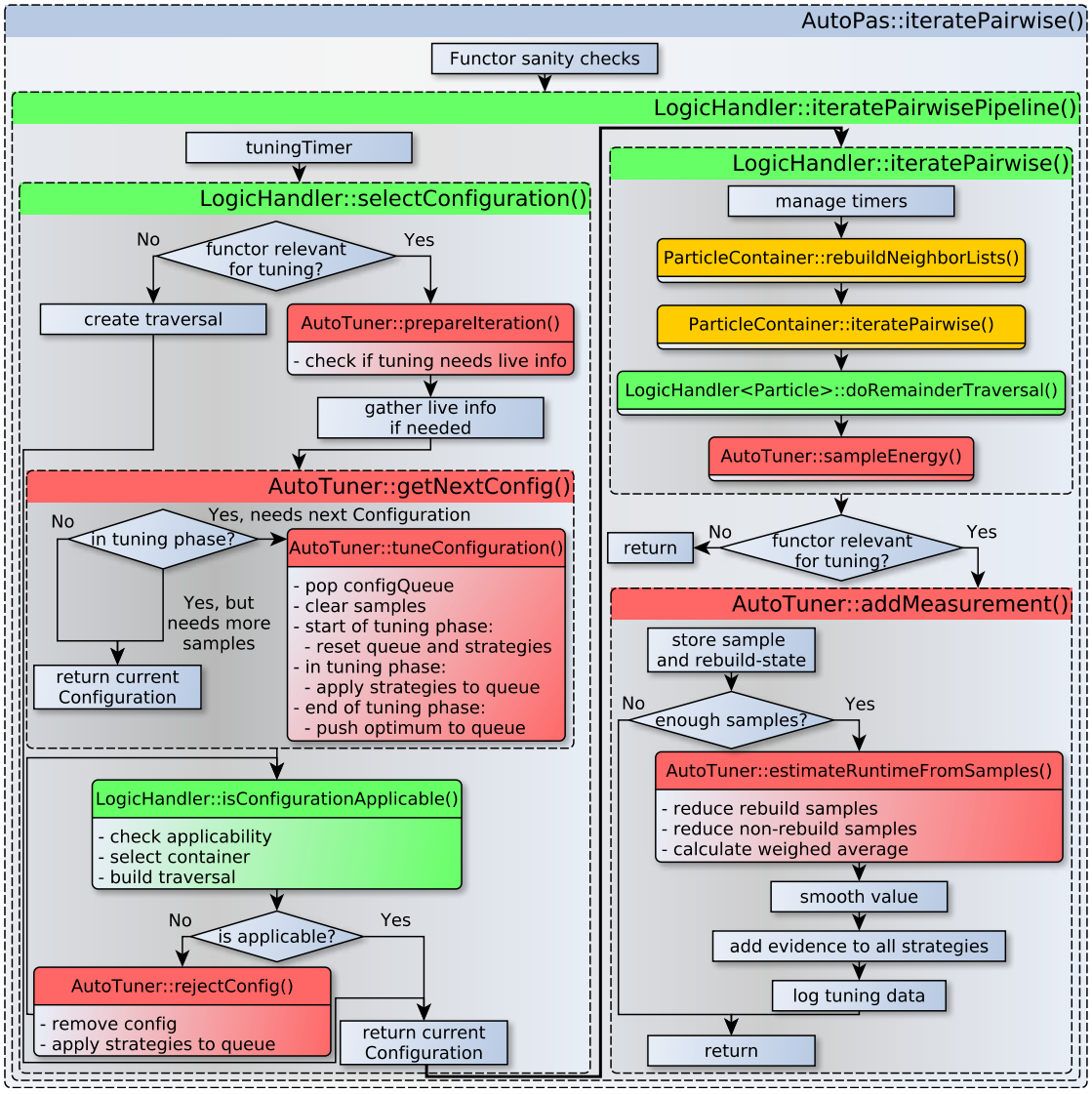


Figure 3.14: The flow of the auto-tuning logic through the internal modules of AutoPas. Gray depicts classes, color-filled boxes functions, and blue nodes summarize several minor steps. The different filler colors indicate the class that implements the step. When the interaction computation is triggered from the main interface via `AutoPas::iteratePairwise()`, it is passed to the `LogicHandler`'s pipeline. First, the `LogicHandler` is responsible for selecting a configuration. For this, it passes through several checks and, in the case of a tuning phase, uses the `AutoTuner` to come up with potential candidates or the optimal configuration. After ensuring that the configuration is also applicable to the scenario at hand, the `LogicHandler` applies the configured traversal on the selected container and takes measurements. These are then passed on to the `AutoTuner` for further processing and as evidence for further tuning.
Green: `LogicHandler`.
Red: `AutoTuner`.
Yellow: `ParticleContainer`.

Following the design goal of optimizing usability by keeping the user side API as small as possible, the whole tuning logic is hidden behind the particle interaction functionality. Whenever `AutoPas::iteratePairwise()` is invoked from the main interface, the top of Figure 3.14, the tuning pipeline is triggered.

After sanity checks about the functor's type and cutoff, the initial function call is passed on to the `LogicHandler::iteratePairwisePipeline()`. The first and most complex step is selecting a suitable configuration. The available discrete configurations are stored as a sorted queue in the `AutoTuner`, and the end of the queue is used as the currently active configuration. Should the functor be marked as irrelevant for the tuning process, for example, an auxiliary functor for FLoating Point Operation (FLOP) calculations, the whole selection process is skipped, and the currently selected configuration is used. In the other, more usual case that the functor is relevant, so-called live info is gathered by the `LogicHandler`, should the tuner need it. This includes data about the current state of the simulation, like the number of particles, density and distribution statistics, and number of threads available. Subsequently, the next configuration to be applied has to be identified. Several cases can arise here: either we are currently not in a tuning phase, then the current configuration is the optimum and can continue to be employed, or we are in a tuning phase. Since time measurements are sometimes very short and thus noisy or because Verlet style containers sometimes have slower iterations for list rebuilds, more than one sample per configuration should be collected. The user can set the number of samples. So when in a tuning phase, if we need more samples from the current configuration, we carry on with it, or if enough samples were collected, the next configuration has to be identified.

This triggers `AutoTuner::tuneConfiguration()`, the heart of the tuner. In short, this function pops the just-used configuration from the queue of configurations and updates it. In practice, this update depends on where in the tuning phase we are, which is identified via internal iteration counters. At the start of a tuning phase the queue would now be empty, has to be repopulated with the whole search space, and then reorganized by the tuning strategies, which will be explained in the following Subsection 3.2.3. During a tuning phase, only the strategies are applied to update the list with newly available information and evidence so that the following configuration that will be tested at the end of the queue. Lastly, at the end of a tuning phase, the queue is empty again, and the optimum must be identified from the collected evidence. It is then reinserted into the queue and used until the next tuning phase.

Note that all operations that manipulate the queue are very cheap since the queue only contains objects which describe the configurations.

Before a configuration can be used, the `LogicHandler` has to confirm that it is actually applicable to the situation at hand. Reasons for inapplicability could include scenarios where the domain is not large enough to support the traversal's shared memory parallelization pattern or the functor does not support the Newton3 mode.

Should the configuration be rejected in this step, it is removed from the queue, and if it was removed for reasons that will not change during the simulation, like the Newton3 support, also from the whole search space. Then, the tuning strategies are applied again,

and a new candidate is tested for applicability until either one is found or an exception is thrown because no configuration can be applied to the scenario.

When an applicable configuration is identified, the `LogicHandler` applies it to the actual particle container after triggering it to rebuild its neighbor lists if necessary. Afterward, the remaining interactions between the container and the logic layer buffers needed for the Linked Cells style black box interface explained in Subsection 3.1.2.1, as well as interactions within them, colloquially referred to as the remainder traversal are computed.

The time it took to execute each of these steps is measured. If the functor is marked as relevant for tuning, these measurements are stored in the so-called evidence collection of the `AutoTuner`. One piece of evidence is a tuple of configurations, measurements, iteration, and tuning phase numbers. When all samples required for the configuration are collected, the tuner calculates a weighted average of the iterations according to Equation 3.4.

$$\bar{s} = \sum_{S_r} s_r/|S_r| + \sum_{S_n} s_n/|S_n| \cdot t_r \tag{3.4}$$

Here, $S_{n|r}$ are the sets of measurements from (non-)rebuilding iterations, t_r the Rebuild Interval, and $\bar{s}$ the averaged sample. This way, if there is at least one sample from a rebuilding and one from a non-rebuilding iteration, an estimate of the average time per iteration can be calculated. The resulting value is then smoothed, as will be explained in the following paragraph, added to the evidence collection, and all tuning strategies will be notified about it, which concludes the `iteratePairwisePipeline()`.

Smoothing Before the averaged value is passed to the evidence collection, it is further smoothed to mitigate the impact of measurement noise. Smoothing can be done since the systems evolve slowly, as explained in Subsection 3.2.1. Hence, any erratic change in the measurements has to be due to external influences, e.g., a third-party program running on the same compute node, and should be mitigated. The smoothing is implemented by applying the locally estimated scatterplot smoothing (LOESS) [SG64] to the sample at hand with smoothed measurements of the same configuration from up to five previous tuning phases. Using this method, trends are preserved because only weighted local information is used, while outliers are mitigated.

3.2.3 Tuning Strategies

The `AutoTuner` itself does not implement any optimization algorithms. The decision which configurations should be evaluated is shifted to a number of so called tuning strategies that can be applied individually or in combination by the tuner. They are selected by the user upon the start of the simulation. Tuning strategies are implemented as independent modules against a plugin style interface so that AutoPas can easily be extended with new ones.

Figure 3.15 shows how tuning strategies are applied to the configuration queue described in Subsection 3.2.2 and how they interact with the evidence collection.

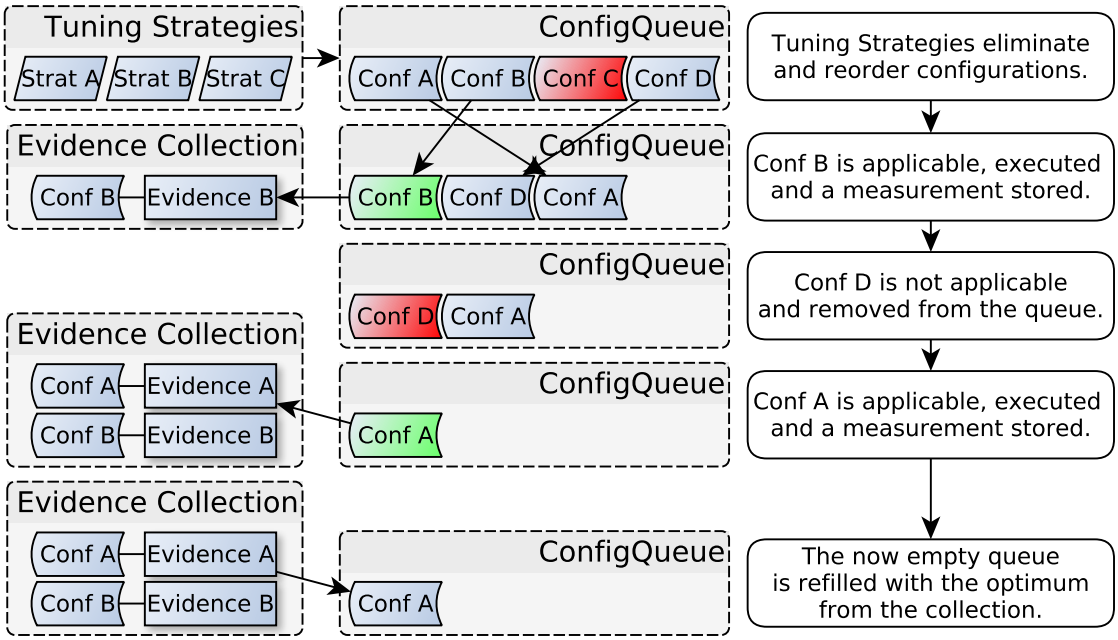


Figure 3.15: Example of the interplay of tuning strategies, configuration queue, and evidence collection. The progression from top to bottom represents advancement in time.

Initially, the queue of configurations contains the complete search space. Then, a number of chosen tuning strategies are applied in a user-given order which can eliminate unpromising strategies. They could also reorder them for example to minimize conversion overhead or to rank and only test the N most promising ones. Next, as already sketched out in Subsection 3.2.2 applicable configurations from the queue are applied, their evidence stored and in the end the best one is reinserted to be used until the next tuning phase. There might be the edge case where a tuning strategy completely wipes the queue empty, for example due to a misconfiguration that deems every configuration unsuitable. In that case, this tuning strategy is ignored for this tuning step and the queue is restored to the state before it was applied.

One can also choose not to apply any tuning strategy. This then leads to the evaluation of the complete search space which is potentially very time consuming because it will also sample every inefficient configuration On the flip side this is guaranteed to find the optimal configuration. A use case for this is to evaluate the quality of the decisions a tuning strategy makes.

In the following a description of tuning strategies implemented in AutoPas is given. For the full list of currently available strategies, we refer to the official documentation[6].

Random Search This strategy randomly selects a configuration to evaluate and terminates the tuning after a given number of configurations has been assessed.

[6] `https://autopas.github.io/doxygen_documentation/git-master/classautopas_1_1options_1_1TuningStrategyOption.html` Accessed: 20.12.2024

Slow Config Filter Usually, for a given scenario, some configurations perform worse by factors > 5 than the optimum. Should any of those be encountered, this strategy removes them from the search space for the rest of the simulation.

Predictive Tuning A straight forward way to estimate the performance of a configuration is to look at its performance in the last tuning phases and apply simple extrapolation methods. Here, methods like linear regression or Newton interpolation can be used to estimate the performance in the current phase. A simpler variant, that only takes into account the performance of the last tuning phase is also available. Configurations are sorted by their predicted performance and only those which are within a given percentage of the optimal predicted performance remain in the queue. Optionally, configurations that have not been evaluated for a long time, measured in a number of tuning phases given by the user, are also put in the queue for reevaluation, to avoid ignoring them due to bad extrapolation. In order to avoid testing extremely slow configurations, this feature should be used together with the slow config filter.

While this strategy provides reasonably good estimates during the simulation, it can not provide any estimates at the start or during the first tuning phases because it needs datapoints to extrapolate from. Thus during the first tuning phases this strategy does nothing.

Rule-Based Tuning Even though AutoPas liberates the user from having to rely on external knowledge, it still offers an interface to provide some. Rule-based tuning works with user-provided rules, passed via a `.rule` file, and formulated in a custom DSL. Parsing the DSL is done via the Antlr4 framework and then evaluated using a virtual machine.

The rules work on a pattern matching basis and take into account information about the current state of the simulation. Some examples are shown in Listing 3.4.

The first rule states that if the number of particles in the AutoPas instance is higher than 1 000, any configuration with the container option being Linked Cells is superior to any configuration using Direct Sum. Thus, if this condition holds during a simulation, if Linked Cells is in the allowed containers, any configuration using Direct Sum will be removed from the queue of configurations.

The second rule is more complex and makes use of live information about the number of particles and empty cells, as well as multiple definitions of variables and lists offered by the DSL. In short, this rule states that in sparse domains, Verlet Cluster Lists are superior to most other containers.

```
1 # Example Rule 1
2 if numParticles > 1000:
3     [container="LinkedCells"] >= [container="DirectSum"];
4 endif
5
6 # Example Rule 2
```

```
7  define maxEmptyCellsPerParticles = 100.0;
8  define isDomainExtremelyEmpty =
       numEmptyCells / numParticles > maxEmptyCellsPerParticles;
9  if isDomainExtremelyEmpty:
10     define_list AllExceptVCL =
           "LinkedCells","VerletLists","VerletListsCells","Octree";
11     [container="VerletClusterLists"] >= [container=AllExceptVCL];
12 endif
```

Listing 3.4: Two examples of rules for the rule-based tuning. Rules take into account dynamic scenario information and apply in a pattern matching fashion. Symbols like `numParticles` and `numEmptyCells` are provided by the tuning strategy. The full rule language grammar is available in the official AutoPas documentation[7]

In practice it is possible to write contradicting rules, for example by supplying any preference relation and their inverse. This would then eliminate all configurations from the queue. Currently, it is up tho the user to only provide non-contradicting rules.

Bayesian Search Approaches that leverage Bayesian statistics are geared to find the maximum of an unknown black-box function [MM89]. The idea is to not directly create an approximation of the function, but to model the uncertainty we have about it via a prior based on a Gaussian process. This prior is then updated step by step, using Bayesian rules, decreasing the uncertainty about the black-box function. For this, the black-box function is evaluated at specifically chosen points, to gain more and more insight about the structure. The trick is, that taking into account the model of the uncertainty, we can choose the evaluation points so that we maximize the information gain about the optimum per evaluation, using so called acquisition functions. This approach can be applied to any function as long as it can be evaluated at any point but works better for smooth functions [GKD19, DOW+22, Gar23].

Following this approach, we interpret our algorithm selection problem as finding the optimum of a function as shown in Equation 2.42. Thus, this tuning strategy uses the technique described above to decide for which configuration to gather evidence next. Since convergence is not necessarily fast for our type of black-box functions because they might not be smooth, the tuning strategy can be restricted to an upper limit of evidence to try.

Bayesian Cluster Search This strategy is an extension to the first Bayesian Search strategy. It accounts for the fact, that while most tunable parameters are discrete choices, some might be continuous, like for example CSF. Here, the idea is to separate discrete values into so called clusters. Within each cluster, the continuous values are then optimized and then the global optimum selected so that an both the continuous and discrete choices are optimized. To avoid having to tune every cluster, we use bayesian statistics to recognize similarities between clusters to avoid explicitly tuning two clusters that behave very similar [TTW19].

MPI Parallelized Strategy In contrast to the other strategies described, this one does not directly trim down the search space or learn anything about the configurations' performance. Instead, it takes advantage from the fact that AutoPas is often used in MPI parallelized software, where there is one instance of AutoPas per rank, as described above in Subsection 3.1.1.4. The strategy distributes the available configurations over available ranks, and lets each evaluate the performance of different ones. Then, evidence is shared between all involved ranks so that the best configuration can be chosen. This has the advantage, that the number of iterations it takes to evaluate all configurations of the search space is reduced, since it is processed in parallel. In practice, it is not so straight forward to compare the performance two algorithms running on different MPI ranks, simulating distinct parts of the domain, that could be very different. Thus, the strategy identifies groups of MPI ranks with similar domain properties, like homogeneity or density, and only lets ranks within a group cooperate.

Currently, in a heterogeneous compute cluster, this could lead to nodes which have different hardware configurations exchanging tuning information which might not be transferable. It is conceivable to extend this grouping mechanism to also consider the underlying hardware to avoid this problem.

3.2.4 Tuning for Energy Efficiency

Up to this point, the sole goal of the tuning process is to reduce the total time to solution. However, over the last few years, power consumption and energy efficiency concerns have moved increasingly toward the center of research interests. In this context, AutoPas also supports tuning for minimal energy consumption.

Looking at Figure 3.16, while there is a clear correlation between energy usage and the time for the particle interaction, it is not so clear cut to tune for minimal time to solution. So, to tune for minimal energy usage, instead of measuring time per iteration, it can measure the energy consumption per iteration using hardware interfaces like Running Average Power Limit (RAPL) Machine State Registers (MSRs). Depending on the processor, the values obtained from the RAPL interface must be scaled to get the actual consumption in Joule. However, AutoPas only needs to find the minimal value hence, no scaling is needed.

Since this change in metric technically only means that the number that needs to be minimized comes from a different source, the whole tuning logic described above can be applied in the same way. Only tuning strategies based on external knowledge, like rule-based tuning, potentially have to adapt the encoded knowledge. Most strategies, however, work without initial assumptions and can thus be used out of the box.

Energy tuning in AutoPas currently is only in its infancy. One future feature could be dynamically adjusting the number of OpenMP threads used. Often, the hardware provides significantly more threads than what can efficiently be mapped on the scenario. So, reducing the number of threads could reduce energy usage while only having a marginal impact on the time to solution.

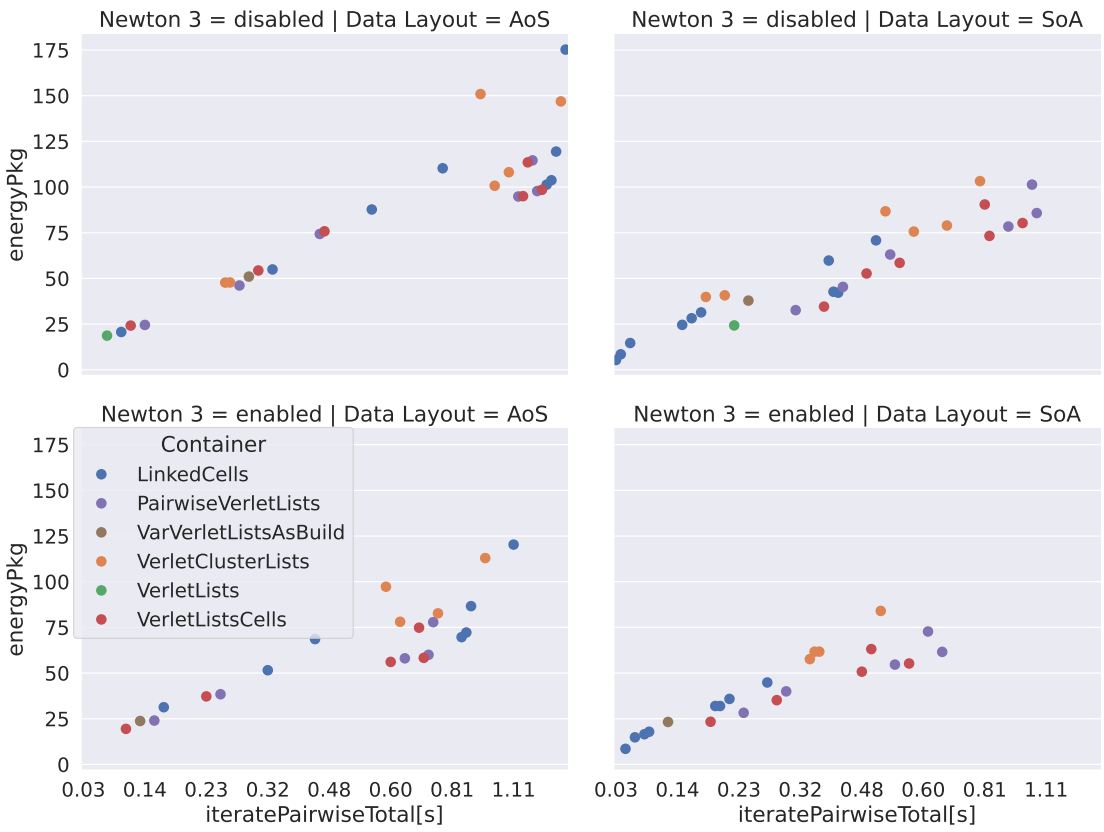


Figure 3.16: Comparison of energy usage vs iteration time for most of AutoPas' configurations. Columns are AoS (left) vs. SoA (right), the rows Newton3 disabled (up) vs. enabled (down). Colors depict the container choice, and multiple dots of the same color are different traversals.
The two quantities energy usage and time to solution seem to correlate but are not completely aligned, because the don't align on a straight line but form a cone. For details about the setup see Section A.1.2.

3.3 Related Work

AutoPas is of course not the first software to implement MD or automated algorithm selection, however, it is, to our knowledge, the first to combine the two. Therefore, no direct comparison to other projects is possible but it is still worthwhile to discuss some that have some overlap in one or the other fields and explore similarities and differences.

3.3.1 Spiritual Predecessor: ls1 mardyn

Particle simulations have a long history at the chair for Scientific Computing in Computer Science (SCCS) at TUM, where AutoPas was conceived and is developed. Here, the simulation code ls1 mardyn has been continuously developed since 2005 [Buc10, Eck14, NBB+14, HEHB15, Sec21] together with partners at the High Performance Computing Center Stuttgart (HLRS), the Rheinland-Pfälzische Technische Universität Kaiserslautern-Landau (RPTU), the Technical University of Berlin, and the Helmut Schmidt University of the Federal Armed Forces Hamburg (HSU). It is written in `C++` and published open-source under a BSD 2-clause licence[8]. The code is optimized for large systems of small rigid (multi-site) molecules using the memory efficient Linked Cells algorithm and very light particle models. This enabled it to achieve several noteworthy simulations featuring massive numbers of particles up to $2 \cdot 10^{13}$, which held the world record for many years [EHB+13, TSH+18]. Further features of the code include efficient SIMD implementations of the Lennard-Jones force, hybrid MPI + OpenMP parallelizations, long-range interactions using FMM, as well as a plug-in interface to dynamically easily extend its simulation capabilities.

ls1 mardyn first implemented the 3D OpenMP domain traversal algorithms like `c18`, `c08`, `c04`, or `sli`(ced) [TSH+18] which were part of the starting point for the Linked Cells implementation of AutoPas. Similarly, the concepts of particle containers, AoS default layout, and (region-)iterators were picked up from ls1 mardyn. Therefore, even though AutoPas was not explicitly designed for ls1 mardyn, the two codes interface relatively easily. From an early point in the development of AutoPas on, it was integrated into ls1 mardyn, replacing its internal particle container.

3.3.2 Popular Molecular Dynamics packages: LAMMPS and GROMACS

Two examples of very well established and widely used codes for MD simulations that are developed actively with extensive funding and many developers are LAMMPS and GROMACS. Despite their superficial similarities, they have slightly different focuses, feature sets, and algorithms. In contrast to AutoPas, neither of them features any dynamic or automated algorithm selection but highly optimized implementations of their respective algorithms for their intended use cases.

[8] `https://github.com/ls1mardyn/ls1-mardyn` Accessed: 20.12.2024

LAMMPS The Large-Scale Atomic/Molecular Massively Parallel Simulator or short LAMMPS[9], is a MD simulation code that was the result of a research and development cooperation between the National Labs of Sandia and Lawrence Livermore and the three companies Cray, Bristol Myers Squibb, and Dupont. Its first version was developed in 1995 using Fortran77 [Pli95]. Over the next decade, Sandia took over the main development and code was eventually ported to `C++`. The code is released under the GPL-2.0 license and publicly available on GitHub[10].

As of today, LAMMPS contains a rich set of potentials, particle models, ensembles, integrators, pre- and post-processing, as well as further specialized features in a plugin-like package structure where a user can pick and choose their components statically. For an up to date reference we refer to their overview paper [TAB+].

Even though LAMMPS can be applied to general MD problems its main focus is on material science simulations as is evident from the high number of publications from this field[11].

Internally, the code is based on the Verlet Lists algorithm and primarily makes use of MPI for parallelization with multiple load balancing strategies, but also implements a few OpenMP shared memory parallelization strategies. It also contains optional packages for GPU support via CUDA, OpenCL, or HIP. General accelerator support is also provided via a package based on Kokkos.

With regard to this thesis, it serves as a comparative implementation of Verlet Lists. To demonstrate the universally applicability of the interfaces of AutoPas, our library was integrated into LAMMPS which will be discussed in Section 4.3.

GROMACS The GROningen MAchine for Chemical Simulations or short GROMACS[12] is a MD simulation code that was originally developed in 1991 at the University of Groningen. Its core was the C reimplementation of an significantly older Fortran code called GROMOS [vGBE+96]. Since the early 2000, development moved to the Royal Institute of Technology (KTH) and Uppsala University Sweden and the code was ported to `C++`. The official version of the code is published on GitLab[13] and released under the GNU LGPLv2.1 licence.

GROMACS primarily focuses on HPC biomolecular simulations with prominent examples being the Folding@Home[14] [VPB23] project, research into COVID 19 [GSK+21, GCBC21, MM22], or in the context of SPPEXA [BRU+20].

As the core driver for the pairwise short-range interactions, GROMACS developed the Verlet Cluster Lists algorithm [PH13] which is also implemented in AutoPas. Even though GROMACS does not feature any dynamic algorithm selection as AutoPas, it features several hand crafted and optimized kernels for this algorithm written in assembly [AMS+15] which are chosen statically before the simulation.

[9] `https://www.lammps.org/` Accessed: 20.12.2024
[10] `https://github.com/lammps/lammps` Accessed: 20.12.2024
[11] `https://www.lammps.org/papers.html` Accessed: 20.12.2024
[12] `https://www.gromacs.org/` Accessed: 20.12.2024
[13] `https://gitlab.com/gromacs/gromacs` Accessed: 20.12.2024
[14] `https://foldingathome.org/` Accessed: 20.12.2024

Furthermore, the code implements a hybrid MPI OpenMP parallelization and can also make use of GPU through CUDA or OpenCL [PZB+20]. Here, it employs auto-tuning for CPU-GPU load balancing[15].

3.3.3 Performance-Portable Algorithms: CoPA Cabana Library

Cabana is a performance portability toolkit library for MD and part of the Co-Design Center for Particle Applications (CoPA) within the Exascale Computing Project (ECP) of the U.S. Department of Energy. Development started in 2018 as a joint cooperation between the US National Labs of Lawrence Livermore, Oak Ridge, Los Alamos, and Sandia.

The library is similar to AutoPas as in it provides data structures like Array of Structures of Arrays (AoSoA) and algorithms like Linked Cells and Verlet Lists for various types of particle based simulations. It is not only constrained to short-range interactions but also implements long-range interactions and particle in cell methods. However, it does not feature any auto-tuning and any algorithm selection has to be done by the user at compile time [SRJ+22]. From the very beginning it was implemented in `C++` with Kokkos as its parallelization backend to enable portability to all architectures relevant in top of the TOP500, including GPU architectures. Furthermore, it is not only concerned with operating on a single node but uses GPU aware MPI for distributed memory parallelism.

The toolkit is publicly available on GitHub[16] under a BSD 3-Clause License.

3.3.4 Particle Toolkit with Parameter Tuning: HOOMD-blue

Another package for running MD simulations is HOOMD-blue[17] with a focus on material science particularly soft matter particle simulations [AGG20]. It's developed by the University of Michigan since 2008 [ALT08].

In contrast to AutoPas HOOMD-blue is primarily developed for GPUs in `C++` and CUDA but can also be executed on CPUs. It uses a Verlet Lists like algorithm that constructs its lists via a bounding volume tree structure [HAN+16, HSM+19]

Even though it does not feature any automated algorithm selection, it has a small tuning component to tune its neighbor list buffer size[18].

The code is publicly available on GitHub under a BSD-3-Clause license[19].

[15] `https://manual.gromacs.org/documentation/5.1/ReleaseNotes/performance.html#improved-performance-auto-tuning-with-gpus` Accessed: 20.12.2024
[16] `https://github.com/ECP-copa/Cabana` Accessed: 20.12.2024
[17] `https://glotzerlab.engin.umich.edu/hoomd-blue/` Accessed: 20.12.2024
[18] `https://hoomd-blue.readthedocs.io/en/v4.1.0/module-md-tune.html#md-tune` Accessed: 20.12.2024
[19] `https://github.com/glotzerlab/hoomd-blue` Accessed: 20.12.2024

3.3.5 Algorithm Selection for Sparse Matrices: Morpheus-Oracle

Matrix operations are a classical field for highly optimized libraries. Morpheus-Oracle is a `C++` library for sparse matrices developed by the University of Edinburgh since 2021 [SW22]. Even though it is designed for a completely different field of application, it employs categorical dynamic auto-tuning for the layout of sparse matrices similar to AutoPas.

Its idea is to learn the performance of available matrix formats in an offline phase by doing profiling runs and feature extraction on the target hardware. Then in the online phase, the actual application, this knowledge is applied by extracting the same features from the target matrix, comparing it to the previously built database and from that predict which format is the most efficient one. The library is then able to convert the given matrix and apply the requested mathematical operation.

With this combination of machine learning and hardware agnostic design Morpheus-Oracle can support a wide range of architectures. In their own benchmarks, the authors report Average speedups from this technique about 1.1 on CPUs and 1.5 on GPUs [SW23].

The library is publicly available on GitHub[20] under a Apach2-2.0 license.

3.4 Interim Summary

This Chapter presented the entire library of AutoPas, from the initial idea, through the implementation and optimizations, to its relation to other codes. It was divided into two main parts, which are the two main pillars of AutoPas. The first is the high-performance library with a user-friendly interface, and the second is the automated dynamic algorithm selection.

The presentation began from the outside, the user's perspective, where Section 3.1 discussed the first pillar. Core design goals were defined, such as usability, customizability, performance, modularity, and maintainability. With these in mind, Subsection 3.1.1.1 analyzed the context for which AutoPas was designed using data from the TOP500 list. Based on this data, in 2018, the decision was made to implement the library for CPUs and to add GPU support later. Furthermore, the high-level structure and terminology of AutoPas were defined.

In Subsection 3.1.1.2, the two main classes a user must implement to use the library were shown with examples. These were a particle class and an interaction functor class.

With this, the user-facing part of the library was complete, and Subsection 3.1.1.3 gave an overview of the internal view and structure. All the implemented algorithmic options and switches were listed and summarized, and their connection to the theoretical concepts explained in Chapter 2 was drawn.

Since AutoPas was designed for shared memory parallelization but intended to be used in software that also makes use of distributed memory parallelization, Subsection 3.1.1.4

[20] `https://github.com/morpheus-org/morpheus-oracle` Accessed: 20.12.2024

sketched the idea of running one instance of AutoPas per MPI rank and how they interacted.

Eventually reaching the deep developer's perspective, Subsection 3.1.2 discussed software engineering challenges and techniques that were applied to overcome them. Here, code design choices like the Verlet Lists style of the main interface were discussed. Different abstraction techniques via inheritance, interface design, and template metaprogramming were shown in selected examples. The last part of the developer's perspective, Subsection 3.1.2.6, highlighted the importance of optimizing especially Verlet Lists-like containers for their memory usage and allocation patterns and showed the performance impact of multiple optimizations.

The subsequent Subsection 3.1.3 took to both the AutoPas user's perspective and the optimization-aware developer. Since the user provides the functor class, which is also the innermost computation kernel, it has to be highly optimized. This section showed and explained the importance and difficulties of hardware-aware SIMD vectorization and its performance impact.

Section 3.2 discussed the implementation of the automated algorithm selection, the second pillar of AutoPas. After Subsection 3.2.1 discussed how the problem fit together with the theory and definitions from Subsection 2.2.1, Subsection 3.2.2 explained the internal logic of the tuning implementation, which was called the tuning loop. The actual selection of strategies to evaluate happens in one or more tuning strategies. These were implemented in a modular way in the gist of the overall design goals of customizability and modularity, enabling the simple development of more strategies in the future.

In the last part about tuning, Subsection 3.2.4 showed that AutoPas can also use energy efficiency as the performance metric that the tuning shall minimize and that this is not necessarily the same as tuning for time to solution.

The last part of the Chapter, Section 3.3, discussed which aspects of AutoPas already existed in other scientific software packages and how they related. The conclusion was that there was currently no other software that brought together simulation and automated algorithm selection, underscoring the value and uniqueness of AutoPas.

4 Examples and Applications

AutoPas is simply a library and cannot function independently. It is designed to be integrated into a simulator, acting as the core data container and driver for particle interactions. This chapter presents four examples of how AutoPas has been integrated into different codes and demonstrates its performance across various scenarios. These simulators include established MD codes as well as new programs developed within the context of this thesis. This serves to verify and validate the library and showcase its versatility.

4.1 md-flexible

The small simulator md-flexible is the primary demonstrator application of AutoPas. It was conceived and is developed together with AutoPas in the same repository[1], by the same authors as part of this thesis, and comes with the library as an example on how to use it. Its primary purpose is not to provide a fully featured particle simulation but to quickly and easily set up various scenario configurations and provide direct configuration access to all features of AutoPas. For this reason, md-flexible should require as little maintenance as possible, even prioritizing maintainability over feature richness and complexity. This greatly helps in the development, testing, and understanding of new features of the library.

4.1.1 Features

Although md-flexible began as a simple implementation consisting of two `for` loops – one for generating particles and another for repeating pairwise interactions, along with some time and memory measurement logic – it was initially not meant to be a fully developed simulator. Over time, however, it has evolved into a comprehensive MD simulator with numerous features:

Forces Since md-flexible is a classic MD simulator, it uses the Lennard-Jones potential. For this, it allows the user to choose from the implementations from the AutoPas application library, which also includes a version written with AVX2 and one with SVE intrinsics. Optionally, a global force in one direction can be added to simulate, e.g., gravity.

Input via CLI and Files In order to configure the simulation and behavior of AutoPas, md-flexible offers two input interfaces: command line parameters and YAML[2]

[1] `https://github.com/AutoPas/AutoPas/blob/master/examples/md-flexible/` Accessed: 20.12.2024
[2] `https://yaml.org/` Accessed: 20.12.2024

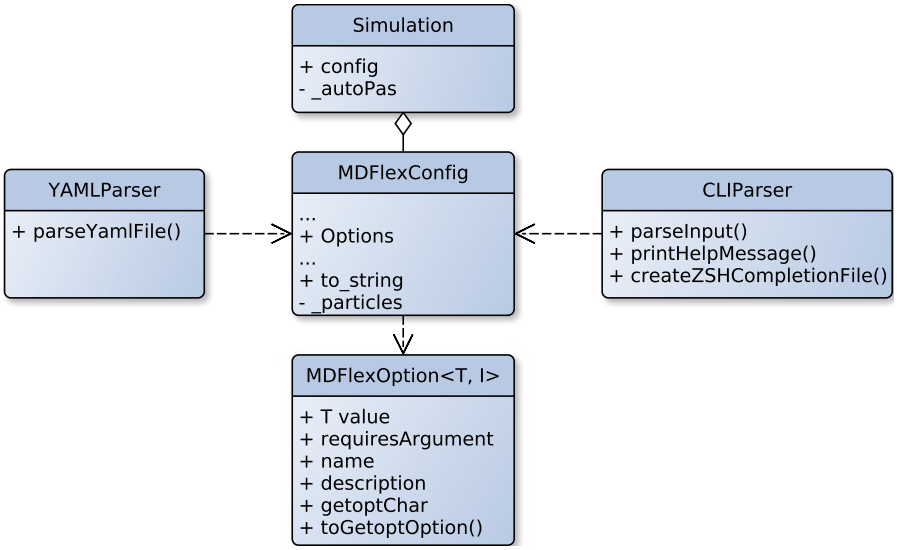


Figure 4.1: The whole simulator is steered by one `MDFlexConfig` file consisting of many `MDFlexOption`, which is filled by (potentially) both parsers.

files. As sketched in Figure 4.1, both parsers can be used to fill one common `MDFlexConfig` object that defines the behavior of the simulation. This allows the user to either only use command line parameters, only a YAML file, or even a mixture of both where the command line parameters take higher precedence than those from the file. The idea for the mixed mode is to provide a simulation template via YAML and easily alter it interactively on the fly by overriding parameters using the command line interface.

All parameter options in the configuration class are realized using the `MDFlexOption` class, which is a collection of all information needed regarding one option. It encapsulates its type `T` and name, which are needed for parsing, as well as a description containing all possible values. Here, all parameters using the `Option` class of AutoPas, which was presented in Figure 3.6, can make use of the provided fuzzy parsing functionality. Furthermore, `getAllOptions()` can be used to provide this list of possible values and `getMostOptions()` to set defaults. The description is used to automatically generate a help message containing all possible parameters, their descriptions, and possible values.

To process the command line arguments, the `getopt`[3] library is used, which is part of the POSIX standard, and thus widely available on UNIX like systems. This library needs one unique integer identifier for every parameter. In order to avoid potential clashes from copy-pasting, `MDFlexConfig` parametrizes each `MDFlexOption` with the line number `I` where the option is declared and `MDFlexOption` uses this as the identifier.

[3] `https://pubs.opengroup.org/onlinepubs/9799919799/` Accessed: 20.12.2024

Finally, all information in `MDFlexOption` can also be used to generate shell autocompletion files. For `zsh`, such a generator is implemented for parameters, their arguments, and descriptions, contributing immensely to an easy and quick workflow with md-flexible.

Distributed Memory Parallelization md-flexible implements a full-shell domain decomposition approach [Sec21] for the distributed memory parallelization using MPI. Dynamic diffusive load balancing is also supported either via A Loadbalancing Library (ALL)[4] or a simple approach based on the idea of moving subdomain boundaries proportionally to the inverse of the subdomain's workload. Both approaches use time measurements as their input and equilibration metric.

To communicate particles via MPI, they must be converted to a series of bits. This process is also referred to as serialization. For this, md-flexible uses the reflections mechanic explained in Subsection 3.1.2.5 in a similar way to the generation of the SoAs. The significant advantage here is that when the particle model changes, e.g., due to an extension to the simulator, only the list of attributes that shall be communicated needs to be updated to adapt the full (de-)serialization process.

Visualization Output The state of all particles can be periodically printed to Visualization Toolkit (VTK)[5] files to be visualized with tools like ParaView[6]. When the simulation is executed using multiple MPI ranks the parallel I/O mechanism of VTK[7] is used. Each rank writes its own partial phase space file per output step, linked together by one metadata file.

The domain decomposition can be visualized for the MPI parallel simulations. This is done by writing VTK files that draw the bounding boxes of all ranks.

Checkpointing Since the VTK output files contain all information about the state of a simulation, md-flexible implements a loading mechanism to (re)start a simulation from such files. Nevertheless, a YAML input file must still be provided to set all further simulation parameters, e.g., time step width or the size of the simulation domain.

Statistics One of the primary use cases of md-flexible is testing the performance of AutoPas. Thus, the wall time of all significant simulation steps is measured, and a summary, including percentages of nested steps, is reported. When the simulation is executed over several MPI nodes, the sum of all timers of the same name and the total wall-clock time are shown.

[4] `https://slms.pages.jsc.fz-juelich.de/websites/all-website/` Accessed: 20.12.2024
[5] `https://vtk.org/` Accessed: 20.12.2024
[6] `https://www.paraview.org/` Accessed: 20.12.2024
[7] `https://www.paraview.org/ParaView/index.php/Parallel_I/O` Accessed: 20.12.2024

Using an interface in the `Functor` class, md-flexible can also count the number of distance calculations and force computations. This is then used to calculate the simulation speed in FLOPs/second per iteration.

Time Discretization md-flexible implements the Velocity Störmer-Verlet algorithm, presented in Subsection 2.1.1.4, to propagate particles through time. The user can set the time step width Δt. If set to zero, time discretization and all functionalities that depend on particle movement are disabled, which means that almost exclusively, the force computation and, thereby, the auto-tuning of AutoPas are executed.

Scenario Generators To provide a simple mechanism to set up various scenarios, md-flexible offers several configurable generators to instantiate objects out of particles:

Regular Grid A 3D grid of equidistant particles.

Sphere This is a spherical cutout of a sphere from a regular grid starting with a particle at the center of the sphere.

Closest Packing A cube of particles arranged in the hexagonal closest packing pattern.

Uniform Box A box filled with uniformly randomly distributed particles.

Gaussian Box A box filled with particles distributed according to a 3D Gaussian distribution with a given mean and standard deviation.

All generators can be used multiple times in one scenario and be combined freely. Since the random-based ones can lead to particles being generated arbitrarily close to each other, it is advisable to set Δt to (near) zero to avoid unphysical behavior.

Boundary Conditions The user can choose the behavior of particles at the boundary of the simulation box. Note that, for simplicity reasons, opposing boundaries will have the same behavior, so there is only one choice per spatial axis.

None Also called outflow condition, particles that cross the domain boundary are deleted.

Reflective Particles bounce back from the domain boundary. In md-flexible, this is implemented by interacting the offending particle with a clone of itself mirrored behind the boundary at the same distance.

Periodic When a particle crosses over the boundary, it is moved to the opposing side of the domain. This behavior is as if it exited on one side and reappeared on the other.

Thermostat In MD, thermostats are often used to alter the energy state of a scenario or keep it constant under the influence of external forces. md-flexible implements a classical velocity scaling thermostat [GKZ07]. The idea is to calculate the system

temperature from its kinetic energy, which comes from the particle movement.

$$E_{kin} = \sum_{i=1}^{N} \frac{m_i \langle v_i, v_i \rangle}{2} \tag{4.1}$$

$$T = \frac{2E_{kin}}{NDk_B} \tag{4.2}$$

Equation 4.1 shows how the kinetic energy is the sum of the products of masses m and scalar products of velocities v of all N particles. Using Equation 4.2, this energy, in combination with the number of particles N, number of simulated dimensions D, and the Boltzmann constant [NT+19] can be used to calculate the temperature T of the system.

With this, we can calculate a factor β that is used to scale the velocity of all particles such that the system temperature is at the desired level T_{target}:

$$\beta = \sqrt{\frac{T_{target}}{T}} \tag{4.3}$$

To simulate gradual heating or cooling, the maximum change in temperature per timestep can be limited.

The same mechanism can also be used to initialize a system with random velocity. For this, Brownian motion is simulated given a temperature, using the Maxwell-Boltzmann distribution, which, in 3D, is the product of three normal distributions.

Multi Site Particles The default operation mode for md-flexible is to simulate particles with a single site using the Lennard-Jones potential. At compile time, it is also possible to activate capabilities for the simulation of multi-site particles. This primarily involves two classes: First, an extended particle class that features multiple sites, as well as torque, angular velocity, and a quaternion, to represent its rotation. Second, a more complex Lennard-Jones functor that interacts each particles' sites and computes torque.

4.1.2 Broad Study of Configurations

With the help of the easy and flexible scenario generation capabilities of md-flexible, and the goal of AutoPas to be a library not only tailored for specific use cases, we created a wide range of configurations to test the simulation and tuning behavior of AutoPas in md-flexible. The parameters and their range of values can be found in Table 4.1 and the template for the remaining configuration file in Section A.1.2.

Figure 4.2a shows the optimal fraction of scenarios for each container involved in the study. We can see that the range of scenarios is sufficiently wide so that each container has at least several dozens of scenarios where it shines. Traversals, shown in Figure 4.2b, are covered less exhaustively. For Linked Cells, `lc_c18` and none of the sliced-based traversals are ever optimal. Similarly, for Verlet Lists Cells, `vlc_c18` and

t_r	r_s/t_r	r_c	box-length	CSF	Num Particles	Distribution
10	0.025	2.5	5/5/5	0.5	534	uniform, 1, 1, 1
20	0.035	3.5	10/20/60	1	12193	uniform, 0.4, 0.6, 1
40			25/25/25	2	71034	gauss, 0.75, 0.25
			80/80/80		230909	closest packed, 0.3, 0.25
					1092804	
					4923403	

Table 4.1: Each column shows one parameter and the employed values. The resulting range of configuration is then generated from the cross-product of all columns. This cross-product is then filtered to remove scenarios that are too dense to compute in a reasonable time. This results in 435 configurations. For the exact process see the generator script[8].
The arguments for the distribution parameters are:
Uniform: Particle object size `X/Y/Z` as a fraction of the domain size.
Gauss: Mean and standard deviation as a fraction of the domain size.
Closest packed: Particle object size and offset from 0/0/0 both as a fraction of the domain size.

the load-balanced sliced variants are never picked. Thus, we will focus more on analyzing the selected containers, since this is also the more impactful decision.

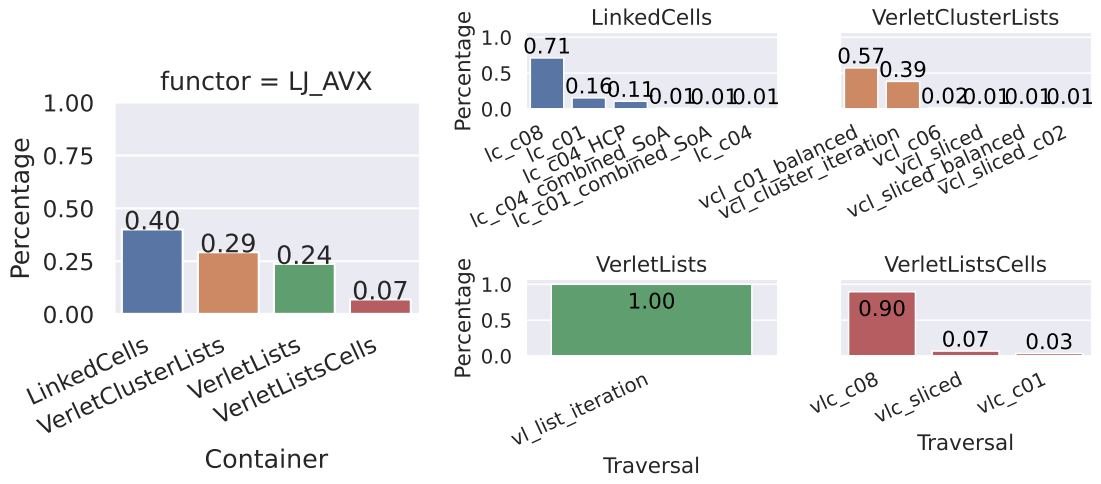
Regarding the dominance of `vlc_c08` within Verlet Lists Cells, it has to be mentioned that in the here presented version of AutoPas[9], this traversal's list rebuild process is significantly better optimized than the others. This mainly revolved around the optimizations discussed in Section A.1.2.

Observations from this plot do not mean that these traversals are useless or that the distribution of containers indicates their general usefulness. The only conclusion we can draw from this is how exhaustive the coverage of the variance of scenarios is but not how balanced it is. We can not say that something is useful or useless in general, but still, we can make statements about the usefulness of configurations in the represented parameter ranges and trends.

Figure 4.3 shows the fraction of how often a container was optimal, grouped by the available domain sizes. The results suggest that an increasing domain size is more disadvantageous for Verlet Lists, and especially large scenarios can benefit from Verlet Cluster Lists.

Looking at Figure 4.4a, we see how average particle number density affects the container choice in the scenarios tested. High densities clearly favor Linked Cells, but the picture is unclear for a density of five and lower. A distorting factor in this plot is that it only looks at average particle densities. However, some scenarios don't fill the entire domain, e.g., those using the distribution `closest packed, 0.3, 0.25` only fill about $1/3^3 = 1/27$ of the domain. This results in a scenario with a small but very dense region and vast parts that are empty. Such conditions can occur, e.g., in the first thousand time steps of a MD simulation or when simulating space debris.

[9] Git commit hash bd22330



(a) Containers **(b)** Traversals

Figure 4.2: The relative distribution of optimal algorithm components over the full range of configurations. While the input parameters of the study lead to scenarios where each container is optimal at least once, this is not the case for all traversals.

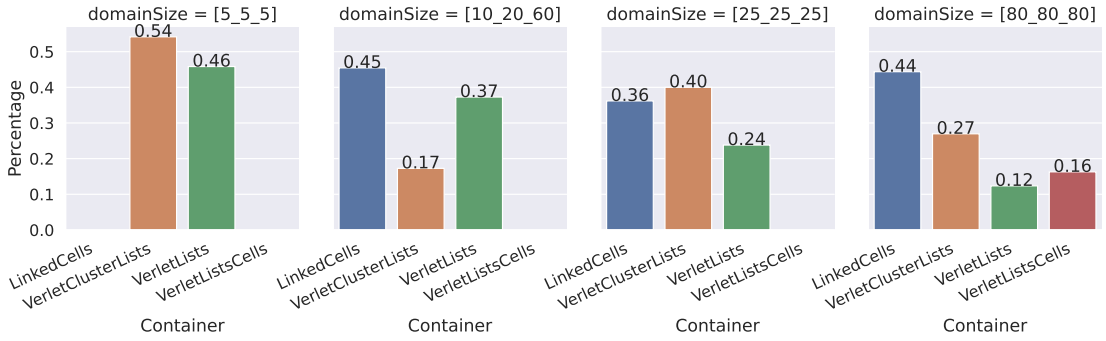
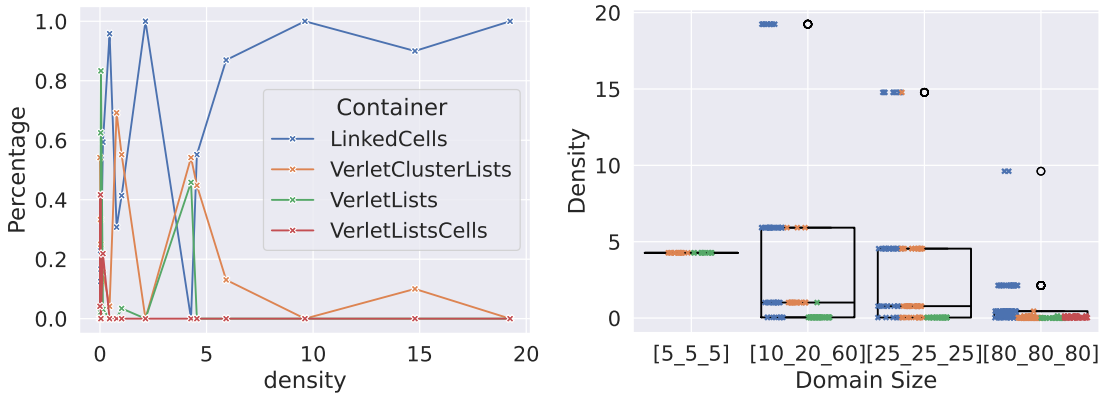


Figure 4.3: Percentages of optimal containers for each category of domain sizes. This figure suggests that Verlet Lists suffer from increasing domain size and Verlet Lists Cells can only be efficient in large domains.



(a) Percentages for each container and how often it is optimal for scenarios of given average densities. High densities favor Linked Cells. Low densities, however, are more challenging to predict.

(b) Average densities of the domain sizes. The smallest domain only features one density since it is too small to be simulated with a wide range of particle numbers. For similar densities, the distribution of container choices changes with different domain sizes.

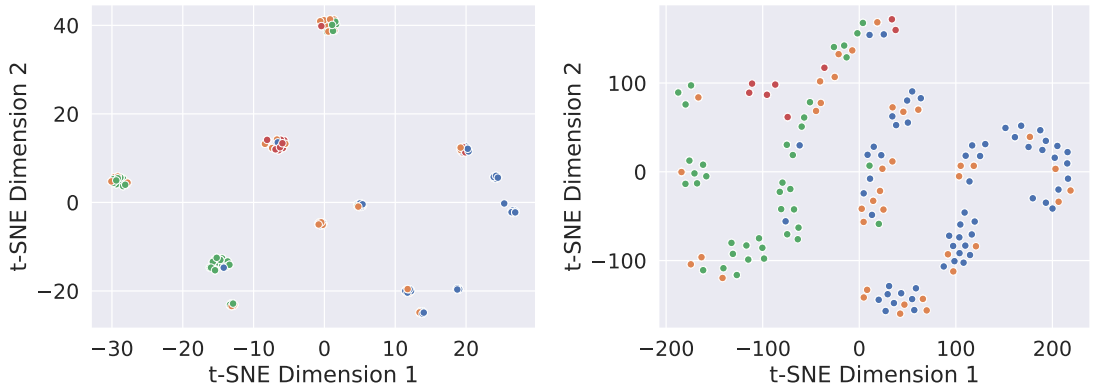
Figure 4.4: Observations of container choices concerning average densities. Here, density is the particle number density as in the number of particles divided by the domain volume.

When we combine the information about density and domain size and look at the container choices, as shown in Figure 4.4b, we see that the two scenario parameters are not entirely independent. This is due to the way the configurations are generated, and scenarios with numbers of interactions that are not feasible for time or memory reasons are eliminated.

So, the data contains trends, but there are no simple correlations. This is captured by the rule-based tuning, described in Subsection 3.2.3.

In order to get a visual grasp of the effectiveness by which these scenario parameters capture the correlation with the container choice, we can look at a t-Distributed Stochastic Neighbor Embedding (t-SNE) plot.

t-SNE [VdMH08] is a dimensionality reduction technique used to visualize high-dimensional data in a low-dimensional space, typically 2D or 3D. As the mapping is particularly effective at preserving local structure and patterns in the data, it can reveal any such in our dataset. The technique works as follows: First, for each point, we calculate the probability that it is the neighbor of another point. This probability is based on the similarity of the pair x_i, x_j, expressed as the conditional probability that x_i would pick x_j as its neighbor under the condition that neighbors are picked proportionally to their probability density using a Gaussian centered around x_i. This process emphasizes local structures by assigning more weight to closer points. Originally, an Euclidean distance is used here, but other distance metrics are possible to counter the effects of the curse of dimensionality [SG17]. Next, the data points are randomly projected into the lower-dimensional space, here 2D, and their similarities are computed,



(a) Scenario Parameters **(b)** Live Infos

Figure 4.5: t-SNE plots showing a projection of the scenarios by the scenario parameters (left) and the gathered live information (right). Each point represents a scenario, and its color is the container that is optimal for it. The color scheme is the same as in Figure 4.4. It can be observed that with these projections, clusters form, which are then populated by mostly one or two containers. This suggests that these parameters can be used to restrict the choice of containers effectively.

but this time using a Student's t-distribution [Stu08]. To achieve the desired arrangement of the projected points, the Kullback-Leibler divergence [KL51] between the two probability distributions is minimized via gradient descent.

Applying this technique to our dataset results in the plots shown in Figure 4.5. Still, these visualizations have to be treated with a bit of care. While t-SNE will group the data into clusters, explaining what these clusters express is unfeasible since the axis in the resulting lower-dimensional space has no direct meaning. Also, their shape, distances, and densities do not necessarily reflect any information originating from the input data [CP23].

Nevertheless, when we plot the scenario generator parameters in a t-SNE and then color each point by the container choice, we can observe whether these input parameters create a structure that correlates with the containers. In Figure 4.5a, we can see that most clusters contain only two containers, suggesting that with this knowledge, the effective search space $\mathbb{A}$ could be restricted severely.

Recreating the same plot using live infos instead of the scenario parameters of the same runs, we obtain Figure 4.5b. This information is gathered by AutoPas at runtime, utilized in its rule-based tuning, and was introduced in Subsection 3.2.2, Even though a few instances of Verlet Cluster Lists appear in most clusters and, therefore, always seem like a potential candidate, the plot hints towards a clear split between scenarios that favor Linked Cells or Verlet Lists, indicating the usability of rule-based in this information to restrict the search space.

4.1.3 Spinodal Decomposition

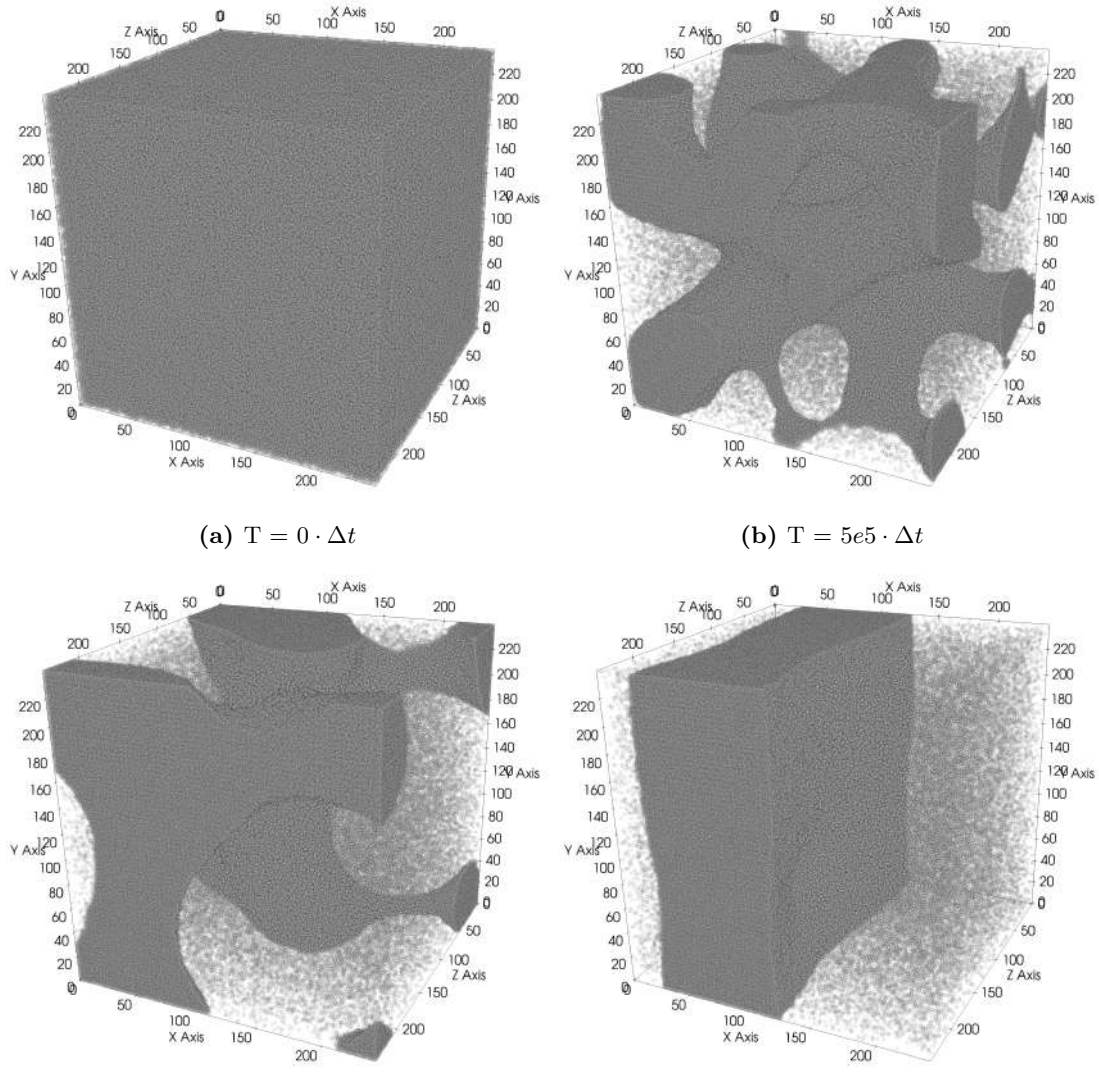
All scenarios from Subsection 4.1.2 were artificial in the sense that their setup does not directly represent a physical experiment. Nevertheless, even though md-flexible is not primarily designed for complex, realistic simulation, it is very capable of conducting them. As an example, the Spinodal Decomposition experiment, which was repeatedly studied with ls1 mardyn [GST+19, SGH+20] is recreated here to show the capabilities of md-flexible, as well as the behavior of AutoPas in complex simulation scenarios with MPI load balancing and checkpointing. The configuration files used are shown in Section A.2.

Equilibration The experiment first requires an equilibration step to generate a homogeneous gas-like particle distribution. 2.6 million time steps were simulated for this. Temperature, and with this pressure, have to be kept at a stable level using a thermostat. Here, a simple velocity scaling thermostat is used, that calculates the current temperature via the kinetic energy of the system from the movement vectors of all particles. The desired system temperature is then set by scaling all these vectors accordingly. Any subsequently shown data does not include the equilibration step.

Decomposition Figure 4.6 shows four steps of the progress of the decomposition simulation over four million time steps. This simulation was conducted on HSUper using five nodes, each running 8 MPI ranks, with nine OpenMP threads each. Due to job runtime restrictions, the simulation was split into two runs with two million iterations each. The total time to solution was $50.35 + 55.2 = 105.55$ hours or 4.4 days. With the Δt corresponding to two femtoseconds, the total simulation covers a period of eight nanoseconds. In the beginning, as seen in Figure 4.6a, the particles are distributed evenly as a homogeneous gas. As the simulation is maintained at a sub-critical temperature, clusters and large structures begin to form rapidly, resulting in local load imbalances, as illustrated in Figure 4.6b. Over time, these structures merge into fewer, dominating structures, such as the large amorphous object depicted in Figure 4.6c. The simulation employs periodic boundary conditions hence, this is only one object split on the XY plane. Consistent with findings in the literature [Cah61, FM08], the scenario progresses toward two stable regions with distinct densities. The denser region forms a wall parallel to the YZ plane, as visualized in Figure 4.6d.

High Level Tuning Analysis In Figure 4.7, the overall distribution of the container choices throughout the whole decomposition part of the simulation over all ranks is shown. It can be seen that Linked Cells based configurations are chosen mostly, 26436 times, followed by a significant share of Verlet Lists Cells with 5285. Next comes Verlet Lists with only 275, and lastly Pairwise Verlet Lists with only four instances of being the optimal container choice on a rank.

Closer Analysis of selected Ranks Taking a closer look at the individual ranks, the dominance of Linked Cells is quickly called into question. Figure 4.8 shows four steps in the simulation, together with the MPI rank subdomains and their currently optimal



(a) T = $0 \cdot \Delta t$ **(b)** T = $5e5 \cdot \Delta t$

(c) T = $1e6 \cdot \Delta t$ **(d)** T = $4e6 \cdot \Delta t$

Figure 4.6: Visualization of the phases of the spinodal decomposition experiment. The experiment starts with a homogeneous gas and a temperature below the critical point. This leads to the formation of clusters that merge into large structures until finally, only a block remains, which due to the periodic boundaries, actually represents an infinite film of liquid.

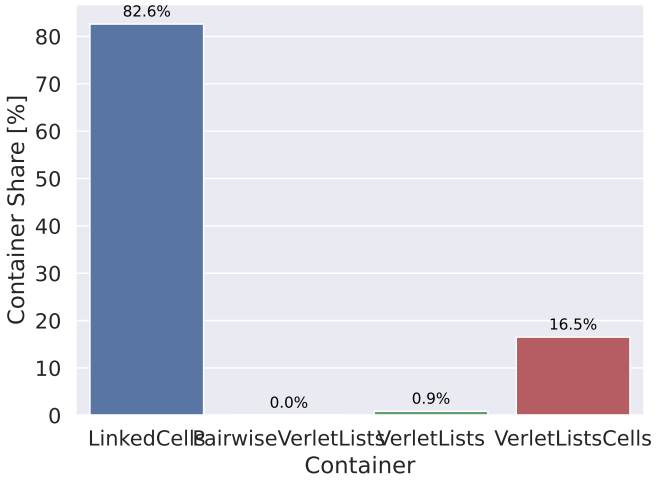


Figure 4.7: Distribution of container choices over the course of the Spinodal Decomposition experiment accumulated from all ranks. Linked Cells is the most common container choice.

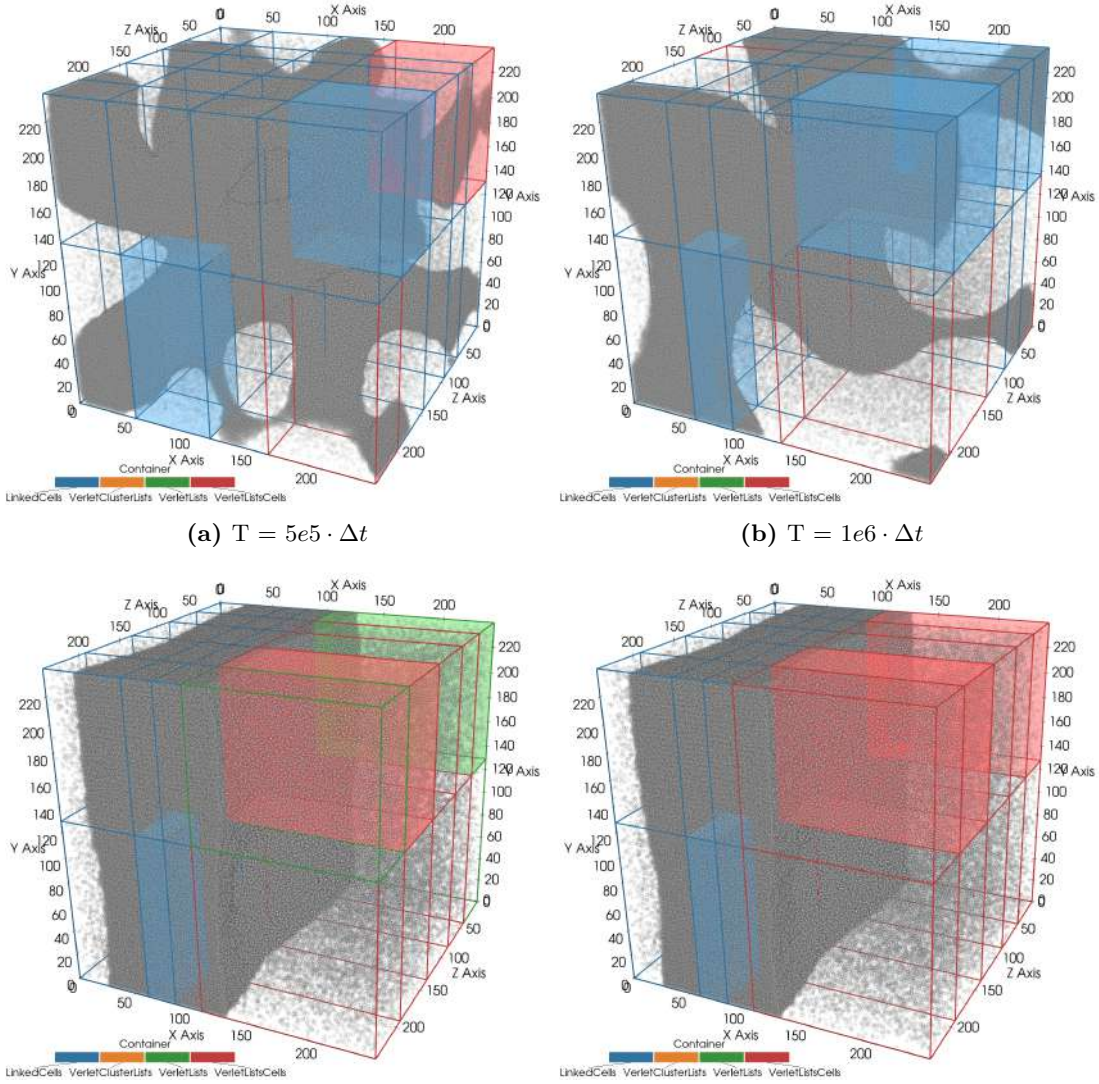
container. It can be seen that denser regions are mostly computed using Linked Cells and more sparser regions using Verlet Lists Cells.

In the following, closer attention is directed towards the three color filled ranks 14, 35, and 38. Looking at their distribution of the container in Figure 4.9 paints a very different picture than the overall average from Figure 4.7. Rank 14 is even more dominated by Linked Cells than the average. In contrast, on the other two ranks, Verlet Lists Cells is used most often, with Verlet Lists even being selected 41 times on rank 35. Returning to Figure 4.9 that shows their spatial location within the simulation, it can be seen that rank 14 is always in denser regions, while the other two are in the inhomogeneous areas of fluctuating density and in the end in a very sparse part of the simulation. Figure 4.10 aims to quantify these observations. For this, two metrics are introduced, which are part of the live information AutoPas collects:

Max Density The domain is subdivided into a dedicated virtual grid of equally sized blocks, which act as bins for particles. Each bin's particle number density is calculated, and the highest value is the maximum density. The number of bins, and thus their size depends on the number of particles. A higher number of particles leads to a finer grid mesh size. This is an indicator of the peak effort that is needed to process the subdomain.

Homogeneity Given the bins from above, the homogeneity is the standard deviation of their densities. Values closer to zero signify a more homogeneous particle distribution. This is an indicator of how well spread the particles are in the subdomain and, thus, how much of its density is close to the maximum density.

The figure consists of three rows, where these metrics are contrasted with the time per iteration from selected traversals. This time is the smoothed result from the samples of



(a) T = $5e5 \cdot \Delta t$ **(b)** T = $1e6 \cdot \Delta t$

(c) T = $3.86e6 \cdot \Delta t$ **(d)** T = $4e6 \cdot \Delta t$

Figure 4.8: Subfigures 4.8a to 4.8d show four points in time the spinodal decomposition experiment. The grid shows the MPI ranks. Colors indicate their selected containers. The rank boxes filled with color are the ranks 14 (lower, front row secon from left), 35 (top, back right corner), and 38 (top, right, second row). Regions containing significant parts of the prominent structure are calculated using Linked Cells-based configurations. The rest employs mostly Verlet Lists Cells.

the tuning process. The described domain metrics were not part of the employed rule file. Thus, conclusions can be drawn on how they correlate to the rules. To improve readability, the shown traversals were restricted such that they represent at least 90% of the optimal configurations on each of the three ranks.

In the first column, the time per iteration for rank 14 only rises moderately as it becomes denser and more homogenous. This aligns well with the previous visual observations and that its volume decreases. Also, the fact that Linked Cells based traversals, especially sliced-based ones, are chosen here aligns well with our expectations as those are primarily designed for dense and homogeneous situations where a lot of benefit can be drawn from vectorization, and no OpenMP load balancing is needed.

The second column shows a very different behavior, which is reflected in very different configuration choices. Rank 35 is situated in a corner of the domain where a hole appears soon after the simulation starts, which can be seen as a sharp drop in maximum density and homogeneity. Even the MPI load balancer can only partially counteract this, and the time per iteration falls below five milliseconds for all traversals. However, soon, the T superstructure, seen in Figure 4.8b, enters the subdomain through the periodic boundary, introducing a very dense object. This leads to an equally sharp rise in the homogeneity value, meaning the subdomain becomes less equally distributed, a rise in the maximum density, and the runtime. During this dense period, Linked Cells-based traversals with load balancing capabilities are selected, which is consistent with expectations. After about 5e5 iterations, the superstructure contracts more towards a wall, and particles recede from the subdomain, reflected in a decline in runtime and the domain metrics. At iteration 2e6, the simulation is interrupted and restarted from a checkpoint due to cluster job runtime restrictions. However, this checkpointing mechanic does neither store the state of the MPI load balancing nor of the AutoTuner. As the MPI domain decomposition is restarted as a regular grid, rank 35 receives a smaller than before chunk of the domain, which is also very sparse, leading to an abrupt fall in runtime. This is less well reflected in the other metrics because this only affected a minor amount of tuning phases and led to optimal configurations, which are not shown as they would pollute the rest of the plot. Thus, a small gap can be observed in the metric plots after iteration 2e6. Unfortunately, the Slow Config Filter seems to eliminate linked cells-based configurations during this ramp-up phase, and they are not tested again, even though the time per iteration subplot suggests that it might have still been a viable strategy, at least for the iterations immediately after the MPI load balancing recalibrated itself again. Nevertheless, looking at the domain metrics, for the remainder of the simulation, Verlet Lists Cells-based configurations seem like a reasonable choice, given the low maximum density smaller one and medium homogeneity.

In the third column, a partially different picture is drawn. There is no initial drop, but a similar rise in density and homogeneity can be seen, resulting in higher runtime. This comes from the initial structure, shown in Figure 4.8a, forming a cluster in this rank's subdomain. However, this soon recedes into the T superstructure, and the MPI load balancer expands the rank to process the emerging sparsely filled space between the periodic split of the T superstructure. As long as parts of the dense cluster are within the domain, Linked Cells-based configurations are superior due to their high efficiency when

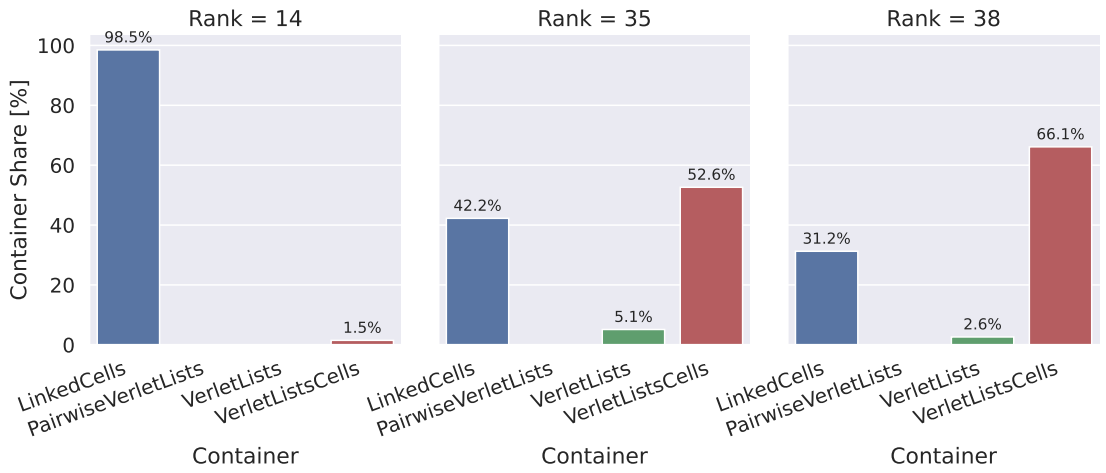


Figure 4.9: Distribution of container choices throughout the Spinodal Decomposition experiment accumulated on ranks 14, 35, and 38. Individual ranks can experience very different distributions than the overall average shown in Figure 4.7. For 35 and 38, Verlet Lists Cells is chosen more often than Linked Cells.

processing dense regions thanks to superior vectorization capabilities. In contrast to the similar Linked Cells phase in rank 35 between iteration 1e6 and 2e6, rank 38 chooses between iterations 0 and 1e6 less but still load balanced traversals, for which no apparent reason can be found in the observed metrics. After the dense clusters are entirely gone from rank 38, the maximum density stabilizes, and the subdomain is significantly more homogeneous. This leads to the switch to Verlet Lists Cells, which again aligns with expectations. Again, a drop in runtime and a gap in domain metrics can be observed directly after restarting from the checkpoint. However, since this time the subdomain was already very sparse before the restart, the subsequent choices by the tuner appear reasonable and support the choices observed on rank 35.

Conclusion This experiment shows that the demonstrator md-flexible is capable of executing complex simulation scenarios with moderate effort, providing an easy way to get performance data to evaluate AutoPas. Together with the study in Subsection 4.1.2, it shows the variety of performance characteristics domains can have. It is shown that there are correlations between the domain metrics defined and collected here but also that they are not trivial to formulate. These observations encourage more knowledge based tuning strategies but also stress the need for a deeper study and understanding of the correlation between domain metrics and algorithm choices.

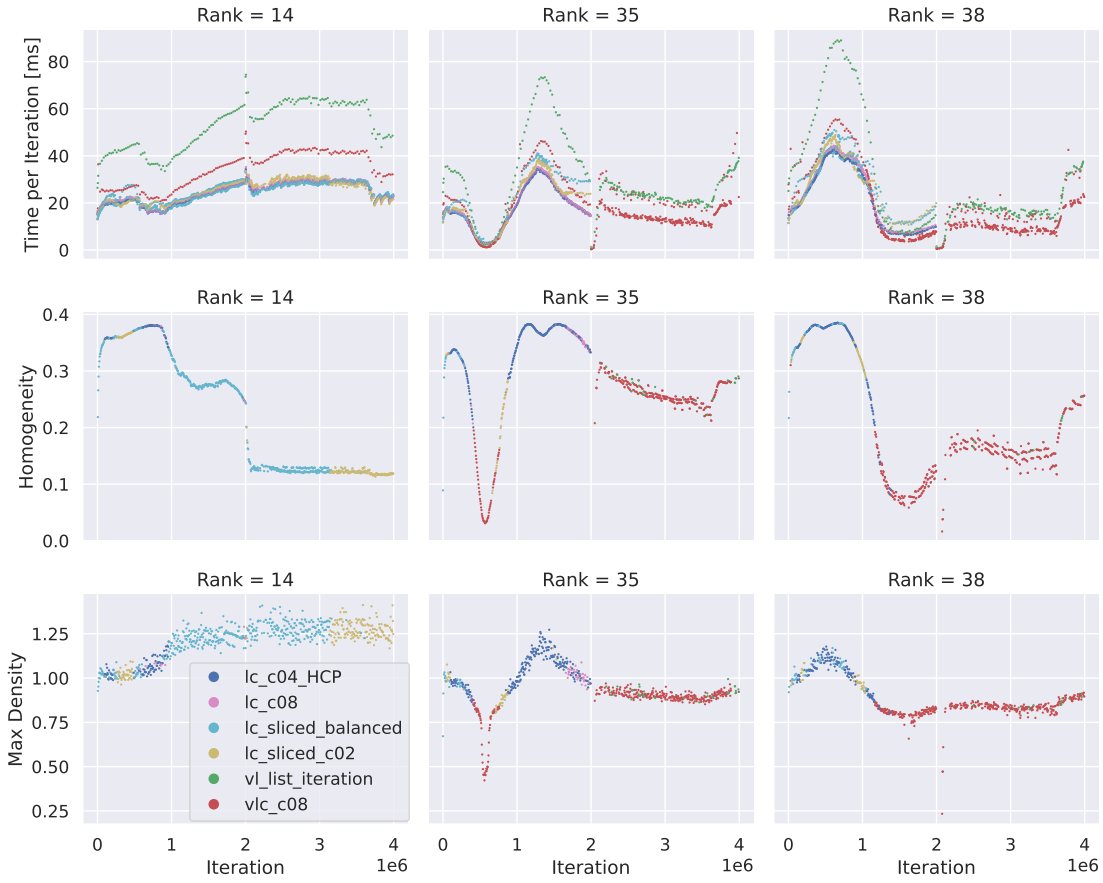


Figure 4.10: Evolution of the properties time per iteration in milliseconds, homogeneity, and maximum density on the ranks 14, 35, and 38 throughout the whole simulation. In the first row, the smoothed time per iteration for all configurations after a tuning phase is shown. In the remaining rows, one domain property is shown colored by the optimal traversal. The cut at two million iterations comes from the restart from the checkpoint, which also resets the MPI load balancing. Observed properties like homogeneity and max density show some imperfect correlation with the container choice. However, the traversal choice within one container remains more complex.

4.2 ls1 mardyn

The MD simulator ls1 mardyn is a established code with a long history of being used successfully for publications. It was already introduced in Subsection 3.3.1 so we refer there for further background.

4.2.1 ls1 mardyn-AutoPas Integration

As mentioned, AutoPas is optionally integrated into ls1 mardyn. This means that during the compilation of the simulator, it is possible to choose AutoPas as the core data container. The library then replaces the existing Linked Cells implementation and the short-range force evaluation. Almost all other features, available via the plugin interface, are still available since the particle class used for AutoPas is an extension of the regular particle class of ls1 mardyn. Features that are not supported are heavily intrusive ones like the reduced memory mode (RMM), which slims down ls1 mardyn to maximize memory efficiency to enable the largest possible simulations [Tch20].

The version without AutoPas will here be referred to as ls1 mardyn vanilla. By default, and everywhere in this thesis, this version uses an implementation of the `lc_c08` traversal.

Our primary objectives in the experiments with ls1 mardyn are:

1. Show that our Linked Cells implementation is at least as fast as the one from the vanilla version, which was optimized over several years.

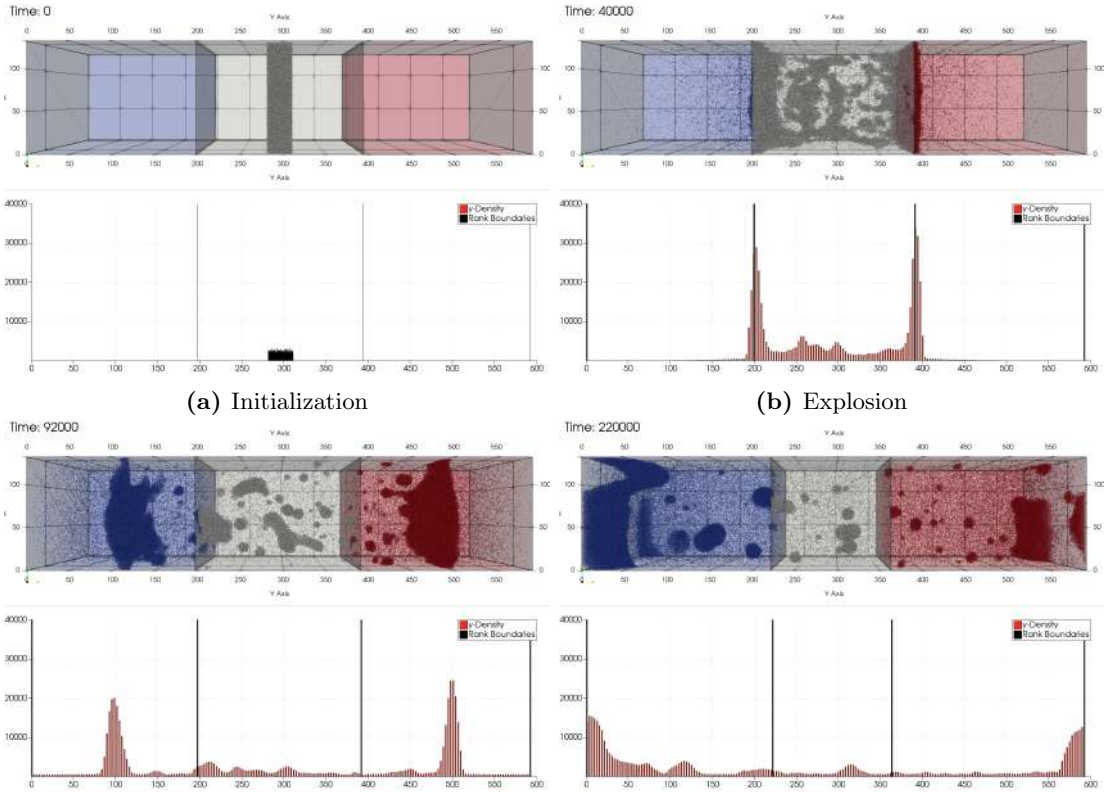
2. Demonstrate that automated dynamic algorithm selection by AutoPas works together with distributed memory balancing by ls1 mardyn.

3. Confirm that using automated dynamic algorithm selection can give an advantage over static simulation methods.

4.2.2 Exploding Liquid

The spinodal decomposition scenario from Subsection 4.1.3 is evolving slowly and towards a stable state. A more challenging scenario is the exploding liquid scenario, which was used for load balancing tests in previous studies [SGH+20, Sec21].

Setup This scenario's domain is a cuboid elongated in the y dimension, with periodic boundary conditions in all directions. In the middle of the y-axis, a dense particle grid is initialized, forming a thin wall in the XZ plane. Considering the periodic boundary conditions, the whole scenario can be understood as a dense wall of liquid stretching infinitely in a vacuum. The initial setup is shown in Figure 4.11a using three MPI ranks distributed along the y-axis and initialized with equally large subdomains. An input file to reproduce the simulation is given in Section A.4.



(a) Initialization **(b)** Explosion

(c) Shock Fronts **(d)** Periodic image collision

Figure 4.11: Subfigures 4.11a to 4.11d show four-time steps of the exploding liquid experiment. Each of the figures' upper halves is a visualization of the scenario where the colors represent the three MPI ranks. The lower halves are a particle number density histogram along the Y axis, with the green lines indicating the rank boundaries. Together, they show the evolution of the scenario, especially the propagation of the shock fronts after the explosion, the formation of droplets behind it, and the overall highly volatile scenario condition. Due to a bug in ls1 mardyn, the dynamic load balancing is not working as intended and the rank boundaries are not following the shock boundaries.

Explosion As soon as the simulation starts, the high-density gradient between the dense liquid and the empty space around it induces a violent rupture, and the liquid explodes symmetrically along the y-axis. A very early stage of this explosion is shown in Figure 4.11b. In each direction, a dense shock front propagates towards the boundaries, preceded by a thick mist of particles that were propelled out of the liquid in the very early steps of the simulation. The initial symmetry of the fronts breaks, and behind them, a sparse mist remains, in which droplets form or break off from the fronts and are elongated by their momentum. This state can be seen in Figure 4.11c.

What is should be visible in this and the previous figure is how the diffusive load balancer ALL automatically follows the shock fronts. Unfortunately the integration of ALL in ls1 mardyn is currently broken and we see an non-optimal domain boundary placement. For the perspective from this thesis, this does not really matter, because this does not affect the tuning, just its outcome. As seen in the respective particle number density histograms, the shock fronts are the most dense regions in the domain, are thus the most compute-intensive. The vast space of droplets between them is entirely left for the third rank, with sometimes even giving the ends of the fronts to it.

Eventually, the two shock fronts reach the periodic y boundaries simultaneously, collide, and bounce off each other. In Figure 4.11d, we can see that the upper part of the right front pushed into the upper part of the left front because it had more momentum due to the break in symmetry. This also leads to the right rank losing a lot of particles to the left one, as can be seen in the histogram. Therefore, the load balancer should give more of the blue region to the gray middle rank and a significant part of the right of the middle from gray to red. Instead the opposite happens and the outer ranks' subdomains are expanded into the middle.

It has to be mentioned that the aforementioned break in symmetry is always a little different with each run of the simulation due to the non-commutativity of the summation of IEEE floating point numbers. Particle force contributions are calculated in parallel. Hence, their order is non-deterministic, leading to different orders of summation. Thanks to the chaotic nature of MD simulations, these tiny changes result in different patterns on the macroscopic scale. This means the overall behavior of forming a mist, shock fronts, and droplets will always occur. Still, their exact positions and shapes will differ.

Tuning Strategy Comparison Since this scenario is evolving very quickly, it is an interesting example to analyze how the tuning strategies presented in Subsection 3.2.3 handle this. Figure 4.12 shows an overview of the performance of different tuning strategies as well as tuning intervals and compares them to the baseline of ls1 mardyn vanilla, which is without AutoPas and thus without any tuning. Performance in this figure is the time spent in the force calculation, which is the step primarily affected by the tuning and free of any communication overheads. In this scenario setup, 5 000 is an upper bound for the tuning interval because this is the rebalancing interval of the MPI load balancer. While this is not necessarily a strict restriction, we consider it necessary here because the subdomains often change substantially. Bayesian and Predictive tuning can both be considered strategies that learn during the simulation from past tuning phases and

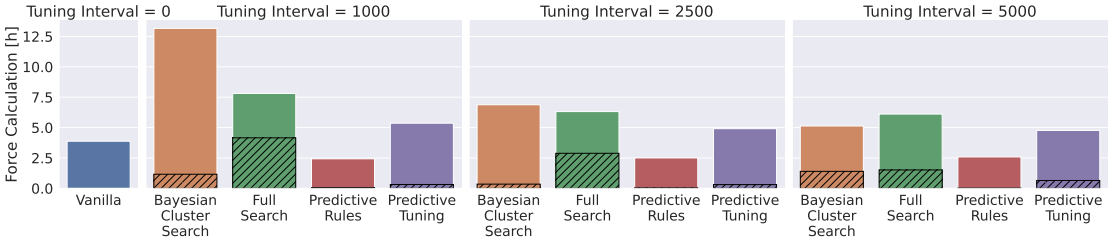


Figure 4.12: The wall time spent in the force calculation for the exploding liquid experiment with ALL using different tuning strategies and intervals. A tuning interval of zero means no tuning because vanilla is ls1 mardyn without AutoPas, which acts as a baseline. Exhaustive full search adds significant overhead and strategies based on data learned on the fly can not sufficiently remedy this. Only the combination of training (predictive tuning) with some expert knowledge (rule-based tuning) yields a significant speedup and beats the baseline.

start without any previous knowledge. It can be seen that in this volatile scenario, this can be better for long tuning intervals than the exhaustive search performed by full-search, however, it is still worse than the baseline, which doesn't tune at all. Even though Bayesian based tuning here spends less time tuning, it but doesn't find a sufficiently good configurations, thus mostly resulting in longer runtimes. When combining the learning strategy predictive tuning with the expert knowledge-based rule-based tuning, a significant improvement is observed. We call the combination of these strategies predictive-rules, Interestingly, in contrast to all other tuning strategies, it is cheaper to tune more often. This suggests that while tuning is very costly for the other strategies due to testing of inefficient configurations, tuning is almost free for predictive rules and benefits from tightly adapted configurations.

To analyze this behavior more thoroughly, Figure 4.13 shows the time per pairwise iteration for the first 50 000 time steps from each rank colored by the active container and for each tuning strategy. The figure shows the data from the runs with a tuning interval of 5 000. Thus, this contains ten tuning cycles, which are visible as vertical clusters of points. Colors between the tuning phases indicate the chosen optimum. Here, the load balancer bug is very visible in the plots of rank zero and one. The fact that the slowest runtimes in the full-search subplots only start to rise shortly before iteration 40 000, which is when the shock fronts hit the subdomain boundaries as can be seen in Figure 4.11b. Especially in the full-search run on rank 1, some very expensive iterations by Verlet Lists Cells can be seen that are up to two orders of magnitude slower than the optimum. It has to be noted, however, that data from Verlet Lists-based containers cluster on two bands. The lower cluster is the actual iteration, and the upper is the iteration plus rebuilding of the lists, so with the here used $t_r = 10$, this is only every tenth iteration. While bayesian-cluster-search barely learns to avoid these configurations, predictive tuning correctly removes the most expensive configurations after it has obtained its minimum of three data points. However, some of them are deemed relevant again, starting from the sixth tuning phase. This initial slowdown by inappropriate configura-

tions and erroneous return to them is wholly eradicated when combining this approach with the restrictions from the rule-based tuning and the Slow Config Filter in the last row. Here, the first tuning phase still tries out a few configurations, but the following only test configurations that are near the optimum. This results in Linked Cells being chosen most of the time with two instances of Var Verlet Lists on rank 0 and one on rank 1.

Conclusion While strategies which are based purely on learning struggle in this scenario, the combination of learned and expert knowledge here achieves a speedup over the vanilla version of almost 1.6. The result is similar and even slightly better than what was already reported for the slower evolving droplet coalescence scenario [SGH+20]. This shows that the algorithm implementations in AutoPas are competitive to the optimized ls1 mardyn code and that algorithm tuning in combination with dynamic load balancing can yield benefits.

4.3 LAMMPS

As ls1 mardyn, the widely popular simulation framework LAMMPS has already been introduced in Section 3.3 paragraph 3.3.2. AutoPas was integrated into LAMMPS in a fork of the main LAMMPS project. This version is publicly available on GitHub[10].

The goals of the experiments below are to demonstrate the following points:

1. LAMMPS-AutoPas is on a comparable level of performance as the vanilla version. LAMMPS has the inherent advantage of being a significantly more mature code with a much larger developer team, which makes it challenging to beat.
2. Although the architecture of LAMMPS is not easily compatible, the effort of integrating AutoPas and achieving reasonable performance is very manageable.
3. Integrating AutoPas and retaining most features of LAMMPS is achievable.

4.3.1 AutoPas Integration

In order to demonstrate the compatibility of AutoPas and to better compare the performance of the implemented algorithms, AutoPas was integrated into LAMMPS. Here, it was slightly more challenging to replace the core with AutoPas because LAMMPS does not have a similar concept of a data container due to its heavy `C` legacy. Instead, particles are stored in SoA style global arrays and accessed directly. To make all LAMMPS styles compatible with AutoPas, variants of them have been created that mostly use iterators to access particles. We acknowledge that this is can often be a source of slowdown, but a more optimal integration would have required a more extensive rewrite, which was not the scope of this work. The primary source of friction like this is when LAMMPS relies on direct random access by array index. Due to the nature of AutoPas not having

[10] `https://github.com/AutoPas/lammps-autopas/` Accessed: 20.12.2024

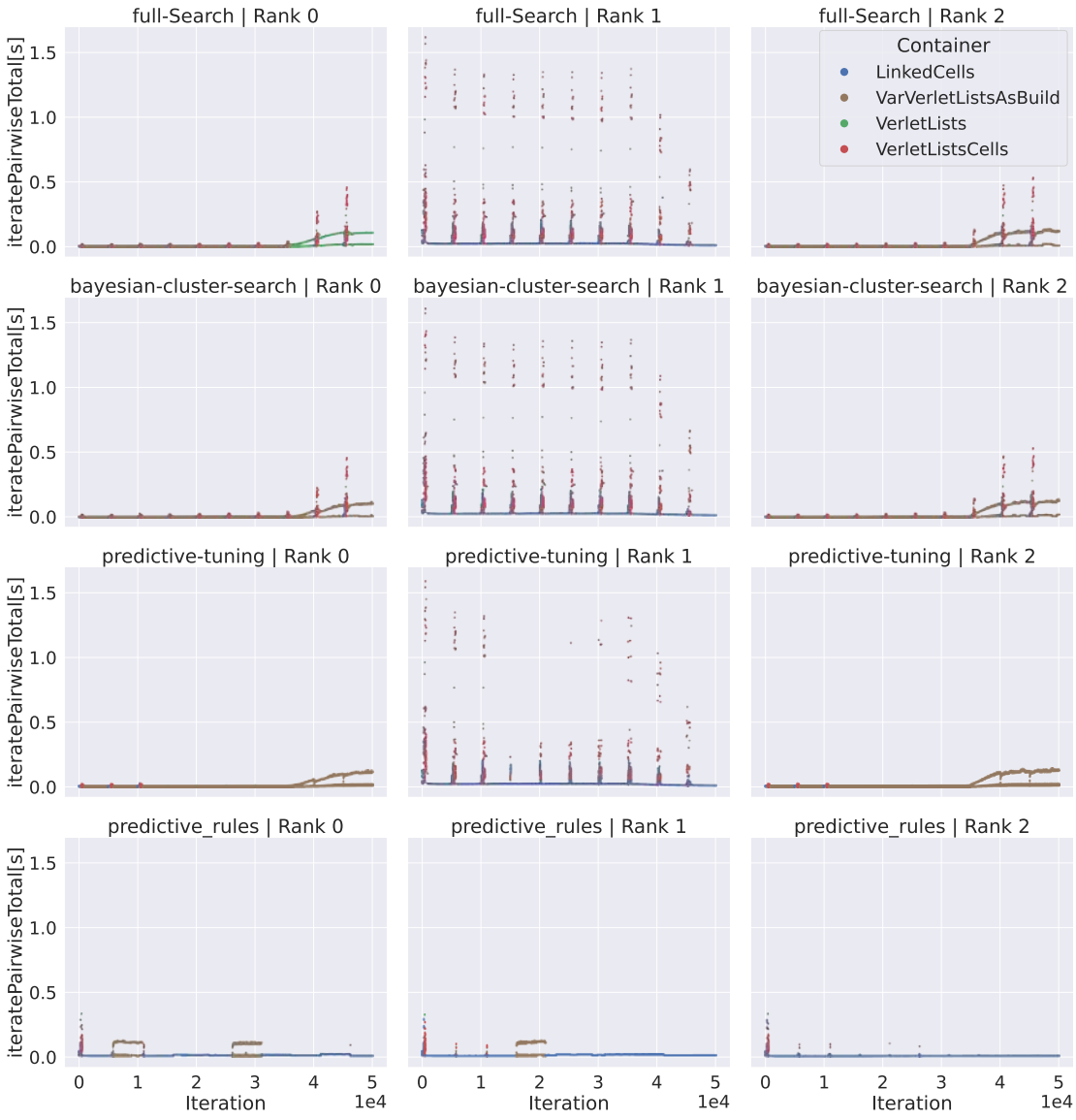
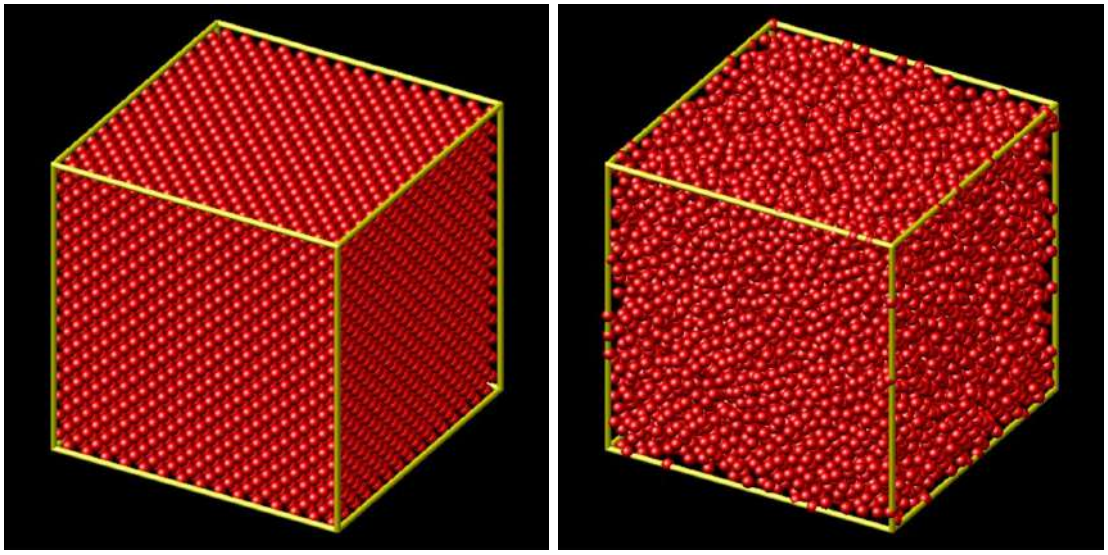


Figure 4.13: The time spent per iteration in the force calculation for different strategies for the first 5 000 iterations of the exploding liquid experiment per rank. Each dot represents one iteration, and its color is the employed container. Every row is one run of the experiment. The vertical clusters are tuning phases. Combining tuning strategies avoids the most expensive outliers.



(a) Initial crystal **(b)** Molten liquid

Figure 4.14: Visualization of the LAMMPS Lennard-Jones liquid benchmark scenario. The left subfigure shows the initialization as a cube of a dense face-centered cubic grid of four million atoms. Subsequently, the grid melts, and a homogeneous but somewhat dynamic liquid ensues.

a permanent container and thus data layout, this is not possible. These instances are resolved by creating an index to particle pointer map, which offers the random access functionality but has to be rebuilt after every full container update.

To underline the usability aspect of AutoPas, the lines of code required for the integration can be compared to those of other accelerator packages like `PKG_USER-OMP` and `PKG_KOKKOS`. The original complete integration of AutoPas into LAMMPS made about 4 800 lines of changes throughout the code [Sau20]. Only about 100 of those are updates of the inner code of LAMMPS. The rest is the creation of the `PKG_USER-AUTOPAS` package. This is significantly less than the absolute size of the OpenMP and Kokkos packages with about 54 000 and 72 000 lines of code each. While both of them might implement more features, the relative difference in lines of code is substantially larger than the relative number of supported features. We thus conclude that the API offered by AutoPas offers a concise way to express functionality and allows for an integration with comparatively little effort.

4.3.2 Lennard-Jones Liquid Benchmark

Setup The performance of the LAMMPS-AutoPas integration is evaluated through the Lennard-Jones liquid benchmark defined on the official LAMMPS website[11]. To make the comparison fairer, the input has been slightly adjusted without changing the

[11] `https://www.lammps.org/bench.html#lj` Accessed: 20.12.2024

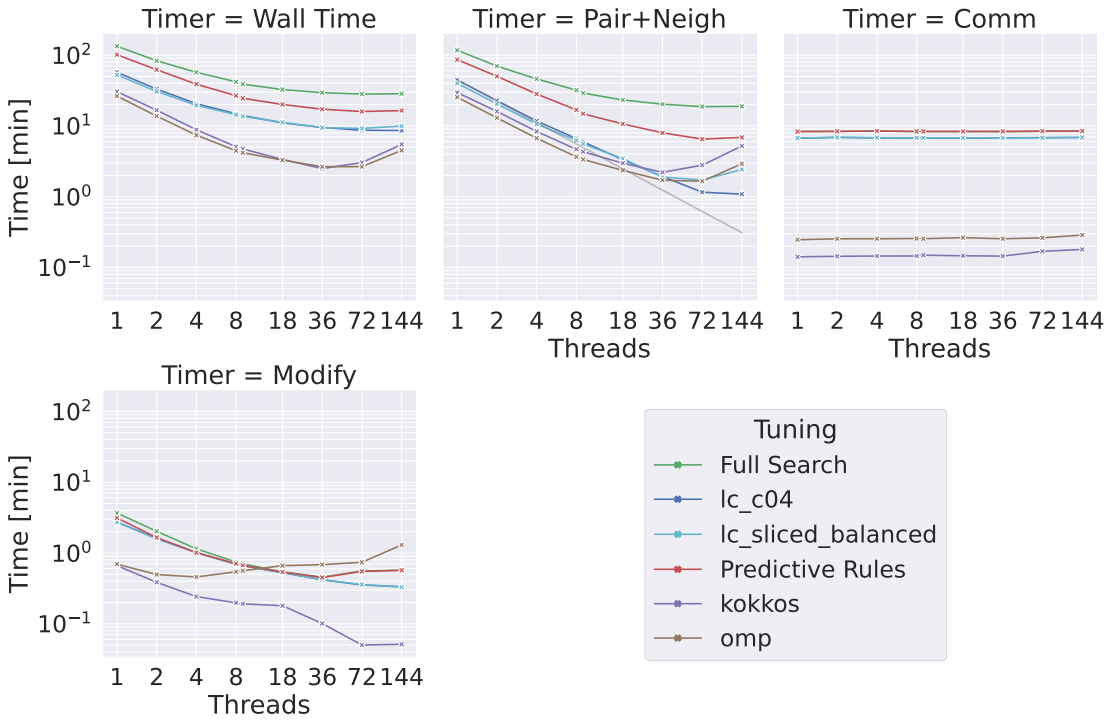


Figure 4.15: Strong scaling of the Lennar-Jones liquid benchmark with LAMMPS. The subplots show timers measuring different parts of the simulation program on a log-log scale. Colors depict different parallelization packages and tuning strategies. omp refers to `PKG_USER-OMP`, kokkos to `PKG_KOKKOS`, and all others to `PKG_USER_AUTOPAS`. The two prefixed with `lc` do not employ tuning and solely use the respective configuration, while the other two employ the respective tuning strategies. In the Pair+Neigh subplot, the gray line without markers illustrates optimal linear scaling. AutoPas without tuning achieves a faster calculation of the pairwise forces due to better scaling. Overall, it is, however, limited by overhead from the Comm phase.

scenario itself. Global indices and thermodynamic information were activated, domain size parameters are scaled more fine grained, and the number of time steps now is also an input value. The exact input file and the commands to launch the benchmark, which also set some parameters, are listed in Section A.5. In this experiment, a cube-shaped domain is initialized with a very dense face-centered cubic grid, as shown in Figure 4.14a, modeling a crystal of Lennard-Jones particles. Through the injection of kinetic energy upon initialization, the crystal melts into a homogeneous liquid, which can be seen in Figure 4.14b.

Strong Scaling Study This experiment was used for a strong scaling study on HSUper with the accelerator packages `PKG_USER-OMP`, `PKG_KOKKOS`, and `PKG_USER_AUTOPAS`. Results from this study are plotted in Figure 4.15, which shows six algorithmic setups and

measurements from four timers. The upper left subplot shows the high-level perspective of the wall time. Looking first at the AutoPas setups, we again see the expected effect that predictive rules tuning beats full search convincingly. However, because this benchmark has very slow dynamics and only lasts for 1 000 iterations and is thus too short to reap the benefits of tuning, it is reasonable to look at the performance of the two AutoPas configurations that perform best in this scenario. These are the two Linked Cells based configurations `lc_c04` and `lc_sliced_balanced`. Both of these configurations only have limited load balancing and schedule big domain chunks as tasks for the threads. Only utilizing these best configurations on such a short simulation gives a speedup of about two over the predictive rules tuning, which is expected as long as the rules are not sharply tailored to the setup. Nevertheless, here, the two packages provided with LAMMPS seem to beat even the optimal AutoPas configurations convincingly by a factor of two to three.

Diving under the hood of the overall time and looking at the timers that show how long the major individual steps take, the picture changes significantly. The upper middle subplot of Figure 4.15 shows the measurements of the timers Pair and Neigh, which had to be added because, from LAMMPS perspective, these two things can not be distinguished in AutoPas. Combined, they represent the time the pairwise force calculation takes and are the main lever for optimizing the parallel efficiency of the overall simulation. Here, we see that the AutoPas algorithms only lack about 30% behind the LAMMPS algorithms for low thread counts. Beyond nine threads, the parallel efficiency of both the OpenMP and Kokkos-based packages fall off, and the AutoPas algorithms break even and `lc_c04` even surpasses them, achieving the overall fastest result when using the full node with hyper-threading. The reason `lc_sliced_balanced` stops scaling beyond 36 threads is that the domain is too tiny, featuring less than 60 cells across. Thus, this coarse-grained algorithm can not distribute the work to all threads, which leads to diminished gains in performance and even a decline when, with 144 threads, significantly more are started than can be used. In their documentation[12], the developer of the LAMMPS Kokkos package acknowledge that their buffer approach, which works by the principles discussed in Subsection 2.1.5, does not scale well beyond eight threads which are confirmed by the here presented findings.

Looking at the plot on the top right, it becomes clear why AutoPas did not beat Kokkos and OpenMP overall. Depicted is the Comm timer, which in this non-MPI based simulation primarily measures the treatment of the periodic boundaries. Because this should not be the critical part of the simulation, no parallelization is employed, which is evident by the performance graph being horizontal and thus not being impacted by the increase in the thread count. Nevertheless, it can be seen that for AutoPas, the time spent is around seven to eight minutes, which thus is a hard scaling limit that is hit and explains the performance gap observed in the wall time plot. This means that due to the integration's overreliance on iterators to replace the aforementioned random access operations, overhead is generated, which limits the performance of the AutoPas integration. These limitations could be overcome either by developing a more complex integration

[12] `https://docs.lammps.org/Speed_kokkos.html` Accessed: 20.12.2024

that avoids these random access operations or with more performance engineering in the AutoPas iterators geared explicitly towards the LAMMPS use case.

Finally, the last plot shows the Modify timer, which measures the time taken for the parallelized update of the particles after the force calculation. OpenMP and Kokkos again start out faster than AutoPas, probably due to simpler to traverse data structures. The OpenMP package relies mainly on data duplication per thread, thus the lack of scaling here.

Conclusion Looking back at the goals set out at the beginning of Section 4.3, some conclusions can be drawn. The overall performance of the LAMMPS-AutoPas integration is not yet at the same level, albeit in the same order of magnitude as native LAMMPS. Reasons for this are the fundamentally different way to access particles and the required overuse of iterators, which in AutoPas still have room for optimization. However, AutoPas shows better scaling behavior in the pairwise force calculation where it can beat LAMMPS. All of this could be achieved with a comparatively lightweight coding effort that still allowed this and more complex simulations [Sau20]. Thus, while there is still room for improvement in this example application, overall, we consider this a successful demonstration of AutoPas being a performant library for pairwise interactions that is easy to integrate into established codes.

4.4 LADDS

Particle simulations can come in different forms on any scale, as was already explored in Section 2.1. All application examples above primarily focus on MD. However, AutoPas is not designed with any built-in assumptions which limit it to this. Hence, this application demonstrates the use of AutoPas in a completely different setting, namely space debris simulation, which is closer to a DEM simulation.

4.4.1 Background

Although alternative approaches exist, the usual approach to simulate and study the evolution of space debris in Low Earch Orbit (LEO) are statistical Monte Carlo simulations [KSH+17, LLA17]. Two highly relevant software models in this context are LEGEND [LHKO04] by National Aeronautics and Space Administration (NASA) and Delta by European Space Agency (ESA) [Vir16]. In the context of an ARIADNA study and this thesis, the Advanced Concept Team of ESA and SCCS studied the feasibility and potential of deterministic evolution and conjunction tracking of large space debris populations. The idea was to challenge the long-standing assumption that these simulations are computationally not feasible with modern algorithms and techniques from MD, brought in by SCCS, where large particle counts in the billions and long simulation times over millions of time steps are familiar and well handled as discussed in Section 2.1. For this, we developed over a short time frame of a few months the Large-scale Determin-

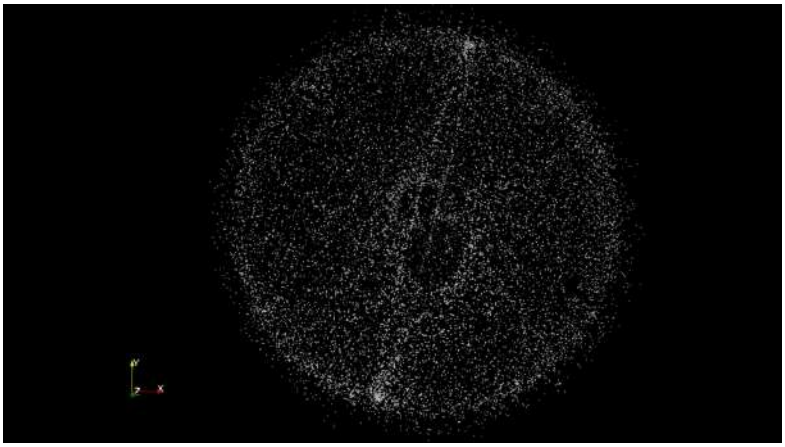


Figure 4.16: Visualization of the reduced dataset used by LADDS of about 16 000 objects in LEO. The rendered perspective looks at the Earth's north pole, which is visible as a small circle of lesser density near the center of the image, and the south pole as a slightly smaller circle behind it. This is because comparatively few satellites are launched in a polar orbit since their launches can not take advantage of the rotation of the Earth and are thus more expensive. From [GG22].

istic Debris Simulation, or short LADDS. This code is written in `C++ 17` and publicly available on GitHub under a GPL-3.0 license[13].

Numerical Model From a modeling perspective, one of the biggest challenges is to find good models for the perturbations influencing the trajectories of particles in orbit. These include but are not limited to, the nonhomogeneous gravity field of Earth, gravitational pull by other celestial bodies like the sun or the moon, solar radiation pressure, and atmospheric drag. Furthermore, in contrast to MD simulations where the exact position is not of relevance, here numerical time integration schemes of higher precision are necessary because we are interested in deterministic conjunctions of objects as small as centimeters in a domain with a diameter of tens of thousands kilometers. Currently, LADDS uses a fourth-order Yoshida integrator, which is an extension of the leapfrog algorithm by some extra steps to achieve higher orders [Yos90]. With this order of precision, it is also possible to propagate the particles for several time steps before searching for conjunctions. The trajectories of particle pairs that have a close encounter between the time steps are then interpolated, and their closest approach is calculated. In LADDS, all these parts are collectively referred to as the propagator, which is described in more detail in the release publications [GGBI22, GGS22]. To enhance the utility of LADDS, the propagator models are plugged into the simulation in a compartmentalized modular way. This enables us to quickly replace, for example, the atmospheric model or the time integration algorithm so their exact impact on precisely repeatable simulation scenarios can be studied. Further code features include modeling and gradual insertion

[13] `https://github.com/esa/LADDS/` Accessed: 20.12.2024

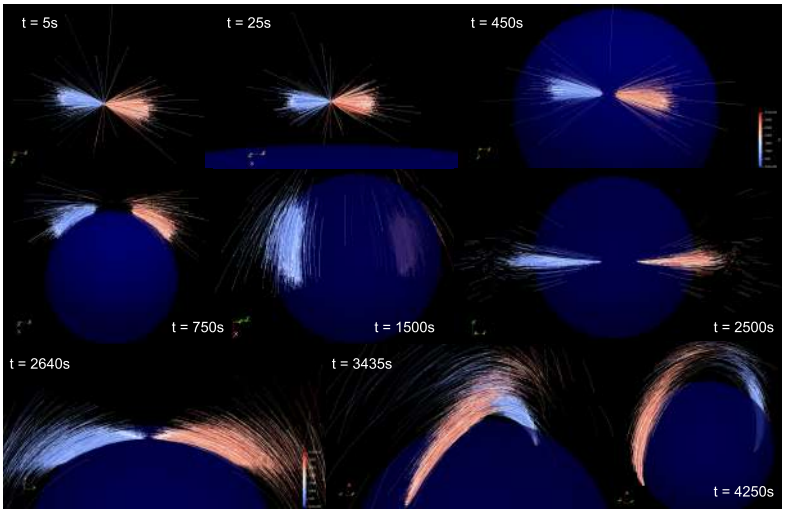
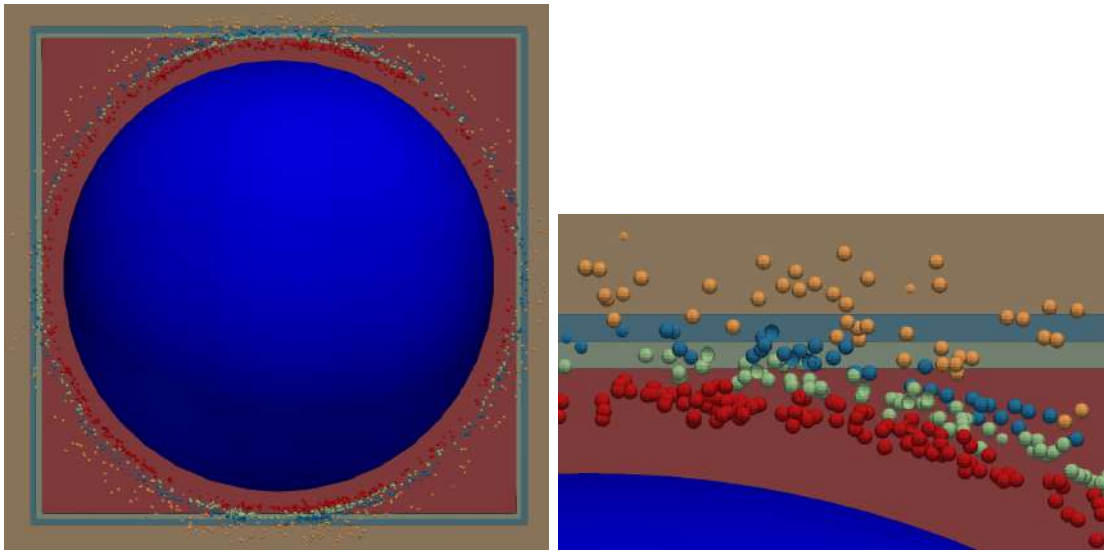


Figure 4.17: Visualization of an isolated collision event at 380 km altitude above ground and the evolution of the debris field over two and a half orbit. Debris particles are colored by their parent object. From [GG22].

of satellite constellations, as well as implementing the NASA breakup model. The latter allows us to either simulate the result of an individual breakup event or, embedded in a population evolution simulation, the collision of two objects, the propagation of the debris fields, and its interaction with the rest of the population. A visualization of an isolated collision simulation is shown in Figure 4.17. From a simulation perspective, such a breakup event presents unique challenges because it suddenly injects potentially thousands of particles into a very tight space, locally heavily increasing the workload.

Computer Science Challenges LADDS is written with AutoPas as its core data structure, allowing for a rapid development process and the first working prototype already featuring OpenMP parallelism. To tackle the goal of simulating a realistic population of all debris of one cm and larger in LEO up to 2 000 km above the ground, a MPI parallelization was added. This proved to be very different from classic MD because of the distribution of particles. In MD, particles are usually spread more or less over the whole domain and move in all directions. Here in the space debris scenario, particles move on an elliptic but close to circular orbit around the Earth as can be seen in Figure 4.16. This makes the classic approach using a grid-based domain decomposition highly inefficient because, for example, in the Earth, which makes up a significant part of the simulation domain, no debris needs to be simulated. Additionally, small subdomains would very quickly be passed through by particles with orbital velocities, inducing high communication overhead. Unfortunately, AutoPas at this time only supports cuboid domain shapes, so the spherical domain of interest has to be simulated with a cube shape, adding the considerable overhead of simulating corners where no orbit can reach. Our alternative approach nests the ranks and thereby the AutoPas instances inside each other like in a Matroshka. Now, each rank can simulate an altitude band of particles, and only very



(a) Full Domain **(b)** Zoom in

Figure 4.18: Visualization of the altitude-based MPI domain decomposition in LADDS. The left figure shows a 2D slab of the entire domain with a reduced dataset. Colors of particles represent the ranks they belong to. Colored regions visualize the extent of the ranks' respective AutoPas containers. From [GG22].

eccentric particles, which are not very frequent, need to be moved between ranks. This approach is visualized as a 2D slab and with a reduced population in Figure 4.18. To achieve a good load balance, the altitudes are selected automatically so that each rank receives approximately the same number of objects.

4.4.2 Benchmark Simulation

Instead of focussing on the value for astrophysics, this thesis reexamines the HPC characteristics, and especially the role AutoPas plays in the program. Therefore, we look at the short benchmark simulation, which is based on the big experiment setup included in the simulator and discussed in the release publication [GGS22].

Setup This simulation comprises around 600 000 objects in LEO, which is based on actual, tracked objects such as satellites or larger pieces of space debris, combined with a population of smaller objects, for which public tracking data is not available, generated according to known distributions [GGS22]. The setup is propagated for 1 000 time steps of ten seconds, and a check for conjunctions happens every tenth iteration. This is, from the particle simulation perspective, the pairwise interaction evaluation. After every iteration, the altitude of each particle is checked. If any are found to be closer than 150 km to the surface of the earth, we consider them to be burning up in the atmosphere and remove them from the simulation. Since the pieces of debris are distributed relatively evenly around the planet and less than one percent deorbits within the short simulation,

tuning is not advisable, similar to in Subsection 4.3.2. Therefore, we fix the configuration to use Linked Cells with `lc_c04_HCP` as this has been evaluated as the best algorithm for this setup [GG22]. Even though large parts of the domain are empty, which typically suggests that Linked Cells is not the best approach, it is still effective because the cells in the domain are huge and thus comprise many particles, leading to effective vectorization. Also, any Verlet Lists-based approaches are at a disadvantage because the high speed of objects in orbit forces large skins or very short rebuild intervals. The complete input configuration is given in Section A.6. Each rank was given 32 OpenMP threads, and each node hosts two ranks, except when there is only one rank. Then, only one node with one rank and 32 threads is used.

MPI Strong Scaling of Components In Figure 4.19, the strong scaling behavior of the most time-intensive components of LADDS are shown. Each plotted datapoint is the sum of the timers of all ranks averaged. Comparing these averages to the measured wall times is exceptionally close to the measured wall time and can be considered equal here. AutoPas primarily drives two components. First, the collision detection through the pairwise interaction evaluation in AutoPas Second, the integrator, which can be considered a compute-intensive for loop over all particles and thus depends on the performance of the `ContainerIterator`. Looking at the slope of these two components' graphs, we see good scaling behavior. The collision detection loses parallel efficiency from 32 ranks because the altitude bands of each rank become very thin, with less than 20 km for the lowest altitudes. Initialization takes a constant time until 32 ranks and then increases rapidly. This is because the initialization was not considered to be important relative to the extremely long run times of the scientifically significant simulations, and thus, no effort was undertaken to optimize it. Currently, the whole input file is loaded and parsed by every rank to find their relevant particles, which does not scale well. The primary bottleneck highlighted by Figure 4.19 is the time spent on communication. Even though the altitude-based decomposition achieved better times to solution than the naive grid-based one, it still suffers from an enormous surface area between the ranks on which particles need to be exchanged. This communication volume is roughly the same for each rank due to the load balancing leading to added overall costs per added rank.

Conclusion LADDS is a small project that employs AutoPas to successfully demonstrate the feasibility of simulating realistic space debris populations deterministically. AutoPas enables good scaling behavior and performance for the components that primarily interact with the library. For larger simulations, better tailored MPI decomposition solutions have to be identified, but that is a task that is not in the realm of responsibility of AutoPas. Nevertheless, it could help to offer the possibility to use polar coordinates instead of only Euclidean ones and non-cuboid domain shapes.

However, with the implementation presented here, significant results are achievable. Our release publication [GGS22] describes the setup for two data sets and our achieved speeds on CoolMUC2 as shown in Table 4.2. With this, we were able to conduct a simulation of the evolution of the small base dataset over 20 years in 218.25 hours or

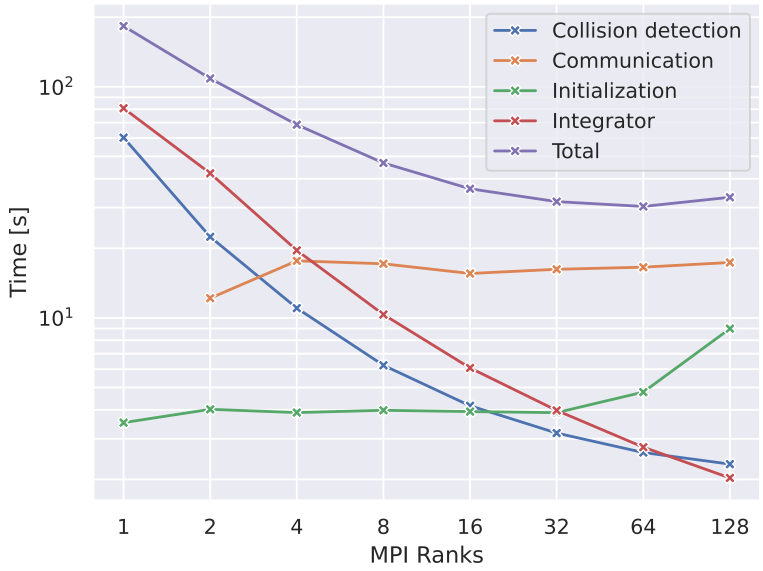


Figure 4.19: Wall times of the main components of LADDS in a strong scaling study of the big debris scenario. Collision detection and the integrator mainly reflect the performance of AutoPas and scale very well. Initialization is generating overhead that rises beyond 32 ranks. With the high number of objects, communication time is a problem that is linear in the number of ranks. Total depicts the total of all individual timers and thus shows the scaling of the whole application.

Population Name	# Objects	s/iter
Base	16 024	0.012
Small Debris	614 515	0.21

Table 4.2: Simulation speed of the two debris populations on CoolMUC2 [GGS22].

just over 9 days. Now, with the current[14] version of AutoPas, the new altitude-based domain decomposition, and on the more modern system HSUper, the simulation speed of the larger population is at 0.026 seconds per iteration with 64 MPI ranks. This brings a long evolution simulation of the big population within reach.

Not captured by the study is the enormous ease of development AutoPas brought to the project. LADDS itself only consists of just over 4 000 lines of code, including all CMake code for the build system. The here used standalone propagator adds another 2 000, but this is not specific to AutoPas and can be switched out. Furthermore, with different stages of development and scenarios, different neighbor identification algorithms proved to be advantageous, which was very simple to adapt to [GG22].

4.5 Interim Summary

This chapter's purpose was to showcase how well the theories behind AutoPas work in reality, but also the usability of its abstractions in terms of the ability to integrate into different simulation environments, and last but not least, its performance. For this, four simulation codes that use AutoPas and examples were demonstrated and discussed.

Performance and Scalability To put the simulation speed into perspective, AutoPas was integrated into the codes ls1 mardyn and LAMMPS. Then, benchmark simulations were conducted with the vanilla versions of the code and the AutoPas versions. In the exploding liquid scenario, discussed in Subsection 4.2.2, AutoPas with the proper tuning approach is faster than vanilla ls1 mardyn as shown in Figure 4.12. The LAMMPS-AutoPas integration proved more challenging. In contrast to ls1 mardyn, the background of the developers is significantly different from ours, which leads to more significant differences in code architecture. This results in some friction overhead, as discussed, that still leaves room for optimization. Nevertheless, the results achieved in the Lennard-Jones Liquid benchmark in Subsection 4.3.2, and Figure 4.15 speed is comparable to native LAMMPS but offers better scalability than both the `KOKKOS` and `USER-OMP` packages. This leads to higher peak performance when using the `USER-AUTOPAS` package and a complete compute node.

Viability of Tuning Methods The different simulation scenarios and benchmarks also offered insights into the potential of automated algorithm selection and the viability of different approaches to it. Subsection 4.2.2 with the exploding liquid demonstrated that,

[14] See Section A.6 for the exact version.

especially in scenarios with quickly changing conditions, the combination of situation-specific learning with prior formulated general expert knowledge is the key to unlocking the full potential of the tuning approach. In order to gather and express this knowledge, metrics are necessary to describe the state of the domain. Subsection 4.1.2 showed a correlation between specific domain properties and the likelihood of certain configurations being picked. Subsequently, Subsection 4.1.3 introduced metrics AutoPas can gather on the fly and in Figure 4.10 shows how it uses them for its rule-based tuning.

API Usability Integrating AutoPas into four different code bases at different stages of their development offers a nuanced perspective on the effectiveness and utility of our API. The test bench md-flexible, which turned into a full simulator capable of conducting MPI parallel simulations involving features like thermostats, checkpointing, dynamic load balancing, output for visualization and analysis of AutoPas, evolved along with AutoPas. Hence, judging the API only based on this can be misleading. Similarly, ls1 mardyn was a significant source of inspiration during the development of AutoPas. Nevertheless, integrating AutoPas into it demonstrates a degree of versatility and compatibility with code initially not in scope when developing AutoPas. The integration into LAMMPS is a much sterner test because of the significantly different architectures touched upon in Section 4.3. Indeed, the integration proved reasonably straightforward compared to other parallelization and data structure enhancement packages. Finally, with LADDS, a new simulator that, in contrast to the others, is not concerned with MD was built when AutoPas was already in a very advanced stage with a stable API. Here, AutoPas significantly accelerated the development process and led to excellent performance from the very start and after every collision model change because of its ability to change algorithms, even if they do not have to change at runtime.

5 Conclusion and Outlook

This chapter brings together all the results and insights of everything written in this thesis until here, summarizes it, and ties it back to the initial research questions and objectives. In light of the presented insights, they are discussed, and answers are proposed. Finally, we glimpse the future of AutoPas on what is conceivable or already planned.

5.1 Recap and Discussion

In Section 1.1, the thesis opened up with a trio of research questions it set out to answer. Now, at the end of the work, they shall be evaluated and answered.

1. *Is there a feasible API interface for all short-ranged particle interaction algorithms?*

   Before the question is tackled directly, in Subsection 2.1.1, the thesis revisited the theoretical foundations of particle interaction algorithms, and in Subsection 2.1.2, the broader particle simulations context they are applied to. The abstract algorithms were discussed, formulated, and compared in Subsection 2.1.3. Here, similarities and overlaps were identified that can be taken advantage of in the implementation or that have to be carefully balanced.

   Throughout Chapter 3, the library AutoPas was presented with comprises of all the presented algorithms and offers a unified API as a high-level response to the research question. To develop a more profound answer, first, in Subsection 3.1.1, the abstract model that AutoPas uses to interact with particle simulations was sketched, and its inner as well as user-facing structure was described. Here, the highest potential for improving usability is the functor's API. While it offers a unified API to interact with all algorithms, the user has to implement their kernel in at least two ways: One for AoS and one for SoA, which offers fine-grained manual optimization access but also requires the technical knowledge to exploit this otherwise uncomfortable design. The more technical aspects, which answer the nuances of this research question, were tackled in Subsection 3.1.2. Our approach to a black box interface that implements a hybrid Verlet Lists-Linked Cells style was defined, and its implementation in AutoPas was explained. This implementation offers a convenient way to interact with any of the presented algorithms without having to know which one is behind the facade of the library. However, due to the scope of AutoPas as the simulation's particle container, the API can not be limited only to the pairwise iteration. Solutions for further abstractions like accessing and traversing particles in serial and parallel are shown, as well as `C++`

template-based code generation for SoA data structures that all algorithms can use. Subsection 3.1.3 discussed how specialized optimizations are still possible behind the API for best performance on different hardware architectures.

Finally, the four integrations presented in Chapter 4 confirm that the high level API is not only feasible from a design perspective but is also usable with comparatively little effort in different code bases. Subsection 4.1.1 showed this for a simulator that was developed alongside and for AutoPas. In contrast, Subsection 4.2.1 and Subsection 4.3.1 demonstrated the ability to integrate into an existing code base, with both having a very different architectural concept. With LADDS in Subsection 4.4.1, a powerful simulator from a different domain was shown that was built on top of AutoPas and highly benefitted from its API by having a small code base that only focuses on the actual application.

2. *Can an automated dynamic selection of the short-ranged particle interaction algorithm bring advantages for particle simulations?*

   In order to formulate an answer to this question, Subsection 2.2.1 defined the theoretical concept of the algorithm selection problem. Subsection 2.2.2 extended this to the automation of it, important properties and key considerations on how to bring down the complexity to a feasible degree were highlighted. The algorithmic level, individual advantages, and algorithmic properties that can lead to advantages under specific circumstances were discussed in Subsection 2.1.4. Subsection 2.1.5 and 2.1.6 discussed the different potentials of parallelization.

   So, from a theoretical point of view, especially these three parts have shown that the algorithms expose differences that can be taken advantage of. Additionally, Section 4.1 demonstrated in synthetic as well as realistic simulations that different scenarios do indeed lead to different algorithm optima.

3. *Is this approach practical and delivers performance while general enough to extend beyond its application in MD?*

   Providing practical access for the user and its ease of integration into simulation engines have already been solved by the answer to the first question. For this question, the internal practicality of the tuning process is also relevant since this is tightly coupled to the overall performance. Section 3.2 explained in detail the implementation starting with Subsection 3.2.1 of the theoretical concepts from Section 2.2. Next, Subsection 3.2.2 described the heart of the automated algorithm selection process, the tuning loop, how it advances the simulation while gathering information, and how it can employ several tuning strategies with a plugin-like interface. These strategies were then explained in Subsection 3.2.3, concluding the first part of the question and demonstrating a practical implementation of the approach.

   Chapter 4 responded to the second and third parts by putting AutoPas to action to demonstrate its performance in different settings. Section 4.2 and Subsection 4.3.2 showed that even established simulators with a long optimization history

can benefit from AutoPas under certain circumstances. The conclusion from this chapter was that depending on the situation, automated dynamic algorithm selection can be beneficial, as in Subsection 4.1.3 and Section 4.2, but in others, like Subsection 4.3.2 static choices are more advantageous because the scenario does not change its state. Thus the potential impact of the AutoPas approach significantly depends on the software and simulations it is integrated into. Nevertheless, since some choice for an algorithm always has to be made, having the ability to choose from a zoo of already implemented algorithms clearly grants flexibility and speedup in the development process. AutoPas offers this ability by simply adapting the arguments passed to it without having to adapt the simulator's interaction with it or even recompilation.

The applicability and performance beyond MD were demonstrated with Section 4.4, where a space debris simulation was developed and successfully employed.

So, in conclusion, we can say that the automated algorithm selection offered by AutoPas, while not providing superior time to solution in every scenario, is a beneficial tool, especially for evaluating new algorithms or facilitating the development of new simulators.

5.2 Future Directions

Even if the presented state and set of features of AutoPas is considerable, there is always more to be done.

The experiments showed that learning without knowledge requires too many trial evaluations of potentially expensive functions to be of great use. However, the here presented solution relies on expert knowledge provided by a user. The next step is to have a data-driven approach that learns and builds a model from a range of synthetic simulations or a series of real simulations and reuses this in subsequent, new simulations. Simple classification algorithms could be considered like random forests. They also have the advantage of high interpretability, so it might be possible to derive rules from a trained forest to feed back into the rule-based tuning.

Subsection 3.2.4 touched upon the possibility of tuning for energy efficiency. When more than one tuning objective is available, the next step is to have a multi-objective tuning goal with constraints instead of choosing between them. For example, it could be conceivable to tune a simulation for an optimal runtime given a limited energy budget or the other way around to finish within a specific time with the least possible energy used.

In Subsection 3.1.1.1, the growing relevance of GPUs was mentioned. Sooner or later, AutoPas should support offloading the interaction calculations to accelerators to be able to fully utilize the newest generation of heterogeneous supercomputers. An implementation for this could be written with a performance portability framework like Kokkos or SYCL to maximize hardware support and minimize the amount of code that has to be maintained. This is interesting for additional reasons. On the one hand, existing

algorithms either have to be adapted to be efficient on GPU architectures or new ones added, necessitating relearning all gathered expert knowledge on new hardware. On the other hand, this capability enables any simulator that integrates AutoPas immediately to utilize supported accelerators, considerably boosting the library's attractiveness.

Bibliography

[ACMR06] Warren Armstrong, Peter Christen, Eric McCreath, and Alistair P Rendell. Dynamic algorithm selection using reinforcement learning. In *2006 international workshop on integrating ai and data mining*, pages 18–25. IEEE, 2006.

[ADPK23] Peter Atkins, Julio De Paula, and James Keeler. *Atkins' physical chemistry*. Oxford university press, 2023.

[AGG20] Joshua A Anderson, Jens Glaser, and Sharon C Glotzer. HOOMD-blue: A Python package for high-performance molecular dynamics and hard particle Monte Carlo simulations. *Computational Materials Science*, 173:109363, 2020.

[ALT08] Joshua A Anderson, Chris D Lorenz, and Alex Travesset. General purpose molecular dynamics simulations fully implemented on graphics processing units. *Journal of computational physics*, 227(10):5342–5359, 2008.

[Ame14] William F Ames. *Numerical methods for partial differential equations*. Academic press, 2014.

[AMS+15] Mark James Abraham, Teemu Murtola, Roland Schulz, Szilárd Páll, Jeremy C Smith, Berk Hess, and Erik Lindahl. GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers. *SoftwareX*, 1:19–25, 2015.

[Bab79] László Babai. Monte-Carlo algorithms in graph isomorphism testing. Technical Report 79-10, Université tde Montréal, 1979.

[BH86] Josh Barnes and Piet Hut. A hierarchical O (N log N) force-calculation algorithm. *nature*, 324(6096):446–449, 1986.

[BHvM09] Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of satisfiability*, volume 185. IOS press, 2009.

[BI21] Francesco Biscani and Dario Izzo. Revisiting high-order Taylor methods for astrodynamics and celestial mechanics. *Monthly Notices of the Royal Astronomical Society*, 504(2):2614–2628, 2021.

[BO00] Max Born and Robert Oppenheimer. On the quantum theory of molecules. In *Quantum Chemistry: Classic Scientific Papers*, pages 1–24. World Scientific, 2000.

[BRP05] Javier Bonet and Miguel X Rodríguez-Paz. Hamiltonian formulation of the variable-h SPH equations. *Journal of Computational Physics*, 209(2):541–558, 2005.

[BRU+20] Hans-Joachim Bungartz, Severin Reiz, Benjamin Uekermann, Philipp Neumann, and Wolfgang E Nagel. *Software for exascale computing-SPPEXA 2016-2019*. Springer Nature, 2020.

[Buc10] Martin Buchholz. *Framework zur Parallelisierung von Molekulardynamiksimulationen in verfahrenstechnischen Anwendungen*. PhD thesis, Technische Universität München, 2010.

[Cah61] John W Cahn. On spinodal decomposition. *Acta metallurgica*, 9(9):795–801, 1961.

[CCH+89] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C Mitchell. F-bounded polymorphism for object-oriented programming. In *Proceedings of the fourth international conference on functional programming languages and computer architecture*, pages 273–280, 1989.

[Cio08] Florina-Monica Ciorba. *Algorithms Design for the Parallelization of Nested Loops*. PhD thesis, Εθνικό Μετσόβιο Πολυτεχνείο (ΕΜΠ). Σχολή Ηλεκτρολόγων Μηχανικών και ..., 2008.

[CKP91] David Callahan, Ken Kennedy, and Allan Porterfield. Software prefetching. *ACM SIGARCH Computer Architecture News*, 19(2):40–52, 1991.

[CLZ+21] Genshen Chu, Yang Li, Runchu Zhao, Shuai Ren, Wen Yang, Xinfu He, Changjun Hu, and Jue Wang. MD simulation of hundred-billion-metal-atom cascade collision on Sunway Taihu light. *Computer Physics Communications*, 269:108128, 2021.

[C/M19] C/MSC - Microprocessor Standards Committee. IEEE Standard for Floating-Point Arithmetic. Standard, IEEE Computer Society, 2019.

[Cop96] James O Coplien. Curiously recurring template patterns. In *C++ gems*, pages 135–144. 1996.

[Cou85] Charles-Augustin Coulomb. First Memoir on Electricity and Magnetism. *A Source Book in Physics*, pages 408–413, 1785.

[CP23] Tara Chari and Lior Pachter. The specious art of single-cell genomics. *PLOS Computational Biology*, 19(8):e1011288, 2023.

[DBK+16] Hans Degroote, Bernd Bischl, Lars Kotthoff, Patrick De Causmaecker, and Brona Brejová. Reinforcement learning for automatic online algorithm selection-an empirical study. *ITAT 2016 Proceedings*, 1649:93–101, 2016.

[DCGGM11] Jose M Domínguez, Alejandro JC Crespo, Moncho Gómez-Gesteira, and Jean C Marongiu. Neighbour lists in smoothed particle hydrodynamics. *International Journal for Numerical Methods in Fluids*, 67(12):2026–2042, 2011.

[DFJ54] George Dantzig, Ray Fulkerson, and Selmer Johnson. Solution of a large-scale traveling-salesman problem. *Journal of the operations research society of America*, 2(4):393–410, 1954.

[DM11] Jacob D Durrant and J Andrew McCammon. Molecular dynamics simulations and drug discovery. *BMC Biology*, 9(1):1–9, 2011.

[DOW+22] Mike Diessner, Joseph O'Connor, Andrew Wynn, Sylvain Laizet, Yu Guan, Kevin Wilson, and Richard D Whalley. Investigating Bayesian optimization for expensive-to-evaluate black box functions: Application in fluid dynamics. *Frontiers in Applied Mathematics and Statistics*, 8:1076296, 2022.

[DYP93] Tom Darden, Darrin York, and Lee Pedersen. Particle mesh Ewald: An N log (N) method for Ewald sums in large systems. *The Journal of chemical physics*, 98(12):10089–10092, 1993.

[EB96] William D Elliott and John A Board, Jr. Fast Fourier transform accelerated fast multipole algorithm. *SIAM Journal on Scientific Computing*, 17(2):398–415, 1996.

[Eck14] Wolfgang Eckhardt. *Efficient HPC Implementations for Large-Scale Molecular Simulation in Process Engineering*. Dissertation, Institut für Informatik, Technische Universität München, München, June 2014. Dissertation erhältlich im Verlag Dr. Hut unter ISBN 978-3-8439-1746-9.

[EEZS+21] Mahmoud A El-Emam, Ling Zhou, Weidong Shi, Chen Han, Ling Bai, and Ramesh Agarwal. Theories and applications of CFD–DEM coupling approach for granular flow: A review. *Archives of Computational Methods in Engineering*, pages 1–42, 2021.

[EHB+13] Wolfgang Eckhardt, Alexander Heinecke, Reinhold Bader, Matthias Brehm, Nicolay Hammer, Herbert Huber, Hans-Georg Kleinhenz, Jadran Vrabec, Hans Hasse, Martin Horsch, Martin Bernreuther, Colin W. Glass, Christoph Niethammer, Arndt Bode, and Hans-Joachim Bungartz. 591 TFLOPS Multi-Trillion Particles Simulation on SuperMUC. In *International Supercomputing Conference*, pages 1–12. Springer, 2013.

[EHL80] James W Eastwood, Roger Williams Hockney, and DN Lawrence. P3M3DP-The three-dimensional periodic particle-particle/particle-mesh program. *Computer Physics Communications*, 19(2):215–261, 1980.

[Eul65] Leonhard Euler. *Theoria motus corporum solidorum seu rigidorum (etc.)(Cum tabulis aeneis.)*. AF Röse, 1765.

[Ewa21] Paul Peter Ewald. The calculation of optical and electrostatic grid potential. *Annalen der Physik*, 64(3):253–287, 1921.

[FHLS10] Peter L Freddolino, Christopher B Harrison, Yanxin Liu, and Klaus Schulten. Challenges in protein-folding simulations. *Nature physics*, 6(10):751–758, 2010.

[FM08] EP Favvas and A Ch Mitropoulos. What is spinodal decomposition. *J. Eng. Sci. Technol. Rev*, 1(1):25–27, 2008.

[Fom11] Eduard S Fomin. Consideration of data load time on modern processors for the Verlet table and linked-cell algorithms. *Journal of Computational Chemistry*, 32(7):1386–1399, 2011.

[Gar23] Roman Garnett. *Bayesian Optimization*. Cambridge University Press, 2023.

[GCBC21] Rajesh Ghosh, Ayon Chakraborty, Ashis Biswas, and Snehasis Chowdhuri. Evaluation of green tea polyphenols as novel corona virus (SARS CoV-2) main protease (Mpro) inhibitors–an in silico docking and molecular dynamics simulation study. *Journal of Biomolecular Structure and Dynamics*, 39(12):4362–4374, 2021.

[GG22] Fabio Alexander Gratl and Pablo Gómez. Exploring the Use of Molecular Dynamics Simulations for High-Performance Space Debris Collision Modelling, Nov 2022. Final presentation.

[GGBI22] Pablo Gómez, Fabio Gratl, Oliver Bösing, and Dario Izzo. Deterministic conjunction tracking in long-term space debris simulations. *arXiv preprint arXiv:2203.06957*, 2022.

[GGS22] Pablo Gómez, Fabio Gratl, and Leopold Summerer. Exploring the Use of Molecular Dynamics Simulations for High-Performance Space Debris Collision Modelling. Technical report, European Space Agency, 2022.

[GKD19] Jochen Görtler, Rebecca Kehlbeck, and Oliver Deussen. A Visual Exploration of Gaussian Processes. *Distill*, 2019. https://distill.pub/2019/visual-exploration-gaussian-processes.

[GKZ07] Michael Griebel, Stephan Knapek, and Gerhard Zumbusch. *Numerical Simulation in Molecular Dynamics: Numerics, Algorithms, Parallelization, Applications*, volume 5. Springer Science & Business Media, 2007.

[GM77] Robert A Gingold and Joseph J Monaghan. Smoothed particle hydrodynamics: theory and application to non-spherical stars. *Monthly notices of the royal astronomical society*, 181(3):375–389, 1977.

[GM09] Sanjay Ghemawat and Paul Menage. Tcmalloc: Thread-caching malloc, 2009.

[GNT04] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: theory and practice*. Elsevier, 2004.

[Gon07] Pedro Gonnet. A simple algorithm to accelerate the computation of non-bonded interactions in cell-based molecular dynamics simulations. *Journal of Computational Chemistry*, 28(2):570–573, 2007.

[GR87] Leslie Greengard and Vladimir Rokhlin. A fast algorithm for particle simulations. *Journal of computational physics*, 73(2):325–348, 1987.

[Gra17a] Fabio Gratl. Implementation and Evaluation of Task-based Approaches for Molecular Dynamics Simulations. Studienarbeit/sep/idp, Institut für Informatik, April 2017.

[Gra17b] Fabio Alexander Gratl. Task Based Parallelization of the Fast Multipole Method implementation of ls1-mardyn via QuickSched. Master's thesis, Technische Universität München, Nov 2017.

[GS10] Matteo Gagliolo and Jürgen Schmidhuber. Algorithm selection as a bandit problem with unbounded losses. In *International conference on learning and intelligent optimization*, pages 82–96. Springer, 2010.

[GSBN22] Fabio Alexander Gratl, Steffen Seckler, Hans-Joachim Bungartz, and Philipp Neumann. N Ways to Simulate Short-Range Particle Systems: Automated Algorithm Selection with the Node-Level Library AutoPas. *Computer Physics Communications*, 273:108262, 2022.

[GSK+21] Sanjay Gupta, Atul Kumar Singh, Prem Prakash Kushwaha, Kumari Sunita Prajapati, Mohd Shuaib, Sabyasachi Senapati, and Shashank Kumar. Identification of potential natural inhibitors of SARS-CoV2 main protease by molecular docking and simulation studies. *Journal of Biomolecular Structure and Dynamics*, 39(12):4334–4345, 2021.

[GST+19] Fabio Alexander Gratl, Steffen Seckler, Nikola Tchipev, Hans-Joachim Bungartz, and Philipp Neumann. Autopas: Auto-tuning for particle simulations. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 748–757. IEEE, 2019.

[GZ14] Ning Guo and Jidong Zhao. A coupled FEM/DEM approach for hierarchical multiscale modelling of granular media. *International Journal for Numerical Methods in Engineering*, 99(11):789–818, 2014.

[Gä19] Ludwig Gärtner. Performance Analysis and Code Generation for the Force Calculation in Molecular Dynamics Simulations. Master's thesis, Technical University of Munich, Oct 2019.

[HA09] XY Hu and Nikolaus A Adams. A constant-density approach for incompressible multi-phase SPH. *Journal of Computational Physics*, 228(6):2082–2091, 2009.

[HAN+16] Michael P Howard, Joshua A Anderson, Arash Nikoubashman, Sharon C Glotzer, and Athanassios Z Panagiotopoulos. Efficient neighbor list calculation for molecular simulation of colloidal systems using graphics processing units. *Computer Physics Communications*, 203:45–52, 2016.

[HEHB15] Alexander Heinecke, Wolfgang Eckhardt, Martin Horsch, and Hans-Joachim Bungartz. *Supercomputing for Molecular Dynamics Simulations: Handling Multi-Trillion Particles in Nanofluidics*. Springer, 2015.

[Her81] Heinrich Hertz. Über die Berührung fester elastischer Körper. *J reine und angewandte Mathematik*, 92:156, 1881.

[HF84] Kai Hwang and A Faye. *Computer architecture and parallel processing*. McGraw-Hill, New York, NY, USA, 1984.

[HKB+21] Keefe Huang, Moritz Krügener, Alistair Brown, Friedrich Menhorn, Hans-Joachim Bungartz, and Dirk Hartmann. Machine learning-based optimal mesh generation in computational fluid dynamics. *arXiv preprint arXiv:2102.12923*, 2021.

[HLW03] Ernst Hairer, Christian Lubich, and Gerhard Wanner. Geometric numerical integration illustrated by the Störmer–Verlet method. *Acta numerica*, 12:399–450, 2003.

[HLYK08] Jeong-Mo Hong, Ho-Young Lee, Jong-Chul Yoon, and Chang-Hun Kim. Bubbles alive. *ACM Transactions on Graphics (TOG)*, 27(3):1–4, 2008.

[HSD12] Barthélémy Harthong, Luc Scholtès, and Frédéric-Victor Donzé. Strength characterization of rock masses, using a coupled DEM–DFN model. *Geophysical Journal International*, 191(2):467–480, 2012.

[HSM+19] Michael P Howard, Antonia Statt, Felix Madutsa, Thomas M Truskett, and Athanassios Z Panagiotopoulos. Quantized bounding volume hierarchies for neighbor search in molecular simulations on graphics processing units. *Computational Materials Science*, 164:139–146, 2019.

[HV19] Matthias Heinen and Jadran Vrabec. Evaporation sampled by stationary molecular dynamics simulation. *The Journal of Chemical Physics*, 151(4), 2019.

[ISO22] ISO Central Secretary. Software, systems and enterprise Architecture description. Standard, International Organization for Standardization, Geneva, CH, 2022.

[Jac21] Rebecca L. Jackson. "The Uncertain Method of Drops": How a Non-Uniform Unit Survived the Century of Standardization. *Perspectives on Science*, 29(6):802–841, 11 2021.

[JH02] Lanru Jing and JA Hudson. Numerical methods in rock mechanics. *International Journal of Rock Mechanics and Mining Sciences*, 39(4):409–427, 2002.

[Kab12] Ivo Kabadshow. *Periodic Boundary Conditions and the Error-Controlled Fast Multipole Method*, volume 11. Forschungszentrum Jülich, 2012.

[KBC20] Seketoulie Keretsu, Swapnil P Bhujbal, and Seung Joo Cho. Rational approach toward COVID-19 main protease inhibitors via molecular docking, molecular dynamics simulation and free energy calculation. *Scientific reports*, 10(1):17716, 2020.

[KHNT19] Pascal Kerschke, Holger H Hoos, Frank Neumann, and Heike Trautmann. Automated algorithm selection: Survey and perspectives. *Evolutionary computation*, 27(1):3–45, 2019.

[Kim14] Sangrak Kim. Issues on the choice of a proper time step in molecular dynamics. *Physics Procedia*, 53:60–62, 2014.

[Kim15] Sangrak Kim. Time step and shadow Hamiltonian in molecular dynamics simulations. *Journal of the Korean Physical Society*, 67:418–422, 2015.

[KL51] Solomon Kullback and Richard A Leibler. On information and sufficiency. *The annals of mathematical statistics*, 22(1):79–86, 1951.

[KMS+11] Serdar Kadioglu, Yuri Malitsky, Ashish Sabharwal, Horst Samulowitz, and Meinolf Sellmann. Algorithm selection and scheduling. In *Principles and Practice of Constraint Programming–CP 2011: 17th International Conference, CP 2011, Perugia, Italy, September 12-16, 2011. Proceedings 17*, pages 454–469. Springer, 2011.

[KMST10] Serdar Kadioglu, Yuri Malitsky, Meinolf Sellmann, and Kevin Tierney. ISAC–instance-specific algorithm configuration. In *ECAI 2010*, pages 751–756. IOS Press, 2010.

[KMT94] Eiichiro Kokubo, Junichiro Makino, and Makoto Taiji. HARP-1: a special-purpose computer for N-body simulation with the Hermite integrator. In *1994 Proceedings of the Twenty-Seventh Hawaii International Conference on System Sciences*, volume 1, pages 292–301. IEEE, 1994.

[KMV22] Gaurav Kumar, Radha Raman Mishra, and Akarsh Verma. Introduction to molecular dynamics simulations. In *Forcefields for Atomistic-Scale Simulations: Materials and Applications*, pages 1–19. Springer, 2022.

[Kos07] Rainer Koschke. Survey of research on software clones. In *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2007.

[KSH+17] Christopher Kebschull, Philipp Scheidemann, Sebastian Hesselbach, Jonas Radtke, Vitali Braun, H Krag, and Enrico Stoll. Simulation of the space debris environment in LEO using a simplified approach. *Advances in Space Research*, 59(1):166–180, 2017.

[Kut01] Wilhelm Kutta. *Beitrag zur näherungsweisen Integration totaler Differentialgleichungen*. Teubner, 1901.

[LD17] David Lowry-Duda. *On Some Variants of the Gauss Circle Problem*. PhD thesis, Brown University Providence, Rhode Island, 2017.

[Lee06] M Lee. Analysis of high-explosive fragmenting shell impact into spaced plates. *International journal of impact engineering*, 33(1-12):364–370, 2006.

[LHH15] Marius Lindauer, Holger Hoos, and Frank Hutter. From sequential algorithm selection to parallel portfolio selection. In *Learning and Intelligent Optimization: 9th International Conference, LION 9, Lille, France, January 12-15, 2015. Revised Selected Papers 9*, pages 1–16. Springer, 2015.

[LHHS15] Marius Lindauer, Holger H Hoos, Frank Hutter, and Torsten Schaub. Autofolio: An automatically configured algorithm selector. *Journal of Artificial Intelligence Research*, 53:745–778, 2015.

[LHKO04] J-C Liou, DT Hall, PH Krisko, and JN Opiela. LEGEND–a three-dimensional LEO-to-GEO debris evolutionary model. *Advances in Space Research*, 34(5):981–986, 2004.

[LJ24] John Edward Lennard-Jones. On the determination of molecular fields. II. From the equation of state of a gas. In *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, volume 106, pages 463–477. The Royal Society, 1924.

[LJ31] John Edward Lennard-Jones. Cohesion. *Proceedings of the Physical Society*, 43(5):461, 1931.

[LKV12] Jaekyu Lee, Hyesoon Kim, and Richard Vuduc. When prefetching works, when it doesn't, and why. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(1):1–29, 2012.

[LL10] MB Liu and GR2593940 Liu. Smoothed particle hydrodynamics (SPH): an overview and recent developments. *Archives of computational methods in engineering*, 17:25–76, 2010.

[LLA17] Aleksander A Lidtke, Hugh G Lewis, and Roberto Armellin. Statistical analysis of the inherent variability in the results of evolutionary debris models. *Advances in Space Research*, 59(7):1698–1714, 2017.

[LLLZ03] MB Liu, GR Liu, KY Lam, and Z Zong. Smoothed particle hydrodynamics for numerical simulation of underwater explosion. *Computational mechanics*, 30:106–118, 2003.

[LMLY10] G Liu, JS Marshall, SQ Li, and Q Yao. Discrete-element method for particle capture by a body in an electrostatic field. *International Journal for Numerical Methods in Engineering*, 84(13):1589–1612, 2010.

[Lon30] Fritz London. Zur theorie und systematik der molekularkräfte. *Zeitschrift für Physik*, 63(3-4):245–279, 1930.

[Luc77] Leon B Lucy. A numerical approach to the testing of the fission hypothesis. *Astronomical Journal, vol. 82, Dec. 1977, p. 1013-1024.*, 82:1013–1024, 1977.

[Lud08] Stefan Luding. Introduction to discrete element methods: basic of contact force models and how to perform the micro-macro transition to continuum theory. *European journal of environmental and civil engineering*, 12(7-8):785–826, 2008.

[MCG10] OK Mahabadi, BE Cottrell, and G Grasselli. An example of realistic modelling of rock dynamics problems: FEM/DEM simulation of dynamic Brazilian test on Barre granite. *Rock mechanics and rock engineering*, 43:707–716, 2010.

[Mic79] Ronald E Mickens. Long-range Interactions. *Foundations of Physics*, 9(3-4):261–269, 1979.

[Mil13] Robert Andrews Millikan. On the elementary electrical charge and the Avogadro constant. *Physical Review*, 2(2):109, 1913.

[MM89] Jonas Mockus and Jonas Mockus. *The Bayesian approach to local optimization*. Springer, 1989.

[MM22] Ranabir Majumder and Mahitosh Mandal. Screening of plant-based natural compounds as a potential COVID-19 main protease inhibitor: an in silico docking and molecular dynamics simulation approach. *Journal of Biomolecular Structure and Dynamics*, 40(2):696–711, 2022.

[MR99] William Mattson and Betsy M Rice. Near-Neighbor Calculations Using a Modified Cell-Linked List Method. *Computer Physics Communications*, 119(2-3):135–148, 1999.

[NBB+14] Christoph Niethammer, Stefan Becker, Martin Bernreuther, Martin Buchholz, Wolfgang Eckhardt, Alexander Heinecke, Stephan Werth, Hans-Joachim Bungartz, Colin W Glass, Hans Hasse, Jadran Vrabec, and Martin Horsch. ls1 mardyn: The massively parallel molecular dynamics code for large systems. *Journal of Chemical Theory and Computation*, 10(10):4455–4464, 2014.

[New87] Isaac Newton. *Philosophiae naturalis principia mathematica*, volume 1. Edmond Halley, 1687.

[NSS+23] Isabel Nitzke, Rolf Stierle, Simon Stephan, Michael Pfitzner, Joachim Gross, and Jadran Vrabec. Phase equilibria and interface properties of hydrocarbon propellant–oxygen mixtures in the transcritical regime. *Physics of Fluids*, 35(3), 2023.

[NT+19] David B Newell, Eite Tiesinga, et al. The international system of units (SI). *NIST Special Publication*, 330:1–138, 2019.

[NW70] Saul B Needleman and Christian D Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, 48(3):443–453, 1970.

[Nyq28] Harry Nyquist. Certain topics in telegraph transmission theory. *Transactions of the American Institute of Electrical Engineers*, 47(2):617–644, 1928.

[Pau25] Wolfgang Pauli. Über den Zusammenhang des Abschlusses der Elektronengruppen im Atom mit der Komplexstruktur der Spektren. *Zeitschrift für Physik*, 31(1):765–783, 1925.

[PH13] Szilárd Páll and Berk Hess. A flexible algorithm for calculating pair interactions on SIMD architectures. *Computer Physics Communications*, 184(12):2641–2650, 2013.

[Pli95] Steve Plimpton. Fast Parallel Algorithms for Short-Range Molecular Dynamics. *Journal of Computational Physics*, 117(1):1–19, 1995.

[PWE+15] Chunlei Pei, Chuan-Yu Wu, David England, Stephen Byard, Harald Berchtold, and Michael Adams. DEM-CFD modeling of particle systems with long-range electrostatic interactions. *AIChE Journal*, 61(6):1792–1803, 2015.

[PYJ+17] Stefan Pantaleev, Slavina Yordanova, Alvaro Janda, Michele Marigo, and Jin Y Ooi. An experimentally validated DEM study of powder mixing in a paddle blade mixer. *Powder Technology*, 311:287–302, 2017.

[PZB+20] Szilárd Páll, Artem Zhmurov, Paul Bauer, Mark Abraham, Magnus Lundborg, Alan Gray, Berk Hess, and Erik Lindahl. Heterogeneous parallelization and acceleration of molecular dynamics simulations in GROMACS. *The Journal of Chemical Physics*, 153(13), 2020.

[RBMC96] Dennis C. Rapaport, Robin L. Blumberg, Susan R. McKay, and Wolfgang Christian. The Art of Molecular Dynamics Simulation. *Computers in Physics*, 10(5):456–456, 1996.

[Ric76] John R Rice. The Algorithm Selection Problem. In *Advances in Computers*, volume 15, pages 65–118. Elsevier, 1976.

[Rou12] Olivier Roussel. Description of ppfolio (2011). *Proc. SAT Challenge*, page 46, 2012.

[Run01] Carl Runge. Über empirische Funktionen und die Interpolation zwischen äquidistanten Ordinaten. *Zeitschrift für Mathematik und Physik*, 46(224-243):20, 1901.

[Sau20] Sauermann, Sascha. Integration of the C++ Node-Level AutoTuning Library AutoPas in the Large-scale Atomic/Molecular Massively Parallel Simulator (LAMMPS). Master's thesis, Technical University of Munich, Aug 2020.

[Sec21] Steffen Seckler. *Algorithm and Performance Engineering for HPC Particle Simulations*. PhD thesis, Technische Universität München, 2021.

[SG64] Abraham Savitzky and Marcel JE Golay. Smoothing and differentiation of data by simplified least squares procedures. *Analytical chemistry*, 36(8):1627–1639, 1964.

[SG17] Erich Schubert and Michael Gertz. Intrinsic t-Stochastic Neighbor Embedding for Visualization and Outlier Detection: A Remedy Against the Curse of Dimensionality? In *Similarity Search and Applications: 10th International Conference, SISAP 2017, Munich, Germany, October 4-6, 2017, Proceedings 10*, pages 188–203. Springer, 2017.

[SGH+20] Steffen Seckler, Fabio Alexander Gratl, Matthias Heinen, Jadran Vrabec, Hans-Joachim Bungartz, and Philipp Neumann. AutoPas in ls1 mardyn: Massively Parallel Particle Simulations with Node-Level Auto-Tuning. In *Journal of Computational Science*. Elsevier, 2020. submitted.

[SI09] Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer: data race detection in practice. In *Proceedings of the workshop on binary instrumentation and applications*, pages 62–71, 2009.

[SKL07] Harold A Scheraga, Mey Khalili, and Adam Liwo. Protein-folding dynamics: overview of molecular simulation techniques. *Annu. Rev. Phys. Chem.*, 58:57–83, 2007.

[SLBT04] Y Sheng, CJ Lawrence, BJ Briscoe, and Colin Thornton. Numerical studies of uniaxial powder compaction process by 3D DEM. *Engineering Computations*, 21(2/3/4):304–317, 2004.

[SMS+14] Nitin Sukhija, Brandon Malone, Srishti Srivastava, Ioana Banicescu, and Florina M Ciorba. A learning-based selection for portfolio scheduling of scientific applications on heterogeneous computing systems. *Parallel and Cloud Computing*, 3(4):66–81, 2014.

[SNK+13] Adarsh Shekhar, Ken-ichi Nomura, Rajiv K Kalia, Aiichiro Nakano, and Priya Vashishta. Nanobubble collapse on a silica surface in water: Billion-atom reactive molecular dynamics simulations. *Physical review letters*, 111(18):184503, 2013.

[Spr10] Volker Springel. Smoothed Particle Hydrodynamics in Astrophysics. *Annual Review of Astronomy and Astrophysics*, 48:391–430, 2010.

[SRJ+22] Stuart Slattery, Samuel Temple Reeve, Christoph Junghans, Damien Lebrun-Grandié, Robert Bird, Guangye Chen, Shane Fogerty, Yuxing Qiu, Stephan Schulz, Aaron Scheinberg, Austin Isner, Kwitae Chong, Stan Moore, Timothy Germann, James Belak, and Susan Mniszewski. Cabana: A Performance Portable Library for Particle-Based Simulations. *Journal of Open Source Software*, 7(72):4115, 2022.

[SS04] K Shintate and H Sekine. Numerical simulation of hypervelocity impacts of a projectile on laminated composite plate targets by means of improved SPH method. *Composites Part A: Applied Science and Manufacturing*, 35(6):683–692, 2004.

[SS22] Qi Shi and Mikio Sakai. Recent progress on the discrete element method simulations for powder transport systems: A review. *Advanced Powder Technology*, 33(8):103664, 2022.

[Stö07] Carl Störmer. Sur les trajectoires des corpuscules électrisés dans l'espace. Applications à l'aurore boréale et aux perturbations magnétiques. *Radium (Paris)*, 4(1):2–5, 1907.

[Stu08] Student. The probable error of a mean. *Biometrika*, pages 1–25, 1908.

[SW22] Christodoulos Stylianou and Michèle Weiland. Exploiting dynamic sparse matrices for performance portable linear algebra operations. In *2022 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pages 47–57. IEEE, 2022.

[SW23] Christodoulos Stylianou and Michele Weiland. Optimizing Sparse Linear Algebra Through Automatic Format Selection and Machine Learning. In *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 734–743. IEEE, 2023.

[TAB+] A. P. Thompson, H. M. Aktulga, R. Berger, D. S. Bolintineanu, W. M. Brown, P. S. Crozier, P. J. in 't Veld, A. Kohlmeyer, S. G. Moore, T. D. Nguyen, R. Shan, M. J. Stevens, J. Tranchida, C. Trott, and S. J. Plimpton. LAMMPS - a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales.

[Tch20] Nikola Plamenov Tchipev. *Algorithmic and Implementational Optimizations of Molecular Dynamics Simulations for Process Engineering.* PhD thesis, Technical University of Munich, 2020.

[Tom17] Milan Toma. The Emerging Use of SPH In Biomedical Applications. *Significances of Bioengineering & Biosciences*, 1(1):1–4, 2017.

[TSH+18] Nikola Tchipev, Steffen Seckler, Matthias Heinen, Jadran Vrabec, Fabio Gratl, Martin Horsch, Martin Bernreuther, Colin W. Glass, Christoph Niethammer, Nicolay Hammer, Bernd Krischok, Michael Resch, Dieter Kranzlmüller, Hans Hasse, Hans-Joachim Bungartz, and Philipp Neumann. Twetris: Twenty trillion-atom simulation. *The International Journal of High Performance Computing Applications*, page 1094342018819741, 2018.

[TSYN21] Yoshiharu Tsugeno, Mikio Sakai, Sumi Yamazaki, and Takeshi Nishinomiya. DEM simulation for optimal design of powder mixing in a ribbon mixer. *Advanced Powder Technology*, 32(5):1735–1749, 2021.

[TTW19] Anh Tran, Minh Tran, and Yan Wang. Constrained mixed-integer Gaussian mixture Bayesian optimization and its applications in designing fractal and auxetic metamaterials. *Structural and Multidisciplinary Optimization*, 59:2131–2154, 2019.

[TYLT23] Jiyuan Tu, Guan Heng Yeoh, Chaoqun Liu, and Yao Tao. *Computational fluid dynamics: a practical approach.* Elsevier, 2023.

[VCG+15] Mauro Vallati, Lukas Chrpa, Marek Grześ, Thomas Leo McCluskey, Mark Roberts, Scott Sanner, et al. The 2014 international planning competition: Progress and trends. *Ai Magazine*, 36(3):90–98, 2015.

[VdMH08] Laurens Van der Maaten and Geoffrey Hinton. Visualizing data using t-SNE. *Journal of machine learning research*, 9(11), 2008.

[Ver67] Loup Verlet. Computer "Experiments" on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules. *Physical Review*, 159(1):98, 1967.

[vGBE+96] Wilfred F van Gunsteren, SR Billeter, AA Eising, PH Hünenberger, PKHC Krüger, AE Mark, WRP Scott, and IG Tironi. Biomolecular simulation: the GROMOS96 manual and user guide. *Vdf Hochschulverlag AG an der ETH Zürich, Zürich*, 86:1–1044, 1996.

[Vir16] B Bastida Virgili. DELTA debris environment long-term analysis. In *Proceedings of the 6th International Conference on Astrodynamics Tools and Techniques (ICATT)*, 2016.

[VKBP02] Ilpo Vattulainen, Mikko Karttunen, Gerhard Besold, and James M Polson. Integration schemes for dissipative particle dynamics simulations: From softly interacting systems towards hybrid models. *The Journal of chemical physics*, 116(10):3967–3979, 2002.

[VKFH06] Jadran Vrabec, Gaurav Kumar Kedia, Guido Fuchs, and Hans Hasse. Comprehensive study of the vapour–liquid coexistence of the truncated and shifted Lennard–Jones fluid including planar and spherical interface properties. *Molecular physics*, 104(09):1509–1527, 2006.

[VPB23] Vincent A Voelz, Vijay S Pande, and Gregory R Bowman. Folding@home: Achievements from over 20 years of citizen science herald the exascale era. *Biophysical journal*, 122(14):2852–2863, 2023.

[WA20] Peter Wriggers and B Avci. Discrete element methods: basics and applications in engineering. *Modeling in engineering using innovative numerical methods for solids and fluids*, pages 1–30, 2020.

[WHH17] Stephan Werth, Martin Horsch, and Hans Hasse. Molecular simulation of the surface tension of 33 multi-site models for real fluids. *Journal of Molecular Liquids*, 235:126–134, 2017.

[WM97] David H Wolpert and William G Macready. No free lunch theorems for optimization. *IEEE transactions on evolutionary computation*, 1(1):67–82, 1997.

[Wol02] David H Wolpert. The supervised learning no-free-lunch theorems. *Soft computing and industry: Recent applications*, pages 25–42, 2002.

[WP05] Jinpeng Wei and Calton Pu. TOCTTOU Vulnerabilities in UNIX-Style File Systems: An Anatomical Study. In *FAST*, volume 5, pages 12–12, 2005.

[WZ14] Chengping Wu and Leonid V Zhigilei. Microscopic mechanisms of laser spallation and ablation of metal targets from large-scale molecular dynamics simulations. *Applied Physics A*, 114(1):11–32, 2014.

[WZFD22] Tuo Wang, Fengshou Zhang, Jason Furtney, and Branko Damjanac. A review of methods, applications and limitations for incorporating fluid flow in the discrete element method. *Journal of Rock Mechanics and Geotechnical Engineering*, 14(3):1005–1024, 2022.

[XHHLB08] Lin Xu, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. SATzilla: portfolio-based algorithm selection for SAT. *Journal of artificial intelligence research*, 32:565–606, 2008.

[Yas17] Shigeki Yashiro. Application of particle simulation methods to composite materials: a review. *Advanced Composite Materials*, 26(1):1–22, 2017.

[Yok13] Rio Yokota. An FMM based on dual tree traversal for many-core architectures. *Journal of Algorithms & Computational Technology*, 7(3):301–324, 2013.

[Yos90] Haruo Yoshida. Construction of higher order symplectic integrators. *Physics letters A*, 150(5-7):262–268, 1990.

[YOYS14] Shigeki Yashiro, Keiji Ogi, Akinori Yoshimura, and Yoshihisa Sakaida. Characterization of high-velocity impact damage in CFRP laminates: Part II–prediction by smoothed particle hydrodynamics. *Composites Part A: Applied Science and Manufacturing*, 56:308–318, 2014.

[YWLC04] Zhenhua Yao, Jian-Sheng Wang, Gui-Rong Liu, and Min Cheng. Improved neighbor list algorithm in molecular simulations using cell decomposition and data sorting method. *Computer Physics Communications*, 161(1-2):27–35, 2004.

[ZS13] Gordon GD Zhou and QC Sun. Three-dimensional numerical study on flow regimes of dry granular flows by DEM. *Powder Technology*, 239:115–127, 2013.

[ZZW+22] Chi Zhang, Yu-jie Zhu, Dong Wu, Nikolaus A Adams, and Xiangyu Hu. Smoothed particle hydrodynamics: Methodology development and recent achievement. *Journal of Hydrodynamics*, 34(5):767–805, 2022.

A Appendix

A.1 Experiment Setups

A.1.1 List of Machines

All experiments were conducted on of the machines listed in Table A.1.

	atsccs93	CoolMUC2	HSUper
Vendor	Intel	Intel	Intel
Micro Architecture	Comet Lake	Haswell	Ice Lake
CPU	i7-10700	Xeon E5-2697 v3	Xeon Platinum 8360Y
Base Frequency [GHz]	2.90	2.60	2.40
Threads per CPU	8 x 2	14 x 2	36 x 2
CPUs per Node	1	2	2
Memory [GB/Node]	32	64	256
Nodes	1	384	571

Table A.1: Specifications of the hardware platforms used in this thesis. In the row for threads, x 2 refers to hyper-threading.

A.1.2 List of Setups

This section lists the setups for all experiments and studies shown in this thesis. Parameters that were varied throughout the study are substituted with a variable prefixed with $. For the range of the variables see the corresponding figures.

Figure 2.9: Impact of Cell Sorting

Versions: AutoPas 2a21817, atsccs93: gcc 11.3.0 HSUper: gcc 12.1.0.

```
$Machine OMP_NUM_THREADS=1 md-flexible                    \
    --box-length                        40                \
    --cutoff                            2.5               \
    --data-layout                       AoS               \
    --deltaT                            0                 \
    --iterations                        20                \
    --newton3                           enabled           \
    --no-flops                                            \
```

```
    --particle-generator                uniform             \
    --particles-total                   $n                  \
    --traversal                         lc_c08              \
```

Figure 2.10: Time per Iteration for LC, LCR sorted, LCR unsorted

Versions: AutoPas 2a21817, atsccs93: Clang 17.0.0

```
atsccs93 OMP_NUM_THREADS=16 md-flexible                     \
    --box-length                        100                 \
    --container                         $LinkedCells        \
    --cutoff                            2.5                 \
    --data-layout                       AoS                 \
    --deltaT                            0                   \
    --iterations                        100                 \
    --newton3                           enabled             \
    --no-flops                                              \
    --particle-generator                uniform             \
    --particles-total                   1e5                 \
    --traversal                         lc_c08              \
    --verlet-rebuild-frequency          1                   \
    --verlet-skin-radius-per-timestep   0                   \
```

Figure 3.10: Verlet Lists Cells Memory Optimizatioons

Versions: AutoPas db2bcfb, atsccs93: Clang 17.0.0

```
atsccs93 OMP_NUM_THREADS=8 md-flexible                      \
    --box-length                        20                  \
    --container                         all                 \
    --cutoff                            2.5                 \
    --data-layout                       AoS                 \
    --deltaT                            1e-23               \
    --iterations                        100                 \
    --newton3                           enabled             \
    --no-end-config                                         \
    --no-flops                                              \
    --particle-generator                uniform             \
    --particles-total                   40000               \
    --traversal                         vlc_c08             \
    --tuning-samples                    10                  \
    --verlet-rebuild-frequency          10                  \
```

Figure 3.11: Verlet Cluster Lists Memory Optimizatioons

Versions: AutoPas db2bcfb, atsccs93: Clang 17.0.0

```
atsccs93 OMP_NUM_THREADS=1 md-flexible                    \
    --box-length                        30                \
    --container                         VerletClusterLists \
    --cutoff                            2.5               \
    --data-layout                       SoA               \
    --deltaT                            1e-23             \
    --iterations                        12                \
    --newton3                           enabled           \
    --no-end-config                                       \
    --particle-generator                closestPacking    \
    --particle-spacing                  1.2               \
    --traversal                         vcl_c06           \
    --verlet-rebuild-frequency          3                 \
```

Figure 3.12: Linked Cells Speedup Vectorization

Versions: AutoPas 7fb70c8, atsccs93: Clang 17.0.0

```
atsccs93 OMP_NUM_THREADS=1 md-flexible                    \
    --box-length                        25                \
    --container                         LinkedCells       \
    --cutoff                            2.5               \
    --data-layout                       $LAYOUT           \
    --deltaT                            0                 \
    --functor                           $FUNCTOR          \
    --iterations                        10                \
    --newton3                           enabled           \
    --no-end-config                                       \
    --particle-generator                uniform           \
    --particles-total                   200000            \
    --traversal                         lc_sliced         \
    --verlet-rebuild-frequency          1                 \
    --verlet-skin-radius-per-timestep   0                 \
```

Figure 3.16: Energy usage vs time

Versions: AutoPas 3189f72, CoolMUC2: Clang 10.0.1

```
CoolMUC2 OMP_NUM_THREADS=1 md-flexible                    \
    --box-length                        25                \
    --container                         LinkedCells       \
    --cutoff                            2.5               \
    --data-layout                       $LAYOUT           \
    --deltaT                            0                 \
    --functor                           $FUNCTOR          \
```

```
    --iterations                          10              \
    --newton3                             enabled         \
    --no-end-config                                       \
    --particle-generator                  uniform         \
    --particles-total                     200000          \
    --traversal                           lc_sliced       \
    --verlet-rebuild-frequency            1               \
    --verlet-skin-radius-per-timestep     0               \
```

Subsection 4.1.2: Configuration Template

Versions: AutoPas bd22330, HSUper: gcc 13.2.0
Template for the YAML input files:

```
functor                           :  Lennard-Jones (12-6) AVX
container                         :  [LinkedCells, VerletLists,
                                      VerletListsCells, VerletClusterLists]
fastParticlesThrow                :  false
verlet-rebuild-frequency          :  $REBUILD_FREQUENCY
verlet-skin-radius-per-timestep   :  $SKIN
verlet-cluster-size               :  4
selector-strategy                 :  Fastest-Absolute-Value
data-layout                       :  [all]
traversal                         :  [all]
tuning-strategies                 :  []
tuning-metric                     :  time
tuning-interval                   :  5000
tuning-samples                    :  3
tuning-phases                     :  1
newton3                           :  [all]
cutoff                            :  $CUTOFF
box-min                           :  [0, 0, 0]
box-max                           :  $BOX_MAX
cell-size                         :  $CSF
deltaT                            :  0.0
Sites:
  0:
    epsilon                       :  1.
    sigma                         :  1.
    mass                          :  1.
Objects:
  $OBJECT
log-level                         :  info
no-end-config                     :  true
no-progress-bar                   :  true
```

For the values that were used in the placeholder variables see Table 4.1.

A.2 Spinodal Decomposition Configurations

Versions: AutoPas bd22330, HSUper: gcc 13.2.0, intel-oneapi-mpi 2021.12.1

A.2.1 Equilibration

```
functor                          :  Lennard-Jones (12-6) AVX
cutoff                           :  2.5
verlet-skin-radius-per-timestep  :  0.05
verlet-rebuild-frequency         :  20
tuning-strategies                :  slow-config-filter, predictive, rule-based
rule-filename                    :  (path to the rule file from Section A.3)
tuning-metric                    :  time
tuning-interval                  :  5000
tuning-samples                   :  3
deltaT                           :  0.00182367  # = 2fs
iterations                       :  2600000
boundary-type                    :  periodic, periodic, periodic
Sites:
  0:
    epsilon                      :  1.
    sigma                        :  1.
    mass                         :  1.
Objects:
  CubeGrid:
    0:
      particle-type-id           :  0
      particles-per-dimension    :  160, 160, 160
      particle-spacing           :  1.5
      bottomLeftCorner           :  0, 0, 0
      velocity                   :  0, 0, 0
thermostat:
  initialTemperature             :  1.4
  targetTemperature              :  1.4
  deltaTemperature               :  2
  thermostatInterval             :  10
  addBrownianMotion              :  true
vtk-write-frequency              :  100000
vtk-filename                     :  SpinDecEqui
no-end-config                    :  true
no-progress-bar                  :  true
```

```
load-balancer                   :  ALL
load-balancing-interval         :  10000
```

A.2.2 Decomposition

The full decomposition was executed in two runs, with 2e6 iterations each. Basically the checkpoint from the end of this configuration was relaunched again with the same configuration.

```
functor                         :  Lennard-Jones (12-6) AVX
cutoff                          :  2.5
verlet-skin-radius-per-timestep :  0.05
verlet-rebuild-frequency        :  20
tuning-strategies               :  slow-config-filter, predictive, rule-based
rule-filename                   :  (path to the rule file from Section A.3)
tuning-metric                   :  time
tuning-interval                 :  5000
tuning-samples                  :  3
deltaT                          :  0.00182367  # = 2fs
box-min                         :  -0.75, -0.75, -0.75
box-max                         :  239.25, 239.25, 239.25
iterations                      :  2000000
boundary-type                   :  periodic, periodic, periodic
Sites:
  0:
    epsilon                     :  1.
    sigma                       :  1.
    mass                        :  1.
thermostat:
  initialTemperature            :  0.7
  targetTemperature             :  0.7
  deltaTemperature              :  2
  thermostatInterval            :  10
  addBrownianMotion             :  false
vtk-filename                    :  SpinDecDecomp
vtk-write-frequency             :  20000
no-end-config                   :  true
no-progress-bar                 :  true
load-balancer                   :  ALL
load-balancing-interval         :  10000
# Change the checkpoint for the second run
checkpoint                      :  SpinDecEqui_2600000.pvtu
```

A.3 Default Rules File

If not stated otherwise, the following rule file was used for all runs that employ rule-based tuning.

```
# Define some aliases
define_list AllContainers = "DirectSum", "LinkedCells", "
    LinkedCellsReferences", "VarVerletListsAsBuild", "
    VerletClusterLists", "VerletLists", "VerletListsCells", "
    PairwiseVerletLists", "Octree";
define_list LinkedCellsContainer = "LinkedCells", "
    LinkedCellsReferences";
define_list VerletListsContainer = "VerletLists", "
    VerletClusterLists", "VerletListsCells", "
    PairwiseVerletLists", "VarVerletListsAsBuild";

# Make sure overhead of empty cells in the LinkedCells
    container does not destroy performance
# Idea: For each empty cell,the LinkedCells implementation
    has overhead. All containers except VerletClusterLists (
    and DirectSum) use
# this container in their implementation. Thus, if the domain
     has a lot more empty cells than particles, do not use
    them, use VerletClusterLists.
# The overhead consists of at least accessing the std::vector
define maxFactorOfEmptyCellsOverNumParticles = 100.0;
define isDomainExtremelyEmpty = numEmptyCells / numParticles
    > maxFactorOfEmptyCellsOverNumParticles;

# Calculate whether skin size makes VerletLists useless in
    normal scenarios
# Idea: From a theoretical perspective, Verlet Lists only
    have the advantage to save some of the neighbor distance
    calculations compared to LinkedCells.
#       If the skin is too large, we know for sure that no
    neighbor distance calculations are saved.
define PI = 3.1415926;
define LC_NeighborVolume = cutoff * cutoff * cutoff * 27;
define interactionLength = cutoff + skin;
define VL_NeighborVolume = 4.0/3 * PI * interactionLength *
    interactionLength * interactionLength;

define neighborVolumeRel = VL_NeighborVolume /
    LC_NeighborVolume;

# Define magic number when it is surely not worth it to use
    VL over LC in terms of the number of neighbor distance
    calculations
```

```
define maxReasonableNeighborVolumeRel = 0.9;

# This holds only if the domain is not extremely empty. Does
    it also hold with not being extremely empty, but
    numParticlesPerCell << 1?
if neighborVolumeRel > maxReasonableNeighborVolumeRel and not
     isDomainExtremelyEmpty:
     [container="LinkedCells", traversal="lc_c08"] >= [
        container=VerletListsContainer] with same dataLayout,
        newton3;
endif

if numParticles > 1000: [container="LinkedCells"] >= [
    container="DirectSum"]; endif

if numParticles < 100: [container="DirectSum"] >= [container
    ="LinkedCells"]; endif

if avgParticlesPerCell < 6:
     [container="VerletListsCells", dataLayout="AoS"] >= [
        container="VerletListsCells", dataLayout="SoA"] with
        same newton3, traversal, loadEstimator;
endif

[container="VarVerletListsAsBuild", dataLayout="AoS"] >= [
    container="VarVerletListsAsBuild", dataLayout="SoA"] with
    same newton3, traversal;

if avgParticlesPerCell < 100:
     [container="VerletListsCells"] >= [container="
        PairwiseVerletLists"];
endif
```

A.4 Exploding Liquid Configuration

The initial state was generated from input data that can be found in the official ls1 mardyn GitHub repository[1].

Versions: ls1 mardyn c5c81d86a, AutoPas 5ffea5b, HSUper: gcc 13.2.0, intel-oneapi-mpi 2021.12.1.

```
<?xml version='1.0' encoding='UTF-8'?>
<mardyn version="20100525" >
  <refunits type="SI" >
    <length unit="nm">0.1</length>
    <mass unit="u">1</mass>
```

[1] https://github.com/ls1mardyn/ls1-mardyn Accessed: 20.12.2024

```
  <energy unit="K">1</energy>
</refunits>

<simulation type="MD" >
  <integrator type="Leapfrog" >
    <timestep unit="reduced" >0.00182367</timestep>
  </integrator>

  <run>
    <currenttime>0.0</currenttime>
    <production>
      <steps>280000</steps>
    </production>
  </run>

  <ensemble type="NVT">
    <temperature unit="reduced" >1.80</temperature>
    <domain type="box">
      <lx>132.6</lx>
      <ly>591.891</ly>
      <lz>132.6</lz>
    </domain>

    <components>
      <include query="/components/moleculetype">
        ls1-mardyn/examples/ExplodingLiquid/components.xml
      </include>
    </components>


    <phasespacepoint>
      <generator name="MultiObjectGenerator">
        <objectgenerator>
          <filler type="ReplicaFiller">
            <input type="BinaryReader">
              <header>ls1-mardyn/examples/ExplodingLiquid/
                 input.header.xml</header>
              <data>ls1-mardyn/examples/ExplodingLiquid/
                 input.dat</data>
            </input>
          </filler>
          <object type="Cuboid">
            <lower> <x>0</x> <y>280.946</y> <z>0</z> </
               lower>
            <upper> <x>131.531</x> <y>310.946</y> <z>
               131.531</z> </upper>
          </object>
        </objectgenerator>
      </generator>
```

```
    </phasespacepoint>
  </ensemble>

  <algorithm>
    <parallelisation type="GeneralDomainDecomposition">
      <updateFrequency>5000</updateFrequency>
      <timerForLoad>SIMULATION_FORCE_CALCULATION</
          timerForLoad>
      <loadBalancer type="ALL"></loadBalancer>
    </parallelisation>
    <!-- For vanilla replace the whole block with:
      <datastructure type="LinkedCells"/>
    -->
    <datastructure type="AutoPas">
        <tuningStrategy>$TUNING_STRATEGY</tuningStrategy>
        <extrapolationMethod>linear-regression</
            extrapolationMethod>
        <blacklistRange>3</blacklistRange>

        <selectorStrategy>fastest-absolute-value</
            selectorStrategy>
        <tuningInterval>$TUNING_INTERVAL</tuningInterval>
        <tuningSamples>10</tuningSamples>
        <rebuildFrequency>10</rebuildFrequency>
        <skin>0.5</skin>
    </datastructure>

    <cutoffs type="CenterOfMass" >
      <defaultCutoff unit="reduced" >2.5</defaultCutoff>
      <radiusLJ unit="reduced" >2.5</radiusLJ>
    </cutoffs>
    <electrostatic type="ReactionField" >
      <epsilon>1.0e+10</epsilon>
    </electrostatic>

    <thermostats>
      <thermostat type="TemperatureControl">
        <control>
          <start>0</start>
          <frequency>1</frequency>
          <stop>0</stop>
        </control>
        <regions>
          <region>
            <coords>
              <lcx>0.0</lcx> <lcy>0.0</lcy> <lcz>0.0</lcz>
              <ucx>16.4414</ucx> <ucy>16.4414</ucy> <ucz>
                  16.4414</ucz>
            </coords>
```

```
            <target>
              <temperature>1.80</temperature>
              <component>0</component>
            </target>
            <settings>
              <numslabs>1</numslabs>
              <exponent>0.4</exponent>
              <directions>xyz</directions>
            </settings>
            <fileprefix>beta_log</fileprefix>
            <writefreq>100000000</writefreq>
          </region>
        </regions>
      </thermostat>
    </thermostats>
  </algorithm>
  <output/>
 </simulation>
</mardyn>
```

A.5 LAMMPS Lennard-Jones liquid benchmark

Slightly adapted LAMMPS input script taken from the offical website[2]. Changes the size of the simulation and activation of global indices to make the native behavior more similar to that of AutoPas.

```
variable      x index 1
variable      y index 1
variable      z index 1
variable      t index 100

variable      xx equal 5*$x
variable      yy equal 5*$y
variable      zz equal 5*$z

units         lj
atom_style    atomic

atom_modify   map yes sort 0 0

lattice       fcc 0.8442
region        box block 0 ${xx} 0 ${yy} 0 ${zz}
create_box    1 box
create_atoms  1 box
```

[2] `https://www.lammps.org/inputs/in.lj.txt` Accessed: 20.12.2024

```
mass          1 1.0

velocity      all create 1.44 87287 loop geom

pair_style    lj/cut 2.5
pair_coeff    1 1 1.0 1.0 2.5

neighbor      0.3 bin
neigh_modify  delay 0 every 20 check no

fix           1 all nve

thermo        100

run           $t
```

Versions: LAMMPS-AutoPas 301250a, AutoPas c76ff3f, HSUper: gcc 13.2.0.

The script was used with the following commands to launch the different simulation setups:

```
# AutoPas with Full Search tuning
lmp -i in.lj -autopas on log debug -sf autopas -v t 1000 -v x 20 -v y 20 -v z 20
# AutoPas with predictive rules tuning
lmp -i in.lj -autopas on strategies 'slowConfigFilter,predictive-tuning,rulebased' rule_file tuningRules.rule -sf autopas -v t 1000 -v x 20 -v y 20 -v z 20
# AutoPas with only lc_c04
lmp -i in.lj -autopas on notune t lc_c04 c LinkedCells d SoA n enabled estimator none -sf autopas -v t 1000 -v x 20 -v y 20 -v z 20
# AutoPas with only lc_sliced_balanced
lmp -i in.lj -autopas on notune t lc_sliced_balanced c LinkedCells d SoA n enabled estimator none -sf autopas -v t 1000 -v x 20 -v y 20 -v z 20
# Kokkos package
lmp -i in.lj -kokkos on t ${OMP_NUM_THREADS} -sf kk -v t 1000 -v x 20 -v y 20 -v z 20
# OpenMP package
lmp -i in.lj  -sf omp -v t 1000 -v x 20 -v y 20 -v z 20
```

A.6 LADDS Benchmark Simulation

Versions: LADDS 1c6c633, AutoPas b39c1c0, HSUper: gcc 13.2.0.

```yaml
sim:
  logLevel: info # Available levels are off, critical, err, warn, info, debug, trace
  iterations: 1000 # Number of simulation iterations
  referenceTime: 2022-01-01 # calendar day associated with simulation start at iteration 0 (yyyy/mm/dd)
  maxAltitude: 10000 # Maximum satellite altitude above earth core. This number times two is the simulation box length. [km]
  minAltitude: 150 # Everything below this altitude above ground will be considered burning up [km]
  deltaT: 10.0 # [s]
  collisionDistanceFactor: 1.0 # Factor multiplied with the sum of radii to over approximate collision distances.
  evasionTrackingCutoffInKM: 0.1 # Distance at which even evaded conjunctions are tracked (in a separate out file)
  timestepsPerCollisionDetection: 10
  decompositionType: Altitude   # MPI decomposition type. Options: "Altitude" for spherical shells, "RegularGrid" standard cartesian grid.

  prop: # Which propagation model components should be applied
    useKEPComponent: true # Keplerian propagation
    useJ2Component: true # J2 spherical harmonic approximation
    useC22Component: true # C22 spherical harmonic approximation
    useS22Component: true # S22 spherical harmonic approximation
    useSOLComponent: false # Solar gravitational pull
    useLUNComponent: false # Lunar gravitational pull
    useSRPComponent: true # Solar radiation pressure
    useDRAGComponent: true # Atmospheric drag
    coefficientOfDrag: 2.2 # c_D for the drag component used in all objects where no BSTAR is available

  breakup:
    enabled: false # (de-)activate the breakup mechanic
    minLc: 0.01 # minimal characteristic length for generated debris [m]
    enforceMassConservation: true # by default the NASA breakup model does not conserve mass

io:
  csv:
    fileName: initial_population_and_1cm_debris.csv   # input population

```

```
33 autopas:
34   logLevel: info
35   cutoff: 80.0 # Cutoff for autopas force interaction
36   rebuildFrequency: 1 # Number of iterations before internal
       data structure is rebuilt. Increases collision search
       radius!
37   desiredCellsPerDimension: 30 # Desired number of cells per
       dimension
38   tuningMode: false # can be used to obtain good values for
       the following parameters
39   Newton3: "enabled"
40   DataLayout: "AoS"
41   Container: "LinkedCells"
42   Traversal: "lc_c04_HCP"
```

A.7 All AutoPas Algorithm Configurations

This table contains all algorithmic configurations of AutoPas, even those like the `Octree` container and `LinkedCellsReferences`, which are, at the time of writing, highly experimental and under active development and, therefore, not discussed in this thesis. For more information on them, refer to the official documentation of AutoPas.

Table A.2: Overview of All Algorithm Configurations of AutoPas.

#	Container	Traversal	Load Estimator	Data Layout	Newton 3
0	DirectSum	ds_sequential	none	AoS	disabled
1	DirectSum	ds_sequential	none	AoS	enabled
2	DirectSum	ds_sequential	none	SoA	disabled
3	DirectSum	ds_sequential	none	SoA	enabled
4	LinkedCells	lc_c01	none	AoS	disabled
5	LinkedCells	lc_c01	none	SoA	disabled
6	LinkedCells	lc_c01_combined_SoA	none	SoA	disabled
7	LinkedCells	lc_c04	none	AoS	disabled
8	LinkedCells	lc_c04	none	AoS	enabled
9	LinkedCells	lc_c04	none	SoA	disabled
10	LinkedCells	lc_c04	none	SoA	enabled
11	LinkedCells	lc_c04_HCP	none	AoS	disabled
12	LinkedCells	lc_c04_HCP	none	AoS	enabled

Continued on next page

Table A.2: Overview of All Algorithm Configurations of AutoPas. (Continued)

#	Container	Traversal	Load Estimator	Data Layout	Newton 3
13	LinkedCells	lc_c04_HCP	none	SoA	disabled
14	LinkedCells	lc_c04_HCP	none	SoA	enabled
15	LinkedCells	lc_c04_combined_SoA	none	SoA	disabled
16	LinkedCells	lc_c04_combined_SoA	none	SoA	enabled
17	LinkedCells	lc_c08	none	AoS	disabled
18	LinkedCells	lc_c08	none	AoS	enabled
19	LinkedCells	lc_c08	none	SoA	disabled
20	LinkedCells	lc_c08	none	SoA	enabled
21	LinkedCells	lc_c18	none	AoS	disabled
22	LinkedCells	lc_c18	none	AoS	enabled
23	LinkedCells	lc_c18	none	SoA	disabled
24	LinkedCells	lc_c18	none	SoA	enabled
25	LinkedCells	lc_sliced	none	AoS	disabled
26	LinkedCells	lc_sliced	none	AoS	enabled
27	LinkedCells	lc_sliced	none	SoA	disabled
28	LinkedCells	lc_sliced	none	SoA	enabled
29	LinkedCells	lc_sliced_balanced	none	AoS	disabled
30	LinkedCells	lc_sliced_balanced	none	AoS	enabled
31	LinkedCells	lc_sliced_balanced	none	SoA	disabled
32	LinkedCells	lc_sliced_balanced	none	SoA	enabled
33	LinkedCells	lc_sliced_balanced	squared-particles-per-cell	AoS	disabled
34	LinkedCells	lc_sliced_balanced	squared-particles-per-cell	AoS	enabled
35	LinkedCells	lc_sliced_balanced	squared-particles-per-cell	SoA	disabled
36	LinkedCells	lc_sliced_balanced	squared-particles-per-cell	SoA	enabled
37	LinkedCells	lc_sliced_c02	none	AoS	disabled
38	LinkedCells	lc_sliced_c02	none	AoS	enabled
39	LinkedCells	lc_sliced_c02	none	SoA	disabled
40	LinkedCells	lc_sliced_c02	none	SoA	enabled
41	LinkedCellsReferences	lc_c01	none	AoS	disabled
42	LinkedCellsReferences	lc_c01	none	SoA	disabled

Continued on next page

Table A.2: Overview of All Algorithm Configurations of AutoPas. (Continued)

#	Container	Traversal	Load Estimator	Data Layout	Newton 3
43	LinkedCellsReferences	lc_c01_combined_SoA	none	SoA	disabled
44	LinkedCellsReferences	lc_c04	none	AoS	disabled
45	LinkedCellsReferences	lc_c04	none	AoS	enabled
46	LinkedCellsReferences	lc_c04	none	SoA	disabled
47	LinkedCellsReferences	lc_c04	none	SoA	enabled
48	LinkedCellsReferences	lc_c04_HCP	none	AoS	disabled
49	LinkedCellsReferences	lc_c04_HCP	none	AoS	enabled
50	LinkedCellsReferences	lc_c04_HCP	none	SoA	disabled
51	LinkedCellsReferences	lc_c04_HCP	none	SoA	enabled
52	LinkedCellsReferences	lc_c04_combined_SoA	none	SoA	disabled
53	LinkedCellsReferences	lc_c04_combined_SoA	none	SoA	enabled
54	LinkedCellsReferences	lc_c08	none	AoS	disabled
55	LinkedCellsReferences	lc_c08	none	AoS	enabled
56	LinkedCellsReferences	lc_c08	none	SoA	disabled
57	LinkedCellsReferences	lc_c08	none	SoA	enabled
58	LinkedCellsReferences	lc_c18	none	AoS	disabled
59	LinkedCellsReferences	lc_c18	none	AoS	enabled
60	LinkedCellsReferences	lc_c18	none	SoA	disabled
61	LinkedCellsReferences	lc_c18	none	SoA	enabled
62	LinkedCellsReferences	lc_sliced	none	AoS	disabled
63	LinkedCellsReferences	lc_sliced	none	AoS	enabled
64	LinkedCellsReferences	lc_sliced	none	SoA	disabled
65	LinkedCellsReferences	lc_sliced	none	SoA	enabled
66	LinkedCellsReferences	lc_sliced_balanced	none	AoS	disabled
67	LinkedCellsReferences	lc_sliced_balanced	none	AoS	enabled
68	LinkedCellsReferences	lc_sliced_balanced	none	SoA	disabled
69	LinkedCellsReferences	lc_sliced_balanced	none	SoA	enabled
70	LinkedCellsReferences	lc_sliced_c02	none	AoS	disabled
71	LinkedCellsReferences	lc_sliced_c02	none	AoS	enabled
72	LinkedCellsReferences	lc_sliced_c02	none	SoA	disabled

Continued on next page

Table A.2: Overview of All Algorithm Configurations of AutoPas. (Continued)

#	Container	Traversal	Load Estimator	Data Layout	Newton 3
73	LinkedCellsReferences	lc_sliced_c02	none	SoA	enabled
74	VarVerletListsAsBuild	vvl_as_built	none	AoS	disabled
75	VarVerletListsAsBuild	vvl_as_built	none	AoS	enabled
76	VarVerletListsAsBuild	vvl_as_built	none	SoA	disabled
77	VarVerletListsAsBuild	vvl_as_built	none	SoA	enabled
78	VerletClusterLists	vcl_c01_balanced	none	AoS	disabled
79	VerletClusterLists	vcl_c01_balanced	none	SoA	disabled
80	VerletClusterLists	vcl_c06	none	AoS	disabled
81	VerletClusterLists	vcl_c06	none	AoS	enabled
82	VerletClusterLists	vcl_c06	none	SoA	disabled
83	VerletClusterLists	vcl_c06	none	SoA	enabled
84	VerletClusterLists	vcl_cluster_iteration	none	AoS	disabled
85	VerletClusterLists	vcl_cluster_iteration	none	SoA	disabled
86	VerletClusterLists	vcl_sliced	none	AoS	disabled
87	VerletClusterLists	vcl_sliced	none	AoS	enabled
88	VerletClusterLists	vcl_sliced	none	SoA	disabled
89	VerletClusterLists	vcl_sliced	none	SoA	enabled
90	VerletClusterLists	vcl_sliced_balanced	none	AoS	disabled
91	VerletClusterLists	vcl_sliced_balanced	none	AoS	enabled
92	VerletClusterLists	vcl_sliced_balanced	none	SoA	disabled
93	VerletClusterLists	vcl_sliced_balanced	none	SoA	enabled
94	VerletClusterLists	vcl_sliced_balanced	neighbor-list-length	AoS	disabled
95	VerletClusterLists	vcl_sliced_balanced	neighbor-list-length	AoS	enabled
96	VerletClusterLists	vcl_sliced_balanced	neighbor-list-length	SoA	disabled
97	VerletClusterLists	vcl_sliced_balanced	neighbor-list-length	SoA	enabled
98	VerletClusterLists	vcl_sliced_c02	none	AoS	disabled
99	VerletClusterLists	vcl_sliced_c02	none	AoS	enabled
100	VerletClusterLists	vcl_sliced_c02	none	SoA	disabled
101	VerletClusterLists	vcl_sliced_c02	none	SoA	enabled

Continued on next page

Table A.2: Overview of All Algorithm Configurations of AutoPas. (Continued)

#	Container	Traversal	Load Estimator	Data Layout	Newton 3
102	VerletLists	vl_list_iteration	none	AoS	disabled
103	VerletLists	vl_list_iteration	none	SoA	disabled
104	VerletListsCells	vlc_c01	none	AoS	disabled
105	VerletListsCells	vlc_c01	none	SoA	disabled
106	VerletListsCells	vlc_c18	none	AoS	disabled
107	VerletListsCells	vlc_c18	none	AoS	enabled
108	VerletListsCells	vlc_c18	none	SoA	disabled
109	VerletListsCells	vlc_c18	none	SoA	enabled
110	VerletListsCells	vlc_c08	none	AoS	disabled
111	VerletListsCells	vlc_c08	none	AoS	enabled
112	VerletListsCells	vlc_c08	none	SoA	disabled
113	VerletListsCells	vlc_c08	none	SoA	enabled
114	VerletListsCells	vlc_sliced	none	AoS	disabled
115	VerletListsCells	vlc_sliced	none	AoS	enabled
116	VerletListsCells	vlc_sliced	none	SoA	disabled
117	VerletListsCells	vlc_sliced	none	SoA	enabled
118	VerletListsCells	vlc_sliced_balanced	none	AoS	disabled
119	VerletListsCells	vlc_sliced_balanced	none	AoS	enabled
120	VerletListsCells	vlc_sliced_balanced	none	SoA	disabled
121	VerletListsCells	vlc_sliced_balanced	none	SoA	enabled
122	VerletListsCells	vlc_sliced_balanced	squared-particles-per-cell	AoS	disabled
123	VerletListsCells	vlc_sliced_balanced	squared-particles-per-cell	AoS	enabled
124	VerletListsCells	vlc_sliced_balanced	squared-particles-per-cell	SoA	disabled
125	VerletListsCells	vlc_sliced_balanced	squared-particles-per-cell	SoA	enabled
126	VerletListsCells	vlc_sliced_balanced	neighbor-list-length	AoS	disabled
127	VerletListsCells	vlc_sliced_balanced	neighbor-list-length	AoS	enabled
128	VerletListsCells	vlc_sliced_balanced	neighbor-list-length	SoA	disabled
129	VerletListsCells	vlc_sliced_balanced	neighbor-list-length	SoA	enabled
130	VerletListsCells	vlc_sliced_c02	none	AoS	disabled

Continued on next page

Table A.2: Overview of All Algorithm Configurations of AutoPas. (Continued)

#	Container	Traversal	Load Estimator	Data Layout	Newton 3
131	VerletListsCells	vlc_sliced_c02	none	AoS	enabled
132	VerletListsCells	vlc_sliced_c02	none	SoA	disabled
133	VerletListsCells	vlc_sliced_c02	none	SoA	enabled
134	PairwiseVerletLists	vlp_c01	none	AoS	disabled
135	PairwiseVerletLists	vlp_c01	none	SoA	disabled
136	PairwiseVerletLists	vlp_c18	none	AoS	disabled
137	PairwiseVerletLists	vlp_c18	none	AoS	enabled
138	PairwiseVerletLists	vlp_c18	none	SoA	disabled
139	PairwiseVerletLists	vlp_c18	none	SoA	enabled
140	PairwiseVerletLists	vlp_sliced	none	AoS	disabled
141	PairwiseVerletLists	vlp_sliced	none	AoS	enabled
142	PairwiseVerletLists	vlp_sliced	none	SoA	disabled
143	PairwiseVerletLists	vlp_sliced	none	SoA	enabled
144	PairwiseVerletLists	vlp_sliced_balanced	none	AoS	disabled
145	PairwiseVerletLists	vlp_sliced_balanced	none	AoS	enabled
146	PairwiseVerletLists	vlp_sliced_balanced	none	SoA	disabled
147	PairwiseVerletLists	vlp_sliced_balanced	none	SoA	enabled
148	PairwiseVerletLists	vlp_sliced_c02	none	AoS	disabled
149	PairwiseVerletLists	vlp_sliced_c02	none	AoS	enabled
150	PairwiseVerletLists	vlp_sliced_c02	none	SoA	disabled
151	PairwiseVerletLists	vlp_sliced_c02	none	SoA	enabled
152	PairwiseVerletLists	vlp_c08	none	AoS	disabled
153	PairwiseVerletLists	vlp_c08	none	AoS	enabled
154	PairwiseVerletLists	vlp_c08	none	SoA	disabled
155	PairwiseVerletLists	vlp_c08	none	SoA	enabled
156	Octree	ot_c01	none	AoS	disabled
157	Octree	ot_c01	none	SoA	disabled
158	Octree	ot_c18	none	AoS	enabled
159	Octree	ot_c18	none	SoA	enabled