Fakultät für Informatik
Technische Universität München

Algorithm and Performance Engineering for HPC Particle Simulations

Steffen Seckler

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktor der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:

Prof. Dr.-Ing. Matthias Althoff

Prüfende der Dissertation:

1. Prof. Dr. Hans-Joachim Bungartz
2. Prof. Dr. Philipp Neumann
3. Prof. Dr. Jadran Vrabec

Die Dissertation wurde am 04.08.2021 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 27.09.2021 angenommen.

Acknowledgements

This thesis would not have been possible without the support of my friends, colleagues, and family who I want to show my gratitude.

First, I would like to thank Prof. Hans-Joachim Bungartz for his supervision. His insights into many aspects of my thesis project have always been perceived as extremely valuable. Next, I would like to thank Prof. Philipp Neumann for his tight supervision even from a remote location. His persistence was very valuable in writing this thesis and previous papers. I especially want to thank Fabio Gratl and Nikola Tchipev for their assistance and close collaboration. Without them, this project would not have been possible. I further want to thank the entire chair of Scientific Computing and Computational Science for the fun times and heated discussions.

Further, I thank all partners of the TaLPas project and the IPCC, especially Matthias Heinen and Prof. Jadran Vrabec for providing many interesting scenarios.

Last, but not least, I want to thank all of my Bachelor and Master students, Julian Spahl, Andreas Schmelz, Simon Griebel, Sascha Sauermann, Daniel Langer, Tim Bierth, Jan Nguyen, Deniz Candas, Joachim Marin, Jeremy Harisch and Wolf Thieme. Their contribution to my thesis was very valuable.

Abstract

Several fields ranging from the simulation of atoms to galaxy clusters use particle simulations. These simulations are very compute-intensive and often require the use of HPC architectures. N-body codes need to be adapted to provide both a good node-level and multi-node performance, allowing faster executions and bigger simulations, thus decreasing the time to get simulation results or enabling more accurate simulations.

Good load balancing and proper communication schemes are essential at the multi-node level. At the node level, choosing the optimal algorithm for the force calculation, e.g., linked cells or Verlet lists, poses a challenge as their performance depends on the simulated scenario and the used HPC architecture. Performance optimizations for particle simulations are an active area of research – our group focusing on the molecular dynamics code ls1 mardyn. Recently, we concentrated on the vectorization and shared-memory parallelization of the linked cells algorithm. Other codes using Verlet lists, however, still outperformed ls1 mardyn for some scenarios. Additionally, only lightly inhomogeneous scenarios were considered, resulting in bad scalability for strongly inhomogeneous cases.

In this thesis, I employ algorithm and performance engineering to improve ls1 mardyn further. Its load balancing is enhanced and now even works on heterogeneous computing systems. Zonal methods and overlapping communication are employed to lift strong scaling limits. Additionally, I integrated the auto-tuning library AutoPas into ls1 mardyn, which selects the optimal node-level algorithm for the molecular simulation. This even allows choosing a different solver algorithm, e.g., for the force calculation, on every MPI rank. These improvements resulted in speedups of more than 3x for real-world scenarios and up to 130x for benchmark problems.

Zusammenfassung

In verschiedenen Bereichen, von der Simulation von Atomen bis hin zu Galaxienhaufen, werden Teilchensimulationen eingesetzt. Diese Simulationen sind sehr rechenintensiv und erfordern oft den Einsatz von HPC-Architekturen. N-Körper-Codes müssen so angepasst werden, dass sie sowohl eine gute Leistung auf Knotenebene als auch auf mehreren Knoten bieten, was schnellere Ausführungen und größere Simulationen ermöglicht, wodurch die Zeit bis zum Erhalt von Simulationsergebnissen verkürzt oder genauere Simulationen ermöglicht werden.

Eine gute Lastverteilung und geeignete Kommunikationsschemata sind auf der Mehrknotenebene unerlässlich. Auf der Knotenebene stellt die Wahl des optimalen Algorithmus für die Kraftberechnung, z.B. Linked-Cells oder Verlet-Listen, eine Herausforderung dar, da deren Leistung vom simulierten Szenario und der verwendeten HPC-Architektur abhängt. Leistungsoptimierungen für Partikelsimulationen sind ein aktives Forschungsgebiet - unsere Gruppe konzentriert sich hierbei auf den Molekulardynamik-Code ls1 mardyn. Zuletzt haben wir uns auf die Vektorisierung und Shared-Memory-Parallelisierung des Linked-Cells-Algorithmus konzentriert. Andere Codes, die Verlet-Listen verwenden, übertrafen ls1 mardyn jedoch weiterhin für einige Szenarien. Außerdem wurden nur leicht inhomogene Szenarien betrachtet. Für stark inhomogene Szenarien konnten wir jedoch schlechte Skalierbarkeit beobachten.

In dieser Arbeit setze ich Algorithmus- und Performance-Engineering ein, um ls1 mardyn weiter zu verbessern. Seine Lastverteilung wurde verbessert und funktioniert nun auch auf heterogenen Rechensystemen. Zonale Methoden und überlappende Kommunikation werden eingesetzt, um die bisherigen Grenzen in der starken Skalierbarkeit aufzuheben. Zusätzlich habe ich die Auto-Tuning-Bibliothek AutoPas in ls1 mardyn integriert, die den optimalen Algorithmus auf Knotebene für die molekulare Simulation auswählt. Damit ist es sogar möglich, auf jedem MPI-Rank einen anderen Algorithmus, z.B. für die Kraftberechnung, zu wählen. Diese Verbesserungen führten zu Geschwindigkeitssteigerungen von mehr als 3x für reale Szenarien und bis zu 130x für Benchmark-Probleme.

Contents

List of Figures

List of Tables

List of Listings

List of Algorithms

Acronyms

CPU	Central Processing Unit.
GPU	Graphics Processing Unit.
HPC	High Performance Computing.
kdd	k-d decomposition.
MD	molecular dynamics.
MPI	Message Passing Interface.
NIC	network interface controller.
P2P	point-to-point.
PCI	Peripheral Component Interconnect.
PCIe	Peripheral Component Interconnect Express.
RDMA	remote direct memory access.
RMM	Reduced Memory Mode.
sdd	standard domain decomposition.
SPH	smoothed-particle hydrodynamics.

1 Introduction

1.1 Motivation

Particle simulations are used in many different application areas ranging from the smallest systems in which atoms are simulated to simulations of cosmological scale that model entire galaxy clusters. These simulations have in common that they are very compute-intensive and are thus often performed on High Performance Computing (HPC) architectures. The efficient execution on HPC platforms enables faster simulations of a fixed scenario, allows to simulate for longer time frames, and makes it possible to simulate bigger systems. This enables scientists to get results faster or to generate better results that are more accurate than before. Over the years many different particle simulation codes have been developed. These codes have in common that much effort was put into an efficient execution on different supercomputers. This effort typically includes algorithm and performance engineering.

Algorithm engineering, hereby, describes changes in the algorithms themselves. At the start, problems, e.g., performance bottlenecks, are found through the implementation of the algorithm and applying it to one or more problems. Next, a change to the algorithm or a switch of the algorithm is performed and its performance is investigated. Often algorithm engineering is a repeated process and restarts after a problem has been fixed. Examples for algorithm engineering in particle simulations include the different particle sorting and neighbor identification (linked cells, Verlet lists, Verlet cluster lists) algorithms.

Performance engineering is the process of optimizing a specific algorithm for (specific) hardware and is useful if multiple application areas can use the same algorithm. Performance engineering typically encompasses things like vectorization or a GPU implementation. Examples for performance engineering include the specific tuning of the fast Fourier transform to different hardware architectures or the vectorization of different force kernels for particle simulations.

Algorithm and performance engineering often go hand-in-hand because some algorithms perform better for specific hardware components than other algorithms, but the algorithm still has to be optimized. Both algorithm and performance engineering additionally benefit not only the code on which they were applied, but results and findings can be applied to other codes as well.

In this thesis, I focus on enhancements through algorithm and performance engineering that are realized in the code *ls1 mardyn* [1], which can, likewise, be applied to other codes.

1.2 Previous Work

ls1 mardyn has been previously optimized with respect to both node-level and multi-node performance. Hereby, Martin Buchholz focused on improvements to the linked cells algorithm by allowing adaptive refinement of the cells, implemented a first shared-memory parallelization using OpenMP and Intel's TBB, and tested various load balancing techniques, including a k-d tree-based, graph-based, diffusive, and space-filling curve-based methods [2]. Wolfgang Eckhardt optimized the k-d tree-based approach and implemented a sliding window-based approach for the linked cells container, which both reduced the memory footprint and improved the caching behavior [3]. The improvements allowed setting the world record for the MD simulation with the biggest number of particles ($4.125 \cdot 10^{12}$ particles) [4]. Additionally, he implemented a GPU parallelization for *ls1 mardyn*, which did, however, not show satisfactory results, and he tested the integration of two libraries for long-range interactions [3]. Nikola Tchipev further improved the node-level performance of *ls1 mardyn*. He introduced new intrinsics wrappers which allow an efficient vectorization of *ls1 mardyn* on all current Intel CPUs, he further enhanced the shared-memory parallelization by introducing new coloring schemes that reduced the number of needed colors from 18 to eight (resp. four) which allowed very efficient shared-memory parallelization on up to 256 threads. Nikola Tchipev's work further allowed extending the previous world record to $2 \cdot 10^{13}$ particles [5]. He additionally implemented a shared-memory and MPI parallel variant of the fast multipole method [6].

1.3 Outline

The main focus of this work lies in improvements of the multi-node performance of *ls1 mardyn*, especially concerning load balancing for extremely heterogeneous scenarios. I also present how the strong scaling limits for small, dense scenarios can be lifted through zonal methods and how overlapping communication can increase the performance further. Additionally, I demonstrate the integration of *AutoPas*, a novel auto-tuned node-level particle simulation library, into *ls1 mardyn* and present the required adaptions to the load balancing.

The thesis starts with an overview of the technical background, describing the foundations this thesis builds upon (chapter 2). Next, I present the MD code *ls1 mardyn* and its state at the start of this thesis (chapter 3). I continue with the improvements to the multi-node performance (chapter 4) and present detailed results. The thesis proceeds with detailing the integration of the *AutoPas* library into *ls1 mardyn* and describes the necessary interface design of the library (chapter 5). Through the integration of *AutoPas*, we show that auto-tuning can be applied to an efficient, grown-up MD code. Finally, I summarize the work and give an outlook into open questions and possible further improvements (chapter 6).

2 Technical Background

2.1 Particle Simulations

Particle or N-body simulations are performed in various fields of research and include very different scales. While in molecular dynamics these mimic the interactions between atoms or molecules, in astrophysics n-body simulations enable the study of interactions between stars or galaxies. Even though these happen on completely different physical scales, the simulations are mostly identical: In all these fields, the atoms, molecules, or stars are represented as particles with a position r, a velocity v, and further domain-specific properties. The interactions between these particles can mostly be represented by pair-wise interactions, i.e., each particle asserts a force on every other particle and that force depends only on the properties of the two interacting particles. The resulting force F_i on a particle i is then the sum of all forces F_{ij} from interactions with other particles

$$F_i = \sum_j F_{ij}. \tag{2.1}$$

This force changes the momentum p and therefore the trajectory of the particles according to Newton's second law of motion

$$\dot{p} = F = m \cdot a = m \cdot \ddot{r}. \tag{2.2}$$

To apply the resulting acceleration a on the particles without the need for analytical integration, the time is discretized using time steps and the positions and velocities of the particles are integrated numerically.

Depending on the scenarios and the area of application, different potentials and thus forces are used to describe the interactions between the particles.

2.1.1 Molecular Dynamics

Molecular dynamics (MD) simulations are one typical application area for N-body simulations. They help understand the movement of atoms and molecules and are, e.g., applied in chemical engineering, biochemistry, biomedicine, and biophysics. In chemical engineering, MD helps to understand material properties, s.t., transport properties of a fluid or the nucleation can be better understood [7]. In biochemistry, MD is used to research microscopic properties of biomolecules [8] explaining the formation of bonds between different atoms or the workings of enzymes. One application area of biochemistry is biomedical engineering in which MD helps in the development of vaccines, most recently for the SARS-CoV-2 virus [9]. In biophysics, MD can help characterize the structure of proteins through protein folding [10].

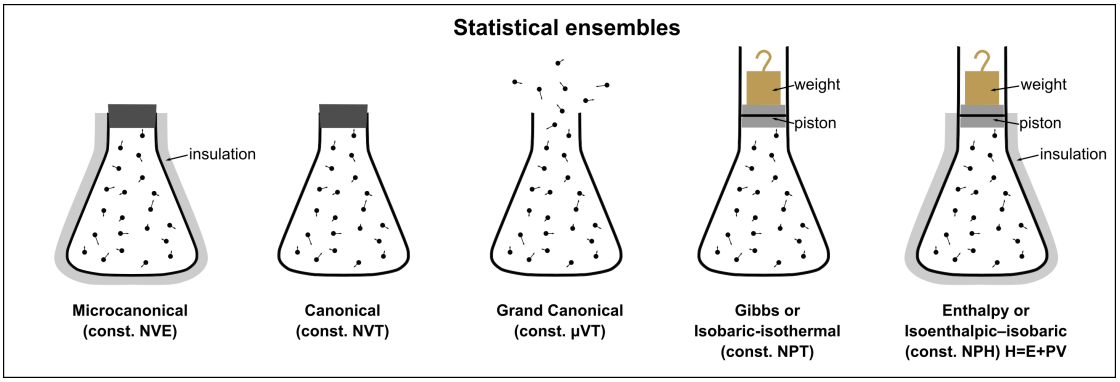


Figure 2.1: Overview of the five different statistical ensembles. Taken from [1]

Ensembles

All these molecular dynamics simulations have in common that an ensemble [11] of particles is simulated. Hereby mainly three different ensembles are used:

Microcanonical Ensemble (NVE) The microcanonical ensemble corresponds to an adiabatic process, i.e., the number of molecules (N), the volume (V), and the energy (E) of the ensemble are preserved, while no energy (heat) can leave the system.

Canonical Ensemble (NVT) The canonical ensemble is used for processes with a fixed number of molecules (N), a fixed volume (V), and a fixed temperature (T). This corresponds to a closed, non-insulated system in a heat bath.

Grand Canonical Ensemble (μVT) The grand canonical ensemble does not use a fixed number of molecules. Instead, along with the volume (V) and the temperature (T), the chemical potential μ is fixed allowing for an exchange of molecules with the environment.

In addition, there exist ensembles with variable volume, namely the Gibbs (isobaric-isothermal ensemble, NPT) and the NPH ensemble. An overview of the different ensembles can be found in Figure 2.1

To realize the different ensembles in a molecular dynamics simulation, multiple tools are necessary. These tools include ways to regulate the temperature of an ensemble, i.e., thermostats, ways to fix a chemical potential, e.g., using particle insertion or deletion, and ways to adapt the volume of a simulation such that the pressure of a system remains constant (barostats). One possible implementation for a thermostat is described in the next section. For the other tools, please see [12, 13].

[1] `https://en.wikipedia.org/wiki/File:Statistical_Ensembles.png`, published under (CC BY-SA 4.0) `https://creativecommons.org/licenses/by-sa/4.0/deed.en`

Thermostats

By making use of the statistical definition of the temperature (Equation 2.3) a thermostat can fix the temperature T of a system of N molecules.

$$\langle E_{\text{kin}} \rangle = \frac{3}{2} N k_{\text{B}} T \qquad , k_{\text{B}} \ldots \text{Boltzmann's constant} \tag{2.3}$$

The simplest thermostat, the velocity-scaling thermostat, directly uses Equation 2.3 and enforces the correct temperature by scaling the velocities v_i of each particle using:

$$v_{i,\text{new}} = \sqrt{\frac{T_{\text{target}}}{T_{\text{current}}}} v_{i,\text{old}}. \tag{2.4}$$

With the definition of the kinetic energy

$$\langle E_{\text{kin,new}} \rangle = \sum_i \frac{1}{2} m_i v_{i,\text{new}}^2 \tag{2.5}$$

and utilizing the scaled velocities

$$= \sum_i \frac{1}{2} m_i \left(v_{i,\text{old}} \sqrt{\frac{T_{\text{target}}}{T_{\text{current}}}} \right)^2 \tag{2.6}$$

$$= \frac{T_{\text{target}}}{T_{\text{current}}} \sum_i \frac{1}{2} m_i v_{i,\text{old}}^2 \tag{2.7}$$

$$= \frac{T_{\text{target}}}{T_{\text{current}}} \frac{3}{2} N k_{\text{B}} T_{\text{current}} \tag{2.8}$$

it is shown that the target temperatur T_{target} is immediately reached:

$$= \frac{3}{2} N k_{\text{B}} T_{\text{target}}. \tag{2.9}$$

This thermostat enforces the temperature, but (especially small) simulations using them do not strictly conform to the canonical ensemble [14]. In addition, when simulating a mixture of multiple molecules, each type of molecule should be thermostated separately to prevent an agglomeration of the kinetic energy in one of the components [15].

An improved version of the velocity scaling thermostat and an overview of other thermostats can be found in [14] and [16].

2.1.2 Particle-Pair Potentials

For molecular dynamics, typical interaction potentials include the Lennard-Jones potential, as well as the forces resulting from Coulomb's law. In astrophysics, mostly Newton's law of universal gravitation is used.

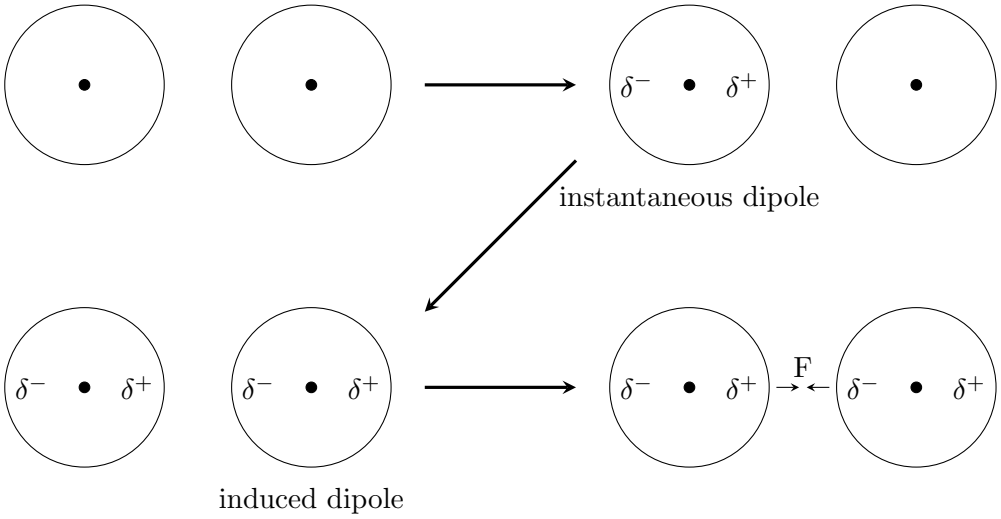


Figure 2.2: London Dispersion Interaction. Top Left: Two atoms with their electron cloud. Top Right: One electron hull fluctuates and thus creates an instantaneous dipole. Bottom Left: This dipole will then induce another dipole on the neighboring atom. Bottom Right: The two atoms are attracted to each other.

Coulomb Force

Coulomb's law [17]

$$F_{\mathrm{C}} = k_e \frac{q_1 q_2}{r^2} \tag{2.10}$$

describes the force between two point charges in vacuum. The force depends on the two point charges q_1 and q_1, the distance r between q_1 and q_2, and Coulomb's constant $k_e \approx 8.987\,55 \cdot 10^9\,\mathrm{N\,m^2/C}$.

Lennard-Jones Potential

The Lennard-Jones potential

$$V_{\mathrm{LJ}} = 4\epsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right] \tag{2.11}$$

describes the interaction of two uncharged atoms [18]. It depends on the potential well ϵ, the atomic cross-section σ, as well as the distance r between the two atoms. The former two are material properties and are experimentally determined.

The Lennard-Jones potential is made of two different terms. The attractive part $V_{\mathrm{attractive}} = -4\epsilon \left(\frac{\sigma}{r}\right)^6$ is a physically accurate term and represents the London dispersion interaction, i.e., the forces between an instantaneous dipole and a second dipole that is induced by the first dipole (cf. Figure 2.2). This interaction is part of the Van der Waals forces. The repulsive portion of the Lennard-Jones potential is an approximation of the Pauli repulsion, which is a phenomenon from quantum chemistry and opposes a too close

Name	Formula	Properties
(explicit) Euler	$v_{i+1} = v_i + a_i \Delta t$ $x_{i+1} = x_i + v_i \Delta t$	first order
Leapfrog	$v_{i+1/2} = v_{i-1/2} + a_i \Delta t$ $x_{i+1} = x_n + v_{i+1/2} \Delta t$	second order, symplectic, time-reversible
Velocity Verlet (Leapfrog kick-drift-kick)	$v_{i+1/2} = v_i + a_i {}^{\Delta t}/_2$ $x_{i+1} = x_i + v_{i+1/2} \Delta t$ $v_{i+1} = v_{i+1/2} + a_{i+1} {}^{\Delta t}/_2$	second order, symplectic, time-reversible

Table 2.1: Time integration methods for N-body simulations.

overlap of the electron hulls of two atoms. The form of this term is not physically driven, but instead chosen for ease of computation.

Newton's law of universal gravitation

Newton's law of universal gravitation

$$F_{\mathrm{G}} = G\frac{m_1 m_2}{r^2} \qquad , G \approx 6.674 \cdot 10^{-11} \, \frac{\mathrm{m}^3}{\mathrm{kg\,s}^2} \ldots \text{gravitational constant} \tag{2.12}$$

describes the gravitational pull between two point masses [19]. The force depends on the masses of the two particles, as well as the distance r between the two particles.

Other interactions

For non-rigid molecules in MD there further exist ways to simulate inner degrees of freedom within a molecule which allow stretching, bending, and torsion of bonds (see, e.g., [20]). These typically are less compute-intensive compared to nonbonded interactions, because they happen between a select number of particles, and are therefore not regarded in this thesis.

Additionally, quantum mechanical simulations can be used to calculate the forces instead of relying on the empirical Lennard-Jones potential. These quantum mechanical simulations are, however more compute-intensive than classical molecular dynamics simulation. To reduce the computational load, there have been approaches that use machine learning to approximate the quantum mechanical simulations for ab initio molecular dynamics [21, 22].

2.1.3 Time Integration

As previously mentioned the time of an N-body simulation is typically discretized, mostly in equally distanced time steps and the positions and velocities of all particles are updated (at least) once in each step. Typical time integration methods are listed in Table 2.1 and used through all different kinds of N-body simulations. Note that in contrast to other applications, like earthquake simulations, N-body simulations normally do not make

use of implicit integrators, as they are too expensive to calculate (note that there are some exceptions [23]). Instead, mostly a variant of the leapfrog or Verlet integration is used [24, 25]. These methods have the benefits of having symplectic properties, i.e., they preserve the energy in a system. The explicit Euler integration does not have this property and therefore is seldomly used within N-body simulations.

2.2 Short Range Algorithms

In particle simulations, the distinction between long-range and short-range potentials is crucial, as the algorithms used for these two different potentials differ greatly. In this section, I first describe the difference between the two classes of potentials and will then introduce the algorithms specific to short-range potentials on which this thesis focuses. Efficient algorithms regarding long-range potentials, e.g., the fast multipole method [26] or the Barnes-Hut method [27] are not discussed in this thesis.

2.2.1 Long-Range vs Short-Range Potentials

While a "long-range potential is one whose range, the distance of effective influence, is unbounded or infinite" [28], a short-range potential's range is bounded. While Mickens uses some more-or-less arbitrary definition for this range [28], Kabadshow uses [29] a more precise definition of such a potential based on the boundedness of the integral

$$I = \int u(r)\mathrm{d}\mathbf{r}. \tag{2.13}$$

Kabadshow defines a potential $u(r)$ to be a long-range potential if it is slowly decreasing with increasing distance r and if the integral I does not converge. A potential is a short-range potential if $u(r)$ decays rapidly with increasing distance r and the integral I converges. Based on this definition, in $\mathbb{R}^n$ long-range potentials decay slower than $1/r^p, p \leq n$. This means that both the Coulomb potential, as well as the gravitational potential, but also dipolar forces, are long-range potentials, while the Lennard-Jones potential is short-ranged.

2.2.2 Cutoff Radius

In general, the calculation of the forces between N particles is an operation that is in $\mathcal{O}\left(n^2\right)$, where $\mathcal{O}$ is the Big O of the Bachmann-Landau notation [30,31]. For short-ranged potentials, it is, however, clear that the interaction between two far-away particles is negligible. Typically, a so-called cutoff radius r_c is introduced, s.t., a potential between two particles is assumed to be zero if the distance between the particles is greater than r_c. For the Lennard-Jones potential the cutoff radius is typically chosen to be between 2.5σ and 3.5σ. The potential is further shifted to prevent discontinuities [12]:

$$V_{\mathrm{LJ,trunc}}(r) = \begin{cases} V_{\mathrm{LJ}}(r) - V_{\mathrm{LJ}}(r_\mathrm{c}) & \text{for } r \leq r_\mathrm{c} \\ 0 & \text{for } r > r_\mathrm{c}. \end{cases} \tag{2.14}$$

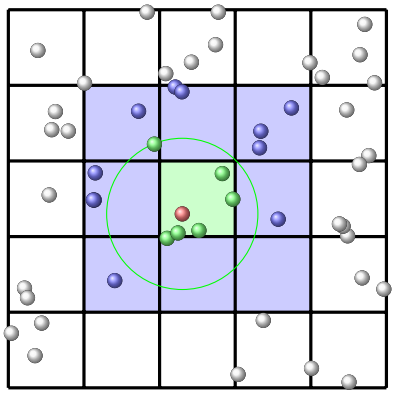


Figure 2.3: Linked cells algorithm: Particles are sorted into cells. To calculate the forces on one particle (here marked red) only the forces of particles in the same cell (green background) or in neighboring cells have to be considered (blue). Particles lying inside of the cutoff are then marked green. Gray particles are not considered for the force calculations of the red particle.

For the adequate calculation of the potential energy, long-range corrections are used [12, 13].

2.2.3 Linked Cells

Using the cutoff radius, only the forces between particles within a distance of at most r_c need to be calculated. This allows for efficient $\mathcal{O}(n)$ algorithms, one of which is the linked cells algorithm [32, 33], where particles are sorted into equally sized cells (c.f. Figure 2.3). Forces between particles then only have to be calculated if the particles lie either in the same cell or if the distance between the two cells is less than the cutoff radius. If the size of each cell is at least the cutoff radius, then only interactions of particles in neighboring cells have to be considered.

2.2.4 Verlet Lists

As can be seen in Figure 2.3 a lot of interactions between particles in neighboring cells are unnecessary as the distance between the particles is bigger than the cutoff radius. The Verlet list algorithm [34] saves on these wasteful distance calculations by maintaining a neighbor list, i.e., a list of all particles within a given distance from each other (c.f. Figure 2.4). This list is typically generated using linked cells. To enable the reusability of the neighbor list over multiple time steps, the list normally does not only include particles within the cutoff radius but instead includes also particles lying slightly outside of the sphere defined by the cutoff radius. This additional distance is normally called the skin radius r_{skin}. For a physically correct result, the list can remain the same, as long as the particles that are not in the list do not move closer than the cutoff radius r_c to each other. This is guaranteed, as long as each particle does not move more than $\frac{r_{\text{skin}}}{2}$, which is assured for a certain time $t_{\text{save}} = \frac{r_{\text{skin}}}{2 \cdot v_{\max}}$ after the last rebuild. Hereby,

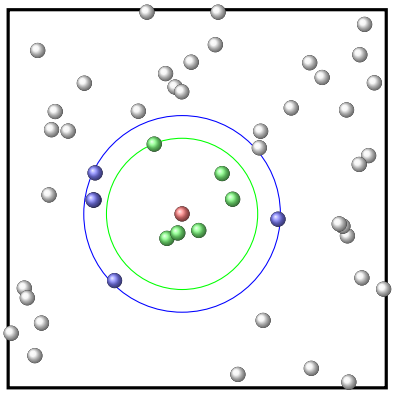


Figure 2.4: Verlet list algorithm: To calculate the forces on one particle (here marked red) only the forces of particles in its neighbor list, i.e., within the blue sphere (circle) with radius $r_c + r_{skin}$ have to be considered. Particles lying inside of the cutoff are then marked green. Gray particles are not considered for the force calculations of the red particle.

$v_{max} = \max_i \max_{\tau \in [t_{rebuild}, t_{rebuild}+t_{save}]} v_i(\tau)$ is the maximal velocity of all particles during the time span.

2.3 HPC Architectures

As particle simulations are very compute-intensive, they are often performed on massively parallel supercomputers [4, 5, 35, 36] to reduce the execution time of specific simulations, which allows to get the results of simulations faster or to simulate longer time spans (assuming a proper parallelization of the code).

Additionally, some simulations can only be executed on big supercomputers as the simulated domains do not fit into the memory of a small computer and the simulation results are too big to store on a conventional storage system [4, 5].

A typical layout of an HPC system consists of many connected nodes, which each host at least one CPU (cf. Figure 2.5).

In this section, I give an overview of typically used hardware and concentrate on aspects relevant to this thesis. For this reason, I will not discuss GPUs in this thesis and also do not go into details about the used storage systems.

2.3.1 CPU Layout

The design of a CPU can be quite diverse. Typically used CPUs do, however, contain multiple compute cores, as well as an (L3) cache that is shared between the cores. The cores additionally include private caches that are smaller in size but faster compared to the shared cache. On each core, there exists a variety of different units that handle distinct tasks. For HPC, especially MD, the most relevant units are the floating-point units (FPU). Traditionally, the FPU is capable of executing a mathematical operation for

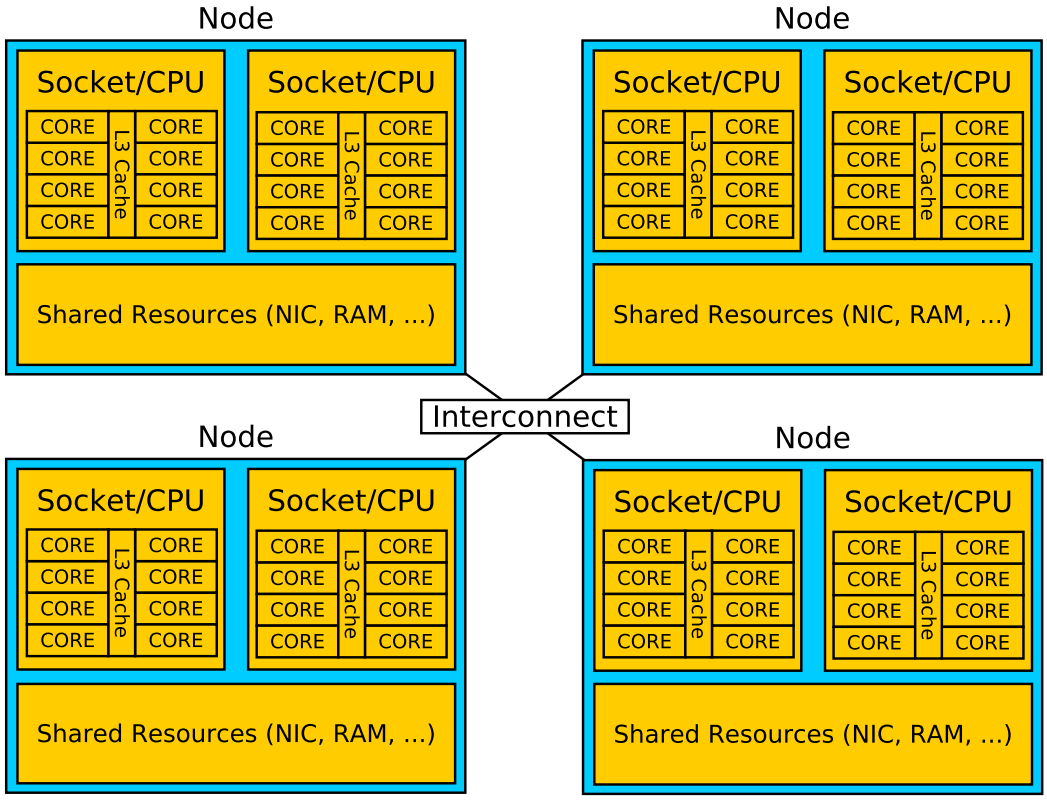


Figure 2.5: Layout of a typical HPC cluster. A typical HPC cluster is made of multiple nodes that are connected by an interconnect. Each of these nodes consists of multiple CPUs that share common ressources like a network interface controller (NIC) or the random-access memory (RAM).

one pair of floating-point values. Nowadays, the FPUs are able to execute these operations on multiple floating-point values at once by the use of SIMD (single instruction multiple data) instructions. Additionally, most FPUs are able to execute fused multiply-add (FMA) operations, where two floating-point values are multiplied and then added to a third value in one operation. FMA instructions are also included in the possible SIMD instructions.

SIMD is, however, not the only concept of parallelism on a core. One other form is instruction-level parallelism in the form of pipelines. Hereby, a single instruction is split into multiple stages (e.g., fetching, execution, write back) which allows the start of an instruction after the first stage of the previous instruction finishes.

For multi-core CPUs another form of parallelism, namely task-level parallelism, is inherent. It describes the execution of multiple processes or threads in parallel and is made possible using the (almost) independently acting cores of a CPU. Recently, task-level parallelism has also found its way into the CPU core using simultaneous multithreading

(SMT, also called hyper-threading). SMT is made possible using superscalar pipelines, which allow the execution of multiple identical pipeline stages in parallel by providing multiple identical execution units (e.g., FPUs or load and store units).

In recent years, Moore's Law, i.e., the doubling of the number of transistors on a microchip every two years, was challenged because the speed at which transistors shrink has significantly slowed down. Instead, vendors tend to increase the size of a CPU by adding more cores to a chip. Intel, e.g., first experimented with their Xeon Phi many-core architecture and added up to 72 cores with four-way hyperthreading (i.e., 288 hardware threads) onto a single chip. While this architecture was discontinued, Intel sells server CPUs with up to 40 cores (80 threads). AMD currently offers CPUs with up to 64 cores (128 threads) and is rumored to add up to 96 cores (192 threads) on the Genoa server CPUs in mid-2022 [2].

2.3.2 Node Layout

A node consists of one or multiple CPUs, which have to share several different resources. These include the main memory (RAM), local storage devices, and the network interface controller.

One important fact regarding the main memory is that it is typically not uniformly accessible from all CPUs of a multi-socket system with the same speed and latency, as each memory slot is typically connected to only one CPU/socket. Access to the memory of another CPU is then still possible, but slower. This effect is called non-uniform memory access (NUMA) and requires care from a user of an HPC system. NUMA effects are also present for the shared LLC (last-level cache) because access from a CPU to the cache of another CPU is slower than access to its cache.

2.3.3 Multi-Node Layout

To enable fast communication between different nodes of an HPC cluster high-performance networks are used. Typical examples of such networks are Intel's OmniPath, Mellanox' (now Nvidia Networking's) Infiniband, Cray's Aries network, and Fujitsu's Torus interconnect. These networks provide lower latencies compared to classical ethernet connections and enable advanced networking techniques such as remote direct memory access (RDMA), where a process of one node can directly access the memory of another node.

The different nodes are often arranged and connected in specific ways that provide short network paths between the nodes, resulting in low latencies. Typically used topologies are fat-trees (e.g., used in SuperMUC-NG) or torus networks (e.g., used in most Cray machines and the Japanese supercomputers K and Fugaku) [37, 38].

[2] `https://twitter.com/ExecuFix/status/1365981401808580614`

2.4 Parallel Programming Models

2.4.1 Flynn's Taxonomy

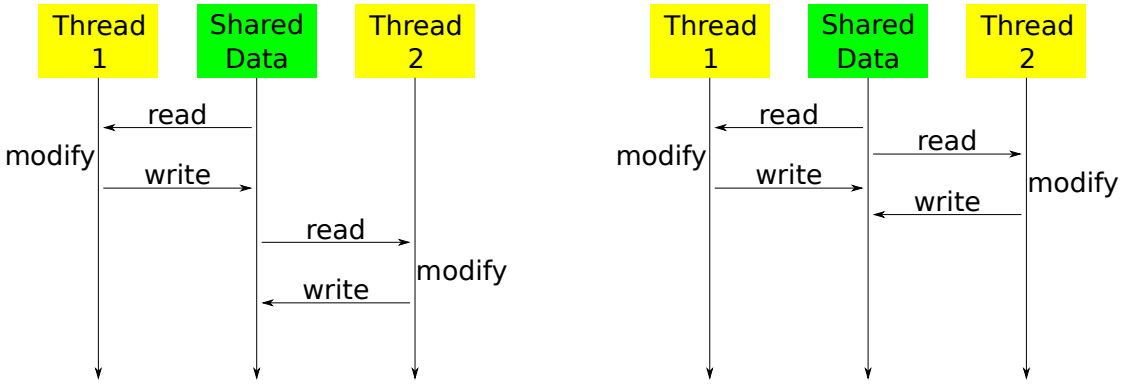
To understand the different parallel programming models, it is first important to understand Flynn's taxonomy [39] which categorizes different HPC architectures. Flynn differentiated based on whether multiple instructions and multiple data were processed in parallel. If both are processed sequentially, he speaks of SISD (single instruction stream, single data stream), which actually corresponds to the von Neumann architecture [40]. SIMD (single instruction stream, multiple data streams) describes architectures in which one instruction is applied to multiple data streams. As previously described, modern HPC machines typically contain multiple SIMD units that are able to perform a single mathematical operation on multiple data using one instruction. Very similar to the SIMD model is the SIMT (single instruction, multiple threads) model, which combines the SIMD idea with multiple threads and is typically used in GPU architectures. A less common architecture is the MISD (multiple instruction streams, single data stream) in which multiple instructions are executed on a single data stream in parallel. This architecture can typically be used for fault-tolerance, e.g., in-flight control computers. Meanwhile, almost any general-purpose CPU implements a MIMD (multiple instruction streams, multiple data streams) architecture, as they are multi-core superscalar processors. Almost all HPC systems are further classified as MIMD, because they are distributed systems (the different nodes have their own memory) in which each processor executes its own instructions on its own data. Thus they execute multiple instructions on different data if viewed as a whole.

In HPC, MIMD is often further divided into SPMD and MPMD. Hereby, SPMD (single program, multiple data streams) represents the more popular one, in which multiple processors execute the same program, but operate on different data. MPMD (multiple program, multiple data streams), meanwhile, refers to a scenario in which multiple different programs are executed that operate on different data. A common example of this is a manager/worker (master/slave) strategy, where one central manager distributes tasks to multiple different workers.

2.4.2 Registry-Level and Shared-Memory Parallelization

To fully leverage the performance of a compute node, i.e., the shared-memory domain, most HPC programs provide support for the SIMD units of a modern CPU. Leveraging SIMD can either be done automatically by the compiler or manually by using intrinsics or inline assembly, through which the SIMD operations are directly specified. On the node-level, shared-memory parallelization (MIMD) in the form of threading is often used to reduce the number of necessary MPI processes, which results in fewer needed messages. Threading can hereby be implemented through a number of means, by leveraging POSIX threads (Pthreads)[3] (other threading libraries) or by using OpenMP [4]. While the former

[3] `https://man7.org/linux/man-pages/man7/pthreads.7.html`
[4] `https://www.openmp.org`



(a) Intended behavior **(b)** Non-intended behavior

Figure 2.6: Example of a race condition. Two threads access a shared variable while both modify them. The modification of thread 1 will not be applied in the right example. If both threads, e.g., increase a zero-initialized shared counter by one, the result on the left is two, while the result on the right is one.

is a simple library, the latter requires code annotation via pragmas and needs to be supported by the compiler. OpenMP does, however, provide easy means of parallelizing existing simulation loops and recently gained the ability to encourage a compiler to vectorize annotated code parts for an efficient use of SIMD units in OpenMP v4.0[5]. OpenMP also gained support for offloading parts of a simulation to a GPU or other accelerators in their recent standards[6]. There additionally exist language standards and libraries that provide possibilities for vectorization and threading support, like OpenCL, Cilk, or Kokkos. These often also allow the efficient utilization of GPUs, which can also be targeted directly by GPU-specific programming languages (e.g., Cuda).

2.4.3 Race Conditions

Race conditions are one of the main problems of shared-memory parallelization [41]. Race conditions can occur if two threads try to access and modify a shared variable. In the intended scenario (cf. Figure 2.6), one thread first reads the variable, then modifies it and writes it back to its address. Afterward, the second thread reads the variable, modifies it, and finally writes it back. Non-intended behavior can occur if the second thread reads the variable before the first thread wrote it back (cf. Figure 2.6). If the final result that is stored in a variable depends on the order in which threads access the variable, one calls this a race condition.

Race conditions can be prevented by the use of locks (mutexes). A mutex can be held by only one thread at a time. It is, therefore, practical to protect a shared variable using a mutex. For this purpose, a thread that wants to access and modify a shared variable, has to lock the mutex. Once it is done, it will unlock the mutex. Afterward, another

[5] `https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf`
[6] `https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-1.pdf`

thread can acquire the lock and modify the variable. If two threads want to access the same variable at the same time, only one thread will be able to lock the mutex. The other thread will have to wait until the lock is released. OpenMP's critical sections are based on locks and offer an easy way to prevent races by allowing only one code to execute a code block.

Another possibility to prevent race conditions is the use of atomics or atomic operations. The latter are operations that cannot be interrupted by concurrent operations and thus guarantee correctness. Atomic objects (or atomics) are objects protected against inconsistencies arising through their parallel access. Atomic objects can be implemented using locks or lock-free atomic CPU instructions. Atomic instructions are implemented in the hardware of a CPU and are thus significantly faster compared to the usage of locks.

There also exist other concepts based on locks or atomics to prevent race conditions. One of those are barriers. A barrier is a point in the program which all threads have to reach before any is allowed to advance beyond that point. Through a barrier, specific code blocks can be separated. An example to prevent race conditions through barriers is allowing access to a shared variable by one thread only before a barrier, while another thread is only allowed to access the variable after the barrier has been passed.

Small examples showing the different methods are represented in Figure 2.7.

2.4.4 Distributed-Memory Parallelization

On HPC systems, out of SPMD and MPMP, the SPMD programming paradigm is prevalent. Hereby, at least one process is started for each node. The processes then use message-based communication to interact, as their memory space is segregated. For this purpose, an implementation of the MPI (message passing interface) is often used.

On the inter-node level, there also exist other programming models besides message passing, which use implicit interaction, e.g., the PGAS (partitioned global address space) model. The PGAS model aims to provide the ease of shared-memory parallelization, in which no messages have to be explicitly sent, for distributed memory systems. Hereby, some memory is made available to all processes through a global address space. Access to that memory requires communication between different processes, which is handled by the runtime or compiler. Additionally, tasking-based models like actor models exist in which tasks and their (data) dependencies are specified and then executed in parallel [42].

A comparison of the different parallel programming models can, e.g., be found in [43].

2.5 MPI

The Message Passing Interface (MPI) is the de facto standard for message-passing in High Performance Computing (HPC). It defines and describes library functions to send messages between different processes of a distributed memory architecture. MPI provides an abstraction of the used networks, such as Infiniband, Intel Omnipath, Cray's interconnects (Aries, Slingshot), Tofu or Ethernet, allowing for portability and ease-of-use.

A first draft of MPI was presented at the Supercomputing conference in November 1993 [44]. A year later in June 1994, the first version of MPI was published [45]. The

```
1 int count{0};
2 auto f = [&count]{++count;};
3 std::thread t1{f}, t2{f};
```

(a) Non-protected code that includes a race condition.

```
1 std::atomic<int> count{0};
2 auto f = [&count]{++count;};
3 std::thread t1{f}, t2{f};
```

(b) Atomic. The shared variable count is protected because it is atomic.

```
1 int count{0};
2 std::mutex m;
3 auto f = [&count]{
4     std::lock_guard lock(m);
5     ++count;
6 };
7 std::thread t1{f}, t2{f};
```

(c) Mutex and lock. When a thread wants to modify the variable `count`, it first acquires the mutex using `std::lock_guard`. Once it is done, `std::lock_guard` goes out of scope and releases the mutex.

```
1  int count{0};
2  // 2 == number of threads.
3  std::barrier b(2);
4  std::thread t1{[&]{
5      ++count;
6      b.arrive_and_wait();
7  }};
8  std::thread t2{[&]{
9      b.arrive_and_wait();
10     ++count;
11 }};
```

(d) Barrier. The second thread can modify and access the shared variable only after the first thread has modified the variable and entered the barrier. (Note: `std::barrier` was introduced in C++20, but a realization with OpenMP is also possible.)

Figure 2.7: Different options to prevent races. Here, two threads access and modify the shared variable `count`. As two threads increment the shared variable by one, one would expect the result to be two. In the unprotected scenario (Figure 2.7a), the result can, however, also be 1.

second version, adding parallel I/O and remote memory operations, followed soon after in 1997 [46]. 2012's MPI-3 adds non-blocking collective operations, as well as additions to the remote memory functionalities [46]. Non-blocking parallel I/O capabilities were added in MPI-3.1 along with neighborhood collectives [46]. The most recent version MPI-4 was approved on June 9, 2021, and includes persistent collective communication, partitioned communication, a new session model among others [46].

The changes from MPI-4, as well as parallel I/O and neighborhood collectives, are not discussed in this thesis. Further, we do not provide details on one-sided communication, as we have tested RDMA in the context of a Master's thesis [47], but did not find it feasible due to the excessive synchronization overhead that is required to guarantee correctness.

2.5.1 P2P Communication

One central building block of MPI is the point-to-point (P2P) communication. For this communication, one pair of communication partners sends one or multiple messages between them.

The simplest form of such a communication is the send-receive pattern (note: do not confuse with `MPI_Sendrecv()`, where two processes execute both a send and a recv operation). While one partner sends a message using `MPI_Send`, the other partner receives it using `MPI_Recv`. An example of this communication pattern is depicted in Listing 2.1. Hereby, the signatures of `MPI_Send` and `MPI_Recv` are

```
int MPI_Send(void* data, int count, MPI_Datatype datatype, int dest,
    int tag, MPI_Comm communicator)
```

and

```
int MPI_Recv(void* data, int count, MPI_Datatype datatype, int src,
    int tag, MPI_Comm communicator, MPI_Status* status).
```

Hereby, it is important to note that an `MPI_Recv` can only receive messages from an `MPI_Send` if the two calls match, i.e., if they both use the same data type, the same number of transmitted elements, the same tag, and the same communicator. Additionally, the destination (dest) of the sending rank has to be the receiving rank and the source (src) of the receiving rank has to be the sending rank. For `MPI_Recv` passing `MPI_ANY_TAG` as tag and `MPI_ANY_SOURCE` as source is possible.

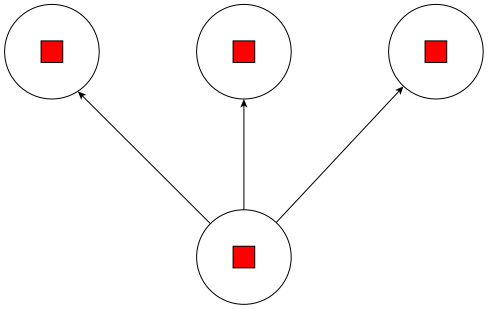
`MPI_Recv` also takes a handle for an `MPI_Status`. It can be used to check for errors. Additionally, it can be used to retrieve the tag and source of a message if `MPI_ANY_TAG` or `MPI_ANY_SOURCE` was used.

In contrast to the source and tag, the size of a message has to be known when issuing an `MPI_Recv`. It is, however, sometimes not possible to know the size of a message before receiving it. To get this size dynamically, `MPI_Probe` can be used. Its signature

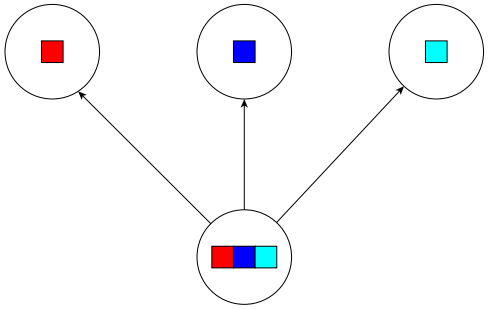
```
int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)
```

```cpp
#include <iostream>
#include <mpi.h>

int main(int argc, char **argv) {
    // Initialize MPI
    MPI_Init(&argc, &argv);

    // Get own rank ID.
    int myrank;
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    // Get number of processes.
    int numprocs;
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    if (numprocs != 2) {
        std::cerr << "Please run this code with two processes!\n";
        return 1;
    }

    if (myrank == 0) {
        // Send 42 from rank 0 to rank 1.
        int myInt = 42;
        MPI_Send(&myInt, 1 /*count*/, MPI_INT, 1 /*dest*/, 1234 /*
            ↪ tag*/, MPI_COMM_WORLD);
    } else if (myrank == 1) {
        int receivedInt;
        // Receive one int from rank 0.
        MPI_Recv(&receivedInt, 1 /*count*/, MPI_INT, 0 /*source*/,
            ↪ 1234 /*tag*/, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        std::cout << "Received int " << receivedInt << " from rank
            ↪ 0.\n";
    }

    MPI_Finalize();
}
```

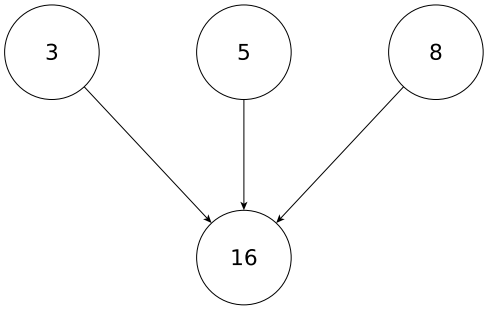
Listing 2.1: MPI Send-Recv example. Rank 0 sends an int with value 42 to rank 1. Rank 1 receives that value and prints it to `std::cout`.



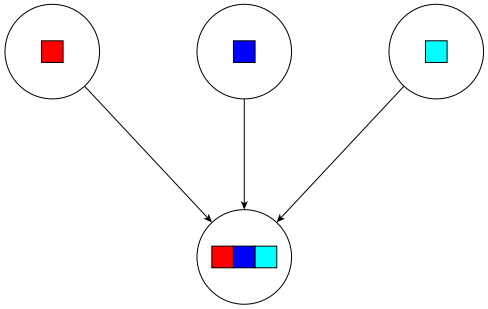
(a) Broadcast. The data from the root note is sent to all ranks.



(b) Scatter. The data from the root note is distributed to all ranks.



(c) Reduce. The data from all ranks is reduced to the root. Here, a sum operation is shown.



(d) Gather. Inverse operation to scatter. The data from each rank is collected at the root node.

Figure 2.8: Overview of typical collective operations in MPI. Here, only operations with a root (bottom) are shown.

is similar to an `MPI_Recv`. It does, however, not require a data pointer, a data type, or a data count as it does not perform the actual receive. `MPI_Probe` blocks until an incoming message matches the source and tag and returns an `MPI_Status` object with information about the incoming message. Using the returned `MPI_Status`, `MPI_Get_count` and the `MPI_Datatype`, the actual number of elements to receive can be reconstructed and an `MPI_Recv` call can be issued.

2.5.2 Collective Communication

In contrast to P2P communication, collectives use all members of an MPI communicator. Typical examples are broadcasts (`MPI_Bcast`), scatters (`MPI_Scatter`), gathers (`MPI_Gather`), and reduction (`MPI_Reduce`) operations (cf. Figure 2.8). Collective MPI operations can roughly be grouped into four categories [48]: All-To-All, One-To-All, All-To-One operations and those that do not fit this scheme (cf. Table 2.2).

[7] `https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report/node98.htm`

Group	Contributing to Result	Receiving Result	Examples

All-To-All	All ranks	All ranks	`MPI_Allgather`
			`MPI_Alltoall`
			`MPI_Allreduce`

All-To-One	All ranks	one rank	`MPI_Gather` `MPI_Reduce`
One-To-All	one rank	all ranks	`MPI_Bcast` `MPI_Scatter`
Other	-	-	`MPI_Scan` `MPI_Barrier`

Table 2.2: Collective MPI operations grouped by the participating ranks [48]. Depending on the source, `MPI_Barrier` can also be viewed as an All-To-All operation[7].

One-To-All (e.g. `MPI_Bcast`) and All-To-One (e.g. `MPI_Reduce`)) operations either have exactly one rank that sends data or exactly one rank that receives data. This rank is called root and specified in the signature of the function call, e.g.,

```
int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype,
    int root, MPI_Comm comm)
```

or

```
int MPI_Reduce(const void *sendbuf, void *recvbuf, int count,
    MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm).
```

Listing 2.2 provides an example using a collective operation to get the global sum of a value.

MPI's collective operations can be implemented using multiple P2P operations. Vendors can, however, provide hand-optimized versions of the collective operations that typically outperform user-written code. These can, e.g., take the topology of a network into account, or work directly on the network interface controller using its RDMA capabilities [49–51]. There additionally exist efforts to offload parts of the MPI collectives directly to the network switches [52, 53].

2.5.3 Non-Blocking Communication

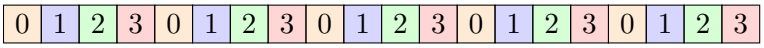
Especially with large numbers of processes and ranks, communication can take up a significant amount of time. One way to circumvent this problem is by overlapping communication with computation. The idea behind this approach is to hide the cost of communication by doing meaningful computational tasks while the communication is taking place. MPI allows this by providing non-blocking variants of their P2P (since MPI v1) and collective operations (since MPI v3). They are easily recognizable by a prefixed `I` to the operation, e.g., `MPI_Isend` instead of `MPI_Send`. The `I` indicates that the call returns immediately and in contrast to the normal variants will never block. Non-blocking MPI

```cpp
#include <iostream>
#include <mpi.h>

int main(int argc, char **argv) {
    // Initialize MPI
    MPI_Init(&argc, &argv);

    // Get own rank ID.
    int myrank;
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    // Get total number of processes.
    int numprocs;
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);

    // Sum all rank-id's together.
    int sumRanks;
    MPI_Reduce(&myrank /*sendbuffer*/, &sumRanks /*recvbuffer*/, 1
        ↪ /*count*/, MPI_INT, MPI_SUM, 0 /*root*/, MPI_COMM_WORLD);

    if (myrank == 0) {
        // Check theoretical result.
        int expectedSum = numprocs * (numprocs - 1) / 2;
        std::cout << (sumRanks == expectedSum ? "successfull" : "
            ↪ wrong") << "\n";
    }

    MPI_Finalize();
}
```

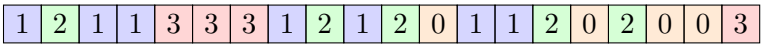
Listing 2.2: Example of a collective MPI operation. Here, a reduction operation is used to sum over the IDs of all ranks. As the root of the collective operation is 0, only the first rank knows about the result of the operation. If all ranks need to know the result, `MPI_Allreduce` should be used instead.

```cpp
#include <iostream>
#include <mpi.h>

int main(int argc, char **argv) {
    // Initialize MPI
    MPI_Init(&argc, &argv);

    // Get own rank ID.
    int myrank;
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    // Get number of processes.
    int numprocs;
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    if (numprocs != 2) {
        std::cerr << "Please run this code with two processes!\n";
        return 1;
    }

    if (myrank == 0) {
        // Send 42 from rank 0 to rank 1.
        int myInt = 42;
        MPI_Request request;
        MPI_Isend(&myInt, 1 /*count*/, MPI_INT, 1 /*dest*/, 1234 /*
            ↪ tag*/, MPI_COMM_WORLD, &request);
        // do some heavy computation
        MPI_Wait(&request, MPI_STATUS_IGNORE);
    } else if (myrank == 1) {
        int receivedInt;
        // Receive one int from rank 0.
        MPI_Request request;
        MPI_Irecv(&receivedInt, 1 /*count*/, MPI_INT, 0 /*source*/,
            ↪ 1234 /*tag*/, MPI_COMM_WORLD, &request);
        // do some heavy computation
        MPI_Wait(&request, MPI_STATUS_IGNORE);
        std::cout << "Received int " << receivedInt << " from rank
            ↪ 0.\n";
    }

    MPI_Finalize();
}
```

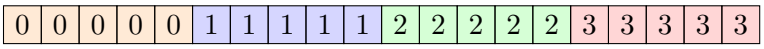
Listing 2.3: MPI Send-Recv non-blocking example. Rank 0 sends an int with value 42 to rank 1. Rank 1 receives that value and prints it to `std::cout`. This corresponds to Listing 2.1, but allows for computation to overlap with the communication.



(a) Round Robin Distribution. The work items are alternately distributed to each compute resource.



(b) Random Distribution. The work items are distributed to a compute resource at random.



(c) Block Distribution. The entirety of all work items is split into blocks. Each block is then assigned to one compute resource.

Figure 2.9: Static load balancing for 20 tasks with equal load onto four compute resources. Each box resembles one work item. Work items with the same color (or number) are assigned to the same compute resource.

calls take a request as an argument. This request can be checked for completion (using `MPI_Test`) or one can wait for communication to finish (using `MPI_Wait`).

A typical pattern of using non-blocking calls is to first start a non-blocking communication (e.g. using `MPI_Isend`), then do some other computation, after which `MPI_Wait` is called (see Listing 2.3).

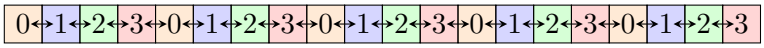
2.6 Load Balancing

Load balancing is an important part of many parallel multi-node simulation programs [54–56]. Load balancing describes the distribution of multiple (sub-)tasks of a program, also called work items, onto multiple compute resources. In general, one differentiates between static and dynamic load balancing.

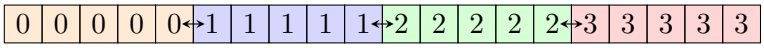
For static load balancing, also referred to as mapping or scheduling problem [57–59], this work distribution is performed only once, at the startup of a simulation. Static load balancing is possible if the load of each (sub-)task is well known. Examples of static mapping techniques are round-robin mappings, randomized mappings, or the distribution of blocks of work (cf. Figure 2.9). Round robin-like approaches are often useful for queues of unknown length [60, 61]. If the size is known, a block distribution can be used and often provides an easier distribution of the values. Random distributions can be useful if the load of the (sub-)tasks varies or no information about them is known.

Dynamic load balancing is required if the load of work items changes over time or is not known a priori [62, 63]. Dynamic load balancing can either be solved using centralized or decentralized approaches. In contrast to centralized approaches, e.g., a central queue, decentralized methods, e.g., work-stealing or diffusive load balancing, provide better scalability as they do not depend on central resources. If the load of each (sub-)task is known, static load balancing techniques can be used in a repeated fashion producing dynamic load balancing.

If one wants to distribute data, one first has to define the work items that are assigned to the different compute resources.



(a) Round Robin Distribution. It requires a lot of communication, as neighboring items always reside on different compute resources.



(b) Block Distribution. Little communication is required as neighboring items mostly reside on the same compute resources.

Figure 2.10: Communication for one dimensional domain partitioning. Here, it is assumed that only information of the neighboring cells/work items is needed. Arrows indicate the necessary information exchange between different compute resources.

2.6.1 Domain Partitioning

Often, a simulation is using spatially ordered data, where data is locally connected, e.g., in terms of stencil operations, force field evaluations, etc. In that case, the domain the data resides on should be distributed to the different compute resources in such a way that close-by data reside on the same compute resource reducing the required communication. In the one dimensional case, this corresponds to using block distributions (cf. Figure 2.10). In the multi-dimensional case, multiple domain partitioning algorithms exist.

2.6.2 Domain Partitioning Algorithms

All domain partitioning (also called spatial decomposition) algorithms have in common that they split the domain into different subdomains, where each compute resource is typically assigned one subdomain.

The easiest domain partitioning algorithm splits the domain into equally sized subdomains using a regular rectilinear grid. It ensures equally sized and formed partitions, but does not provide room for dynamic load balancing.

Other algorithms that can guarantee cubic partitions are bi-partitioning (or bisection) algorithms. They recursively split the domain into two, until enough partitions for the number of compute resources exist. Hereby, they create a (potentially unbalanced) binary tree, where each leaf of the tree corresponds to the subdomain assigned to one compute resource. By moving the partitioning-planes this algorithm can provide good load-balancing.

Instead of allowing only bi-partitions, multi-partitioning methods allow splitting the domain into more than two partitions per step. The multi-section method [64] implements this behavior. They first create a factorization of the total number N of compute resources with d factors (d is the dimension), s.t., $\prod_{i=1}^{d} n_i = N$. The domain is then split into d recursive calls, wherein the i-th call, each partition is split into n_i subpartitions.

Space-filling curve-based [65–67] partitioning methods reduce the multi-dimensional problem into a one-dimensional partitioning problem that is easy to solve [68, 69]. One disadvantage of these methods is the resulting non-cubic subdomains. For this reason, they are not considered in this thesis.

As an alternative to these approaches, it is possible to generalize the partitioning problem to a graph-partitioning problem by discretizing the domain into cells. This allows for relatively easy inclusion of a cost model for the communication. To solve (graph-)partitioning problems, multiple libraries like (Par)METIS[8] or Zoltan[9] can be used.

An overview of the different partitioning methods is shown in Figure 2.11.

The domain partitioning methods differ in their requirements. While the recursive partitioning methods mostly require a priori knowledge of the load, to properly split the domain, both the space-filling curve-based approach and the graph partitioning can work with time measurements of the specific subdomains and work using diffusive methods or using already known loads [70, 71]. A diffusive approach is also possible based on the regular partitioning or the recursive multi-partitioning approach [72]. Hereby, the domain borders are shifted towards the subdomains with a higher load.

Significant differences between diffusive and non-diffusive methods lie in the frequency and the cost of the rebalancing. Diffusive approaches normally have to be performed more often. They are, however, mostly cheaper to calculate as they require fewer communication [71].

A detailed comparison of different domain partitioning methods can, e.g., be found in [73, 74].

2.6.3 Force Decomposition

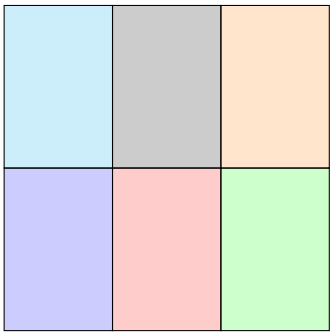
For the force calculation of a particle simulation, instead of interpreting specific subdomains of a simulation as a work item, one can interpret the calculation of the force between each particle pair as one work item. If these are then distributed, one often speaks of force decomposition [75]. This kind of decomposition is favorable if the total number of particles is small or the cutoff large and a high degree of parallelism is needed [76]. Figure 2.12 provides a matrix view of the force decomposition. One disadvantage of force decompositions is the higher amount of communication, scaling with $O\left({}^{N}/_{\sqrt{p}}\right)$, compared to $O\left({}^{N}/_{p}\right)$ for spatial decomposition methods [76], here N is the particle number and p the number of processes.

Domain partitioning algorithms can be interpreted as force decomposition algorithms, where some matrix entries, i.e., force-calculation pairs are calculated multiple times.

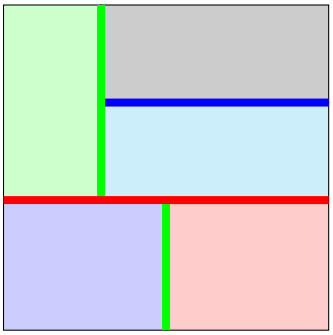
2.6.4 Load Estimation

For simulation domains in which grids are used in combination with a stencil-based algorithm, the domain can be split into subdomains with an equal number of grid elements or cells. This guarantees a good load balance as mostly the cost for the calculation of a subdomain scales linearly with the number of grid elements. Unfortunately, for particle-based simulations, the cost of the simulation of a cell depends on the number of particles inside and around that cell and it is not possible to easily split a domain into

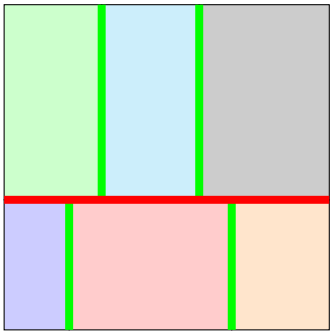
[8] `http://glaros.dtc.umn.edu/gkhome/metis/metis/overview`
[9] `http://www.cs.sandia.gov/Zoltan/`



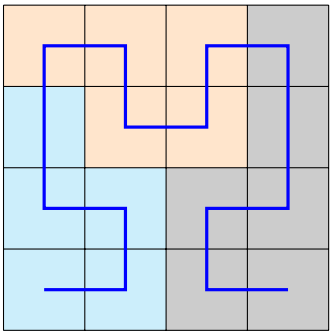
(a) Regular rectilinear Partitioning. The domain is split into equally sized subpartitions using a regular grid consisting of equally sized cuboids.



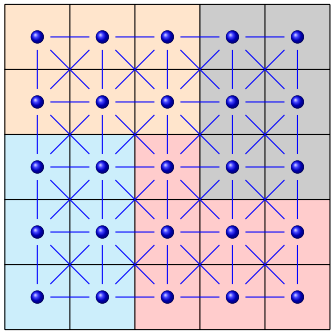
(b) Recursive Bi-Partitioning. The domain is recursively split into two subdomains. Lines of the same color represent the different levels of the splitting.



(c) Recursive Multi-Partitioning. The domain is recursively split into multiple subdomains. Lines of the same color represent the different levels of the splitting. The level of the tree corresponds to the dimension of the domain.

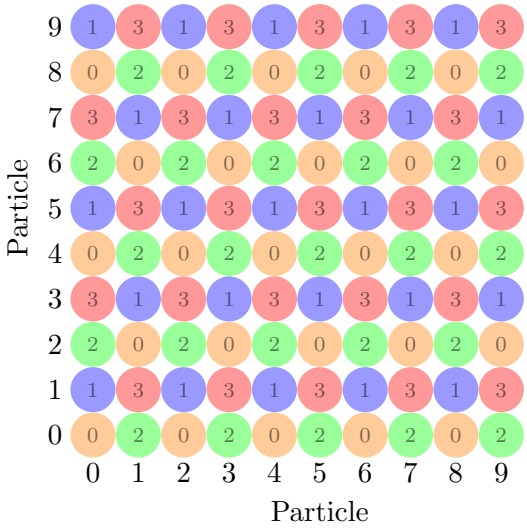


(d) Space-Filling-Curved-based Partitioning. The domain needs to be discretized. Afterwards, a space-filling curve is used to find a good partitioning.

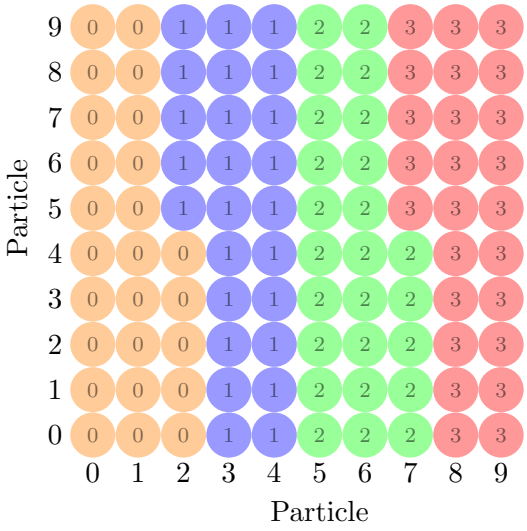


(e) Graph Partitioning. The domain needs to be discretized. Afterwards, graph partitioning algorithms can be used to split the domain. Communication costs (lines) can be incorporated into the splitting.

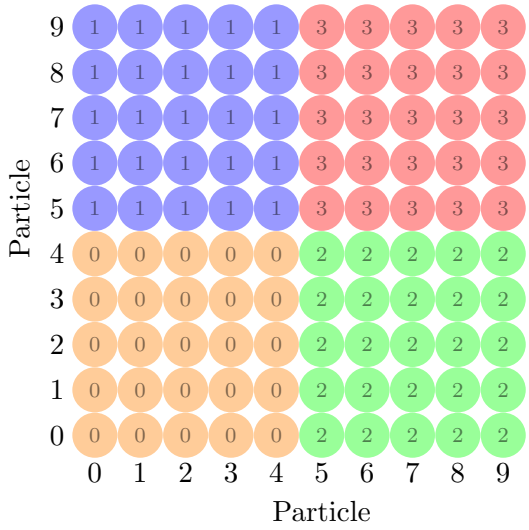
Figure 2.11: Overview of the different domain partitioning algorithms.



(a) Round Robin Distribution.

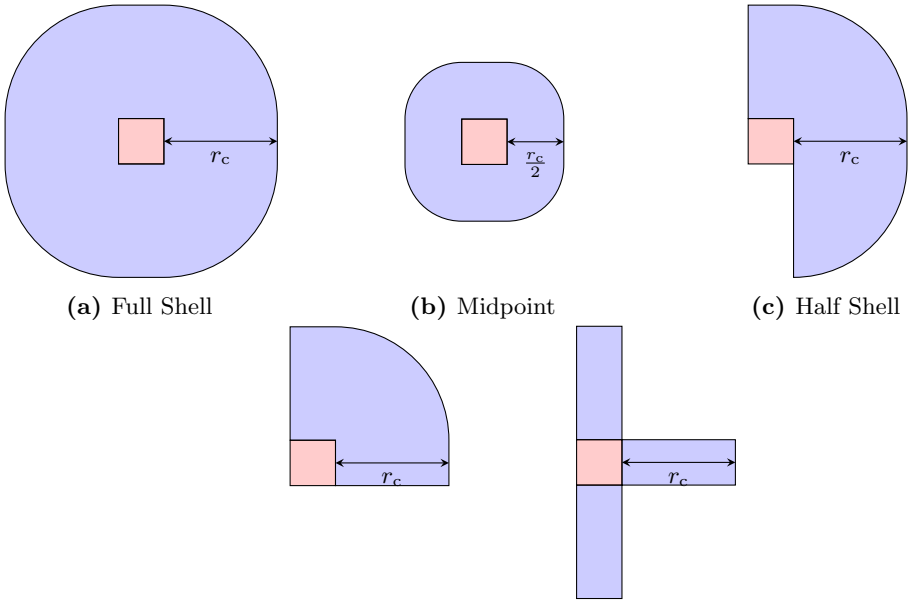


(b) Block Distribution v1.



(c) Block Distribution v2. In contrast to v1, not all compute resources need knowledge of all particles.

Figure 2.12: Matrix view of the force decomposition for 10 particles. A total of 100 interactions are calculated among these particles which are then assigned to four different compute resources.



(a) Full Shell **(b)** Midpoint **(c)** Half Shell

(d) Eighth Shell **(e)** Neutral Territory

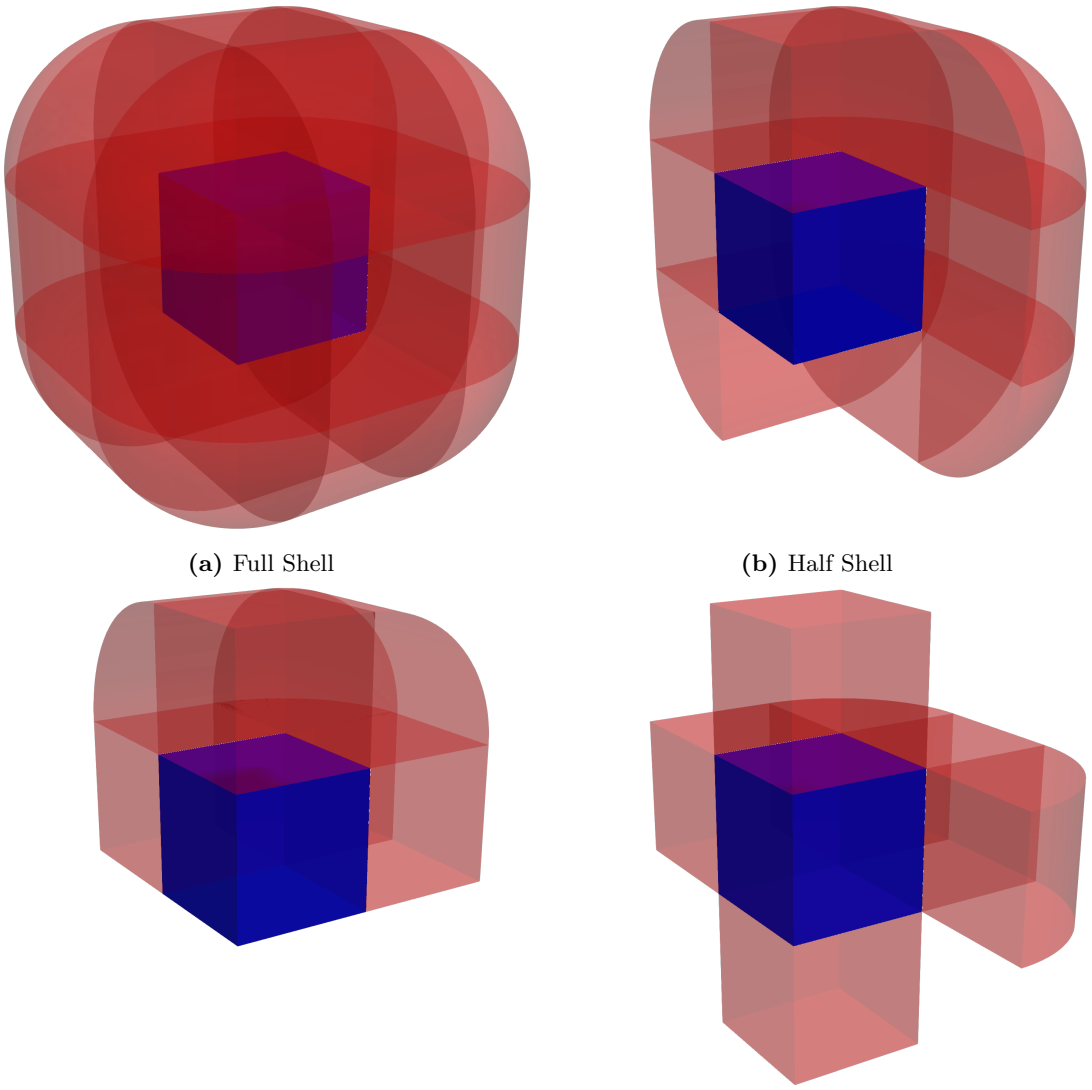
Figure 2.13: Zonal methods in 2d.

partitions based on the number of cells. Instead, the particle density has to be taken into account. For this purpose cost-models can be derived to estimate the cost based on the particle numbers [2, 77]. In addition to the particle number, it is possible to consider the cost of communication for the load balancing, which is relatively straightforward for graph-based partitioners, where it can be specified using the edges of the graph [62]. It is, e.g., cheaper to split a domain within a low-density region than a high-density region, because fewer particles have to be communicated. Additionally, it is better to produce subdomains with a higher volume to surface ratio, because large surface areas imply more communication.

2.6.5 Zonal Methods

Zonal methods [78] represent a mix of spatial and force decomposition algorithms. Like domain partitioning methods, each compute resource is assigned a specific subdomain. But, instead of calculating some particle-pairs multiple times, like in the spatial decomposition methods, zonal methods communicate values of specific force calculations, like force decomposition methods. Zonal methods typically consist of so-called home boxes, i.e., the areas owned by a compute resource, and different imported zones, which make up the import or halo region.

Two-dimensional and three-dimensional overviews of the different zonal methods are presented in Figure 2.13, resp. Figure 2.14. A comparison of the sizes of the halo regions is shown in Table 2.3 and Figure 2.15.



(a) Full Shell

(b) Half Shell

(c) Eighth Shell

(d) Neutral Territory

Figure 2.14: Zonal methods in 3d. Shown in blue is the home box, while the import region is red.

Name	r_I	F	E	C	V_I	$\lim_{a\to 0} V_\mathrm{I}$
Full Shell	r_c	6	12	8	$6a^2r_c + 3\pi ar_c^2 + 4/3\pi r_c^3$	$4/3\pi r_c^3$
Midpoint	$r_c/2$	6	12	8	$3a^2r_c + 3/4\pi ar_c^2 + 1/6\pi r_c^3$	$1/6\pi r_c^3$
Half Shell	r_c	3	6	4	$3a^2r_c + 3/2\pi ar_c^2 + 2/3\pi r_c^3$	$2/3\pi r_c^3$
Eighth Shell	r_c	3	3	1	$3a^2r_c + 3/4\pi ar_c^2 + 1/6\pi r_c^3$	$1/6\pi r_c^3$
Neutral Territory	r_c	4	2	0	$4a^2r_c + 1/2\pi ar_c^2$	0

Table 2.3: Properties of the import regions of zonal methods in 3d. All values are for their three-dimensional version, assuming a side length of a for the home box. The radius of the import region is given by r_I and the volume by V_I. The volumes for one face import (F) is $V_\mathrm{face} = a^2 r_\mathrm{import}$, for an edge import (E) $V_\mathrm{edge} = a\pi r_\mathrm{import}^2/4$ and for each corner import (C) $V_\mathrm{corner} = \pi r_\mathrm{import}^3/6$. See also Figure 2.15. Out of the shown methods, the neutral territory method is the only method with no lower bound for the import volume.

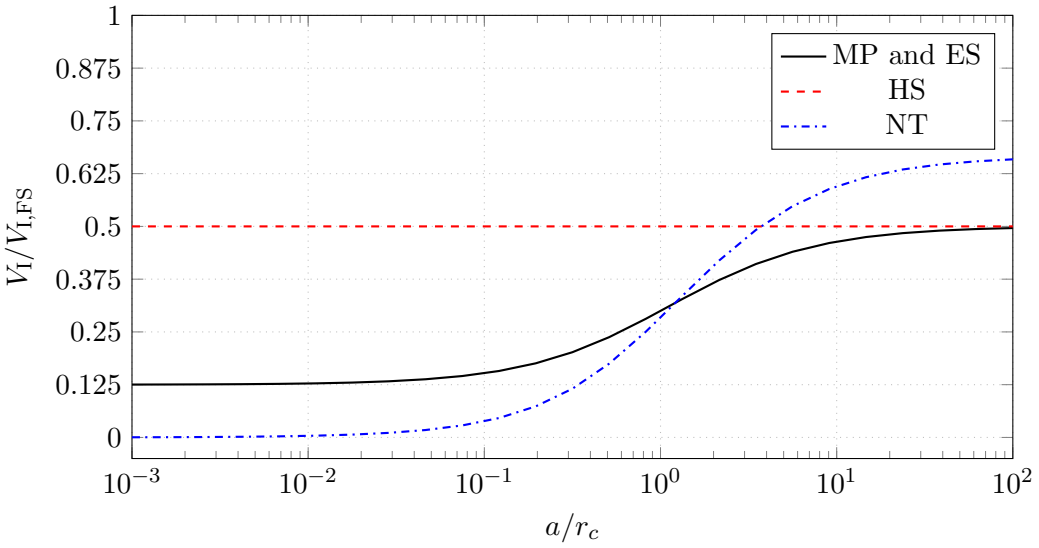
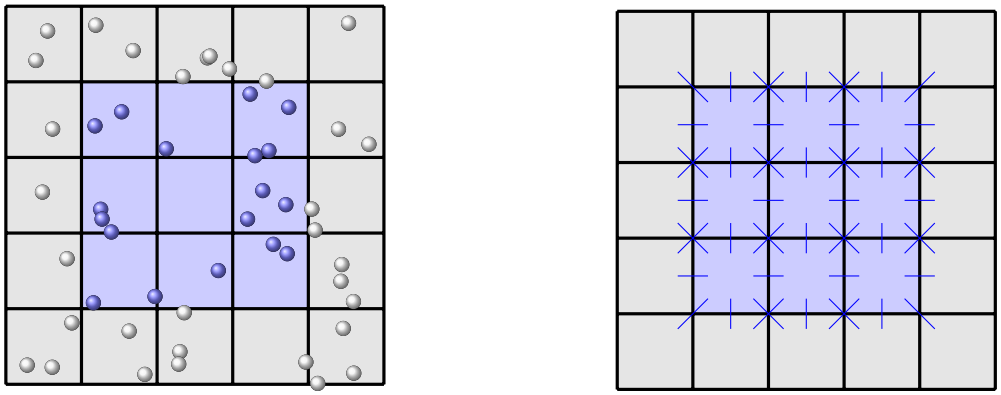


Figure 2.15: Comparison of the sizes of the import regions for different zonal methods. See also Table 2.3. For relative box sizes a/r_c larger than $\pi/8 + \sqrt{9\pi^2 + 96\pi}/24 \approx 1.216$ the midpoint and eighth shell method provide the smallest import region. For smaller box sizes, the neutral territory method provides the smallest import region.



(a) Particles in a linked cells algorithm. Ghost cells (gray) lie outside of the domain of a process (blue)

(b) Necessary force calculations (blue lines). The forces between halo cells do not need to be calculated.

Figure 2.16: Visualization of the full shell method. In a typical MD simulation the forces between particles only need to be calculated if at least one particle is not a ghost particle.

Full Shell The full shell method (cf. Figure 2.16) is another name for the normal spatial decomposition method, where no forces are communicated between different compute resources. For a correct force calculation, all forces of owned and halo particles thus need to be calculated. The home box is thus interacted with all halo zones. The import region of this method is always bigger than a sphere whose radius is the cutoff.

Midpoint Method Using the midpoint method [79], the compute resource an interaction is calculated on is decided by calculating the midpoint of said interaction. It is then calculated by the compute resource that owns the midpoint. Because particles that are interacted on one compute resource have to lie within $r_{\text{cutoff}}/2$ of its subdomain, the necessary import volume is reduced. Instead of calculating the actual midpoint for each interaction, it is possible to use a pseudo-midpoint method by interacting only specific cell-pairs [79]. This, however, requires equally sized cells throughout the entire simulation domain of all compute resources. The size of the import region of the midpoint method is one-eighth of the full shell method (cf. Table 2.3).

Half Shell As the name indicates, the half shell method's import region is halved compared to the full shell method. Using the half shell method, interactions between two zones are calculated inside of the compute resource whose home box has the lower x coordinate (or y coordinate if the x coordinate is equal, or z coordinate if both x and y coordinates are equal) [78]. Alternatively one can define a three-dimensional index I

$$I = i_x \cdot N_y \cdot N_z + i_y \cdot N_z + i_z \tag{2.15}$$

that is calculated based on the one-dimensional indices $i_x, i_y and i_z$ of the grid of boxes (with N_x, N_y, N_z boxes in the respective dimensions). The interaction then takes place on the compute resource with the lower three-dimensional index I.

Eighth Shell While the full shell and half shell method only assign an interaction to a specific compute resource if at least one of the interacting particles lies in its home box, the eighth shell method also assigns interactions where both particles do not lie inside of the home box of a process. In the ES method, the compute resource owning the position, which is given by the element-wise minimum of the two interacting particle positions, calculates an interaction. In 3d, the import region of this method equals an eighth of the import region of the full shell method and resembles one corner of a sphere.

Neutral Territory As all previously mentioned methods' import regions are at least an eighth of a cutoff sphere, the neutral territory methods were introduced in [80] that does not have this restriction and, instead, has no lower limit. The neutral territory method takes the base idea of the eighth shell a step further – most performed pairwise interactions are calculated on a compute resource on which neither particle resides in its home box.

For determining on which compute resource the computation of an interaction takes place, each pair of particles is categorized in one tower and one base atom:

1. First, if the x coordinates of the particles' home boxes differ, the one lying in the lower box is the tower atom.
2. Otherwise, if the y coordinates of the home boxes differ, the one lying in the lower (in y-direction) box is the tower atom.
3. If x and y coordinate of the home boxes are identical and the z coordinates are different, the one in the lower (in z-direction) box is the plate atom.
4. If all three coordinates are the same, choosing the categorization is unimportant.

An interaction of tower and plate atoms then takes place in the compute resource with x and y coordinates of the tower and z coordinate of the plate atom.

3 ls1 mardyn — An Overview

The massively parallel molecular dynamics code *ls1 mardyn*[1,2] is capable of simulating very large systems of small rigid molecules on HPC clusters [1, 4, 5].

ls1 mardyn allows us to simulate single or multi-centered molecules that consist of one or multiple sites (Lennard-Jones, charge, dipole, or quadrupole). In addition, support for the Mie potential [81] (a generalized version of the Lennard-Jones potential) has been added recently [82].

The simulation of various processes and scenarios from chemical and process engineering is possible using *ls1 mardyn*: The program helps understanding the formation of gas bubbles and other nucleation processes [83–89]. Furthermore, *ls1 mardyn* is used to better describe the surface tension of fluids [84, 90–96]. *ls1 mardyn* can simulate adsorption processes [97, 98] and the flow through nanoporous membranes [99]. The program is also used to develop novel long-range correction schemes, e.g., for planar interfaces [94, 100]. In addition, the *MaMiCo* tool was used to couple *ls1 mardyn* with mesh-based fluid dynamics solvers to allow multi-scale simulations [101]. Recently, support for the long-range parts of the Coulomb potential using the fast multipole method has been added [6].

3.1 Algorithms, Parallelization and Optimizations

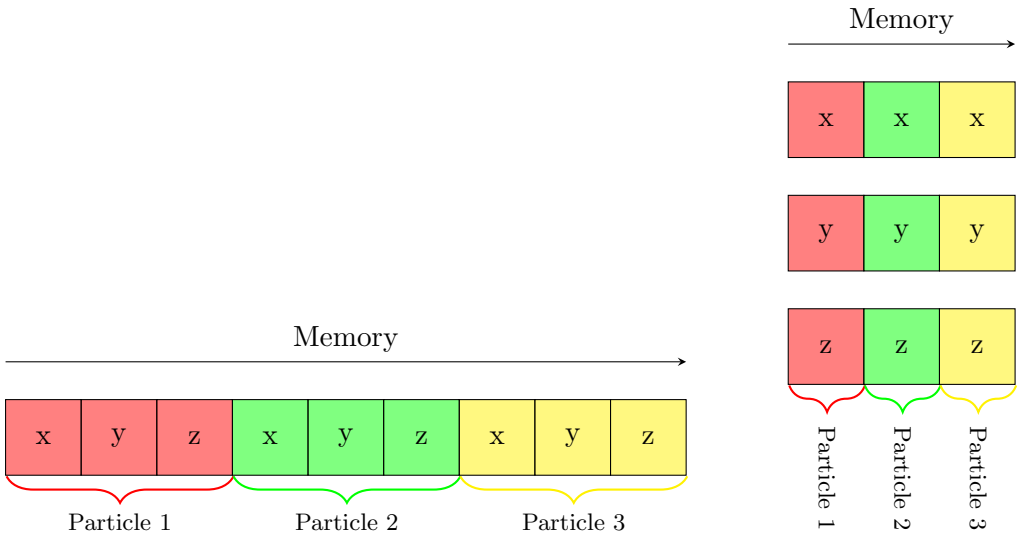
To provide good performance and reasonable time-to-solutions for this large variety of scenarios *ls1 mardyn* employs efficient algorithms, which are tuned to modern HPC architectures [1, 4–6, 102]. In comparison to other molecular dynamics codes, *ls1 mardyn* does not use Verlet lists, but uses linked cells (c.f. subsection 2.2.3) as the underlying data structure [1]. Since linked cells do not store any neighbor lists, they have a significantly reduced memory footprint compared to Verlet lists. Using the linked cells algorithm, *ls1 mardyn* holds the world record of the largest molecular dynamics simulation since 2013 [4] with a total of four trillion particles. In 2019 we broke this record, again using *ls1 mardyn*, and were able to simulate a total of twenty trillion atoms [5]. The optimizations, to enable these and other simulations, can be split into node-level and multi-node-level optimizations.

3.1.1 Node-Level Optimizations

For an in-depth view of the node-level optimizations, please refer to [6], which discusses them in great detail.

[1] Abbreviation for *large systems 1: molecular dynamics*.

[2] `https://www.ls1-mardyn.de/`



(a) Array of Structures (AoS) **(b)** Structure of Arrays (SoA)

Figure 3.1: Data Layouts: The AoS data layout stores each particle as a structure (e.g., a C++ class). Multiple of these structures are then stored in an array (e.g., a C++ vector). The SoA data layout does not store particles as structures. Instead, the information a particle contains is split up into multiple arrays. A combination of these arrays into one structure is then called SoA.

3.1.1.1 Scalar-Level Optimizations

Data Structures *ls1 mardyn* stores the cells of the linked cells algorithm within the so-called LinkedCells container. Each of these cells contains a C++-vector of molecules which are stored in an Array of Structures (AoS) format (cf. Figure 3.1). Storing the molecules directly within the cells guarantees data locality when iterating through the particles of a cell. *ls1 mardyn* also maintains an SoA storage of the molecules for each cell. It is needed for an easier vectorization of the force kernel as it provides non-strided access to memory.

Precision *ls1 mardyn* is capable of simulating molecular dynamics scenarios in different floating-point precisions. By default, it uses double precision, i.e., 64-bit for all floating-point values. In [5] support for single precision (32-bit) floating-point values was added, reducing the memory requirements by 50%. Using this lower floating-point precision also for the accumulation of forces and energies can lead to significant precision loss of the overall simulation [103,104]. As a compromise mixed-precision can be used which provides speed-ups without too much loss in precision [103, 104]. Using mixed-precision, particle properties, e.g., position and velocity, are stored in single-precision, while accumulated values, e.g., the force acting on a particle, are stored in double-precision. Support for mixed-precision was previously added to *ls1 mardyn* [6].

SoA-only Storage For the world record from 2019, we implemented further improvements to memory consumption. This included a new reduced memory mode (RMM) for simulations of single-centered Lennard-Jones molecules. Using RMM, *ls1 mardyn* does not use AoS storage. Instead, it saves all information of a particle inside of SoAs. This reduces the memory footprint by roughly 50% compared to maintaining both AoS and SoA storage. Maintaining both further requires synchronization overhead between them. The existence of only one storage eliminates this overhead.

3.1.1.2 Vector-Level Parallelization

The SoA storage enables sequential memory access patterns which is a requirement for good vectorization. We use self-written intrinsics wrappers for vectorization. In comparison to auto-vectorization or pragma-based vectorization, these need slightly more implementation efforts. They do, however, guarantee good vectorization. Our intrinsics wrappers support all current x86_64 architectures, i.e., they include support SSE3, AVX, AVX2, AVX512. As of the writing of this thesis, they do not support ARM intrinsics.

3.1.1.3 Shared-Memory Parallelization

ls1 mardyn supports multiple parallelization models for shared memory. It mostly uses OpenMP for shared-memory parallelism [5, 6, 102]. In addition, [6, 105] describe tests using the QuickSched library to enable task-based parallelism. In the following, we shortly describe the parallelization using OpenMP.

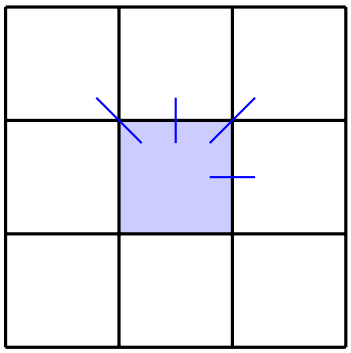
ls1 mardyn uses the Newton3 optimization, i.e., forces between two particles are only calculated once and applied to both particles. Therefore, if two threads calculate forces on the same particle at the same time, race conditions would occur. To prevent race conditions, *ls1 mardyn* uses coloring schemes at the cell-level by employing barriers (cf. subsection 2.4.3) and supports schemes with eight (c08) or four colors (c04) [5, 102] (cf. Figure 3.2). In addition, support for a lock-based traversal has been added which uses a one-dimensional splitting of the domain into equally sized slices [5]. Based on this partitioning, we will henceforth call it sliced traversal. In comparison to the c08 and c04 traversals, the sliced traversal does not provide any load balancing but offers less scheduling overhead. One should therefore only use it for homogeneous scenarios.

3.1.2 Distributed-Memory Parallelization

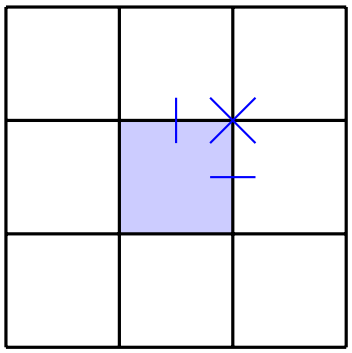
In addition to the shared-memory parallelization, *ls1 mardyn* uses domain partitioning schemes for distributed memory parallelism.

3.1.2.1 Standard Domain Decomposition

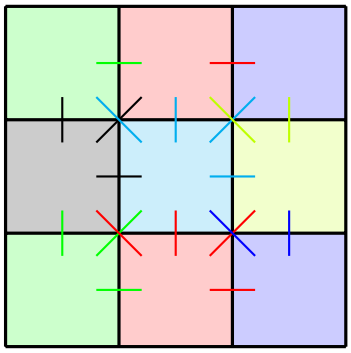
The standard domain decomposition (sdd) uses a regular rectilinear mesh to distribute the domain to the different processes (cf. Figure 3.3). To get a proper distribution of the processes, *ls1 mardyn* uses MPI's `MPI_Dims_create` in combination with `MPI_Cart_create`. `MPI_Dims_create` first generates a three-dimensional division of N processes into a Cartesian



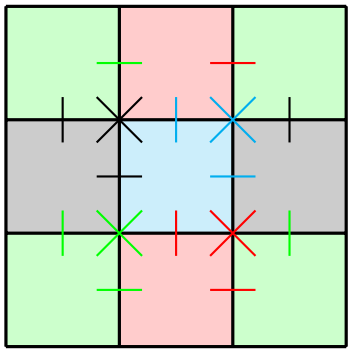
(a) 2-d c06 stencil for one cell.



(b) 2-d c04 stencil for one cell.

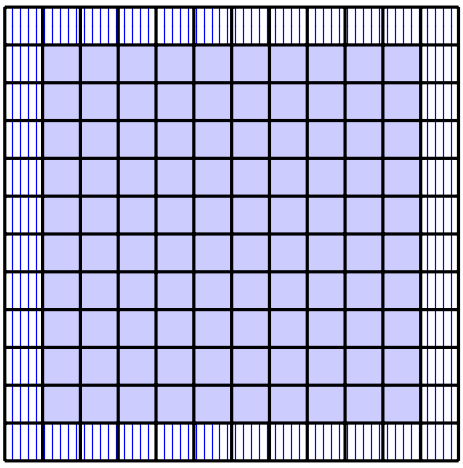


(c) 2-d c06 stencil over multiple cells.

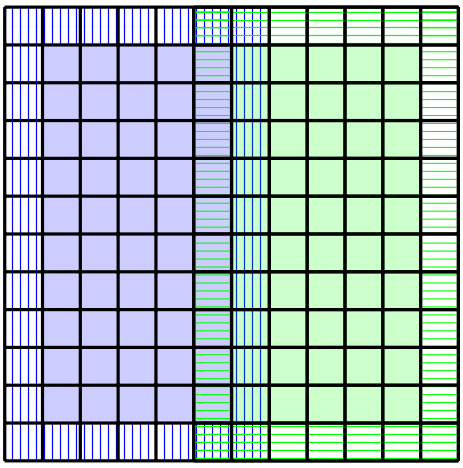


(d) 2-d c04 stencil over multiple cells.

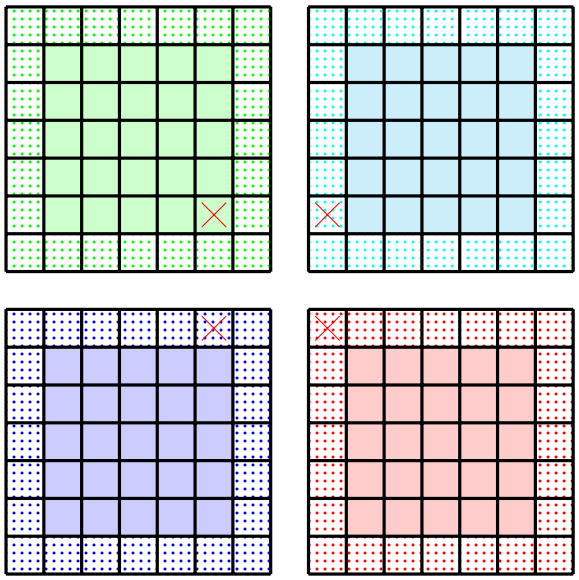
Figure 3.2: Comparison of the 2-dimensional stencils for one cell (top) and multiple cells (bottom). The 2-d c06 stencil corresponds to the 3-d c18 stencil. The 2-d c04 stencil corresponds to the 3-d c08 stencil. We define the so-called base-step as the calculation of the interactions within a base-cell and with cells connected by the same color as the base-cell. This base-step is performed by one thread. The base-steps of cells with the same color can be executed in parallel.



(a) Domain using one process.

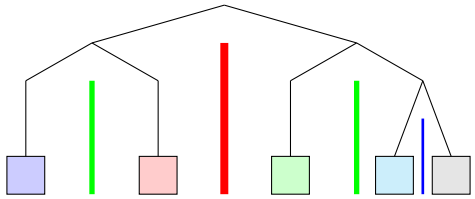


(b) Domain using two processes.

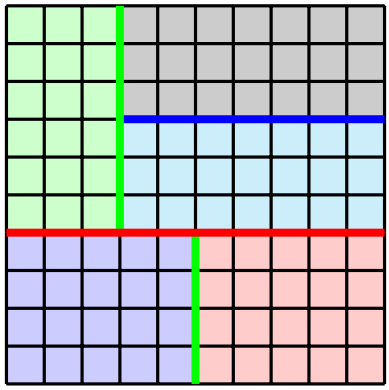


(c) Domain using four processes in an exploded-view. The red cross marks the same position for all processes.

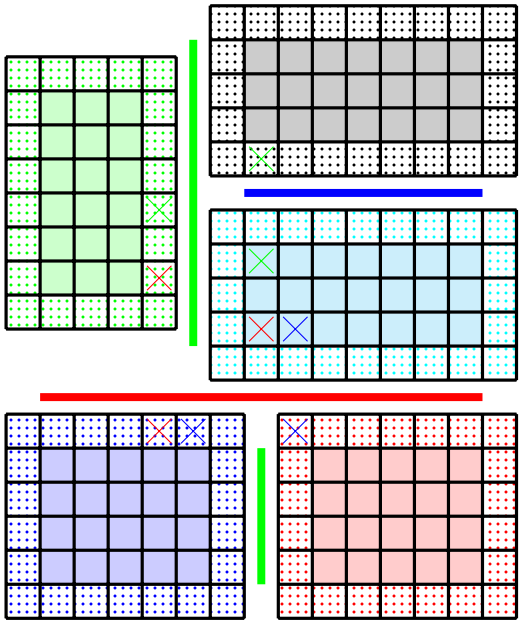
Figure 3.3: Standard domain decomposition (sdd) for a varying number of processes. One process calculates the interactions of all cells with the same color, while ghost cells around a specific domain are marked using patterns. Particles in ghost cells are only needed for the correct calculation of the particle interactions.



(a) Tree of the partition.



(b) Partition of the domain.



(c) Partitioning in an exploded-view, also visualizing ghost cells. Crosses of the same color mark the same positions on different subdomains.

Figure 3.4: k-d tree based domain decomposition of a domain into five subdomains. The domain is recursively split along the different colored lines in the order red, green, blue.

grid of $n_1 \times n_2 \times n_3$ processes. The function will try to select a balanced distribution of processes, s.t., $n_1 \approx n_2 \approx n_3$. `MPI_Cart_create` uses the grid generated by `MPI_Dims_create` and creates a new MPI communicator that contains information about the Cartesian topology.

After the generation of the processor-grid, it is applied to the simulation domain. *ls1 mardyn* then splits the domain in $n_1 \times n_2 \times n_3$ equally sized partitions.

To be able to calculate interactions of particles that belong to different processes, *ls1 mardyn* exchanges particle information between the processes using MPI. Hereby, *ls1 mardyn* communicates the information of particles that lie close to the border of one partition to all processes lying close to that particle. This information is saved in so-called ghost cells (cf. Figure 3.3).

3.1.2.2 K-D Decomposition

Molecular dynamics simulations often include scenarios with heterogeneous particle distributions. These can range from lightly inhomogeneous scenarios like droplet formation and condensation to heavily inhomogeneous scenarios like droplet coalescence and vapor-liquid-equilibrium simulations [106, 107].

These inhomogeneous particle distributions result in heterogeneous load distributions, as the local load depends on the local particle density. A partitioning into equally sized partitions thus generally produces partitions with unequal load. To provide good performance the domain has to be split into partitions with (almost) equal load. For this, [2] introduced a k-d tree-based domain partitioning technique. It uses a recursive bi-partitioning scheme to split the domain (cf. Figure 3.4). [2] also compared k-d tree-based against diffusive-based, graph-based and space-filling curve-based approaches and came to the conclusion that the k-d tree-based approach provides the best performance for *ls1 mardyn*.

In the context of this thesis the k-d tree-based approach was further optimized. These optimizations, together with a more in-depth view of the k-d tree-based decomposition are described in chapter 4.

3.2 Control Flow

The control flow of *ls1 mardyn* consists out of two phases. First, the simulation is initialized according to the parameters given by an input file and through the command line. Next, the actual simulation is carried out through the main simulation loop. At the beginning of each loop iteration, the initial kick of the (rotational) leapfrog integrator [108, 109] (synchronized form) is performed

$$v \mathrel{+}= \frac{\mathrm{d}t}{2} \cdot \frac{F}{m} \tag{3.1}$$

$$x \mathrel{+}= \mathrm{d}t \cdot v \tag{3.2}$$

$$(+ \text{ rotational terms}). \tag{3.3}$$

After the initial kick, the halo particles are exchanged, as they are needed for the following step, i.e., the force calculation. As they are no longer needed, the halo particles are removed after the force calculation. The final step of the loop iteration consists of the drift step of the integrator

$$v \mathrel{+}= \frac{\mathrm{d}t}{2} \cdot \frac{F}{m} \tag{3.4}$$

$$(+ \text{ rotational terms}). \tag{3.5}$$

ls1 mardyn is highly extensible through plugins. They are initialized in the initialization phase of *ls1 mardyn* and are repeatedly called in each loop iteration at specific extension points (cf. Figure 3.5).

3.3 Structure

ls1 mardyn provides a modular design to ease maintenance and extensibility. For this purpose, the different functionalities of *ls1 mardyn* are well separated into different classes.

[3] Taken from the *ls1 mardyn* documentation and adapted slightly.

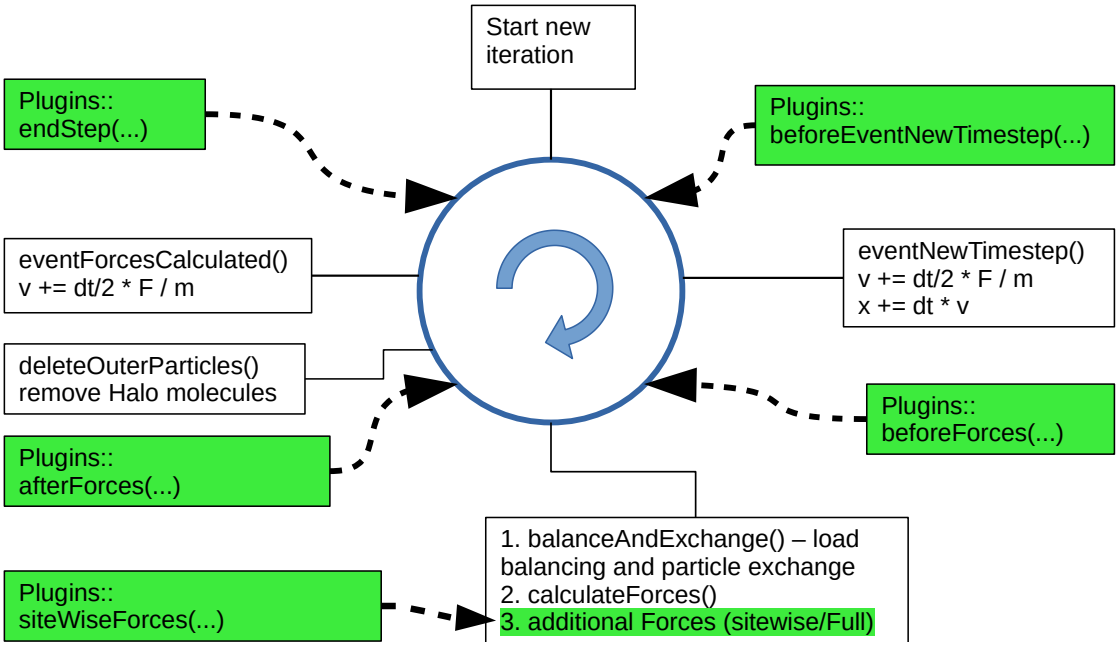


Figure 3.5: Overview of the main simulation loop of *ls1 mardyn*. Highlighted in green are the different extension points at which plugins are called.[3]

These classes allow switching the decomposition schemes, s.t., the Cartesian decomposition, or the k-d tree-based decomposition can be used. Additionally, all plugins inherit from a common *PluginBase* class which provides an interface that provides extensibility for *ls1 mardyn*. These plugins and also other parts of the code can easily access the particles stored in a particle container through the means of iterators (introduced in [5,6]). They hide the inner workings of the particle storage and the used container and were initially implemented to provide an agnostic way to iterate over both an AoS and an SoA storage of particles [5]. The flexible structure of *ls1 mardyn* helped us in extending it further, e.g., we added a new class for coupling arbitrary load balancers (see section 5.5).

An overview of the different classes is given below.

Molecule The *Molecule* classes *FullMolecule* and *MoleculeRMM* contain the position, velocity and force information of a molecule. While *FullMolecule*s are normally stored inside of the *ParticleContainer*, *MoleculeRMM* is not actually stored. Instead, an SoA representation of the molecules is used. *MoleculeRMM* is used whenever the *Molecule*s are added to the container or accessed through the iterators.

(Region)ParticleIterator The ParticleIterator and RegionParticleIterator provide an easy way to access all particles inside of *ParticleContainer* [6]. [5] introduced *ParticleIterator*s to provide a uniform interface to access molecules both in normal and RMM modes.

ParticleContainer *ParticleContainer* provides an interface for containers to store and manage molecules. A *ParticleContainer* further provides the methods *iterator()*, *regionIterator()*. These methods will generate the appropriate iterator to access the particles from outside of the *ParticleContainer*.

LinkedCells The class *LinkedCells* implements the *ParticleContainer* interface. It stores the molecules inside of the cells of the linked cells.

CellPairTraversals The *CellPairTraversals* are used to define a scheme to iterate over all pairs of neighboring cells of a *LinkedCells* container. A short overview of the different traversals can be found in subsubsection 3.1.1.3.

CellProcessor A *CellProcessor* defines the interactions between the particles in one or within a pair of cells. *ls1 mardyn* mainly implements two different *CellProcessor*s: The *LegacyCellProcessor* and the *VectorizedCellProcessor*. The former is used for AoS-like access of the particles, while the latter accesses the SoAs of particles. It uses a hand-written intrinsics wrapper to enable vectorization of the force calculation.

PluginBase *ls1 mardyn*'s capability for extension is realized by a modular plugin architecture. For this purpose, *ls1 mardyn* provides the interface *PluginBase* from which all plugins inherit. It defines specific entry points which the program calls at different times during the simulation loop. These entrypoints provide different information as they resemble different states throughout the simulation (cf. Figure 3.5). There, e.g., exists steps for adding additional forces and a step that is mainly used for the output of the simulation. A plugin typically only uses one or two of those extension points. For a detailed view of the plugin structure in *ls1 mardyn*, please refer to [110].

DomainDecompBase *ls1 mardyn* uses periodic boundary conditions. *DomainDecompBase* provides the sequential implementation for them. These methods are used if MPI is not available.

DomainDecompMPIBase This class provides the MPI-parallel particle-exchange methods for both *DomainDecomposition* and *KDDecomposition*.

DomainDecomposition The standard domain decomposition (c.f. Figure 3.3) is implemented in the class *DomainDecomposition*.

KDDecomposition This class adds the k-d tree based domain decomposition (c.f. Figure 3.4) to *ls1 mardyn*.

Simulation The class *Simulation* orchestrates the entire simulation. It calls the different classes for input, particle exchange and domain partitioning, force calculation, output, and the registered plugins.

4 Multi-Node Optimizations

4.1 Related Work

For most particle codes, some effort was put into providing good multi-node scalability through load balancing and various MPI optimizations.

Load balancing in the form of orthogonal recursive bisection methods, like k-d trees, is used not only in *ls1 mardyn* but also in other codes such as *NAMD* [111–113] or *LAMMPS* [75]. *NAMD*, hereby, combines spatial and force decomposition methods by assigning cells to specific processes, but allowing the calculation of forces between two cells (called compute) on arbitrary processes. This enables possibilities for load balancing but also suggests topology-aware placement of these compute objects, which was implemented in [114]. Besides the bisection method, *LAMMPS* also provides options to produce rectilinear grids and generally aims to distribute an equivalent number of (weighted) particles to each process. A weighting is possible to equally distribute the computational load instead of the particle number and is needed if, e.g., different particle potentials are used. *LAMMPS* provides a wide variety of configurable weighting algorithms ranging from direct input over the number of neighbors to timing data [1].

GROMACS [115] uses a staggered grid approach, where the partition boundaries of an initially regular grid are moved in a diffusive-like hierarchical approach, which aims to work without global communication. A similar approach is also implemented in *A Load Balancing Library* [116], which is subject of the work at hand.

FDPS [117] uses a multi-section scheme which allows for non-expensive re-balancing [64]. Like the staggered-grid approach, the multi-section method provides good locality, i.e., upon a small change of the load distribution, only a small move of the partition boundaries is performed, which is often not guaranteed for bisection methods [64].

Some codes provide the ability to combine different spatial partitioning methods. An implementation of a graph-based partitioner for *ESPResSo* [118] that combines full rebalancing with diffusive rebalancing has been showcased in [77]. *ESPResSo* is also capable of joined partitioning for multi-physics simulations (e.g., lattice-Boltzmann flow simulations and MD simulations) [119].

There also exist codes like *ms2* [120–123] that use force decomposition instead of spatial decomposition approaches. These are, however, mostly limited in the maximal number of particles and provide bad scalability.

Some codes do, however, use zonal methods or neutral territory methods which are a mixture of spatial and force decomposition methods (cf. subsection 2.6.5). Hereby, the eighth shell method is the most popular, and is, e.g., used by *GROMACS* [54]. Some

[1] `https://docs.lammps.org/balance.html`, `https://docs.lammps.org/fix_balance.html`

codes, like *LAMMPS*, do not support neutral territory methods, because developers claim that typically run models would not gain much performance through their use [2].

Support for overlapping communication also varies between the different simulation codes. While *LAMMPS* and *ms2* provide no real overlapping, *GROMACS* provides overlapping of the GPU computations of the non-bonded interactions with the calculations of the bonded interactions and the communication on the CPU [35]. *NAMD* supports overlapping communication through the use of the task-based parallel programming paradigm *Charm++* [124] [112].

4.2 Load Balancing and k-d Decomposition in ls1 mardyn

In the context of this dissertation, *ls1 mardyn*'s k-d decomposition (kdd) was optimized and adapted to handle additional use cases that it could not handle before. Among other optimizations, this includes the handling of heterogeneous compute architectures and better load estimation.

4.2.1 K-D Decomposition in ls1 mardyn

In *ls1 mardyn*, the kdd works at cell level, i.e., it splits the domain of size $a \times b \times c$ into a regular rectilinear grid of $\lfloor \frac{a}{r_c} \rfloor \times \lfloor \frac{b}{r_c} \rfloor \times \lfloor \frac{c}{r_c} \rfloor$ cells and distributes these cells among the different processes.

For the distribution, *ls1 mardyn* uses the small class `KDNode` that stores the recursive structure of the kdd. Each instance of `KDNode` stores the area of owned cells, the owning process, the number of processes assigned to this node or its children, and pointers to the two children of the node. Using these nodes, the full information of the k-d tree can be stored and an efficient splitting of the domain can be found.

To calculate a balanced partition, information about the computational load of each partition is needed. To track the quality of a possible partitioning, the deviation D_i of the assigned load C_i from the optimal load C_{opt} is tracked for each process i.

$$C_i = \sum_{\text{cells j assigned to process i}} c_j \tag{4.1}$$

$$C_{\text{total}} = \sum_{\text{processes i}} C_i \tag{4.2}$$

$$C_{\text{opt}} = \frac{C_{\text{total}}}{n_{\text{procs}}} \tag{4.3}$$

$$D_i = (C_i - C_{\text{opt}})^2 \tag{4.4}$$

The sum of the deviations over all subdomains $D_{\text{total}} = \sum_i D_i$ is then used as a measure for the quality of a partition. In the k-d tree, these subdomains correspond to the leaves of the entire tree. For the non-leaf nodes of the tree, the sum of the deviations of its

[2] `https://sourceforge.net/p/lammps/mailman/message/34807376/`

children is stored.

$$D_{\text{parent}} = D_{\text{child},1} + D_{\text{child},2} \tag{4.5}$$

Applying this formula from the bottom-up to a tree will store the total deviation of each subtree at the root node of it and the root of the entire tree contains its total load deviation. If this formula were used exclusively, searching for the best possible subdivision would require creating all possible and complete subdivisions, as it can only be applied bottom-up to the tree. To circumvent this requirement, the lower bound $\inf D$ for the deviation is used. It specifies the lowest possible deviation $\inf D_{\text{parent}}$ from the optimal load for an arbitrary node given a number of processes n and the load C assigned to it (derivation in section A.1). The infimum includes all possible further subdivisions of the node and all its children.

$$\inf D = n \left(\frac{C}{n} - C_{\text{opt}} \right)^2 \tag{4.6}$$

$$\inf D = n \left(C_{\text{avg}} - C_{\text{opt}} \right)^2 \tag{4.7}$$

$$\tag{4.8}$$

Using this formula, the infimum for a node assuming knowledge of its children is

$$\inf D_{\text{parent}} = n_1 \left(C_{\text{avg,child},1} - C_{\text{opt}} \right)^2 + n_2 \left(C_{\text{avg,child},2} - C_{\text{opt}} \right)^2 , \tag{4.9}$$

where n_1 and n_2 are the number of processes assigned to child 1 and child 2, and $C_{\text{avg,child},1} = \frac{C_{\text{child},1}}{n_1}$ and $C_{\text{avg,child},2} = \frac{C_{\text{child},2}}{n_2}$ are the average loads of the two children. The above formula will henceforth be called *quadratic model*.

The creation of the tree follows a recursive, parallel scheme and is implemented in the function `decompose(fatherNode)`. In this function, if only one process is assigned to `fatherNode`, i.e., if the node is a leaf, the load deviation from the optimal load is calculated and the function returns. Otherwise, i.e., if more than one process is assigned to `fatherNode`, possible partitions into two parts are calculated using the function `calculateAllPossibleSubdivisions()` and stored in `possibleSubdivisions`. These subpartitions are sorted according to $\inf D_{\text{parent}}$ (Equation 4.9).

After the possible subdivisions are calculated, they are further analyzed. Starting with the subdivision with the lowest $\inf D$, recursive calls to `decompose()` are issued for each of the two partitions of the subdivision. Hereby, only processes that are assigned to a specific partition partake in the recursive call (handled using different MPI Communicators). After the recursive call returns, the sum of the deviations of the two partitions is calculated using `MPI_Allreduce`. If the current subdivision's load deviation is lower than a previous subdivision, it is saved as the best subdivision. Further subdivisions of `possibleSubdivisions` are tested if their $\inf D_{\text{parent}}$ is lower than the one of the best subdivision. After all subdivisions are calculated, the best subdivision is returned. The recursion is started with the root node of the tree which contains the entire simulation domain. Using this recursion, the tree with the lowest overall deviation of the optimal load is constructed.

Even with the usage of inf D_{parent} the total number of possible subdomains is very big and a few optimizations to reduce its size are used. First, if more than `fullSearchThreshold` processes are assigned to `fatherNode`, instead of calculating multiple subdivisions, the number of processes is split in half and an appropriate subdomain is assigned to the children in function `calculateAllPossibleSubdivisions()`. This is possible, as the high number of processes guarantees the availability of a good subdivision. Second, `fullSearchThreshold` also provides a limit on whether all calculated subdivisions are actually tested. If we are close to the root node and have enough processes ($N_{\text{procs}} < 2^{\text{level+fullSearchThreshold}}$) only the subdivision with the lowest inf D is tested. Typical values of fullSearchThreshold lie between 3 and 8 (we use a value of 3 in this thesis). The function `decompose()` is depicted in algorithm 4.1.

For `decompose()` to work, the load c has to be known for each cell. The cost c for a cell is calculated using the empirical formula

$$c = N_{\text{own}}^2 + \frac{1}{2} \sum_{\text{neighbor} \in \text{neighbors}} N_{\text{own}} \cdot N_{\text{neighbor}}, \tag{4.10}$$

where N_{own} is the number of particles in the respective cell and N_{neighbor} the number of particles in a neighboring cell.

4.2.2 Limitations

During this thesis, a couple of limitations of the algorithm described above have been revealed and resolved.

1. The algorithm does not incorporate heterogeneous compute systems into the domain partitioning. This results in a bad load balancing when using clusters with differently powerful compute elements.

2. The algorithm only works well for slightly inhomogeneous particle distributions, e.g., for droplet formation scenarios. For very inhomogeneous particle distributions, a good load balancing cannot be observed.

3. The algorithm does not take multiple particle types into account. This can be seen in Equation 4.10, where only the particle number, but not the particle type is considered. If there exist at least two spatially separated groups of particles, where the calculation of forces is more expensive for one of the particle types, large load imbalances can arise.

4.2.3 Adaptions for Heterogeneous Compute Systems

4.2.3.1 Motivation

At the time of writing this thesis, most HPC clusters contain large amounts of identical nodes. However, there exist some clusters which consist of multiple partitions, where only within a partition the used nodes are identical, while nodes of different partitions might differ.

Algorithm 4.1: Parallel algorithm to evaluate the best partitioning. This algorithm is used by the k-d tree-based decomposition.

```
1  Function decompose (fatherNode)
       input  : fatherNode: a KDNode consisting of the subdomain and the assigned
                processes (starting process and numprocs)
       output : complete partitioning of fatherNode
2      if fatherNode.numprocs == 1 then
3          fatherNode.calculateDeviation ()
4          return fatherNode
5      // subdividedNodes are sorted by a lower bound for the deviation (infD), from
         smallest to biggest.
6      subdividedNodes ← calculateAllPossibleSubdivisions (fatherNode)
7      bestSubdivision ← nullptr
8      bestDeviation ← ∞
9      foreach node in subdividedNodes do
10         if node.infD > bestDeviation then
11             // If the lower bound for the deviation is bigger than the best
                 measured deviation, we end the loop, as all following subdivision will
                 also be worse.
12             break
13         // Each process (with MPI rank myrank) only partakes in the
             decomposition of one of the children.
14         if myrank in node.child1.processes then
15             node.child1 ← decompose (node.child1)
16         else
17             node.child2 ← decompose (node.child2)
18         // The deviations of the two children are collected using MPI_Allreduce.
19         partialAllReduceDeviations ()
20         node.deviation ← node.child1.deviation + node.child2.deviation
21         if node.deviation < bestDeviation then
22             bestSubdivision ← node
23             bestDeviation ← node.deviation
24     return bestSubdivision
```

One example of these clusters was CoolMAC[3] (see section B.2), which consisted of five different partitions. Three of these partitions were CPU-only, using Intel's SandyBridge architecture (snb), one using AMD's Bulldozer architecture (bdz), and another using Intel's Westmere architecture (wsm). The other two partitions also featured GPUs in combination with Intel's SandyBridge CPU architecture. The partition "nvd" additionally included NVIDIA Tesla GPUs, while the partition "ati" included AMD's FirePro GPUs.

Another example was the SuperMIC, a cluster consisting of multiple nodes, each containing an Ivybridge Host-CPU and Xeon Phi accelerator cards. To properly leverage the performance of the accelerators, they should be used in their native mode, i.e., an entire simulation process should run on the accelerator. To also leverage the Host-CPU, another MPI process will run on this CPU.

In the case of CoolMAC, nodes of different partitions provide different performance, while for SuperMIC, MPI processes on the host and on the accelerators have a different simulation speed.

In general, we model these performances as P_i. If a load balancer simply distributes the loads evenly without taking the varying performance into account, the different processes i will take a different amount of time t_i for the same work load C.

$$t_i = C/P_i \tag{4.11}$$

Instead, the load balancers have to take the performance of the different compute resources into account (cf. Figure 4.1) and assign appropriate loads, here called C_i to the compute resources, s.t., the computations all take the same amount of time.

$$t_i = C_i/P_i \stackrel{!}{=} const \tag{4.12}$$

4.2.3.2 Implementation

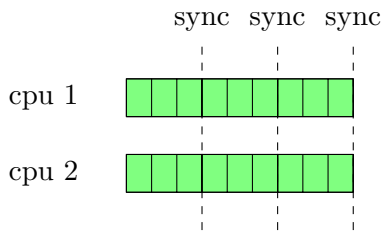
To adapt the algorithm to incorporate the performances of the compute resources correctly, the formulas for the calculation of the deviations, i.e., Equation 4.4 and Equation 4.9, had to be adapted (for a derivation, see section A.2) and now incorporate the performances P of the children of a node and the total mean performance of all ranks (P_{avg}):

$$D_i = \left(C_i - \frac{C_{\text{opt}} \cdot P_i}{P_{\text{avg}}}\right)^2 \tag{4.13}$$

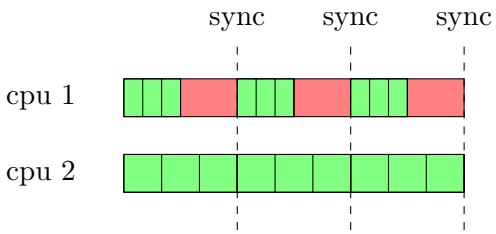
$$\begin{aligned} \inf D_{\text{parent}} = {} & n_1 \left(C_{\text{avg,child,1}} - \frac{C_{\text{opt}} \cdot P_{\text{avg,child,1}}}{P_{\text{avg}}}\right)^2 \\ & + n_2 \left(C_{\text{avg,child,2}} - \frac{C_{\text{opt}} \cdot P_{\text{avg,child,2}}}{P_{\text{avg}}}\right)^2 \end{aligned} \tag{4.14}$$

(4.15)

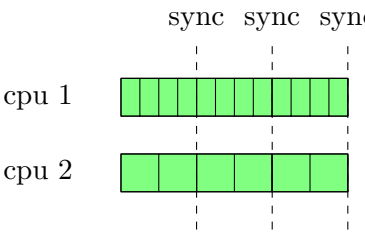
[3] http://www.mac.tum.de/wiki/index.php/MAC_Cluster



(a) Equally powerful compute resources. Cpu 1 and cpu 2 provide the same performance and have to calculate tasks with the same computational load.



(b) Differently powerful compute resources with performance-unaware load balancing. Here, cpu 1 is twice as powerful as cpu 2. The load is equally, but – for the performance – inappropriately distributed.



(c) Differently powerful compute resources with performance-aware load balancing. Here, cpu 1 is twice as powerful as cpu 2 and the load is distributed accordingly.

Figure 4.1: Load balancing for heterogeneous architectures. If the performance of processes vary, performance-aware load balancing is necessary. In green, the different tasks are displayed. Every six tasks, synchronization is necessary. This corresponds to the synchronization after each time step in which neighborhood information is exchanged. If a process has no work to do and has to wait for the next synchronization step, it is idling (marked in red). The total load and the total compute power is the same for each graphic.

To correctly use these adapted formulas, the performance of each process has to be known. These could either be input directly by the user of *ls1 mardyn* or could be measured directly by the program. We decided to implement the latter and implemented a means to measure the performance of each process. Hereby, for each process the number of flops needed for a simulation step is counted and the time needed for one simulation step is measured. This flop rate is then used as P_i.

4.2.3.3 Results[4]

Estimation First, a theoretical estimate of the speedup is given. This is needed to quantitatively evaluate the performance gains. Assuming performance-unaware, but otherwise perfect load balancing, the time used per simulation loop is limited by the process with minimal performance.

$$t_{\text{unaware}} = \frac{C_i}{\min P_i} = \frac{C_{\text{total}}}{n_{\text{procs}} \min P_i} \tag{4.16}$$

For performance-aware load balancing, this time is given as

$$t_{\text{aware}} = \frac{C_i}{P_i} = \frac{C_{\text{total}}}{P_{\text{total}}}. \tag{4.17}$$

The speedup S is thus given as

$$S = \frac{t_{\text{unaware}}}{t_{\text{aware}}} = \frac{P_{\text{total}}}{n_{\text{procs}} \min P_i} = \frac{P_{\text{avg}}}{\min P_i}. \tag{4.18}$$

The speedup can thus be expected to be large, if the average performance P_{avg} is much bigger than the minimal performance $\min P_i$. The most extreme case is reached, if only one process performs significantly worse than all other processes. In that case, the expected speedup is $S_{\max} = \frac{P_{\text{fastest}}}{P_{\text{slowest}}}$.

Clusters We have tested our scenario on CoolMAC and SuperMIC. The former was an inhomogeneous compute system consisting of five different partitions containing two generations of Intel CPUs and one generation of AMD CPUs.

In contrast to CoolMAC, SuperMIC's nodes are all the same and the cluster also only provides one partition. Instead, the nodes themselves are inhomogeneous, as they consist of a host CPU and an accelerator (Xeon Phi, Knights Corner), which can both run simulation code. The individual cores of the host CPU and the accelerator each have their distinct performance characteristics, as they run at significantly different clock speeds.

Due to their different types of inhomogeneity, CoolMAC and SuperMIC were thus perfectly suitable for our experiments.

In addition to CoolMAC and SuperMIC, we performed measurements on CoolMUC2. As it is, however, not a heterogeneous system, we emulated different performances by assigning a varying amount of threads to the MPI processes.

For a more detailed description of the clusters, see Appendix B.

[4]Disclaimer: Some of the presented results have previously been published in [125].

Scenarios We have tested our new method using three homogeneous scenarios:

LJ-CGG The first chosen scenario consists of single-centered Lennard-Jones-12-6 particles. They are generated using the cubic grid generator with a density of 0.785, and a cutoff of 3.5, resulting in around 35 particles per cell. As it is a scenario generated by a generator, the size of the scenario and the total number of particles can vary and is indicated when used.

E512k This scenario is a relatively small scenario. It consists of a pre-equilibrated set of 512 000 rigid ethane molecules (C2H6), which are modeled using two Lennard-Jones-12-6 sites. The scenario consists of 25 x 25 x 25 cells, resulting in around 37 molecules per cell.

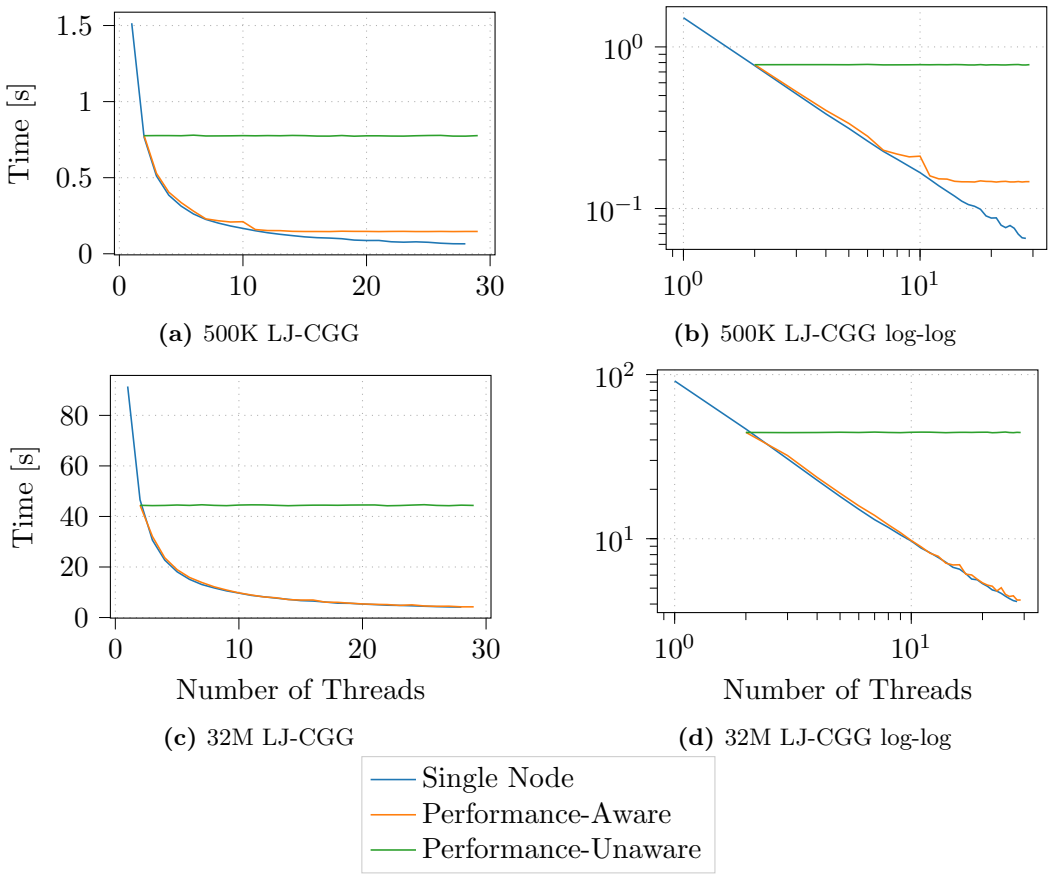
LJBig This scenario is significantly bigger with 344 million single-centered Lennard-Jones molecules, resembling, e.g., a scenario simulating argon. A total of 200 x 200 x 200 linked cells with around 43 molecules per cell are simulated.

Results We evaluated the developed method using two nodes of CoolMUC2 and the LJ-CGG scenario. On the first node, one MPI rank was started with one OpenMP thread. On the second node, we used a variable number of OpenMP threads. That way we can model huge performance differences between the two ranks ranging from 1x to 28x.

As the performance-unaware method does not consider the performance differences between the different ranks, it distributes an equal workload to the two ranks. The slower of the two ranks (using one OpenMP thread), limits the performance of the overall simulation. We, therefore, observe equal performance independent of the total number of used OpenMP threads (cf. Figure 4.2).

The performance-aware method, however, properly incorporates the performance ratio of the different MPI ranks and thus good scalability is observed. Compared to the experiment run on a single node (pure OpenMP scalability), it provides a bit worse performance for the smaller of the two scenarios. As the smaller scenario uses 24 x 24 x 24 inner cells and always at least a minimum of two cells in each dimension needs to be distributed to each process, the smallest possible subdomain has the size of 2 x 24 x 24 cells which corresponds to 1/12 of the simulation domain. The strong scaling limit of around 12 threads is thus expected. The single-node OpenMP parallelization (c08) uses a finer-grained parallelization and therefore better scalability is reached for the small scenario. For the larger scenario, both the single-node parallelization and the performance-aware load balancing provide equal performance.

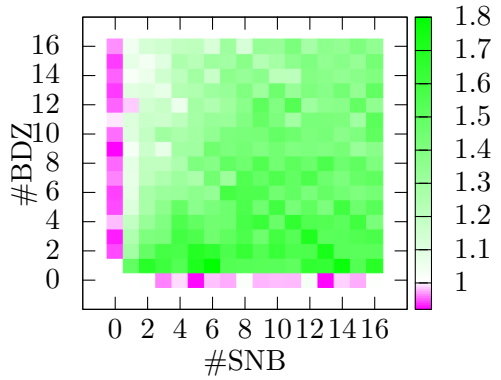
We further tested the performance-aware load balancer on CoolMAC (cf. Figure 4.3, [125]). Using two nodes (1 x AMD Bulldozer, 1 x Intel Sandy Bridge), we observed that significant speedups could be achieved that lie close to the theoretical maximum. We noticed that the performance-aware load balancer could introduce performance degradation if only ranks of a single type are used. This decline is, however, expected as measurement errors can make equally performant compute resources appear to provide different performances which result in a bad load balancing. Additionally, the performance drops were smaller than 10%, showing that the measurements were reasonable. We



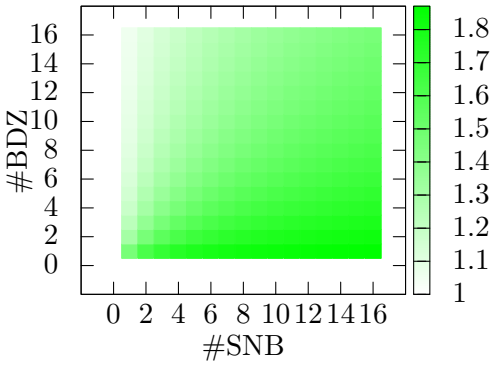
(a) 500K LJ-CGG **(b)** 500K LJ-CGG log-log

(c) 32M LJ-CGG **(d)** 32M LJ-CGG log-log

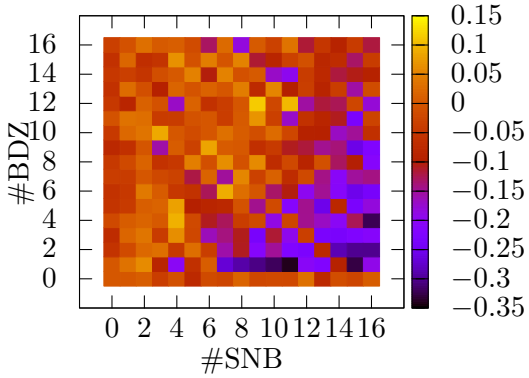
Figure 4.2: Scalability of the performance-unaware and the performance-aware load balancing for two different scenarios using two nodes of CoolMUC2. For *Performance-Aware* and *Performance-Unaware*, the simulation has been carried out on two nodes, where the MPI process on the first node is limited to one OpenMP thread and the MPI process on the second node has a variable number of OpenMP threads ranging from one to 28. This results in two MPI-ranks with largely varying performances. The x-Axis always shows the total number of threads used, i.e., 2 threads of the *Performance-Aware* or *Performance-Unaware* load balancing correspond to 2 MPI ranks, each using 1 OpenMP thread. *Single Node* is used as comparison and shows the performance values if only one node is used with a single MPI rank.



(a) Speedup in dependence of the number of ranks.

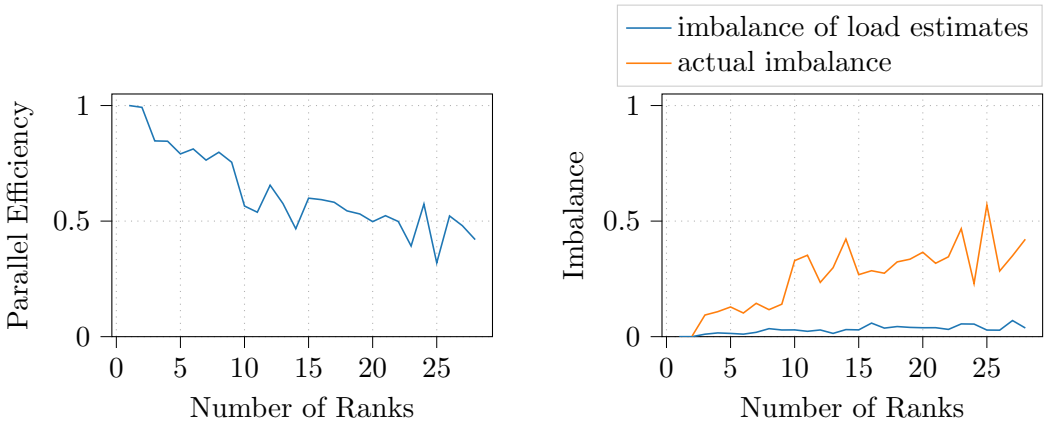


(b) Theoretical speedup (Equation 4.18).



(c) Speedup minus theoretical speedup.

Figure 4.3: Speedup and theoretical speedup for the E512k scenario on two nodes of CoolMAC. Shown is the speedup using the performance-aware load balancing compared to a performance-unaware load balancing in dependence on the number of ranks used on the two nodes. The performance of the ranks on the Intel Sandy Bridge (SNB) node is roughly 1.9 times as fast compared to the ranks on the AMD Bulldozer (BDZ) node. Left graphics taken from [125].



(a) Parallel efficiency.

(b) Measured and predicted imbalance.

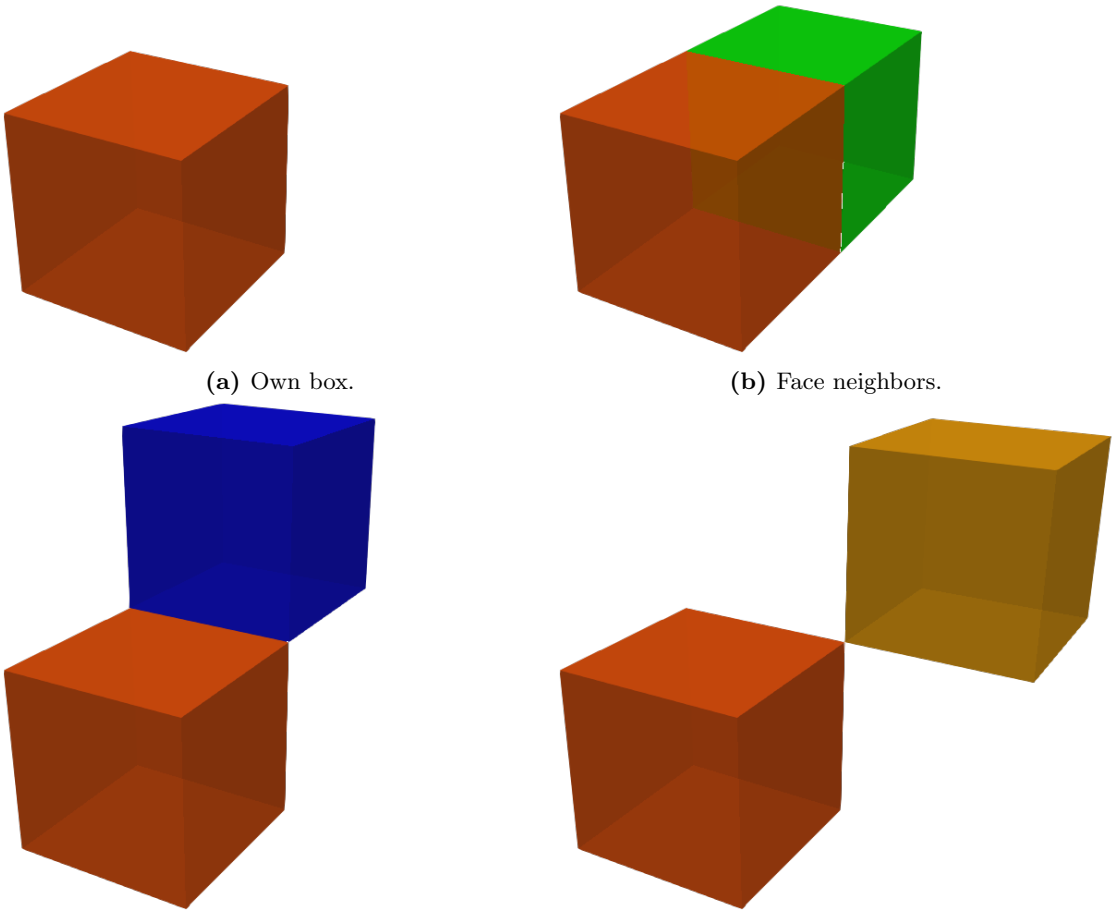
Figure 4.4: Strong scaling for a droplet coalescence scenario with around 3 million particles on one node of CoolMUC2 using the default load estimator (Equation 4.10). Bad scalability is observed. The large deviation from the predicted load imbalance (based on the load estimates) and the actually measured imbalance shows that a bad load estimation is causing the limited scalability.

further executed a large simulation (LJBig) on CoolMAC that included a total of 1056 ranks on three different partitions of the cluster (nodes: 15x Intel Sandy Bridge, 8x AMD Bulldozer, 1x Intel Westmere). For this setup, we observed a speedup of 1.3 for the performance-aware load balancer, which was a good result compared to the theoretical speedup of 1.4.

In [125] we went into more details on this topic and further showed speedups on SuperMIC, which showcased that the load balancer can also account for large performance differences of 5x.

4.2.4 Better Load Estimation

As noted in subsection 4.2.2 (limitation 2), we observed bad load balancing in *ls1 mardyn* when simulations with strongly varying particle densities were executed. One such scenario is a droplet coalescence simulation, i.e., two neighboring droplets embedded in a gaseous phase merge together and form a larger droplet. The problematic part about this simulation is that the particle density in the liquid is about 35 times higher compared to the gaseous phase and most cells of the simulation are (almost) empty, but still significantly contribute to the total time needed for the simulation. This effect can, however, not be appropriately resembled using Equation 4.10, which assumes that empty cells produce no load. This is, however, not the case and leads to a bad load estimation and therefore bad load balancing (cf. Figure 4.4).



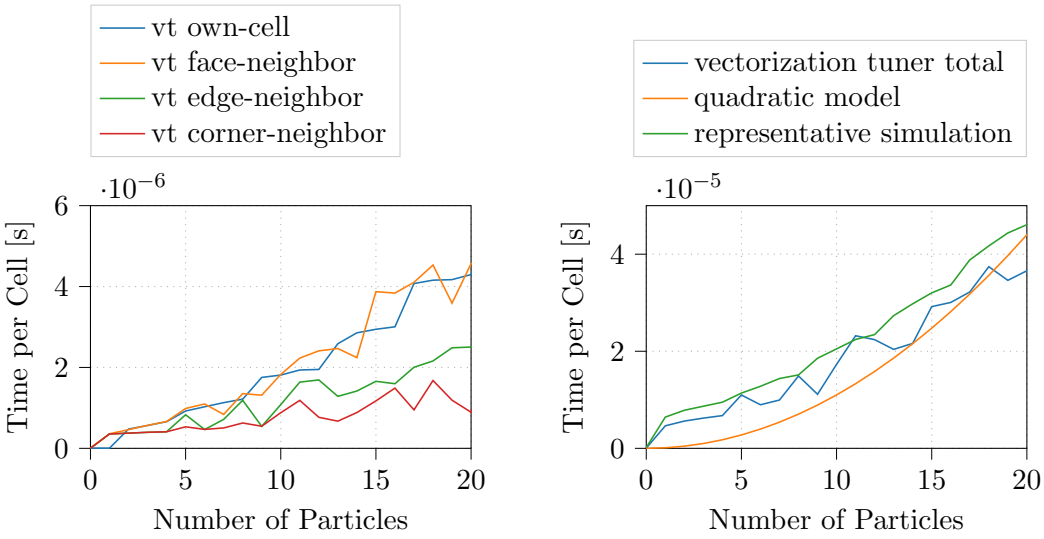
(a) Own box.

(b) Face neighbors.

(c) Edge neighbors.

(d) Corner neighbors.

Figure 4.5: Different box relations for which the vectorization tuner samples the performance/-time.



(a) Vectorization tuner (vt) output.

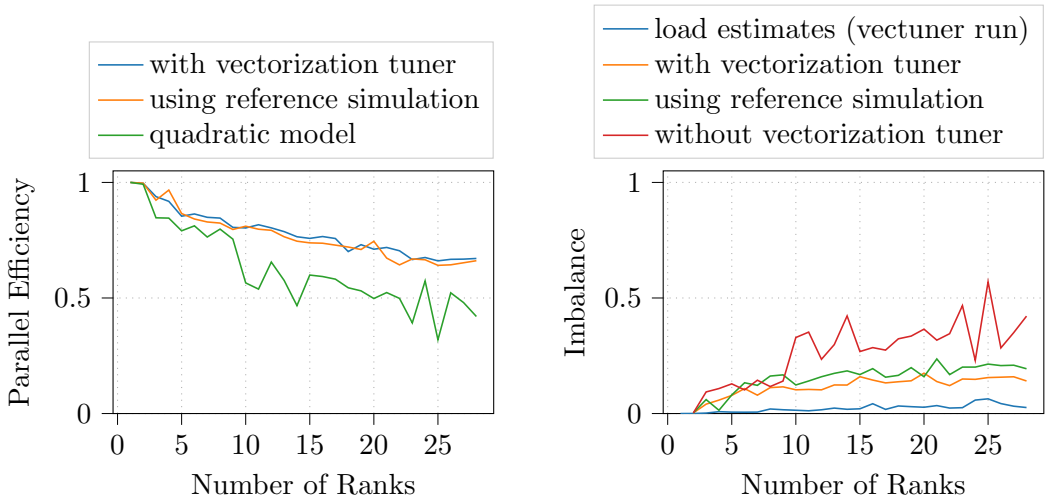
(b) Different load estimations.

Figure 4.6: Load estimation for the vectorization tuner. The tuner generates time measurements for the force calculation within one cell and with the different neighbors (left). On the right, the vectorization tuner is compared to the quadratic model and a representative simulation. The representative simulation contains exactly the given number of particles per cell and its time measurements only include the force calculation.

4.2.4.1 Vectorization Tuner as Load Estimator

Instead of the quadratic model, we use the output of the vectorization tuner, which was initially developed to understand the performance characteristics of *ls1 mardyn*'s force kernel in the context of [6]. It generates data that describes the time needed for the force calculation depending on the number of particles within a cell. The accumulated data additionally considers the relations between cells, i.e., it generates data for interactions within one cell and for interactions between neighboring cells, which are further distinguished between face, edge, and corner neighbors (cf. Figure 4.5). Some drawbacks of this method stem from the implemented sampling, which measures the time of repeated force evaluations using dummy cells for different numbers of particles. This measurement thus only includes the time for the actual force calculation and cannot represent any time spent in other parts of the simulation (e.g., plugin calls). Additionally, it always reuses the same cells. This might result in deviations from the actual force calculation because the cells will always be cached during the sampling but not necessarily during the actual force calculation.

Results When comparing the loads estimated by the quadratic model and the vectorization tuner, one can observe that the vectorization tuner assigns more loads to sparsely populated cells (cf. Figure 4.6). To verify the correctness of these estimates we performed measurements using a reference simulation. As reference, we simulated 10000



(a) Parallel efficiency.

(b) Measured and predicted imbalance.

Figure 4.7: Strong scaling for a droplet coalescence scenario with around 3 Million particles on one node of CoolMUC2 using the vectorization tuner as load estimator. Better load balance is observed compared to the default load estimator.

cells which contain a constant number of particles per cell for 100 time steps, and took time measurements for the force calculation. We then used these measurements as load estimator.

Using the vectorization tuner reduces the load imbalances by half, thus resulting in better overall load balancing and parallel efficiency (cf. Figure 4.7).

4.2.5 Load Estimation via Inverse Problem (MeasureLoad)

We presented another possibility to estimate the load for the cells in [110]. Using it, we try to estimate the load not based on a reference or dummy problem but, instead, on the actual problem at hand. Therefore, we measure the time needed for the force calculation on each rank. We then assume that for each rank I its time T_I can be written as

$$T_I = \sum_{i=0}^{n_{\text{max per cell}}} n_{I,i} \cdot t_i \qquad \forall I \in \{0, ..., n_{\text{ranks}}\}, \tag{4.19}$$

where $n_{\text{max per cell}}$ is the maximal number of particles per cell, n_{ranks} the number of ranks and $n_{I,i}$ the number of cells with i particles on rank I. If the cell statistics $(n_{I,i})$ are known, the time needed for a cell with i particles can be reconstructed. For the reconstruction, it is advantageous to rewrite the above system of equations in its matrix-vector form

$$T = N \cdot t, \tag{4.20}$$

where T corresponds to the time measurements of entire ranks, N to the cell statistics, and t to the sought time values for single cells. The reconstruction can only be performed if enough measurements are performed ($n_{\text{ranks}} \geq n_{\text{max per cell}}$), as otherwise the system is under-determined. To get positive t_i, we use a non-negative least squares algorithm [126, 127] to solve the typically over-determined system.

Additionally, one can include constraints on t_i. One such constraint could be that the time needed for the calculation of the interactions of a cell with more particles should take longer than that of a cell with fewer particles, i.e, $t_i > t_j$ if $i > j$. This constraint can be implemented by requiring that the vector d

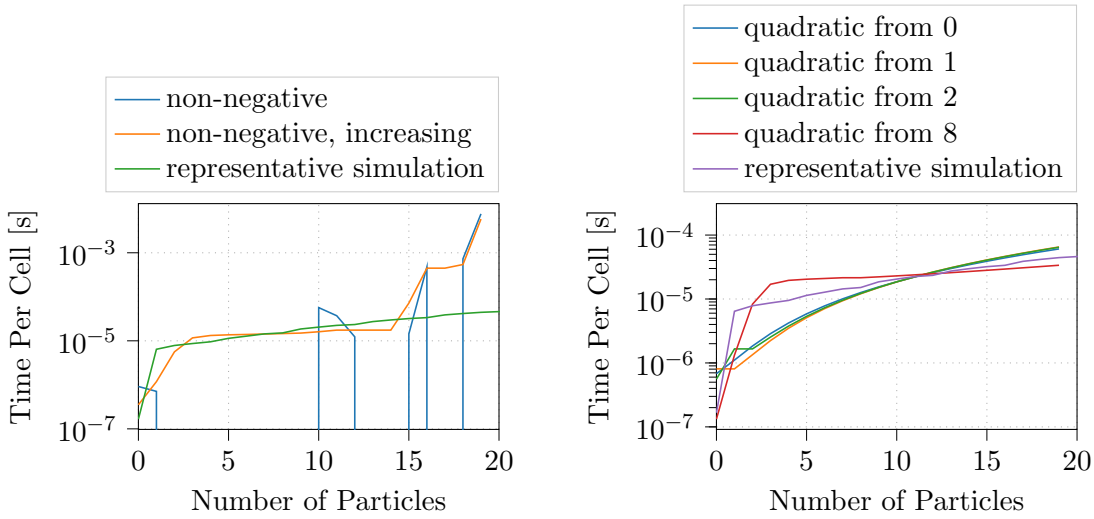
$$d = \left(t_0, t_1 - t_0, t_2 - t_1, \cdots, t_{n_{\text{max per cell}}} - t_{n_{\text{max per cell}}-1}\right)^t \tag{4.21}$$

$$d = \underbrace{\begin{bmatrix} 1 & 0 & \ddots & & 0 \\ -1 & 1 & \ddots & & \\ 0 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & -1 & 1 \end{bmatrix}}_{C} \cdot t, \tag{4.22}$$

which can be included into the matrix equation (4.20) using the inverse of the matrix C

$$T = N \cdot t \tag{4.23}$$

$$T = N \cdot \underbrace{\begin{bmatrix} 1 & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ & \ddots & \ddots & 0 \\ 1 & & \cdots & 1 \end{bmatrix}}_{C^{-1}} \cdot d, \tag{4.24}$$



(a) Reconstructed time values assuming non-negativity. If decreasing time values are allowed large jumps can be observed, as some time values are zero. If decreasing time values are not allowed jumps are reduced and no time values evaluate to zero. For large particle numbers, unrealistic jumps still occur.

(b) Reconstructed time values using quadratic interpolation.

Figure 4.8: Load estimation using inverse calculations for the coal-3M scenario on CoolMUC2. Based on time measurements for entire ranks the time needed for cells with specific amounts of particles is reconstructed. Shown is the time needed for one cell in dependence on the number of particles contained within the cell. For comparison, the runtime of representative simulations is displayed. Those simulations contain only cells with the specified number of particles.

is component-wise non-negative by solving the equation using a non-negative least-squares algorithm for d. The values of t can then be reconstructed using $t = C^{-1}d$.

As the matrix N can be quite ill-conditioned if few cells with specific particle counts exist, the measured time values can include jumps and sometimes produce unrealistic timing results (cf. Figure 4.8a). We thus introduced the assumption that the time needed for the force calculation can be modeled as a quadratic function in dependence on the particle count i

$$t_i = ai^2 + bi + c \tag{4.25}$$

and applied it to Equation 4.19. The system of linear equations then becomes

$$T_I = \sum_{i=0}^{n_{\text{max per cell}}} n_{I,i} \cdot \left(a \cdot i^2 + b \cdot i + c\right) \qquad \forall I \in \{0, ..., n_{\text{ranks}}\}\,, \tag{4.26}$$

where t_i can be written as matrix equation

$$\begin{bmatrix} t_0 \\ \vdots \\ t_{n_{\text{max per cell}}} \end{bmatrix} = \begin{bmatrix} 0^2 & 0^1 & 0^0 \\ \vdots & \vdots & \vdots \\ n^2_{\text{max per cell}} & n^1_{\text{max per cell}} & n^0_{\text{max per cell}} \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix}. \tag{4.27}$$

As this assumption, does, however, not hold true for small particle counts (cf. Figure 4.6 and Figure 4.8a, especially for empty cells), we use the quadratic interpolation only for cells with at least q particles.

$$\begin{bmatrix} t_0 \\ \vdots \\ t_{q-1} \\ t_q \\ \vdots \\ t_{n_{\text{max per cell}}} \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & 0 & \cdots & 0 & 0 & 0 & 0 \\ 0 & \ddots & \ddots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \ddots & \ddots & 0 & \vdots & \vdots & \vdots \\ 0 & \cdots & 0 & 1 & 0 & 0 & 0 \\ 0 & \cdots & \cdots & 0 & q^2 & q & 1 \\ \vdots & & & \vdots & (q+1)^2 & (q+1) & 1 \\ \vdots & & & \vdots & \vdots & \vdots & \vdots \\ 0 & \cdots & \cdots & 0 & n^2_{\text{max per cell}} & n_{\text{max per cell}} & 1 \end{bmatrix}}_{Q_{\text{partial}}} \begin{bmatrix} t_0 \\ t_1 \\ \vdots \\ t_{q-1} \\ a \\ b \\ c \end{bmatrix} \tag{4.28}$$

To ensure increasing time-values, we introduce a helper matrix in a similar fashion to Equation 4.24 which changes the solution vector to the differences of consecutive values $(t_{i+1} - t_i)$. We additionally ensure that the quadratic equation evaluated at q ($t_{\text{quad.,q}} \coloneqq aq^2 + bq + c$) is bigger than t_{q-1}.

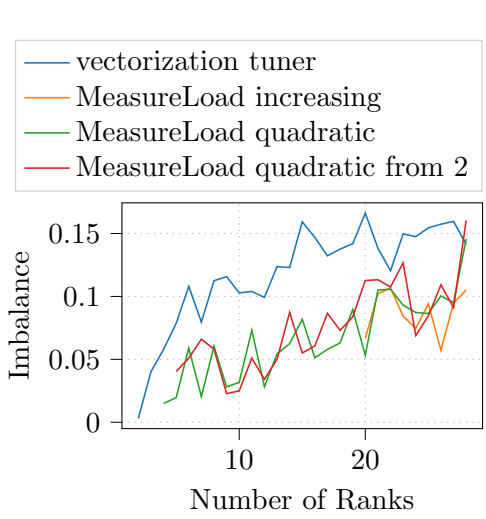
$$\underbrace{\begin{bmatrix} t_0 \\ t_1 \\ \vdots \\ t_{q-1} \\ a \\ b \\ c \end{bmatrix}}_{t_{\text{partial}}} = \underbrace{\begin{bmatrix} 1 & 0 & \cdots & 0 & 0 & 0 & 0 \\ 1 & \ddots & \ddots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \ddots & \ddots & 0 & \vdots & \vdots & \vdots \\ 1 & \cdots & 1 & 1 & 0 & 0 & 0 \\ 0 & \cdots & \cdots & 0 & 1 & 0 & 0 \\ 0 & \cdots & \cdots & 0 & 0 & 1 & 0 \\ 1 & \cdots & \cdots & 1 & -q^2 & -q & 1 \end{bmatrix}}_{C_{\text{partial}}} \underbrace{\begin{bmatrix} t_0 \\ t_1 - t_0 \\ \vdots \\ t_{q-1} - t_{q-2} \\ a \\ b \\ t_{\text{quad.,q}} - t_{q-1} \end{bmatrix}}_{d_{\text{partial}}}. \tag{4.29}$$

Solving the combined matrix equation

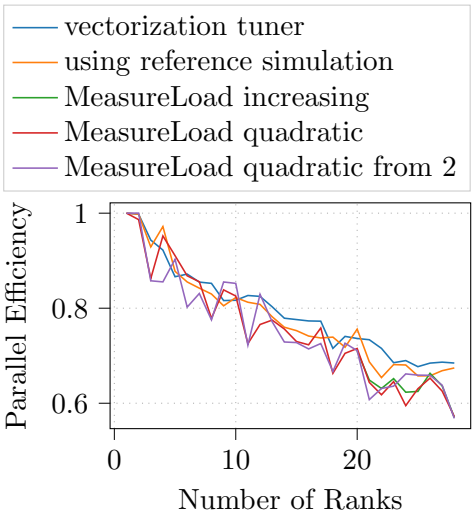
$$T = N \cdot Q_{\text{partial}} \cdot C_{\text{partial}} \cdot d_{\text{partial}} \tag{4.30}$$

for d_{partial} using a non-negative least squares algorithm, allows to get the desired solution t_{partial}.

We tested this approach using the inhomogeneous droplet coalescence scenario coal-3M that contains around 3 million particles. In this scenario, two droplets are embedded into



(a) Imbalance of the force calculation in dependence of the number of ranks used. Using the inverse problem less imbalance is observed compared to the usage of the vectorization tuner.

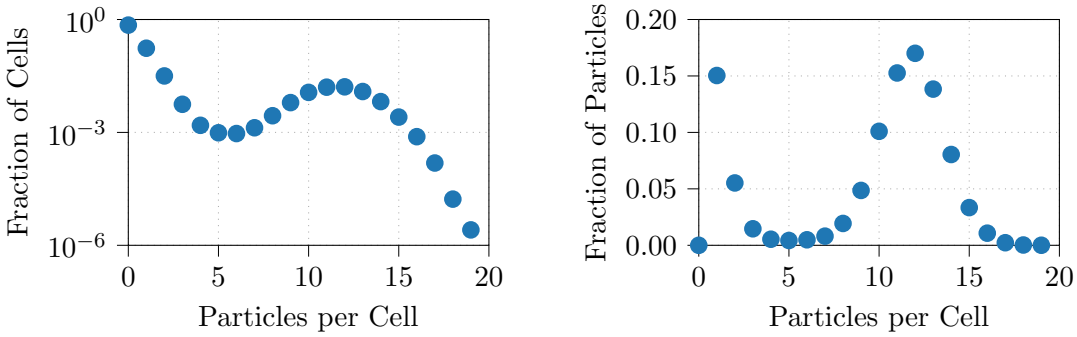


(b) Strong scaling on one node of CoolMUC2. The parallel efficiency using the inverse problem is mostly equivalent compared to the vectorization tuner.

Figure 4.9: Comparison of the actual performance of the inverse problem (MeasureLoad) and the vectorization tuner for the coal-3M scenario. For MeasureLoad, three different variants are displayed that describe different assumptions for the measured time values (cf. Figure 4.8). *increasing* indicates that cells with more particles need more time than cells with fewer particles. *quadratic* describes that the times for the cells follow a quadratic dependence based on the particle number. *quadratic from 2* only assumes the quadratic dependence starting with two particles in a cell.

a gaseous phase with a significantly reduced density (cf. Figure 4.10). In Figure 4.8 we display estimates for the time needed to calculate the interactions of a cell in dependence on the number of particles within that cell. We observe that the time predictions using unrestricted non-negative values result in multiple time values to evaluate to zero. Compared to the unrestricted case, restricting the time values to be increasing, significantly increases the quality of the solution. However, large jumps can still occur for large particle counts. Using the quadratic model jumps for high particle counts can no longer be observed.

Using the load estimation from the inverse problem, we observe a reduction of imbalances (cf. Figure 4.9) and similar performance values compared to the usage of the vectorization tuner. We further note that the different restrictions (increasing, quadratic) do not have a significant influence on the imbalance and parallel efficiency measurements, even though they heavily influence the prediction for the cell timings. A simple reason for both the jumps of the predictions and the relatively low influence of bad predictions can be found in the cell statistics of the coal-3M scenario (cf. Figure 4.10), where mainly cells with zero, one, eleven, 12 or 13 particles can be found. Only for these cells, sufficient



(a) Statistics showing the fraction of cells with a specific number of particles. Most cells ($\approx$ 70%) are empty.

(b) Statistics showing the fraction of particles stored within cells containing a specific number of particles. Cells containing 1 or 12 particles contribute the most particles to the simulation.

Figure 4.10: Cell statistics for the coal-3M scenario.

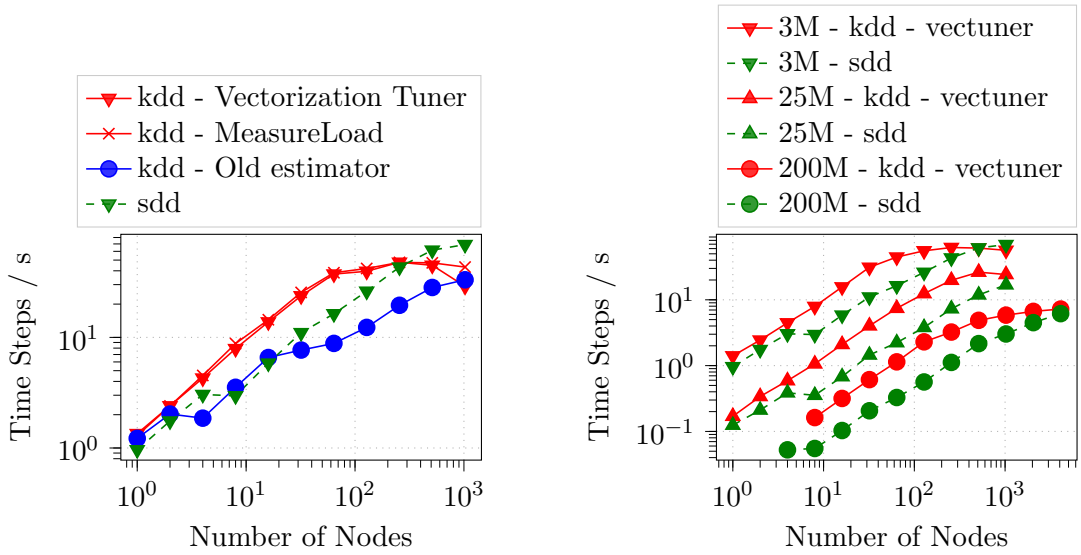
statistical data exists and the inverse problem can get meaningful results. Therefore, large jumps can occur for cells with different amounts of particles. However, these incorrect values will not influence the overall solution much, as they will only slightly change the predicted load of a subregion, as too few cells are affected by the wrong predictions.

In addition to the small runs on one node, we performed large scale strong-scaling experiments of the coal-3M, coal-25M and coal-200M scenarios on Hazel Hen (cf. Figure 4.11). We observe that the old pure-quadratic model produces very bad scalability and performs worse than a simple Cartesian decomposition, whereas the new load estimators perform reasonably well up to a certain number of nodes. Compared to the old load estimator, speedups of up to $4.3x$ and compared to the Cartesian decomposition, speedups of up to $3x$ could be observed(for coal-3M). The Cartesian decomposition can outperform the kdd using one of the new load estimators only for very high node counts. This behavior is, however, expected, as at some point very precise splittings of the inner droplet are needed to retain good load balancing and the Cartesian decomposition automatically uses better partitions in the strong scaling limit. Also note that for the coal-3M scenario this turning point is at 256 nodes, where only around 500 particles are assigned to each CPU core. For the larger scenarios, the kdd now performs consistently better than the Cartesian decomposition.

4.2.6 Load Estimation for Multiple Particle Types

Another problem with the quadratic model for the load estimation occurs when more than one particle type is used in a simulation and the computational complexity of the force calculation depends heavily on these particle types. This is the case if the two particle types are modeled with a varying number of sites or if the sites resemble different physical properties (Lennard-Jones sites, charges, dipoles, quadrupoles; cf. Figure 4.12).

The quadratic model (Equation 4.10) does not incorporate information about the used particle model, as it implicitly assumes that the costs for the force calculation between



(a) Strong scaling for the coal-3M scenario comparing the different load estimators.

(b) Strong scaling for the new vectorization tuner estimator.

Figure 4.11: Strong scaling for droplet coalescence scenarios of different size on Hazel Hen. For comparison, the Cartesian domain decomposition (sdd) is given.

particles is always the same. This becomes problematic if the two different particle types are unequally distributed within the physical domain, e.g. if one particle type is more dominant in one part of the domain and the other particle type in a different part of the domain.

We have adapted the load estimator in *ls1 mardyn* to be able to cope with up to two particle types using the vectorization tuner. Previously, the vectorization tuner measured the performance for $n_{\text{cell 1}}$ particles in the first cell and $n_{\text{cell 2}}$ particles in a second cell for $n_{\text{cell 1}}, n_{\text{cell 2}} \in [0, n_{\text{max per cell}}]$ resulting in $(n_{\text{max per cell}} + 1)^2$ measurements. Hereby, only particles of one type were inserted. A one-to-one adaption to two particle types would incorporate measuring the performance for $n_{\text{cell 1, type 1}}$ particles of component one, resp. $n_{\text{cell 1, type 2}}$ particles of component two, in cell one and $n_{\text{cell 2, type 1}}, n_{\text{cell 2, type 2}}$ particles in cell two resulting in significantly more measurements (assuming $n_{\text{cell i, type 1}} + n_{\text{cell i, type 2}} \leq n_{\text{max per cell}}$, $\frac{1}{4}(n_{\text{max per cell}} + 1)^2 \cdot (n_{\text{max per cell}} + 2)^2$). To reduce the number of measurements, we assume that the density of both components does not change much over neighboring cells and that the density change can be neglected for the load estimation. We therefore only measure the performance assuming that both cells contain the same number of particles for each component, reducing the number of necessary measurements to $\mathcal{O}n^2_{\text{max per cell}}$. A further increase in components would result in $\mathcal{O}n^{n_{\text{components}}}_{\text{max per cell}}$ necessary measurements and was therefore not implemented. We additionally note that using MeasureLoad as load estimator is not feasible for more than one particle type, as the degrees of freedom for the load estimation are too high for any meaningful results. Additionally, too many MPI ranks are needed to determine the unknowns.

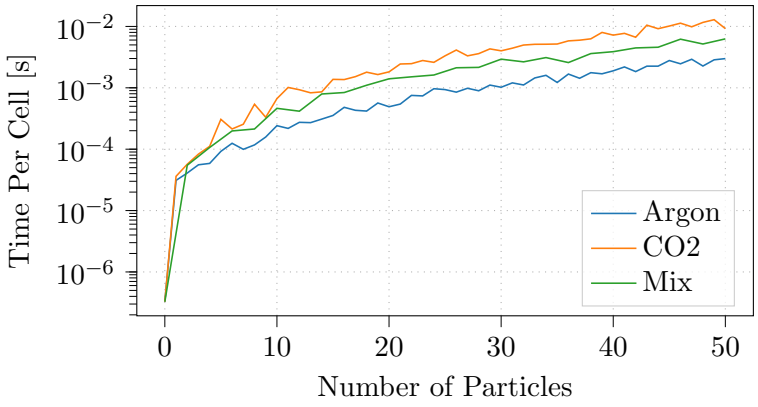


Figure 4.12: Load in dependence of the number of particles for argon and carbon dioxide (CO2). While the argon atoms are modeled using one Lennard-Jones site, each CO2 molecule consists of two Lennard-Jones sites and an additional quadrupole site at the center, resulting in a longer time to compute the interactions for the CO2 molecules. The x-values for the mix (number of CO2 molecules and argon atoms is the same) resemble the total number of particles.

Results are presented for the following two scenarios in Figure 4.13.

Arg-CO2 A scenario consisting of both argon and carbon dioxide particles. The argon particles are modeled using one Lennard-Jones-12-6(LJ) site, while the carbon dioxide particles are represented by two LJ sites and one quadrupole using the model from [128].

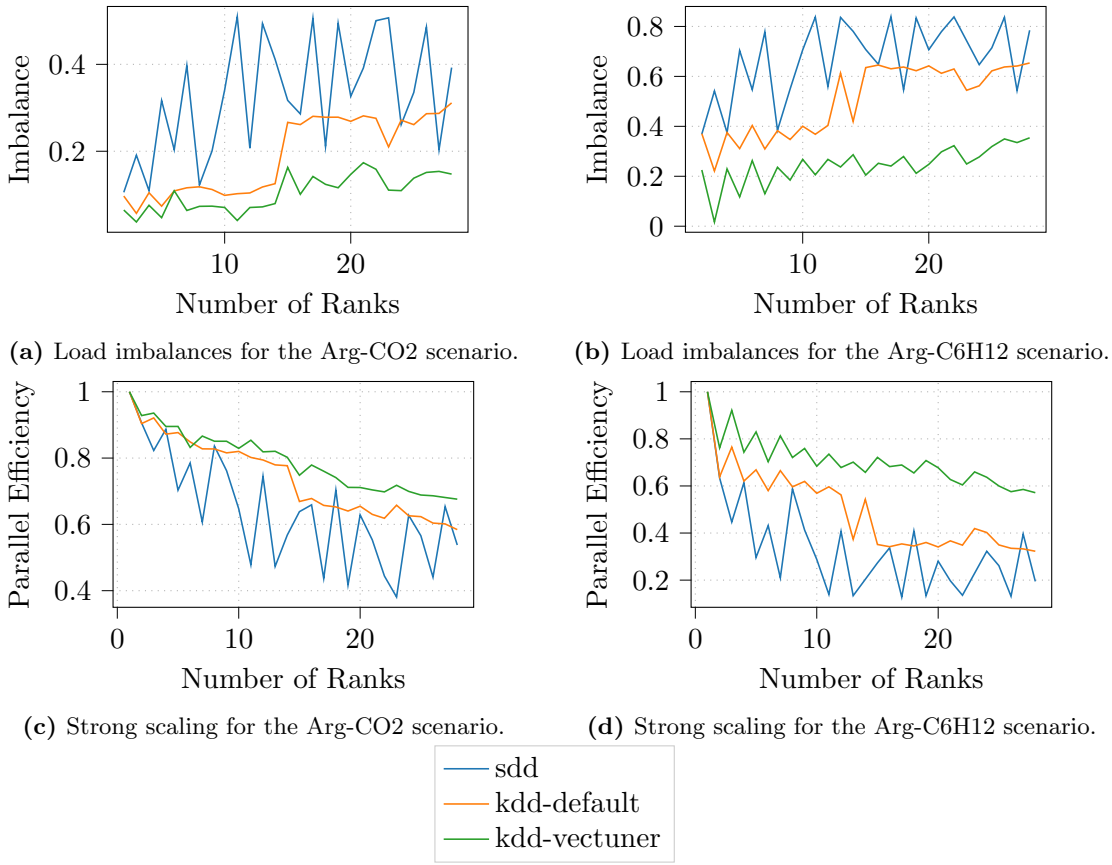
Arg-C6H12 A scenario using both argon and cyclohexane, where the latter is modeled using six LJ sites.

Both scenarios include around 1.5M particles in 180k cells with an overall uniform particle distribution. In the lower 10% of the domain (in x-direction) particles of the second type (CO2, resp. C6H12) are used instead of argon.

For both scenarios, the imbalances have been reduced by up to 50% and speedups of up to 2x could be observed compared to the old load estimator.

4.2.7 Fully Heterogeneous Load Balancing

To further test our software, we have run experiments on CoolMAC using its AMD Bulldozer and Intel Sandy Bridge partitions (cf. Figure 4.14). We can observe that using the vectorization tuner as load estimator leads to a decrease in the time to solution of around 50%, i.e., the simulation speed is roughly doubled. This showcases that our improvements using the vectorization tuner as load estimator are well suited for heterogeneous simulations using multi-component particle systems on heterogeneous hardware.



(a) Load imbalances for the Arg-CO2 scenario.

(b) Load imbalances for the Arg-C6H12 scenario.

(c) Strong scaling for the Arg-CO2 scenario.

(d) Strong scaling for the Arg-C6H12 scenario.

Figure 4.13: Load imbalances and strong scaling for the Arg-CO2 and Arg-C6H12 scenarios on one node of CoolMUC2 for the kdd using the new (vectorization tuner) and old load estimator. As comparison results using the standard Cartesian domain decomposition (sdd) are given.

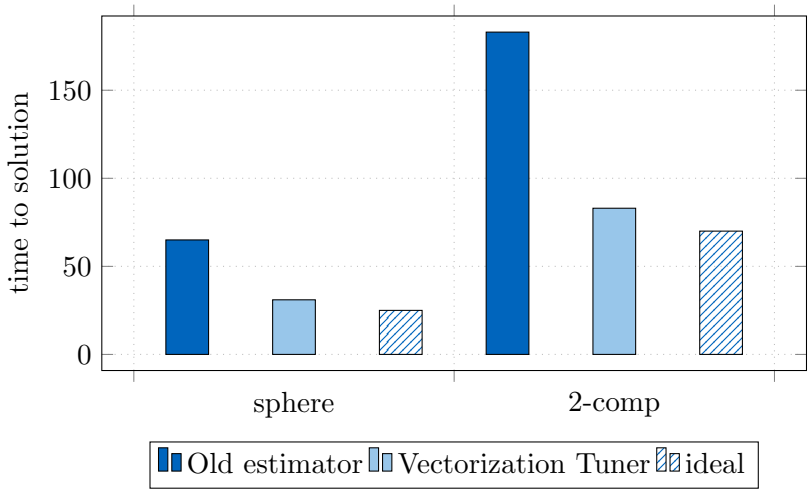


Figure 4.14: Comparison of the time to solution for two different scenarios on CoolMAC. The sphere scenario consists of a spherical higher density region in one corner of the simulation domain and a total of around 4.2 million particles in 350k cells, where only one particle type (1 centered LJ) is used. The 2-comp scenario is made up of two different phases (density and particle type), each filling half of the simulation domain with different densities. In this case, we used argon (1 centered LJ) and cyclohexane (6 LJ + 1 quadrupole). The sphere scenario was run using 64 processes on the SNB partition and 128 processes on the BDZ partition, while the 2comp scenario uses 96, resp. 384 cores. Numerical results taken from [129].

4.3 Zonal Methods

4.3.1 Motivation

In MD the easiest and most common way to handle simulations across a distributed memory system is using domain partitioning methods (cf. subsection 2.6.2) in combination with the full shell method (cf. subsection 2.6.5). This method is also used in *ls1 mardyn*. It does, however, provide a strong scaling limit, i.e., the minimal workload per process is finite, as the import region is limited to at least a cutoff sphere (cf. subsection 2.6.5). Each process thus has to import a certain amount of particles, which is a workload that does not shrink with a decreased home box of a process. To circumvent this limit, we have implemented and analyzed a couple of zonal methods in this thesis.

4.3.2 Implementation

For the zonal methods, multiple components of *ls1 mardyn* had to be modified.

Cell Structure The linked-cells structure had to be modified to support cells smaller than the cutoff. This is required for a sensible application of the midpoint and neutral territory method and supported only for those methods.

Halo Exchange The different zonal methods provide different import volumes. The halo exchange had to be modified, to properly support this behavior, as otherwise unnecessary particles were communicated.

Force Exchange Zonal methods do not calculate forces of particle pairs that span two different MPI ranks. Instead, the force is calculated on only one process and then communicated to the processes that require the information.

Traversal Compared to the full-shell method, the import volumes of the zonal methods are reduced. For the half-shell and eight-shell methods, this allows skipping specific cell interactions. For the midpoint and neutral-territory methods completely new ways to traverse the particle cells were necessary, and we thus implemented different traversals for each of the zonal methods.

For the halo and force exchange, new classes describing the import and export regions of the zonal methods have been implemented. They are used by the newly created `NeighborAcquirer` class. This class uses global communication to determine the necessary communication partners for the halo, force, and leaving particle exchanges and to specify the regions from which particles are sent. For that the `NeighborAcquirer` uses a three-stage process (cf. Figure 4.15):

1. Each process specifies the regions, for which it needs to know the communication partner. These regions are then communicated to all other processes. This step is implemented using one `MPI_Allgather` to specify the number of desired regions, followed by an `MPI_Allgatherv` to send the actually desired regions.

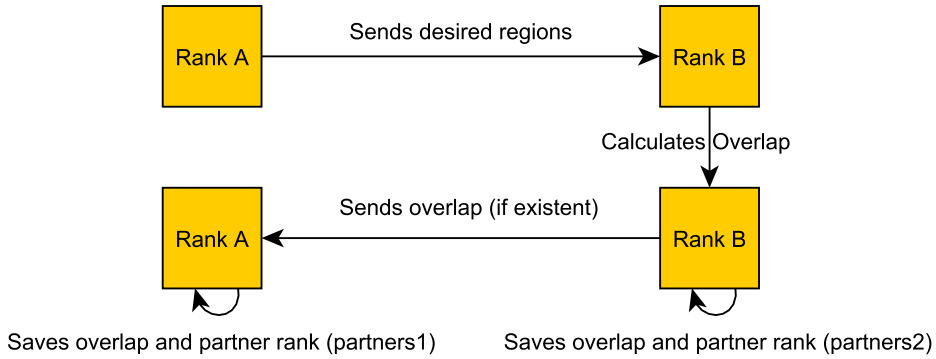


Figure 4.15: Schematics of the Neighbor Acquirer

2. Each process checks whether it owns parts of the received regions and saves the overlap(intersection) with its owned area as `partners2`.

3. The processes send the overlaps to the processes that desire them. The receiving processes will save those partners as `partners1`. This step is implemented using one `MPI_Allreduce` to first notify each process how many regions it should receive, followed by P2P communication in the form of `MPI_Isend`, `MPI_Probe` and `MPI_Recv` to send the actual data. Hereby, the receiving processes query for incoming messages until they have received as many regions as specified through the reduction operation.

After the neighbor acquisition process, each rank holds two lists of communication partners. `partners1` holds the neighbors that own the desired regions and `partners2` contains the neighbors that desire particle data from the rank. If the neighbor acquirer is called with the import region of a zonal method, `partners2` contains the list of neighbors to which halo particles have to be copied and `partners1` contains the list from which neighbors halo particles are received. For the force exchange, the role of the two lists is reversed. `partners1` now contains the list where to send particles and `partners2` contains the list of particles to receive. The exchange of leaving particles is independent of the zonal method, as particles can move in any direction. For a correct acquisition of the required neighbors, the full shell import region has to be used. If knowledge about the maximal speed of the particles is existent, this region can, however, be significantly decreased, s.t., the number of communication partners can be reduced (if the size of a neighbor is smaller than the cutoff radius). For the leaving exchange, `partners1` represents the sending and `partners2` the receiving list.

Compared to the previous implementation of the neighbor exchange, we now directly communicate with the respective neighbors. In the original code, the neighbor communication contained three steps in which a process always only communicated with its face-neighbors. This communication pattern is now no longer possible, as a process now no longer communicates with all face-neighbors. One advantage of the new approach to directly communicate is the reduction from three steps to now only one step, which

reduces the time for the communication, as we only communicate small messages and are mostly latency bound.

Additionally, in the previous implementation, halo and leaving particles were always sent together to each process. This was possible because the side length of a process was always guaranteed to be at least $2r_{\text{cutoff}}$ and a particle that enters one process cannot be in the halo of any other process except the sending one or its neighbors. For zonal methods, we want to lift the subdomain size requirement and therefore communicate first the leaving particles and then the halo particles. This way, the subdomain size can be arbitrarily small.

4.3.3 Results

Once the aforementioned modifications were implemented, we tested them on CoolMUC2 (for the cluster description see section B.1).

First, we checked the performance on a very small example consisting of a cubic domain with a side-length of slightly more than two cutoff radii (exact factor $\frac{7.1}{3.5}$). As *ls1 mardyn* restricts its domain to be bigger than two cutoff radii, because of its periodic boundary conditions, it is almost the smallest possible example that can be simulated. Using this scenario, the strong scaling behavior of the different methods and their potential gains can be estimated depending on the particle density.

For small particle densities, zonal methods with fewer cells are beneficial (cf. Figure 4.16), as having more cells results in overhead due to an increased amount of cells that have to be traversed during the force calculation. In addition, having more cells results in fewer particles per cell and thus less potential for vectorization.

However, more cells provide better overall parallelizability, which allows using more processes for the same simulation, and thus an overall smaller time-to-solution can be observed when many processes are used. This is especially apparent if the particle density becomes higher and the overhead of having additional cells shrinks in relation to the time needed for the actual calculation of the forces.

When comparing the different zonal methods, one can observe that the eighth shell, half shell, and neutral territory (1 cell per cutoff) methods need roughly half the time compared to the full shell method if only the time for the force calculation is observed. Because large enough densities result in a much more compute-intensive force calculation, the midpoint and neutral territory methods will also outperform the full shell method by a factor of two if more than one cell per cutoff is used.

The force calculation alone does, however, not provide the complete picture, as other parts of the simulation, especially, the communication, are influenced by the choice of the zonal method. This influence is especially visible for a small number of particles (cf. Figures 4.17, 4.18). On the one hand, the half shell method is not able to outperform the full shell method. The eighth shell and neutral territory method (1 cell per cutoff), on the other hand, are able to outperform the full shell method even for relatively small particle numbers. This difference can be explained, as the latter two methods use a significantly reduced import region and thus also need to communicate with fewer neighboring processes.

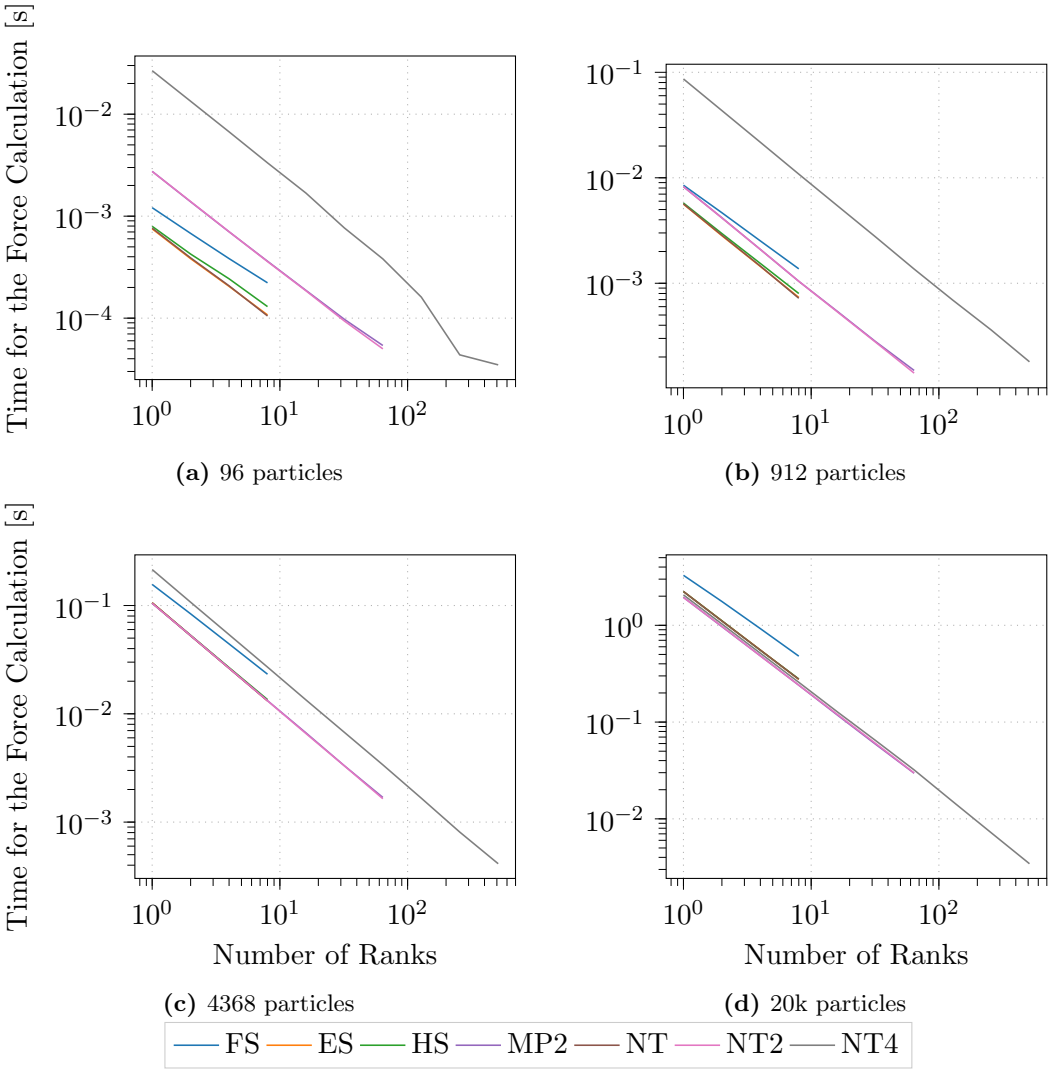


Figure 4.16: Strong scalability of the force calculation for different implemented zonal methods (Full Shell, Eighth Shell, Mid Point, Neutral Territory) using a homogeneous scenario with fixed domain size ($\approx 2\,r_{\text{cutoff}}$) and varying density and number of ranks. The number behind the zonal method indicates the number of cells per cutoff. Note that only the time taken for the force calculation is displayed. These measurements were performed on CoolMUC2.

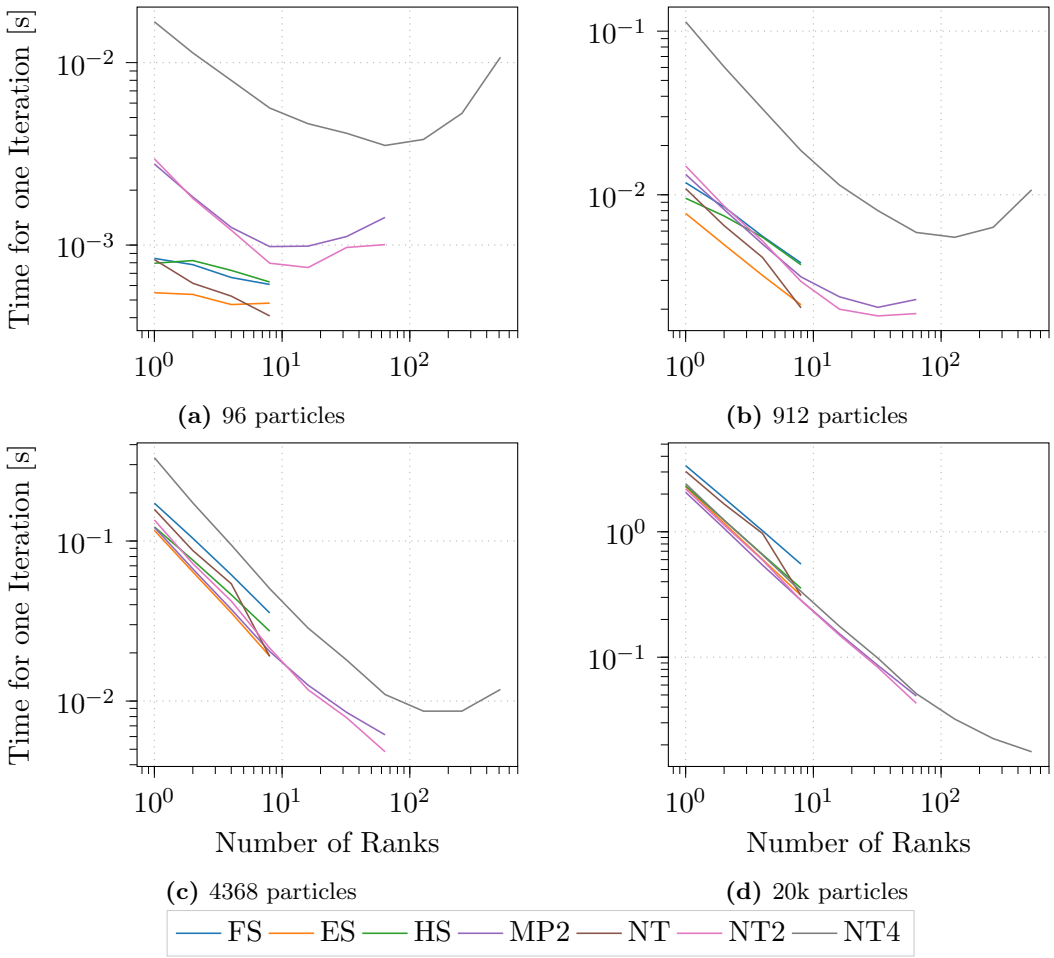


Figure 4.17: Strong scalability of the different implemented zonal methods (Full Shell, Eighth Shell, Mid Point, Neutral Territory) for a homogeneous scenario with fixed domain size ($\approx 2r_{\text{cutoff}}$) and varying density and number of ranks. The number behind the zonal method indicates the number of cells per cutoff. These measurements were performed on CoolMUC2.

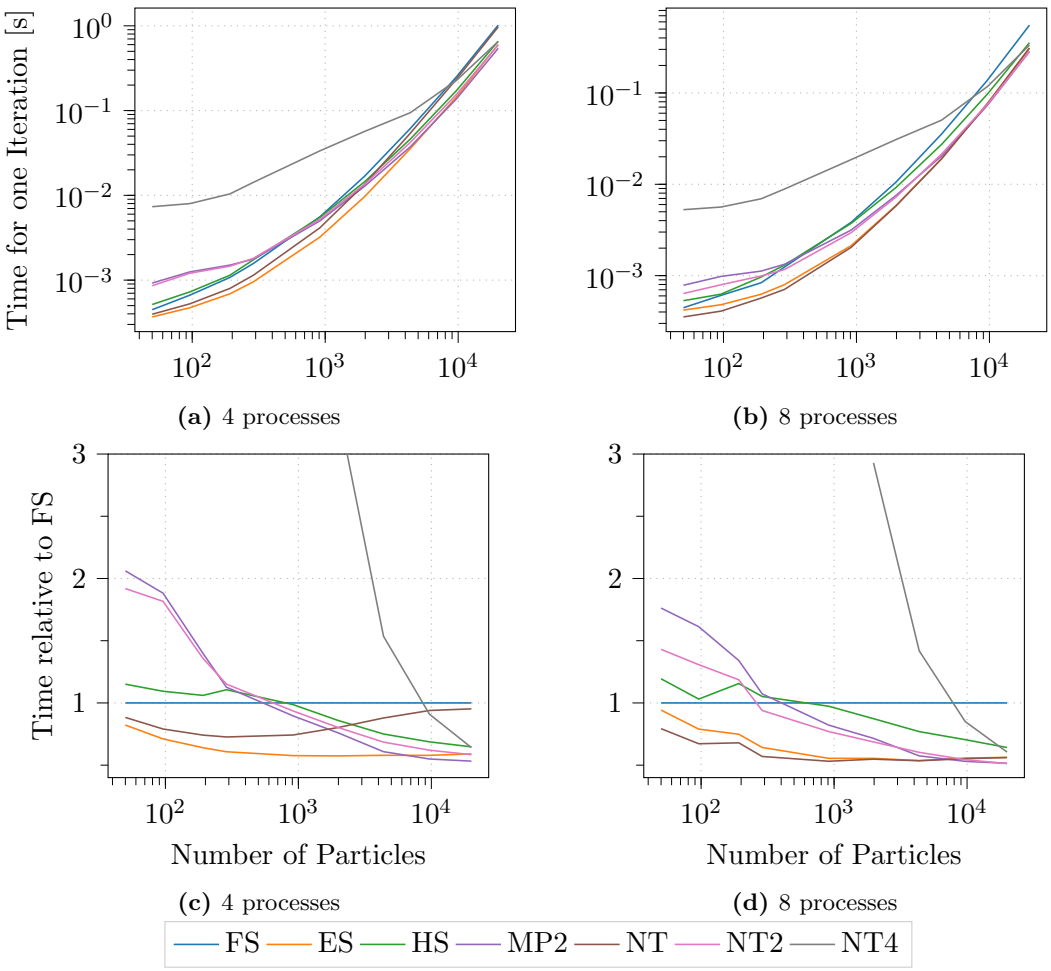


Figure 4.18: Performance of the different implemented zonal methods (Full Shell, Eighth Shell, Mid Point, Neutral Territory) in dependence of the density for a homogeneous scenario with fixed domain size ($\approx 2\,r_{\text{cutoff}}$). The number behind the zonal method indicates the number of cells per cutoff. On the top, the absolute time per iteration is given, while on the bottom the times compared to the full shell method are provided. These measurements were performed on CoolMUC2.

The maximal measured speedup could be observed for the scenario with 20k particles. Hereby, the neutral territory method (4 cells per cutoff, 256 processes) outperforms the full shell method by a factor of 17.6, as the latter is only able to utilize up to 8 processes. Note that before this thesis, the full shell method only allowed the usage of 1 process, as $2 \times 2 \times 2$ cells were required for each process. Thus, compared to the state before this thesis, a speedup of almost 130x is achieved.

4.3.4 In Short

The eighth shell method is mostly outperforming all other methods and runs up to 2x faster than the full shell method because no duplicated calculations are performed. Only for very high particle densities, it is worthwhile to use multiple cells per cutoff radius and the neutral territory and midpoint method provide a clear advantage as they allow for deeper parallelization.

4.4 MPI optimizations

In this section, we describe further optimizations regarding the MPI communication, which include non-blocking collectives and non-blocking p2p communication.

4.4.1 Non-blocking Collectives

One of the main problems using *ls1 mardyn* in the strong-scaling limit is the repeated use of collective communications across all MPI processes. These operations imply synchronization points and require a significant amount of time for large-scale simulations in which many MPI processes partake. To circumvent these restrictions, we have introduced an easy-to-use way to handle non-blocking collective operations, which is based on the reuse of values from the previous time step instead of the current values. This reuse is of course only possible if the communicated values do not change significantly over one time step. The reuse then allows overlapping collective communications throughout an entire time step, because the current values are only needed in the next time step. To identify the correct values from a previous time step, each call to a collective communication is identified by a unique key, e.g., the calculation of the temperature could use key 20, while the communication within some plugin could use key 2002. On the first communication with a specific key, a blocking collective call is performed to get initial values for the call. The second call to said communication will start a non-blocking communication, while the values from the first communication are returned. The third and all following calls will finish the non-blocking calls from the previous time step, return the associated values and initialize a non-blocking call to the next time step.

To allow these modifications seemlessly, we have restructured the code concerning the collective operations. Previously, the collective operations were all performed by the DomainDecompMPIBase class, which handled all functionality common to the DomainDecomposition (Cartesian decomposition) and KDDecomposition classes, while the class DomainDecompBase provided a dummy interface for collective communications.

We have extracted the methods using the strategy design pattern by defining a common interface for the collective operations (CollectiveCommunicationBaseInterface). We have further moved the old, blocking communication pattern and the dummy methods into new classes (CollectiveCommunication and CollectiveCommunicationBase) and implemented the new, non-blocking behavior in the class CollectiveCommunicationNonBlocking. The latter stores multiple instances of the class CollectiveCommunicationSingleNonBlocking, which implements the actual functionality of the overlapped communication and reuses the behavior of CollectiveCommunication by inheriting from it.

For MPI-parallel programs, either the blocking (CollectiveCommunication) or non-blocking (CollectiveCommunicationNonBlocking) variants of the collective communication are then used by DomainDecompMPIBase, which acts as an adapter for the collective communication and retains the original interface for the collective operations.

A UML-like diagram of the related classes can be found in Figure 4.19.

Speedups through global collectives can be achieved for two reasons. First, load imbalances within the different phases between two collective operations can be reduced. And second, communication and computation can be overlapped, thus the time needed for the global data exchange is hidden behind the computation.

To showcase the first ability to overlap collectives, we created a scenario with an artificial load imbalance by letting one process sleep before and one after an `MPI_(I)Allreduce` call. The corresponding trace can be found in Figure 4.20, in which one can observe that the length of one time step can be significantly reduced using overlapping collectives, as the two sleep calls can be overlapped. For this scenario, the load imbalances are exaggerated for better visualization. In normal use cases, these imbalances are, e.g., introduced by

CPU speed fluctuations Typical CPUs provide frequency scaling. While it is intended for energy savings, the frequency sometimes is reduced because of overheating of the system or if the overall system draws too much power. The CPU frequency is thus not uniform across processes over time and can occur on some processes before and on some processes after a reduction operation, thus introducing load imbalances.

Background tasks While background tasks are typically reduced on HPC systems compared to normal workstations, they still exist and can lead to load imbalances as they can occur on the systems at different times and frequencies.

Different scaling of plugins and force calculation While the force calculation scales locally with $O\left(\rho^2\right)$, most plugin calls scale with $O\left(\rho\right)$ (where ρ is the density). If the force calculation is perfectly balanced, most likely a load imbalance exists for the plugins and vice-versa. A common load balancing is not possible if blocking communication is used. If the plugin or thermostat calls do not need much time compared to the force calculation, this effect is, however, only small.

The first two reasons typically average out over the long run and are not visible if averaging is used or if many particles are present on each MPI rank. In the strong scaling limit, i.e., if the time between different global communication steps is small, the load imbalances do, however, tend to become visible.

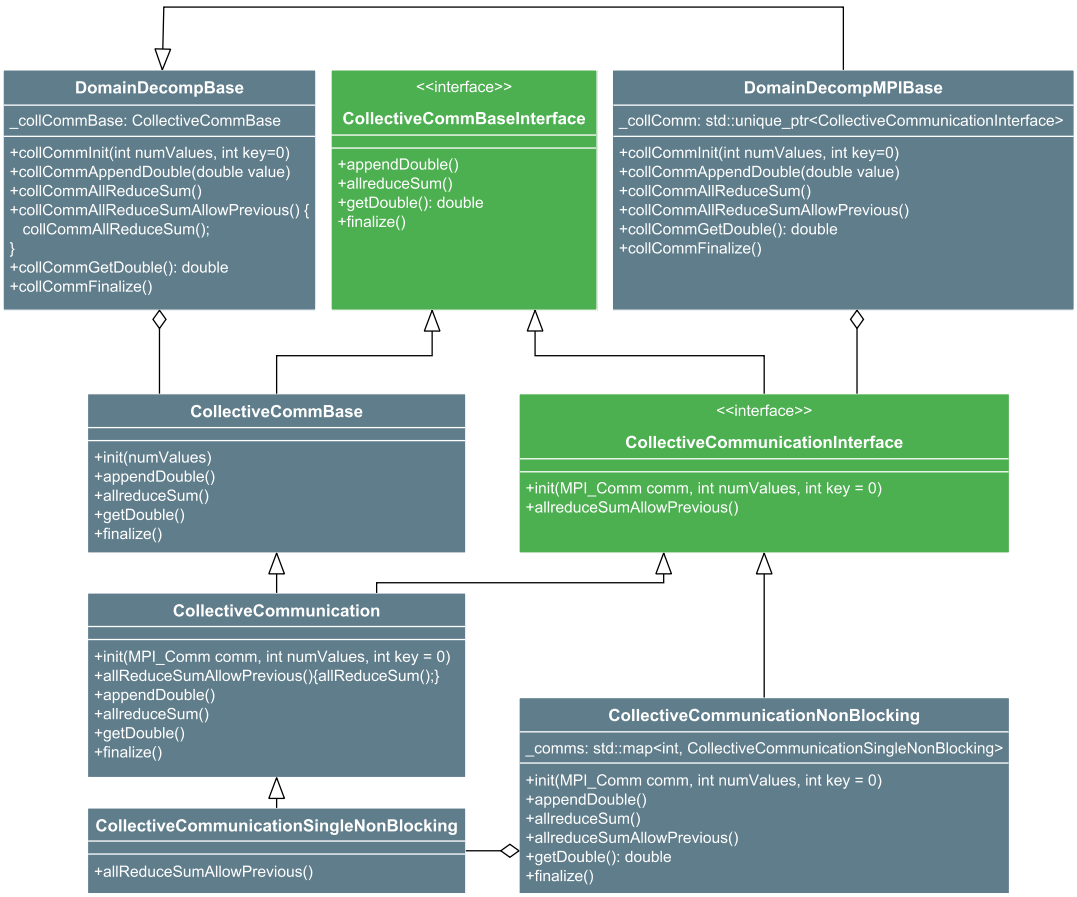
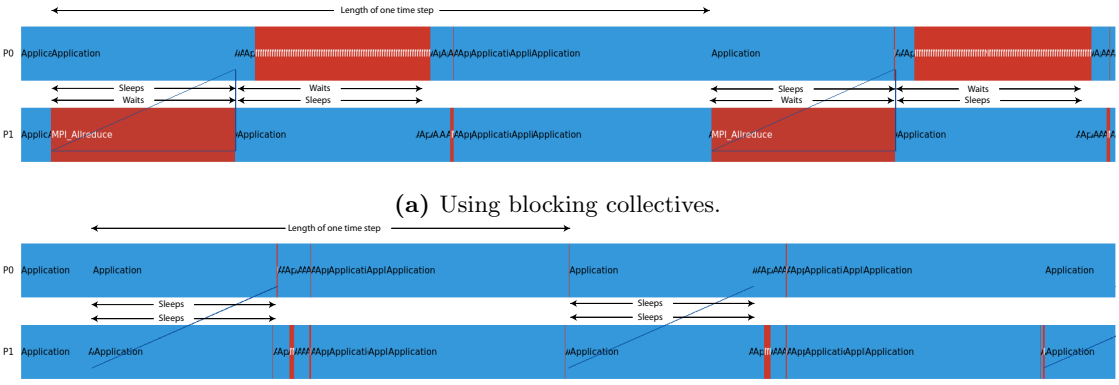


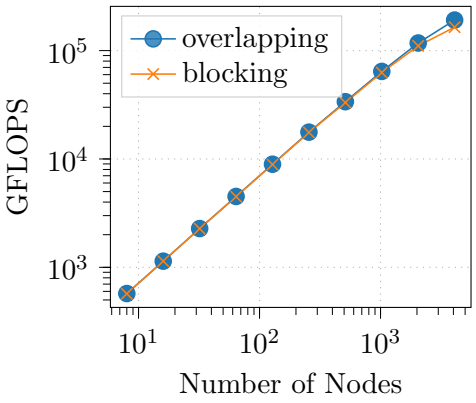
Figure 4.19: UML-like diagram for classes related to collective communication. DomainDecompMPIBase acts as an adapter for the CollectiveCommunicationInterface, which is realized by either CollectiveCommunication or CollectiveCommunicationNonBlocking. The latter uses multiple CollectiveCommunicationSingleNonBlocking instances that are stored in an `std::map` and identified using a key.



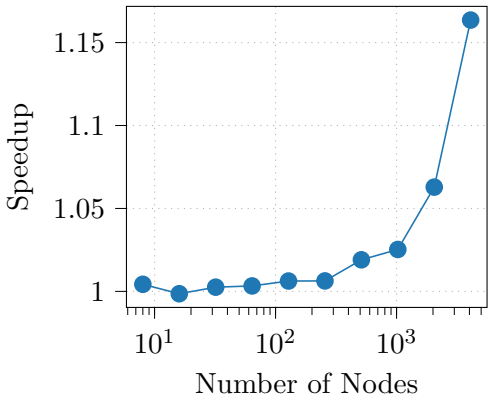
(a) Using blocking collectives.

(b) Using non-blocking collectives.

Figure 4.20: Comparison of the traces of two simulation runs with and without the use of overlapping collectives. In this test case, artificial load imbalance is introduced (P0 sleeps before the call to MPI_(I)Allreduce and P1 after the call). Using overlapping collectives, the sleeps can be overlapped and the time for one iteration (distance between the start of two sleeps) is significantly reduced. The diagonal lines indicate the collective operations. The load imbalances in this example are chosen very high (around the time needed for one time step) for better visualization.



(a) Flops.



(b) Speedup overlapping vs blocking.

Figure 4.21: Comparison of overlapping and blocking collectives for a scenario of 64 million benzene molecules, which are generated on a grid. The simulation was executed on SuperMUC Phase 1 using 2 MPI ranks per node. For 4096 nodes a speedup of 16% could be measured when using overlapping collectives compared to blocking collectives.

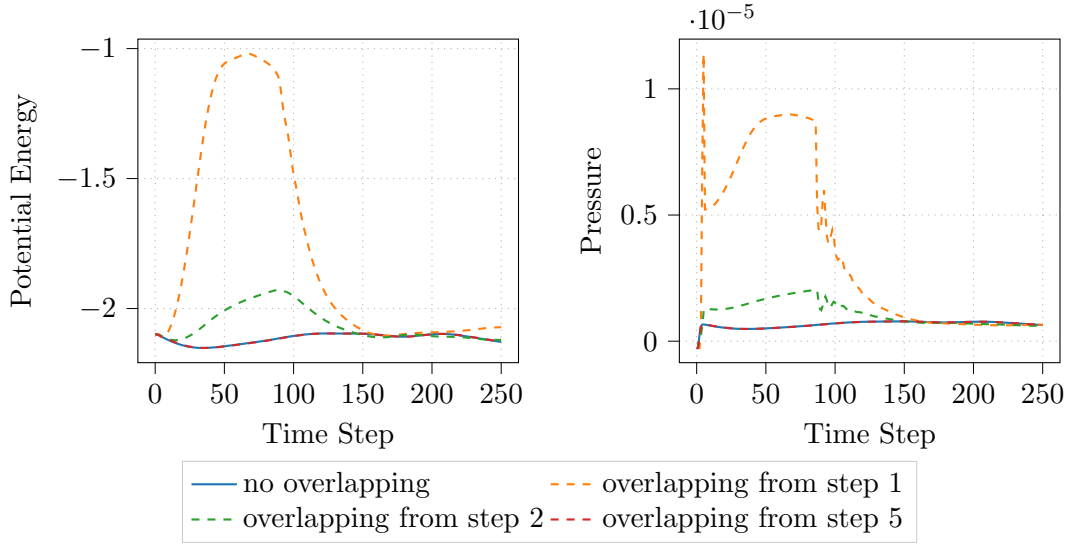
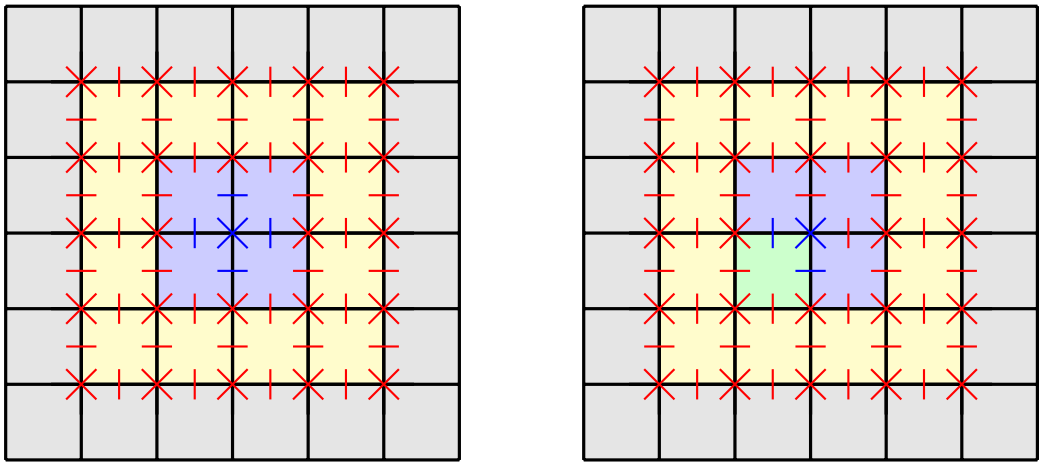


Figure 4.22: Comparison of macroscopic quantities depending on the start of global overlapping collectives. This test uses the argon example provided in the *ls1 mardyn* repository.

In addition to this simple test, we tested the use of overlapping collectives on a real-world scenario using 64 million benzene molecules (cf. Figure 4.21). This scenario showcases the second reason for global collective, as we have disabled all plugin calls and only one collective call is issued in every time step. For this scenario, speedups could be observed starting with 512 nodes (2% speedup). As blocking collectives tend to require more time with an increase of processes, these speedups increase with an increased number of nodes. The highest speedup of 16% could be observed when using 4096 nodes.

The described use of non-blocking collectives can actually change the physical results because values are used from a previous time step. This can, e.g., change the behavior of a thermostat. We have therefore checked an exemplary scenario for its change in global variables (cf. Figure 4.22). We hereby noticed significant deviations if overlapping collectives are used compared to a simulation without the use of overlapping collectives. Reasons for this deviation are the initial deviation of the temperature of the system ($2.5 \cdot 10^{-5}$) from the target temperature of the thermostat ($6 \cdot 10^{-4}$) and the sharp increase of the pressure at the start of the simulation. To prevent these deviations, we included an adjustable offset that defines after how many time steps overlapping collectives are allowed in a simulation. Using this offset, no significant deviations from the actual simulation can be observed. We have therefore shown that *ls1 mardyn* can use overlapping collectives and reuse macroscopic values from a previous time step without significant changes to the simulation. To our knowledge, *ls1 mardyn* is the first MD code using overlapping collectives in such a way.



(a) General ability to overlap the force calculations.

(b) Ability to overlap when using the c08 (c04 in 2d) base step. Fewer calculations can be overlapped, as all calculations that are associated with a specific base cell are calculated at once. In this case, only interactions "belonging" to one base cell (green) can be calculated.

Figure 4.23: Overview of the ability to overlap the force calculation. Force calculations (lines) that can be overlapped with the communication are displayed in blue. Force calculations that depend on the halo cells (gray) or cells at the boundary of the domain (yellow) are marked in red and cannot be overlapped with the communication. They can only be calculated after the particles are communicated. At least 4x4x4 cells are required, s.t., any calculations can be overlapped.

4.4.2 Overlapping P2P Communication

Overlapping communication in *ls1 mardyn* has already been employed for peer-to-peer communication, i.e., the exchange of particles. Hereby, the communication of particles is overlapped with their unpacking from the MPI-buffers and their insertion into the particle container.

We further tested the overlap of p2p communication with the actual force calculation. Hereby, force calculations that are not affected by the particle exchange routines are performed while the communication takes place. On the cell-level, force calculations can only be performed after the particle exchange, if particles are inserted into one of the partaking cells. If a force calculation includes halo cells, then this force calculation can only be performed after the particle exchange. Particles that move from one process's subdomain to another's subdomain are, however, also inserted into cells that lie on the boundary of a process's subdomain. Force calculations that include those cells are therefore also not possible to overlap with the communication. A visual representation of which force calculations can be overlapped and which cannot is displayed in Figure 4.23.

We tested the overlapping p2p communication for the same scenario as in Figure 4.21, where we did, however, not notice any significant speedup. One reason for this is that for the p2p communication to become noticeable, the subdomains of each process have to

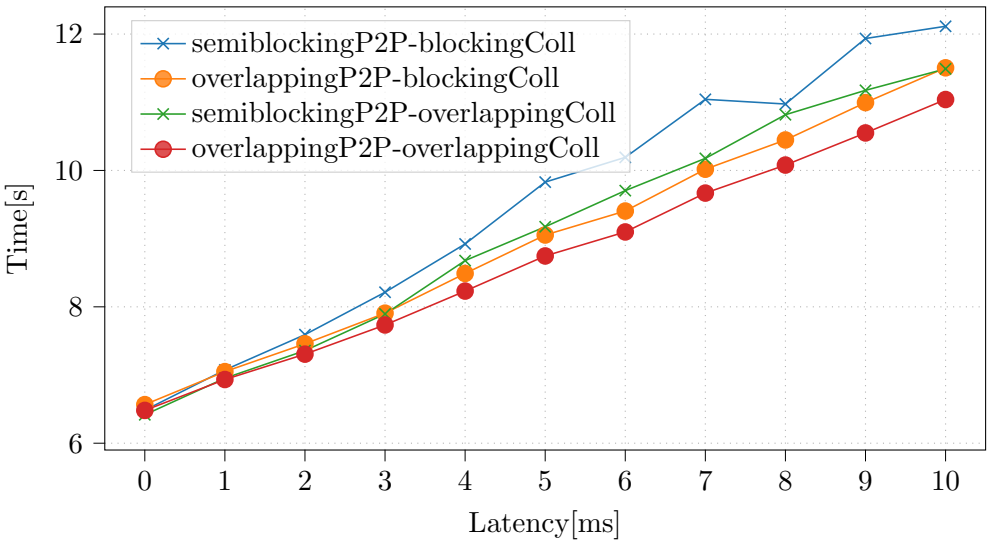


Figure 4.24: Time needed for a simulation with 8 MPI ranks in dependence of the latency between the ranks. The simulated system contains 1.7M single-centered molecules (LJ-12-6) in 36x36x36 cells. The simulation contains 10 time steps which were executed on a machine containing an AMD Ryzen 7 3700X 8-core processor. SemiblockingP2P indicates that the p2p communication is only overlapped with the unpacking and insertion of the particles, while overlappingP2P means that the particle exchange is overlapped with the force calculation. A blocking base line for the P2P communication is not displayed, as the particle exchange in *ls1 mardyn* first initiates all send calls (ISend), after which the receive calls are triggered (IRecv). For blocking communication, the sends and receives would have to be always matching, which in this case is never worthwhile and also error-prone to implement.

be relatively small, s.t., the communication takes a significant amount of time. For very small subdomains, the ability of overlap is, however, relatively small, as most interactions can only be calculated after the communication took place, as information of halo or boundary cells is required for their calculation. One possibility, where the overlap can be useful is given, is if the fabric is relatively slow and the overlap with the buffer extraction and particle insertion is not enough for the communications to finish. We, therefore, continued to investigate under which circumstances speedups can be expected and tested the p2p communication on a workstation with artificially introduced latency. To introduce the latency we employed the netem (Network Emulation)[5] utility provided in the Linux kernel using `tc qdisc add dev lo root netem delay 1ms`. For this delay to become visible for a program using OpenMPI, it has to be started with the TCP protocol, i.e. `mpirun -mca btl self,tcp`, thus disabling the different shared-memory byte transfer layers (BTL), such as *vader* or *sm*, using OpenMPIs modular component architecture (MCA).

The test showed that the overlap of p2p communication with the force calculation is profitable for high latencies and is not very useful for low-latency networks (cf. Figure 4.24). This test also showcases the usefulness of overlapping collectives for high-latency networks in conjunction with overlapping p2p communication.

We can thus conclude that overlapping p2p communication is useful on slow networks, especially over the internet, as latencies above 1ms are normally not to be expected in local networks. On local networks, low latencies might also occur if the network is flooded by messages, as the network switches might take some time to process the requests.

[5] `https://wiki.linuxfoundation.org/networking/netem`

5 Auto-Tuning for ls1 mardyn

5.1 Disclaimer

Most of the results presented in this chapter have previously been published in [130]. The focus of this work is the integration of the *AutoPas* library into *ls1 mardyn* and the required interface design of the library. The actual implementation of the *AutoPas* library itself is the work of Fabio Gratl and described in [131, 132].

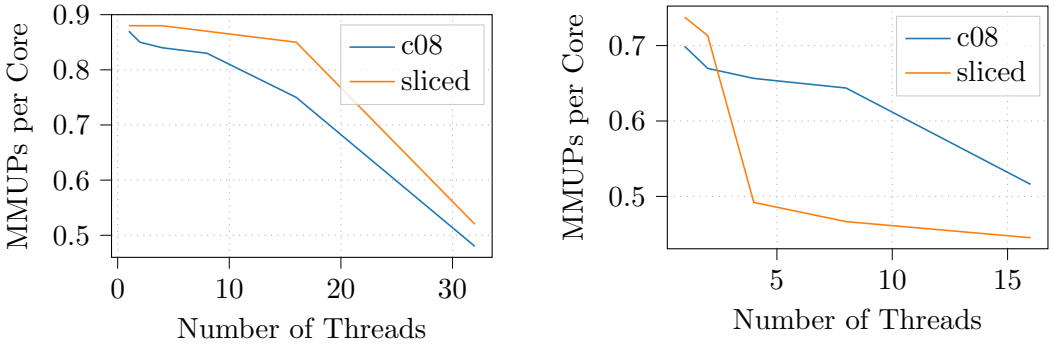
5.2 Motivation

During simulations published in [5], we have shown that, depending on the simulated scenarios and the number of used threads, different shared-memory parallelization strategies (see subsubsection 3.1.1.3) provide better performance when using *ls1 mardyn*. Figure 5.1 displays this for two scenarios, one with and one without a uniform particle density.

For the heterogeneous scenario, the sliced traversal provides better performance if one or two threads are used, while the c08 traversal outperforms the sliced traversal if more than two threads are used. The reason for this behavior lies in the dynamic scheduling of the c08 traversal, which allows to schedule the computational load evenly throughout the threads. The sliced traversal instead splits the domain statically into equally sized chunks. Each thread is then assigned one chunk, providing low-overhead scheduling with better memory access patterns at the cost of non-existing load balancing. The sliced approach, therefore, performs better if only one thread is used. Because the particle distribution is symmetric, the sliced traversal outperforms the c08 traversal for two threads, as it splits the domain into two subdomains with equal load.

For the homogeneous particle distributions, the sliced traversal will, however, always outperform the c08 traversal, as the dynamic scheduling cannot provide any benefit.

As most users of *ls1 mardyn* do not know the details of the implemented traversals and other algorithms, we have decided to automate the selection of these algorithms. We, therefore, concluded to extract the force calculation from *ls1 mardyn* into a new library called *AutoPas*. The library allows auto-tuning the traversals and other parameters of the simulation. Through the extraction into the new library, we were able to enable auto-tuning not only for *ls1 mardyn* but also for other short-ranged N-body simulation codes.



(a) Homogeneous scenario.

(b) Inhomogeneous scenario.

Figure 5.1: Qualitative comparison of the performance of the c08 and sliced traversal for a homogeneous and an inhomogeneous scenario. Simulations executed on one node of SuperMUC Phase 1 using one thread per core. Numerical results taken from [5].

5.3 Related Work

In short-ranged N-body simulations, there exist multiple codes, each targeting one or more application fields. These simulation packages have in common that they use some algorithms to calculate the forces between the particles. In almost all codes, the algorithms are implemented from scratch without relying on an external library, and thus their implementations, used data structures, and algorithms all look a bit different.

LAMMPS (Large-scale Atomic/Molecular Massively Parallel Simulator), a classical molecular dynamics code, e.g., uses Verlet lists (cf. subsection 2.2.4) as data structure [75], while *GROMACS* (Groningen Machine for Chemical Simulations) uses a variant of neighbor lists that works on clusters of particles [133] (cf. Figure 5.2) thus providing better mapping to SIMD (vectorization for CPUs) or SIMT units, i.e., GPUs. *ls1 mardyn*, in contrast, uses linked cells (cf. subsection 2.2.3) which allows for good vectorization (cf. subsubsection 3.1.1.1), however at the cost of additional distance calculations compared to Verlet lists.

In addition to the underlying data structures, the codes provide different implementations for the force calculation. For the traversal, *LAMMPS* provides different accelerator packages using CUDA, OpenCL, ROCm/HIP, OpenMP threads, hand-optimized code, generically optimized code, and code using the KOKKOS library, which promises performance-portable code [1]. *GROMACS* also provides multiple implementations using OpenMP, CUDA and OpenCL [134]. In contrast to *LAMMPS* and *GROMACS*, *ls1 mardyn* is not modified to work on GPUs. It does, however, provide multiple different traversals (c04, c08, sliced) using OpenMP and also provides a parallelization using Quicksched's tasking mechanisms [135, 136].

All of these options are typically chosen at compile-time or program startup, and the choice of the correct, most performant parameters is necessary. These parameters might

[1] `https://docs.lammps.org/Speed_packages.html`

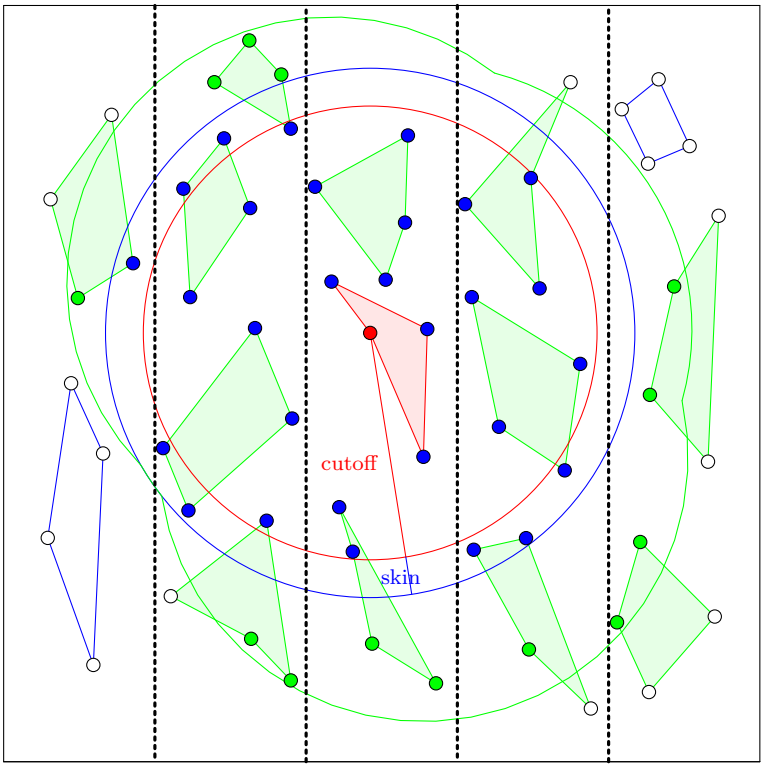


Figure 5.2: Verlet clusters. Instead of calculating neighbor lists for each particle, multiple particles are conglomerated into clusters, and lists are constructed for the clusters. Shown are the cutoff-sphere (red circle) and the cutoff+skin-sphere (blue circle) around the red particle. For the red cluster, all clusters for which at least one particle lies within the combined cutoff+skin-sphere (greed) of any particle in the red cluster are colored green. Particles are colored depending on their relation to the red particle. If they lie within the cutoff+skin-sphere, they would be in the interaction list of the red particle for a normal neighbor-list approach and are colored blue. If they are normally not interacted with the red particle, they are left uncolored.

depend not only on the used hardware but also on the simulation setup. For this purpose, the codes do provide some comfort by choosing and tuning a few things. *GROMACS*, e.g., uses auto-tuning to choose an appropriate load balance between the CPU and GPU[2]. Additionally, the codes can detect the used hardware at compile-time and select a few parameters wisely. Expert knowledge is, however, still required, as many parameters are not chosen automatically.

There exist some libraries that can help with that problem by providing dynamic auto-tuning functionalities. One example for such a library is *Active Harmony* [137], which uses a server-client infrastructure to explore a search space. Hereby, the clients will sample specific points of the search space according to the sample points provided by the server. Libraries, like *Active Harmony*, are, however, not actively used in MD.

5.4 Design and Implementation

AutoPas is the first library to enable dynamic auto-tuning for particle simulations. We developed *AutoPas* to achieve multiple purposes:

- Reducing the expert knowledge that is required to use particle simulation software most efficiently. This is achieved by auto-tuning over a range of parameters and choosing the parameters with the best performance automatically.
- Easing the writing of efficient particle codes by providing an easy-to-use library that automatically produces performant code.
- Enable speedup of existing simulation codes through dynamic auto-tuning by allowing the selection of different algorithms over time or by different (MPI-)processes of a running simulation.

This is achieved by the dynamic auto-tuning and selection of the following parameters:

Particle Container The particle container handles the storage of particles. We have implemented containers using linked cells, Verlet lists, and Verlet cluster lists. In addition, we implemented a direct sum container. It does not perform sorting and interacts a particle with every other particle. The direct sum container thus offers worse scalability compared to the other containers.

Traversal The traversal defines how particles are traversed for the force calculation. For each particle container, a different set of traversals is applicable. The traversals mainly differ by the way race conditions are circumvented. While some use coloring, others use locks. An overview of the different traversals is depicted in Table 5.1.

Data Layout The data layout describes the way particles are stored for the force calculation. We hereby differentiate between SoA and AoS storage (cf. Figure 3.1).

[2] `https://manual.gromacs.org/documentation/5.1/ReleaseNotes/performance.html#improved-performance-auto-tuning-with-gpus`

Other Parameters In addition to the parameters listed above, *AutoPas* allows tuning over some parameters of the traversals. It can decide whether the traversals should use newton3 for the force calculation, which reduces the number of needed force calculations by half, but requires proper handling of the parallelization options to circumvent data races, e.g., by introducing coloring.

We call a combination of the different algorithm parameters configuration.

AutoPas consists of a modular design that enables the dynamic switching of the different options. This design is also very extensible, s.t., it is easy to add new options and test them against the existing ones.

For a user of the *AutoPas* library, this flexibility is hidden behind the `autopas::AutoPas` class by applying the facade pattern, s.t., an easy usage can be guaranteed. This class represents the main interaction point for a user of *AutoPas*, through which almost all user interactions with *AutoPas* are performed. The main interaction points of the class are:

- The addition of particles via the `autopas::AutoPas::addParticle()` function.
- Accessing the particles through iterators. For this purpose we provide the functions `autopas::AutoPas::begin()` and `autopas::AutoPas::getRegionIterator()`, which return iterators that either iterate over all particles, or only those lying in a specific region of the domain. Using the iterator, particles can also be removed from a container.
- Executing the force calculation using a functor-like object via `autopas::AutoPas::` ↪ `iteratePairwise()`. The functor, hereby, describes the force potential. Currently, we have implemented one functor describing the Lennard-Jones potential, as well as two functors that are used for SPH calculations. A user of the library is, however, free to add further functors.
- Updating the container through `autopas::AutoPas::updateContainer()`. This function updates the container by resorting the particles into correct cells. Additionally, particles that moved outside the container are returned by this function.

In addition, some options to describe the outline of the domain of *AutoPas* and some auto-tuning options can be handed to the `autopas::AutoPas` object after its construction.

The actual auto-tuning process of *AutoPas* is hidden behind the `autopas::AutoPas::` ↪ `iteratePairwise()` call, in which the force calculation takes place. The behavior of *AutoPas* hereby differs depending on whether a good configuration has been found or not. If one has already been found, *AutoPas* simply uses the configuration to calculate the forces. If a good configuration has, however, not been found yet, *AutoPas* potentially changes the used configuration. *AutoPas* will always measure the time needed for the force calculation using the currently selected configuration. A switch of the configuration is performed if it has already been measured often enough. Once all configurations are then tested for performance, the auto-tuning process is stopped, and the best configuration is selected. The auto-tuning process is then restarted after a certain number of `autopas::AutoPas::iteratePairwise()` calls.

Traversal	Advantages	Disadvantages	DLB	CS
ds	low overhead for managing particles	$O(n^2)$, bad parallelizability		
lc_sliced	very low scheduling overhead	each chunk of work consists of at least two slices of cells	no	XL
lc_c01	only one barrier, best parallelizability	no support for Newton3, bad caching	yes	XS
lc_c04	4 barriers, caching, low scheduling overhead	lower parallelizability than c08	yes	M
lc_c08	best parallelizability with Newton3 support	more barriers than c04	yes	S
lc_c18		many barriers	yes	S-M
vcl_c06	some vectorization support, low scheduling overhead	potentially too large chunks	yes	L
vcl_sliced			no	XL
vl_list		no Newton3 support	yes	XS
vlc_sliced	very low scheduling overhead	bad for small domains	no	XL
vlc_c18	Newton3 support	many barriers	yes	XS
vlc_c01	only one barrier	no Newton3 support	yes	S-M
vvl_as_built	good parallelizability	static LB not necessarily accurate	partial	

Table 5.1: Comparison of the traversals implemented in *AutoPas* with respect to their advantages and disadvantages. We further indicate whether dynamic load balancing (DLB) is supported on the node level or not and give a qualitative indication of the work chunk size (CS), which is an important characteristic of the traversal. Large chunk sizes indicate lower scheduling overhead, while small chunk sizes lead to better parallelizability and are better for small subdomains or if many threads are used. The traversals are prefixed with the container they are used with, where ds stands for direct sum and lc for linked cells. The Verlet cluster lists(vcl) container uses neighbor lists of clusters, while the other containers starting with v use classical particle-wise Verlet lists. They do, however, differ in the way how the lists are built and stored. The Verlet list (vl) container stores the neighbor list globally, while the Verlet list cells (vlc) container stores the lists for each cell of the underlying linked cells container. The VarVerlet (vvl) container generates the Verlet lists in a parallel fashion using the c08 traversal. To circumvent race conditions, each thread saves the particle pairs that it traversed when building the lists. Hereby, the pairs are stored in separate lists for each color, resulting in eight different lists for each thread. For the actual force calculation, these lists are then traversed by the same threads that generated the lists, resulting in a static load distribution that does, however, not necessarily correspond to the optimal load distribution, as the actual force calculation is not regarded for the generation. Especially for compute-intensive force kernels, this load balancing is often incorrect because the generation that includes only distance calculations is inherently memory-bound (for most systems).

We further had to decide on a common behavioral interface for all containers and had to decide between a linked cells-like approach and a Verlet list-like approach to be able to independently tune different *AutoPas* objects on different MPI ranks. While the former is native to the linked cells container, where particles are sorted into the correct cells and particles moving across cell or domain boundaries have to be handled in every time step, the latter is native to Verlet list-like containers. For them particles are not moved between cells in every time step, allowing the reuse of neighbor lists over multiple time steps. To decide between the two options, we checked the possibility of implementing the linked cells-like approach for Verlet list containers and the possibility of implementing the Verlet-like approach for the linked cells container.

Implementing the Verlet-like approach for linked cells is straightforward and can be achieved by two changes:

- Particles are not moved into the correct cells at every step. This way, particles that are still owned by a cell can be slightly outside of it (at most by $r_s/2$, where r_s is the skin radius).
- As particles don't necessarily lie within the boundaries of a cell, distance calculations between different particles are necessary for the force calculation if the distance between the cells the particles reside in is less than $r_c + r_s$ (instead of simply r_c, where r_c is the cutoff radius). We implemented this change by increasing the minimum size of a cell.

Using this approach, the performance of the linked cells is slightly reduced, as more distance calculations become necessary. Additionally, the increase in cell size is beneficial if the particle density is low, as the overhead of iterating through cells is reduced.

In contrast, using a linked cells-like approach for Verlet lists requires that particles are copied to other instances of *AutoPas* in every time step. Additionally, particles are not actually removed from an *AutoPas* instance if Verlet lists are used, as, otherwise, the neighbor lists could not be reused. The following changes are necessary to implement this behavior:

- The particle ownership state has to be tracked explicitly. This allows marking particles that have left the domain of a process as dummy/halo without actually removing them. This prevents invalidating the used neighbor lists, as the data structure remains unchanged.
- The force calculation has to be adapted, s.t., particles marked as dummy are not considered for it.
- Adding a particle vector containing all particles that would have been added while the neighbor lists are reused. This particle vector has to partake in the force calculation and, additionally, has to be considered by the iterators.
- And optionally: When the neighbor lists are built, they could include all particle pairs, where one particle could potentially become an owned particle. This makes it

possible to add particles that have previously been halo particles as owned particles. If this change is not implemented, they have to be added to the additional particle vector, while the old halo particle has to be marked as dummy.

There additionally exists a mixed approach for implementing a linked cells-like interface for Verlet lists, which, however, is significantly more complicated to implement and maintain and will thus not be discussed in this work. For details of the mixed approach please see [130].

Currently, we have implemented a Verlet-like approach in *AutoPas*, as it allows skipping MPI communications of leaving particles in most time steps and because it is easier to maintain. In the future, we will probably switch to a linked cell-like approach, as it allows the addition of particles in every time step and is thus more flexible with regard to the Grand Canonical ensemble, in which particles can be added and removed. In addition, a linked cell-like approach is easier to handle for a user of *AutoPas*. We are further thinking about a dynamic tuning of the Verlet skin radius and the rebuild frequency, i.e., the frequency at which the neighbor lists are rebuilt. This is, however, only possible if a linked cell-like interface is chosen, as the rebuild frequency and skin have to be constant throughout multiple MPI processes for a Verlet-like approach.

5.5 Integration into ls1 mardyn

As we designed *AutoPas* with an integration into *ls1 mardyn* at mind, the integration itself was relatively straightforward. To integrate *AutoPas* into *ls1 mardyn*, the following changes had to be made:

- Create a new molecule class that is compatible with *AutoPas* and *ls1 mardyn*. We have done this by creating a molecule that inherits from `autopas::MoleculeLJ` which is a molecule class provided by *AutoPas*. The new class additionally implements the common interface for all molecule classes in *ls1 mardyn* (`MoleculeInterface`) and is thus also compatible with *ls1 mardyn*.

- Replace the used linked cells particle container and the force computation with *AutoPas* and its functors. The newly created particle container (`AutoPasContainer`) acts as a wrapper (adapter pattern) of *AutoPas* and also encapsulates the force calculation. Outside this class, only the particle iterators are needed and no other functionalities of *AutoPas* are exposed.

- Replace the used particle iterators with the ones provided by *AutoPas*. As the interfaces of the iterators in *ls1 mardyn* and *AutoPas* are almost identical, we simply added a class that inherits from *AutoPas*' ParticleIteratorWrapper. Only the iterator behavior, i.e., an enum that indicates which cells should be included in the iteration, had to be converted between the behavior used in *ls1 mardyn* and *AutoPas*.

- Adapt the domain decomposition, s.t., it works with *AutoPas*' Verlet-like interface. Using this interface, particles aren't moved between different processes at every step.

Instead, they are only moved before the neighbor lists are rebuilt. Additionally, we increased the area of communicated halo particles by the skin radius, as they are needed for a correct assembly of the Verlet lists.

AutoPas further requires a change in the load balancing algorithms used in *ls1 mardyn*, as *ls1 mardyn*'s load balancing algorithms require an a-priori load estimate. The problems with this estimate are twofold. First, the estimate works on a cell level, i.e., it predicts the time needed for the computation of a cell in dependence on the number of particles inside it and its neighboring cells. *AutoPas* does, however, internally not necessarily use cells. Additionally, the size of the cells might vary between processes. Second, the algorithms used within *AutoPas* can change the time needed for the calculation of a cell. The load of a cell does not only depend on the cell itself. Instead, it also depends on the algorithms used by the process that owns the cell. As the algorithm is, however, not known before a load balancing is made, an a-priori load estimate becomes infeasible to use with *AutoPas*.

We, therefore, decided to use diffusive load balancing when using *AutoPas*. For this purpose we included *A Load Balancing Library* (A Load Balancing Library, [3]) into *ls1 mardyn*. *A Load Balancing Library* is a library which is developed at the Forschungszentrum Jülich that provides dynamic domain-based load balancing for particle simulation codes. In our case, we use *A Load Balancing Library*'s staggered grid approach which implements a diffusive version of the multi-section method (cf. subsection 2.6.2).

To integrate *A Load Balancing Library* into *ls1 mardyn*, we have written the wrapper class `ALLLoadBalancer`, which implements the new interface `LoadBalancer`. We have added the latter, s.t., we can easily add new load balancing libraries to *ls1 mardyn*. The `LoadBalancer` only handles information about the domain and does not know about particles. To handle the particle exchange, we have created another new class, called `GeneralDomainDecomposition` which uses a `LoadBalancer` for the load balancing.

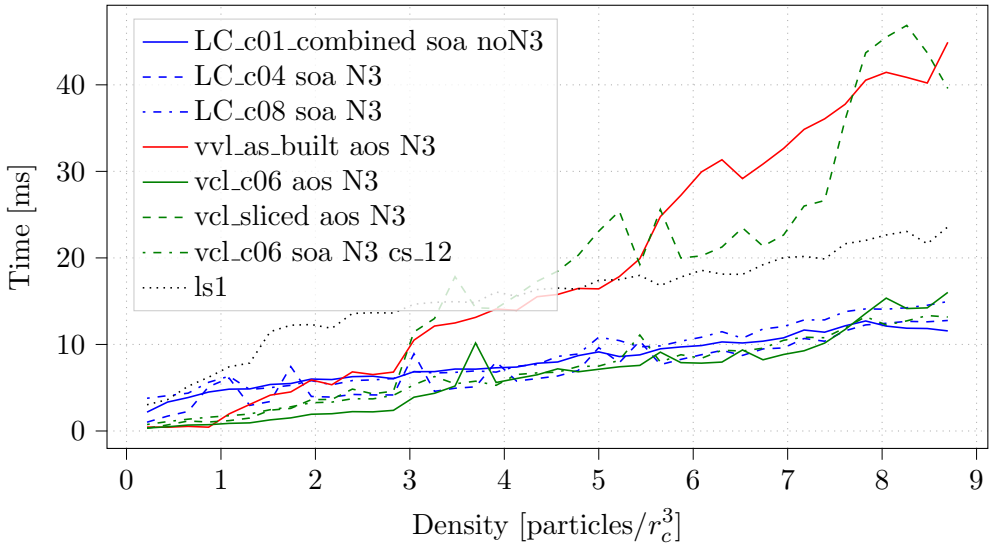
A Load Balancing Library generates rectilinear partitions whose boundaries can take arbitrary values bigger than a certain minimal size. When using *AutoPas*, this does not pose a problem. Without *AutoPas*, *ls1 mardyn*, however, requires that its cells are aligned on a grid. For this purpose, we have implemented a latching of the domain boundaries to a given grid, thus allowing the use of *A Load Balancing Library* even if *AutoPas* is not used.

5.6 Results

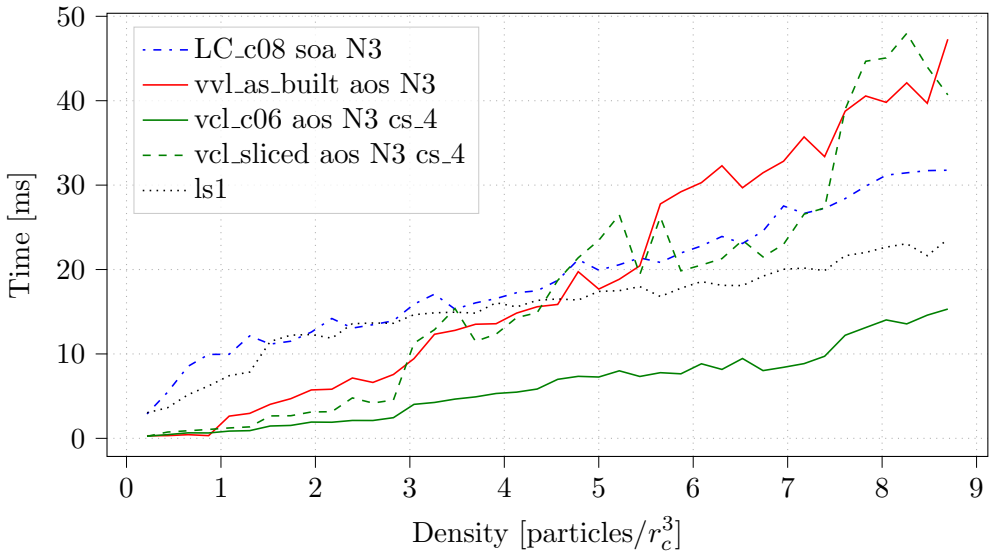
5.6.1 Node-Level

To showcase some benefits that *AutoPas* enables, we show performance results of *ls1 mardyn* with and without *AutoPas* in Figure 5.3. Additionally, we compare the performance of the two provided functors LJFunctor and LJFunctorAVX. They both model the same force field but use different means of vectorization. The LJFunctor uses `#pragma omp simd`, while LJFunctorAVX uses intrinsics that guarantee proper vectorization.

[3] `https://gitlab.version.fz-juelich.de/SLMS/loadbalancing`



(a) Using the intrinsics functor (LJFunctorAVX).



(b) Using the functor without intrinsics (LJFunctor).

Figure 5.3: Performance of different configurations for a homogeneous scenario (single-centered LJ) in dependence of the particle density. Shown are the performances of all configurations that are chosen for some density, as well as the configuration of *AutoPas* (LC_c08 soa N3) that matches *ls1 mardyn*'s default behavior (ls1). In this simulation that has been carried out on one node of CoolMUC2 using 28 threads, we chose a skin radius of $0.1r_c$ and a rebuild-frequency and sampling interval of 10 time steps. The configurations include linked cells (LC), Verlet lists (vvl), and Verlet cluster lists (vcl, size of the clusters indicated at the end). Shown is the time needed for the force calculation only. We additionally created these results for both Lennard-Jones functors provided by *AutoPas*, one using intrinsics to guarantee vectorization and one using OpenMP's *#pragma omp simd*.

First, we describe the results using the intrinsics functor. Hereby, the chosen algorithm configuration depends on the used particle density. For low densities, algorithms using Verlet lists provide better performance compared to linked cells, as the latter require iterating over multiple almost empty cells, which is not worthwhile. For higher densities, however, linked cells using the SoA data layout provide better performance compared to Verlet lists, because they allow better vectorization. The Verlet cluster lists provide some variant in-between linked cells and Verlet lists. They thus tend to provide reasonable performance for most densities. The performance of the sliced approach for the Verlet clusters is not as good for this scenario, because the domain is relatively small ($\approx (26(r_c + r_{\text{skin}}))^3$) for the large number of used threads (28).

If the LJFunctor is used, good vectorization is not achieved and linked cells provide only around half the performance compared to the intrinsics functor. While the algorithms using the linked cells data structure perform better than the classical Verlet list approach (vvl) for high densities, they cannot reach the performance provided by the Verlet cluster approach and are, therefore, never chosen as the best configuration. The LJFunctor is thus generally less performant compared to the version using intrinsics.

5.6.2 Multi-Node

We have chosen three different scenarios to analyze the behavior of *ls1 mardyn* when using *AutoPas* on multiple nodes and executed them on SuperMUC-NG.

Spinodal decomposition In this scenario, an equilibrated homogeneously distributed fluid is rapidly cooled below its critical temperature. Due to this, the fluid contracts and builds regions of higher and lower density, forming an inhomogeneous scenario.

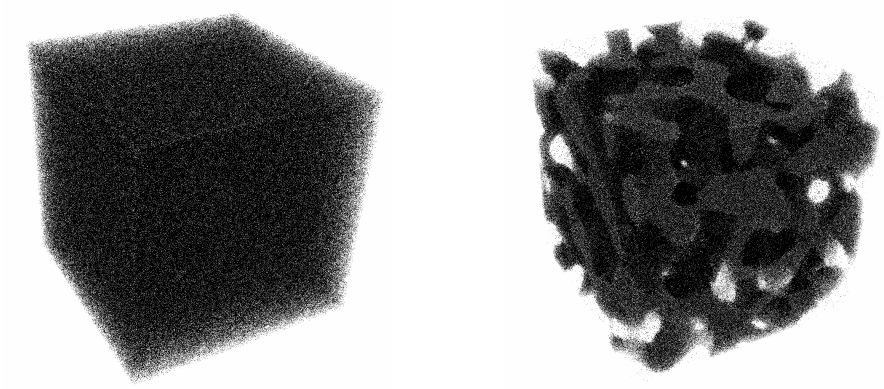
Exploding liquid This scenario starts with a liquid film that is placed into vacuum. Upon the start of the scenario, the film rapidly expands and forms two shock fronts with some filaments and droplets in between.

Droplet coalescence This scenario resembles the process of two neighboring droplets that merge into one bigger droplet.

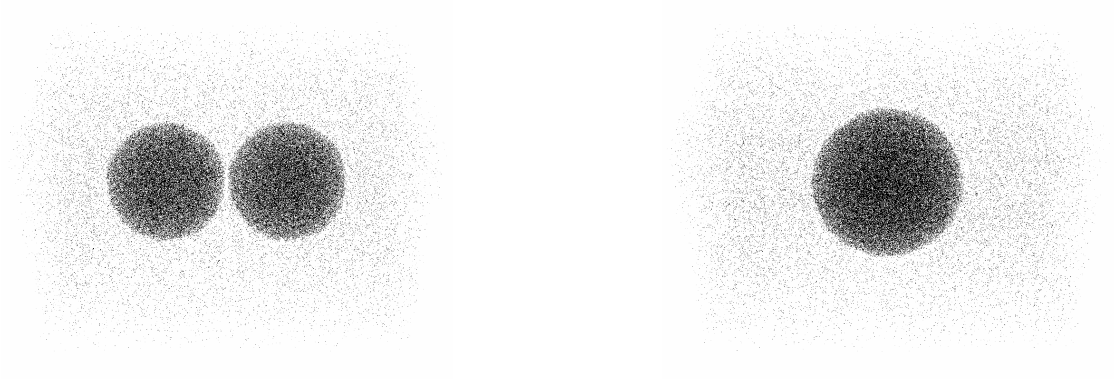
All of these scenarios use single-centered Lennard-Jones particles and are visualized in Figure 5.4. We decided to use these scenarios, as they offer a wide range of problems ranging from slowly changing scenarios with global inhomogeneity (droplet coalescence) over faster changing scenarios (spinodal decomposition) to rapidly progressing scenarios (exploding liquid). These scenarios need different considerations, as the load balancing for a slowly changing scenario can be almost static, while the load balancing for the exploding liquid scenario has to be very responsive.

Note that the first two experiments (spinodal decomposition and exploding liquid) were performed with a version of *AutoPas* that did not yet include the intrinsics functor or the Verlet cluster approach. We, therefore, expect further speedups if they were used.

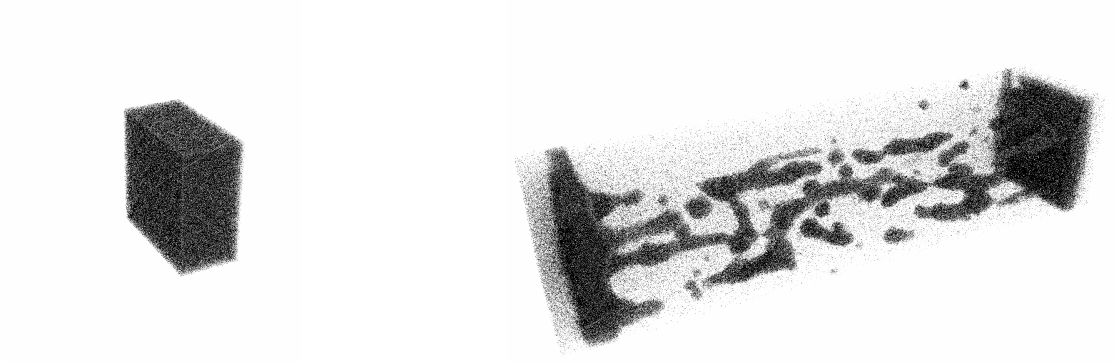
For the droplet coalescence scenario, the current version of *AutoPas* was used. We therefore enabled both the intrinsics functor, as well as the Verlet cluster approach.



(a) Spinodal decomposition scenario.
`https://youtu.be/yarl9028dEc`



(b) Droplet coalescence scenario.
`https://youtu.be/1tlxDapmgbI`



(c) Exploding liquid scenario.
`https://youtu.be/u7TE5KiSQ08`

Figure 5.4: Visualization of the scenarios used for the multi-node experiments when using *ls1 mardyn* with *AutoPas*. On the left, the initial setup is shown, while the right shows the state at the end of the simulation. Previously published in [130].

5.6.2.1 Spinodal decomposition

Using the spinodal decomposition scenario, we show that it is possible to use *ls1 mardyn* and *AutoPas* together in an MPI-parallel simulation for a relatively homogeneous scenario. We hereby restrict *AutoPas* to use the linked cells container and the SoA data layout with enabled newton 3 and allow it to only choose between two traversals, the c08, and the sliced traversal.

The sliced traversal does not provide any internal load balance and provides the biggest possible work chunks with minimal scheduling overhead. The sliced traversal should thus be chosen if the particle density within one rank is uniform. If the density is non-uniform the c08 traversal typically performs better. The c08 traversal should also be chosen if the domain of each process is relatively small, i.e., if the sliced traversal is unable to split its domain into equally sized partitions. In a strong scaling scenario, the c08 traversal should thus be chosen more often than the sliced traversal if more processes are used. This effect is also driven by the smaller subdomains in which small load imbalances result in larger relative imbalances within a subdomain compared to bigger subdomains.

We could observe all of these preliminary considerations for the spinodal decomposition scenario (cf. Figure 5.5). The scenario starts with an (almost) homogeneous particle distribution and becomes more and more inhomogeneous. This change is perfectly reflected in the ratio of the chosen traversals. At the start, the sliced traversal is chosen. With an increasing inhomogeneity, the c08 traversal is chosen more often. In addition, the hard limits of the sliced traversal regarding the size of the subdomains are visible for the scenario with 2 million particles and 32 nodes, where the sliced traversal is hardly chosen. For the larger of the two scenarios, the sliced traversal is, however, still chosen even if 128 nodes are used. For the large scenario and using only 16 nodes, the small inhomogeneities within a subdomain are averaged out, as the subdomains themselves are big. Therefore, the sliced traversal is chosen more often in comparison to the measurements with more nodes, for which the subdomains are smaller and small inhomogeneities become more relevant.

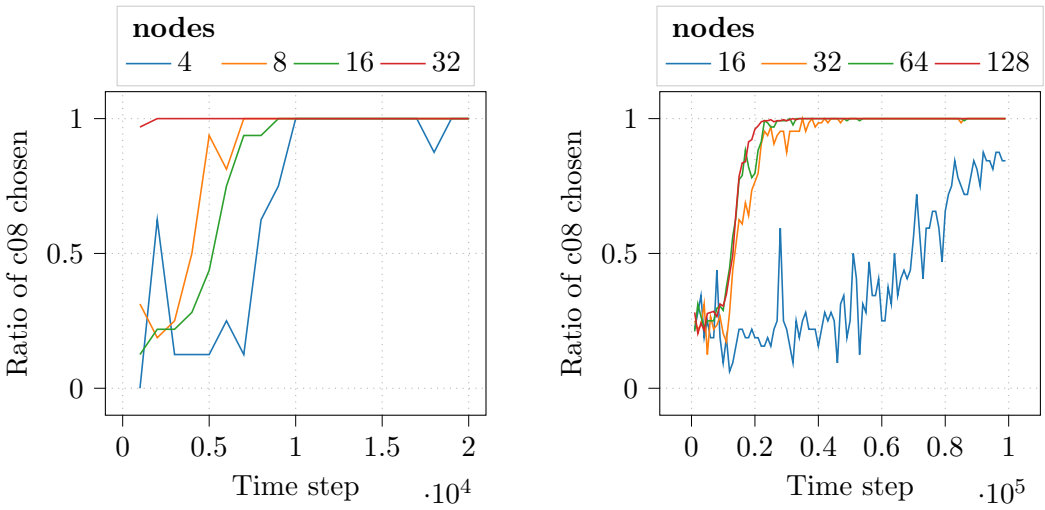
In comparison to pure *ls1 mardyn*, using *AutoPas* can enable a significant speedup of up to 50% (cf. Figure 5.6). This speedup can, however, mostly be attributed to a force kernel that is more optimized for single-centered instead of multi-centered molecules.

5.6.2.2 Exploding Liquid

Using the exploding liquid scenario with three MPI processes, we show that the diffusive load balancing works well together with the auto-tuning of *AutoPas*.

For this example, we always start a tuning phase after a rebalancing happens. We further ensure that the rebalancing is only started once the tuning of the previous tuning phase has ended. This is important to get a load balancing that is adjusted to the performance of the optimal traversal.

Figure 5.7 shows that for three processes the domain decomposition reacts properly to the load imbalances, s.t., the time needed on each node is almost identical. We also



(a) 2M particles.

(b) 157M particles.

Figure 5.5: Ratio of the chosen traversal for the spinodal decomposition scenario using the Cartesian decomposition on SuperMUC-NG. For this example, we only allow the selection of either the sliced or c08 traversal. Numerical results previously published in [130].

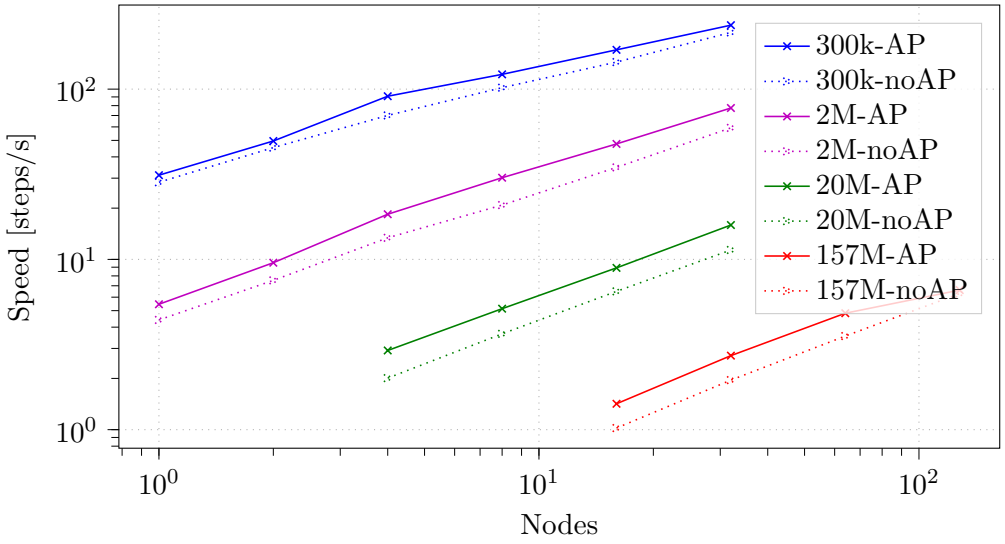


Figure 5.6: Strong scaling for the spinodal decomposition scenario using the Cartesian decomposition for *ls1 mardyn* with and without *AutoPas* on SuperMUC-NG. For this example, we only allow the selection of either the sliced or c08 traversal. Numerical results previously published in [130].

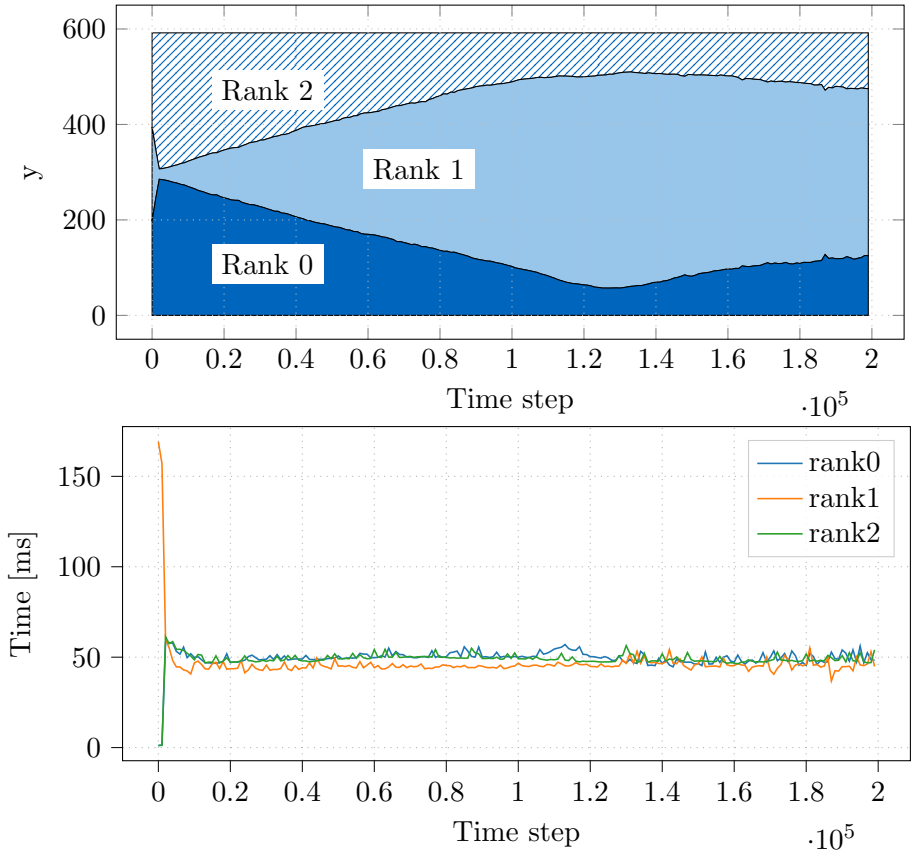


Figure 5.7: Decomposition and load balancing for the exploding liquid scenario when using the ALL load balancer on three MPI nodes of SuperMUC-NG. The simulation starts with equally distributed subdomains. The inner of these subdomains will then contract due to a load concentration in the middle of the domain (rank1), while the outer subdomains increase in size. After the contraction of the inner subdomain, the liquid expands and the inner subdomain follows this expansion.

observe that the load balancer can quickly find a good initial load balancing. For this example, we start a rebalancing almost immediately after a tuning phase has finished.

Using the exploding liquid scenario, we additionally performed strong-scaling experiments, for which we observe a speedup of up to 43% compared to the pure *ls1 mardyn* code (cf. Figure 5.8). In this case, the speedup can mostly be attributed to a better load balancing, which is able to follow the liquid. Meanwhile, k-d tree-based load balancing cannot follow the liquid. Instead, a partition is mostly completely changed after a rebalancing step and all particles have to be exchanged to the new owning process.

The use of *AutoPas* is, however, not always better. This can be seen when looking at the Cartesian decomposition. Runs on many nodes that use the Cartesian decomposition are actually faster when *AutoPas* is not used (25% faster for 256 nodes). The reason for this decrease in performance can be partially attributed to the tuning overhead, as some

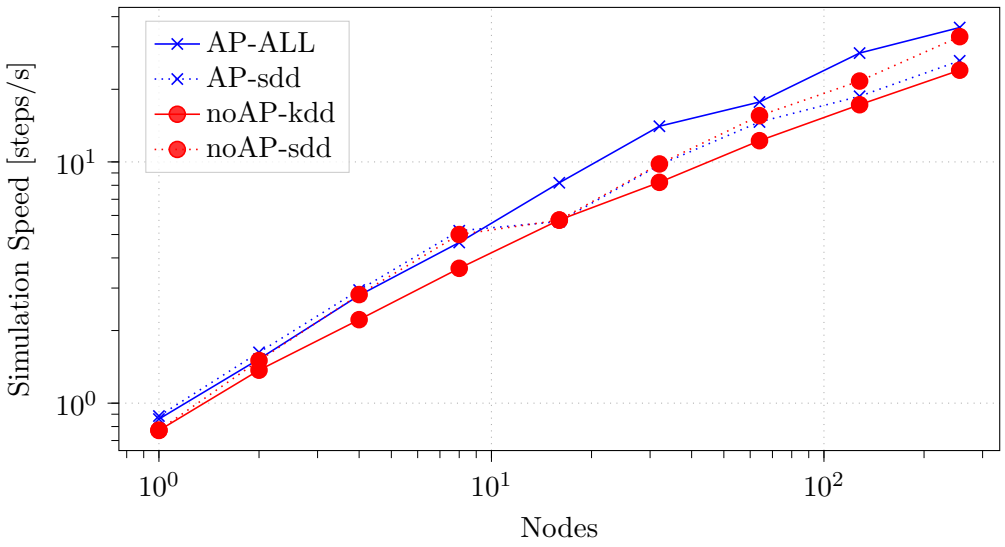


Figure 5.8: Strong scaling for the exploding liquid scenario with 8 million particles on SuperMUC-NG (2 MPI ranks per node). Shown is the behavior with the use of *AutoPas* (AP) and without (noAP) and its behavior when using the diffusive load balancing (ALL) and the k-d tree-based load balancing (kdd). In this simulation, *AutoPas* uses 5 tuning samples per configuration, which is also the rebuild frequency of the containers. If ALL is used, a rebalancing is performed every 2500 time steps (resp. every 1000 time steps for the first 10000 time steps). The entire simulation comprises 100000 time steps. For the kdd, rebalancing is also performed every 10000 time steps. Numerical results previously published in [130].

configurations are suboptimal and a time step takes significantly longer when using such a configuration compared to the pure *ls1 mardyn* code. Especially the use of non-vectorized code (configurations using AoS) in the dense region of the domain results in a significant loss of performance.

Using the diffusive load balancing together with *AutoPas* does, however, consistently perform better compared to the original *ls1 mardyn* code.

5.6.2.3 Droplet Coalescence

The droplet coalescence scenario represents a slower progressing scenario compared to the exploding liquid or spinodal decomposition scenario. It, therefore, is possible to rebalance and retune less frequently. We make use of this by setting a rebalancing interval of 20000 time steps. To still get a good initial partition, we start with ten rebalancing steps with a shorter interval of 1000 time steps allowing to equilibrate the load quickly.

Figure 5.9 shows such a simulation with its domain decomposition at the start of the simulation and after the initial load balancing phase. First, we observe that the domain decomposition results in smaller subdomains where the droplet is located (center) and larger subdomains on the outside of the domain.

We further note that the chosen configurations vary in an almost symmetric way, in which the low-density regions (corners) are simulated using a Verlet list approach, while the high-density regions, for which vectorization is beneficial, use linked cells. The high-density regions, hereby, show a behavior where the c04 traversal is used on bigger subdomains, while the c08 traversal is used for smaller subdomains, because the c08 traversal uses smaller work items, thus providing better parallelizability, while the c04 traversal uses bigger work items resulting in less scheduling overhead.

When looking at the tuned configurations (cf. Figure 5.10) in dependence on the time, we notice that, while initially, a relatively large part of the processes use Verlet-like approaches, they tend to be less favored, after the initial rebalancing phase. The reason for this is that most processes will partially calculate parts of the droplet for which a good vectorizability is needed and thus linked cells are more efficient. Hereby, for most processes, the c04 traversal tends to be the optimal choice.

In Figure 5.11 a comparison of the strong scaling is shown. Hereby the newly introduced diffusive load balancing is able to outperform the k-d tree-based load balancing by up to 60% (8 nodes) and the Cartesian decomposition by up to 100% (128 nodes). For this scenario, the use of *AutoPas* is only beneficial for the overall performance if few nodes are used. For a large number of processes, using *AutoPas* can decrease the performance by up to 30%, assuming no Verlet cluster lists are used. There are multiple reasons for this. First, *AutoPas*' cell-based containers use a larger cell size. This cell size also restricts the minimal subdomain size which is bigger compared to a simulation without *AutoPas* and thus decreases the maximal level of parallelization. When using *A Load Balancing Library* on 256 nodes, only around 315 cells can fit into the smallest subdomain, which corresponds to 13 cells, resp. 250 particles per thread. Second, the iterators implemented within *AutoPas* are slower compared to the native iterators of *ls1 mardyn* and result in worse performance of the communication. The reason for this slowness is the higher

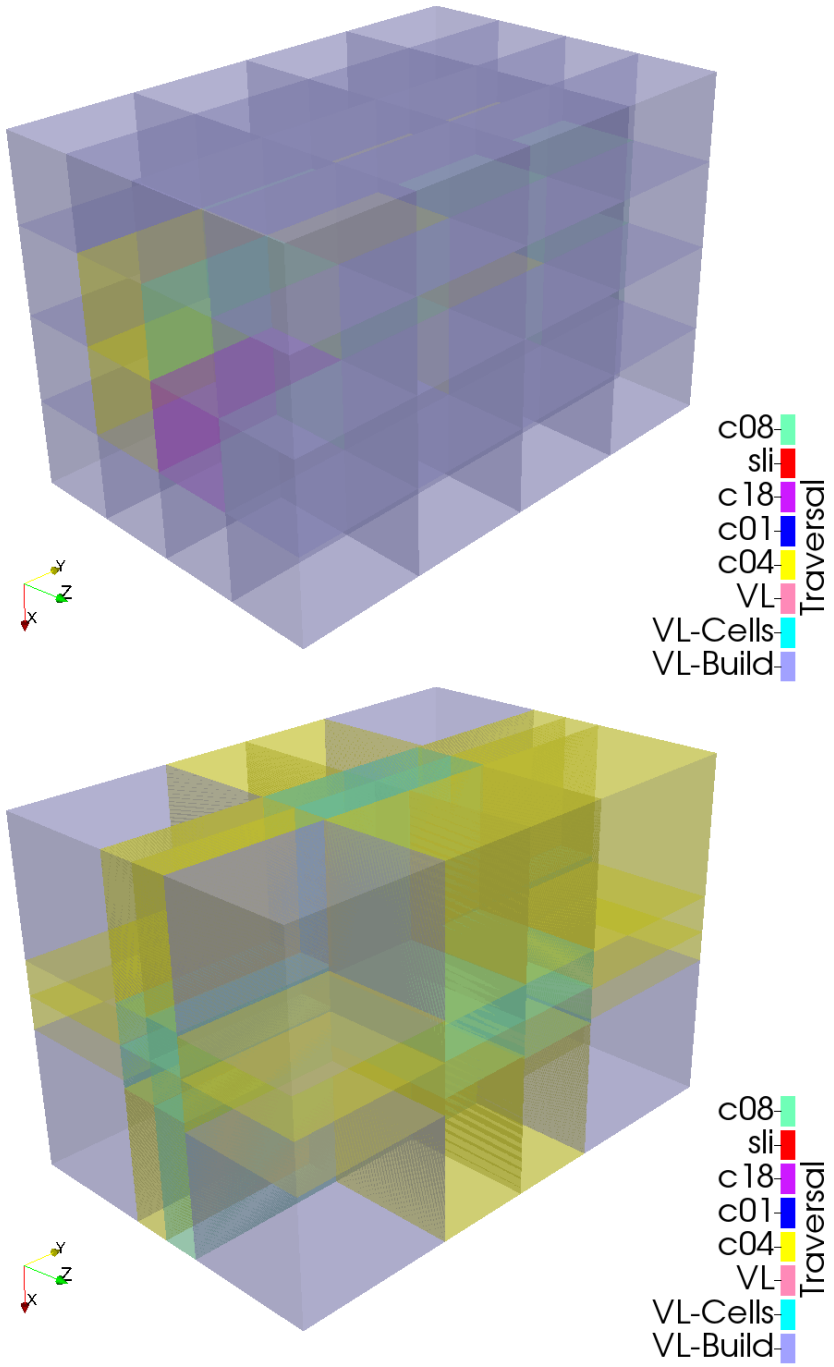


Figure 5.9: Domain decomposition and tuning results for the droplet coalescence scenario on 64 processes (32 nodes) of SuperMUC-NG at the start of the simulation (top) and after an initial rebalancing (bottom) when using diffusive load balancing. Figure previously published in [130].

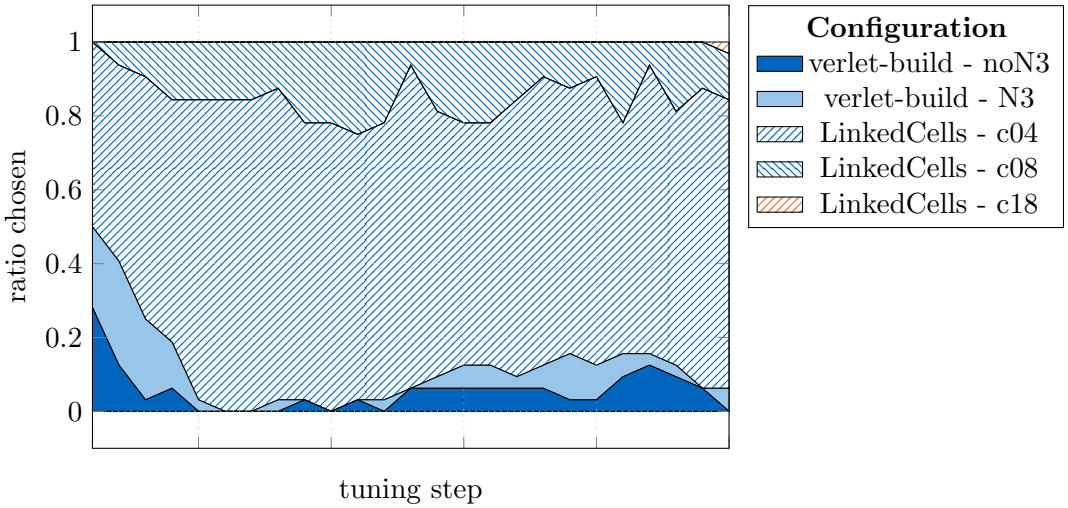
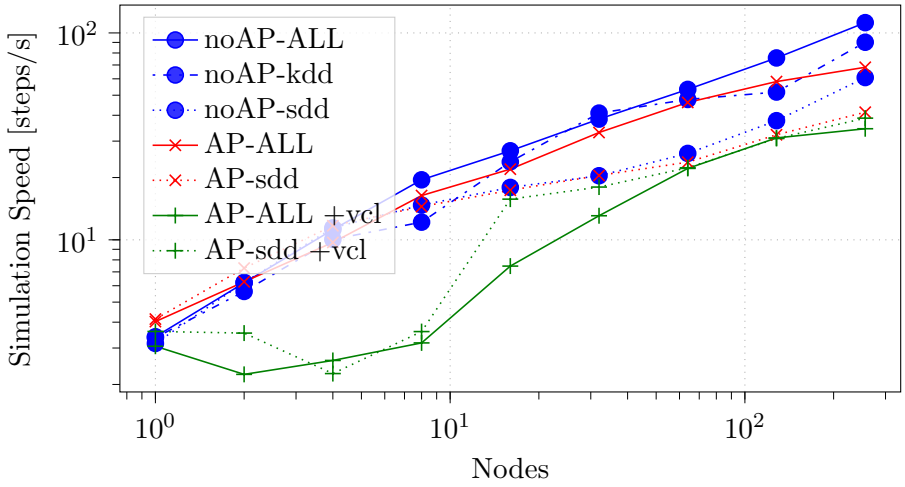


Figure 5.10: Ratio of the selected configurations on 32 processes (16 nodes) of SuperMUC-NG for the droplet coalescence scenario. The simulation uses diffusive load balancing. Numerical results previously published in [130].

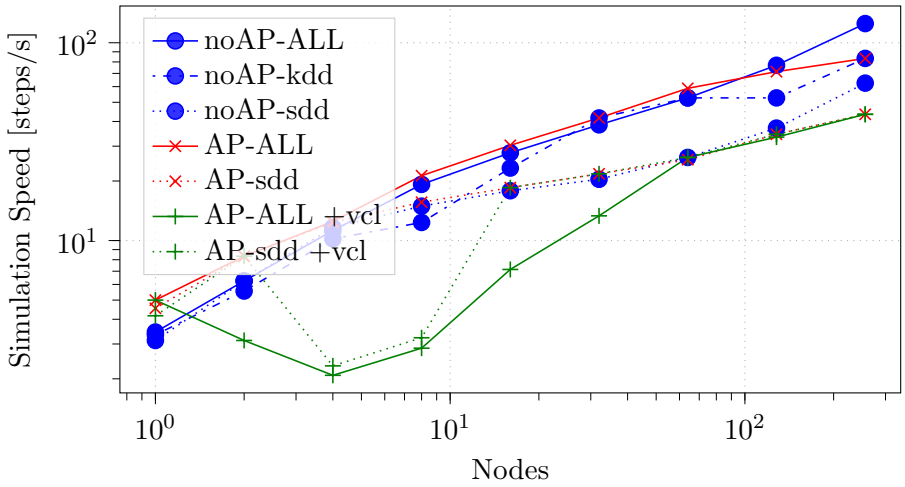
number of indirections within the iterators that are needed to cope with the different containers. And third, *AutoPas* introduces a tuning overhead that arises from iterating with non-optimal configurations. If this overhead could be eliminated, the use of *AutoPas* would be beneficial. This can be seen in Figure 5.11b, where only the performance of the last 10 time steps is given, for which an optimal configuration has already been selected. Once the tuning has finished, using *AutoPas* can provide speedups of up to 50% on 1 node. When using many nodes, only minor speedups are possible.

The force calculation does not become faster if *AutoPas* is used (cf. Figure 5.12). One reason for this is the relatively small scenario in which the smallest subdomains quickly reach the strong scaling limit. We do, however, observe that the average time needed for the force calculation is significantly (up to 45%) smaller when employing *AutoPas*. The use of different algorithms can thus decrease the accumulated time which potentially reduces energy consumption.

We additionally note that while the Verlet cluster container was quite performant on the node level, it significantly reduced the performance on the multi-node level. The main reason for this is that, on the node level, we only looked at the time for the force calculation and not the entire simulation. For an entire simulation, particle addition and deletions are, however, problematic for the Verlet cluster list container, because they, currently, trigger container rebuilds, and thus the entire container is rebuilt more often than needed. We are, however, working on resolving this bug to allow good performance of this container for an entire simulation run. If just the force calculation is considered, the cluster lists do not decrease the performance much and can sometimes even increase the performance a bit for the Cartesian decomposition. For the diffusive decomposition, using the cluster lists does, however, still decrease the performance, which indicates that the load balancing does not work properly when the Verlet cluster lists are used. This



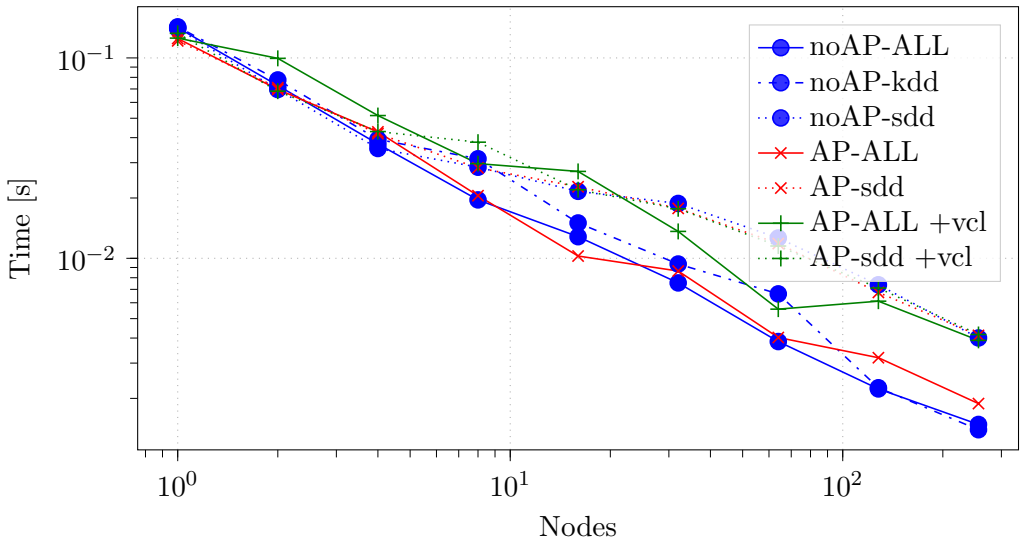
(a) Average simulation speed for the entire simulation.



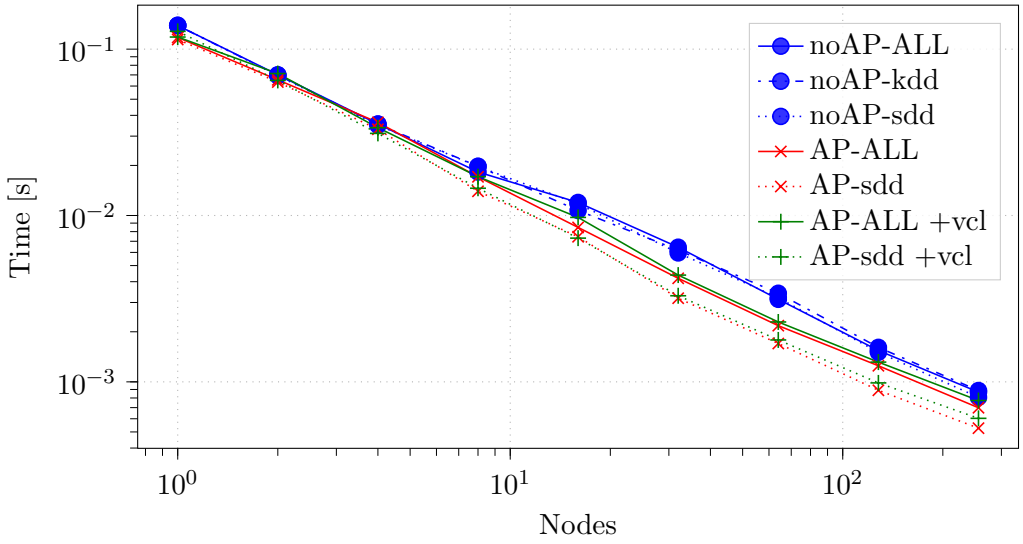
(b) Average simulation speed for the last 10 steps. For *AutoPas*, the optimal configuration has already been found.

Figure 5.11: Strong scaling for the droplet coalescence scenario with 3 million particles on SuperMUC-NG (2 MPI ranks per node). Shown is the behavior with the use of *AutoPas* (AP) and without (noAP) and its behavior when using the diffusive load balancing (ALL), the k-d tree-based load balancing (kdd), and the Cartesian domain decomposition (sdd). This run uses a list rebuild frequency of 10 time steps and also uses 10 samples for the tuning for each configuration. For ALL, initially, 20 rebalancings are performed every 1000 time steps. After this initial rebalancing phase, a rebalancing is performed only every 20000 time steps. For kdd, a rebalancing is performed every 20000 time steps. The entire simulation was performed for 100000 time steps. For AP-ALL and AP-sdd, the Verlet cluster lists were disabled, as they have not yet been optimized for the entire simulation cycle and unnecessary rebuilds of their structure are performed, slowing down the simulation significantly. If +vcl is specified, the cluster lists were enabled.

can also be seen in Figure 5.12b, where the time needed does not depend on the usage of the Verlet cluster lists. One possible reason for the poor load balancing might be that the underlying assumption, that the time needed for the computation should not change much depending on the selected configuration, is broken because the change is too big.



(a) Maximum of the time needed by the ranks. This corresponds to the strong scaling because the slowest rank limits the simulation speed.



(b) Average of the time needed by the ranks. This corresponds to the optimal time needed if perfect load balancing were possible.

Figure 5.12: Strong scaling for the droplet coalescence scenario with 3 million particles on SuperMUC-NG (2 MPI ranks per node). Here, only the time needed for the force calculation of the last 10 time steps is shown. Otherwise, the same parameters have been used as in Figure 5.11.

6 Summary and Outlook

6.1 Summary

I presented various enhancements regarding the multi-node performance of *ls1 mardyn*.

First, the load balancing that uses the k-d tree-based decomposition has been enhanced by providing a precise load estimation. The load balancing can now handle different particle types, very inhomogeneous particle densities, and even heterogeneous computing clusters. For a fully heterogeneous scenario on the CoolMAC cluster, we could observe that the simulation speed has more than doubled.

Second, I have showcased the usage of zonal methods in *ls1 mardyn*. This allowed lifting the strong scaling limits that exist for small, but dense scenarios. Using the eighth shell method, speedups of up to 100% could be observed for realistic scenarios. When using the neutral territory method, the simulation was sped up by up to 130x (13000%). This speedup was, however, measured for an unrealistically high density (2500 particles per r_{cutoff}^3) and should be taken with a grain of salt.

Third, I have enabled overlapping communication within *ls1 mardyn*. Hereby, overlapping collectives were employed to overlap global communication over an entire time step. This allowed a speedup of 16% for a large-scale production run. The actual overlap of the neighbor exchange of particles with the force computation using overlapping P2P communication, did, however, not show any meaningful improvements compared to the already existing overlap with the particle extraction. Only for high-latency networks, speedups could be observed.

In addition, I have integrated the node-level auto-tuned particle simulation library *AutoPas* into *ls1 mardyn*. This integration allows the dynamic selection of the optimal algorithm configuration for the force calculation and can provide significant speedups of the force calculation. At the node level, we observed that the time needed for the force calculation could be halved if *AutoPas* was used. For the usage of *AutoPas* in MPI parallel simulations, I integrated the load balancing library *A Load Balancing Library* into *ls1 mardyn* which enables diffusive load balancing. This diffusive load balancing can outperform the tree-based partitioning for the droplet coalescence scenario by up to 60%. The use of *AutoPas* in MPI parallel simulations can increase performance, as demonstrated in the exploding liquid scenario. This increase is, however, restricted due to the tuning-overhead of the exhaustive search. I have shown that the use of *AutoPas* and its different algorithm configurations can decrease the resource consumption of the force calculation by up to 45%.

The shown speedup values can always also be directly translated to a possible increase of the system size while leaving the required time unchanged (if a simulation was sped up by X%, an X% bigger simulation can now be simulated as fast as the original scenario

before the optimizations). This is especially relevant as most HPC clusters restrict the run-time of jobs and thus indirectly limit the maximal simulation size if no checkpoint-restart mechanics is used. With the speedup, it is now possible for bigger simulations to finish within the time limits. Additionally, a speedup is equivalent to a reduction of needed core-hrs on an HPC system and thus allows more simulation runs for a given compute budget. If these simulations provide samples (e.g., of rare events or for parameter identification), a higher accuracy of the sampled process is possible.

6.2 Outlook

While the use of diffusive load balancing in *ls1 mardyn* already brought good speedups, it might not be the best choice for the exascale era, as the initial load balancing might take too long. G. Sutmann proposed a way to circumvent this problem by generating an underlying load density field and distributing the domain according to it [72]. In *FDPS*, another way to solve this problem is implemented using particle samples and was shown to scale onto up to 20000 nodes on Sunway TaihuLight [138–140]. Therefore, further load balancing strategies should be tested and compared. This testing is made relatively easy because, with the integration of *A Load Balancing Library*, I designed an interface that allows the easy addition of other load balancing or partitioning libraries into *ls1 mardyn*.

Furthermore, we are planning to fully replace the force kernel of *ls1 mardyn* with *AutoPas*. For this purpose, we need to make it possible to use all essential features of *ls1 mardyn* with *AutoPas*. One of the major features that *ls1 mardyn* supports but is currently not possible with *AutoPas*, is the simulation of multi-centered molecules. We, therefore, intend to adapt *AutoPas* accordingly. Currently, it is still unclear whether the efficient support of multi-centered molecules requires a change of the interface or not. Basic support is, however, possible through a modification of the force functor.

For *AutoPas* itself, there are a couple of open work items to increase performance and make it easier to use.

First, we are aiming to integrate the Kokkos library into *AutoPas* to enable performance portability and the efficient use of GPUs. In addition, using Kokkos would allow a user of *AutoPas* to easily write new force interactions by specifying only the pairwise interaction. Using Kokkos will most likely change the interface of *AutoPas* from the provided iterators to `for_each` function calls that take a lambda as input. To use *AutoPas* inside of *ls1 mardyn*, we would then need to change the usage of *ls1 mardyn* accordingly.

Second, Fabio Gratl is working on enhancing the auto-tuning to reduce the overhead of testing bad options. He is working on methods that replace the exhaustive search (which was used in this work) with other strategies that utilize methods ranging from Bayesian optimization to linear regression and also include MPI-parallel tuning. We are further working on enabling tuning for other parameters besides the presented ones. Among others, these include the chunk size of OpenMP regions and the size of the Verlet clusters. Especially for the latter, we could already observe that their value can influence the performance (cf. Figure 5.3). The auto-tuning changes will not change the interface of *AutoPas* and are thus transparent to the integration within *ls1 mardyn*.

And third, we are thinking of restructuring *AutoPas* to utilize C++20's concepts. This would allow us to significantly shorten possible compilation errors when *AutoPas* is used (e.g. in *ls1 mardyn*), which eases both its maintainability and usability.

Bibliography

[1] C. Niethammer, S. Becker, M. Bernreuther, M. Buchholz, W. Eckhardt, A. Heinecke, S. Werth, H.-J. Bungartz, C. W. Glass, H. Hasse, et al. ls1 mardyn: The massively parallel molecular dynamics code for large systems. *Journal of chemical theory and computation*, 10(10):4455–4464, 2014.

[2] M. Buchholz. *Framework zur Parallelisierung von Molekulardynamiksimulationen in verfahrenstechnischen Anwendungen*. PhD thesis, Technical University of Munich, 2010.

[3] W. F. Eckhardt. *Efficient hpc implementations for large-scale molecular simulation in process engineering*. PhD thesis, Technical University of Munich, 2014.

[4] W. Eckhardt, A. Heinecke, R. Bader, M. Brehm, N. Hammer, H. Huber, H.-G. Kleinhenz, J. Vrabec, H. Hasse, M. Horsch, et al. 591 tflops multi-trillion particles simulation on supermuc. In *international supercomputing conference*, pages 1–12. Springer, 2013.

[5] N. Tchipev, S. Seckler, M. Heinen, J. Vrabec, F. Gratl, M. Horsch, M. Bernreuther, C. W. Glass, C. Niethammer, N. Hammer, et al. Twetris: Twenty trillion-atom simulation. *The International Journal of High Performance Computing Applications*, 33(5):838–854, 2019.

[6] N. P. Tchipev. *Algorithmic and Implementational Optimizations of Molecular Dynamics Simulations for Process Engineering*. PhD thesis, Technical University of Munich, 2020.

[7] E. J. Maginn and J. R. Elliott. Historical perspective and current outlook for molecular dynamics as a chemical engineering tool. *Industrial & Engineering Chemistry Research*, 49(7):3059–3078, 2010. URL: `https://doi.org/10.1021/ie901898k`, `doi:10.1021/ie901898k`.

[8] K. Dubey and R. P. Ojha. Molecular dynamics simulations: Applicability and scopes in computational biochemistry. In *Nanoscience and Computational Chemistry: Research Progress*, 2013.

[9] L. Casalino, A. C. Dommer, Z. Gaieb, E. P. Barros, T. Sztain, S.-H. Ahn, A. Trifan, A. Brace, H. Ma, H. Lee, et al. Ai-driven multiscale simulations illuminate mechanisms of sars-cov-2 spike dynamics. *BioRxiv*, 2020.

[10] H. A. Scheraga, M. Khalili, and A. Liwo. Protein-Folding Dynamics: Overview of Molecular Simulation Techniques. *Annual Review of Physical Chemistry*, 58(1):57–83, 2007. PMID: 17034338. URL: `https://doi.org/10.1146/annurev.physchem.58.032806.104614`, `doi:10.1146/annurev.physchem.58.032806.104614`.

[11] J. W. Gibbs. *Elementary principles in statistical mechanics: developed with especial reference to the rational foundations of thermodynamics*. C. Scribner's sons, 1902.

[12] D. Frenkel and B. Smit. *Understanding molecular simulation: from algorithms to applications*, volume 1. Elsevier, 2001.

[13] M. P. Allen and D. J. Tildesley. *Computer simulation of liquids*. Oxford university press, 2017.

[14] G. Bussi, D. Donadio, and M. Parrinello. Canonical sampling through velocity rescaling. *The Journal of chemical physics*, 126(1):014101, 2007.

[15] S. C. Harvey, R. K.-Z. Tan, and T. E. Cheatham III. The flying ice cube: velocity rescaling in molecular dynamics leads to violation of energy equipartition. *Journal of computational chemistry*, 19(7):726–740, 1998.

[16] P. H. Hünenberger. *Thermostat Algorithms for Molecular Dynamics Simulations*, pages 105–149. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005. URL: `https://doi.org/10.1007/b99427`, `doi:10.1007/b99427`.

[17] C. Coulomb. Histoire de l'Académie Royale des Sciences. *Premier et second memoire sur l'electricite et le magnetisme, ed. by AR des Sciences, Paris, France: L'Imprimerie Royale*, pages 569–611, 1785.

[18] J. E. Jones and S. Chapman. On the determination of molecular fields. II. From the equation of state of a gas. *Proceedings of the Royal Society of London. Series A, Containing Papers of a Mathematical and Physical Character*, 106(738):463–477, 1924. URL: `https://royalsocietypublishing.org/doi/abs/10.1098/rspa.1924.0082`, `doi:10.1098/rspa.1924.0082`.

[19] I. Newton. *Philosophiae naturalis principia mathematica*, volume 3. Apud Guil. & Joh. Innys, Regiæ Societatis typographos, 1726.

[20] S. Genheden, A. Reymer, P. Saenz-Méndez, and L. A. Eriksson. Computational chemistry and molecular modelling basics. *Computational Tools for Chemical Biology*, 2017.

[21] Z. Li, J. R. Kermode, and A. De Vita. Molecular dynamics with on-the-fly machine learning of quantum-mechanical forces. *Physical review letters*, 114(9):096405, 2015.

[22] V. Botu and R. Ramprasad. Adaptive machine learning framework to accelerate ab initio molecular dynamics. *International Journal of Quantum Chemistry*, 115(16):1074–1083, 2015.

[23] C. S. Peskin and T. Schlick. Molecular dynamics by the backward-euler method. *Communications on pure and applied mathematics*, 42(7):1001–1031, 1989.

[24] B. J. Leimkuhler, S. Reich, and R. D. Skeel. Integration methods for molecular dynamics. In *Mathematical Approaches to biomolecular structure and dynamics*, pages 161–185. Springer, 1996.

[25] E. Hairer, C. Lubich, and G. Wanner. Geometric numerical integration illustrated by the störmer-verlet method. *Acta numerica*, 12:399–450, 2003.

[26] V. Rokhlin. Rapid solution of integral equations of classical potential theory. *Journal of Computational Physics*, 60(2):187 – 207, 1985. URL: `http://www.sciencedirect.com/science/article/pii/0021999185900026`, `doi:https://doi.org/10.1016/0021-9991(85)90002-6`.

[27] J. Barnes and P. Hut. A hierarchical o (n log n) force-calculation algorithm. *nature*, 324(6096):446–449, 1986.

[28] R. E. Mickens. Long-range interactions. *Foundations of Physics*, 9(3):261–269, Apr 1979. URL: `https://doi.org/10.1007/BF00715182`, `doi:10.1007/BF00715182`.

[29] I. Kabadshow. *Periodic boundary conditions and the error-controlled fast multipole method*, volume 11. Forschungszentrum Jülich, 2012.

[30] P. Bachmann. *Die analytische zahlentheorie*, volume 2. Teubner, 1894.

[31] E. Landau. *Handbuch der Lehre von der Verteilung der Primzahlen*, volume 1. Ripol Classic, 2000.

[32] M. ALLEN. Computer simulation of liquids. *Oxford Science Publications*, 1987.

[33] D. Rapaport. The art of molecular dynamics simulation, 1996.

[34] L. Verlet. Computer "Experiments" on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules. *Phys. Rev.*, 159:98–103, Jul 1967. URL: `https://link.aps.org/doi/10.1103/PhysRev.159.98`, `doi:10.1103/PhysRev.159.98`.

[35] S. Páll, M. J. Abraham, C. Kutzner, B. Hess, and E. Lindahl. Tackling exascale software challenges in molecular dynamics simulations with gromacs. In *International conference on exascale applications and software*, pages 3–27. Springer, 2014.

[36] R. Yokota and L. A. Barba. A tuned and scalable fast multipole method as a preeminent algorithm for exascale systems. *The International Journal of High Performance Computing Applications*, 26(4):337–346, 2012.

[37] C. E. Leiserson. Fat-trees: universal networks for hardware-efficient supercomputing. *IEEE transactions on Computers*, 100(10):892–901, 1985.

[38] Y. Ajima, S. Sumimoto, and T. Shimizu. Tofu: A 6d mesh/torus interconnect for exascale computers. *Computer*, 42(11):36–40, 2009.

[39] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, 1972. doi:10.1109/TC.1972.5009071.

[40] J. Von Neumann. First draft of a report on the edvac. *IEEE Annals of the History of Computing*, 15(4):27–75, 1993.

[41] R. H. Netzer and B. P. Miller. What are race conditions? some issues and formalizations. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(1):74–88, 1992.

[42] C. Hewitt. Actor model of computation: scalable robust information systems. *arXiv preprint arXiv:1008.1459*, 2010.

[43] J. Diaz, C. Munoz-Caro, and A. Nino. A survey of parallel programming models and tools in the multi and many-core era. *IEEE Transactions on parallel and distributed systems*, 23(8):1369–1386, 2012.

[44] The MPI Forum, CORPORATE. MPI: A Message Passing Interface. In *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, Supercomputing '93, pages 878—883, New York, NY, USA, 1993. Association for Computing Machinery. URL: https://doi.org/10.1145/169627.169855, doi:10.1145/169627.169855.

[45] MPI Forum. Mpi: A message-passing interface standard, 1994.

[46] MPI Forum [online]. 2021. URL: https://www.mpi-forum.org/ [last checked 2021-06-15].

[47] A. Schmelz. Optimization of Molecular Dynamics Simulations Using One-Sided MPI-Directives. Master's thesis, Technical University of Munich, Jun 2017. URL: https://mediatum.ub.tum.de/1462220.

[48] A. Skjellum, N. E. Doss, and K. Viswanathan. Inter-communicator extensions to mpi in the mpix (mpi extension) library. *Submitted to ICAE Journal special issue on Distributed Computing*, 1994.

[49] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of collective communication operations in mpich. *The International Journal of High Performance Computing Applications*, 19(1):49–66, 2005.

[50] J. Pješivac-Grbović, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra. Performance analysis of mpi collective operations. *Cluster Computing*, 10(2):127–143, 2007.

[51] T. Schneider, T. Hoefler, R. E. Grant, B. W. Barrett, and R. Brightwell. Protocols for fully offloaded collective operations on accelerated network adapters. In *2013 42nd International Conference on Parallel Processing*, pages 593–602. IEEE, 2013.

[52] Q. Xiong, C. Yang, P. Haghi, A. Skjellum, and M. Herbordt. Accelerating mpi collectives with fpgas in the network and novel communicator support. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 215–215. IEEE, 2020.

[53] J. Stern, Q. Xiong, A. Skjellum, and M. Herbordt. A novel approach to supporting communicators for in-switch processing of mpi collectives. In *Proc Workshop on Exascale MPI*, 2018.

[54] B. Hess, C. Kutzner, D. Van Der Spoel, and E. Lindahl. Gromacs 4: algorithms for highly efficient, load-balanced, and scalable molecular simulation. *Journal of chemical theory and computation*, 4(3):435–447, 2008.

[55] M. Garcia, J. Labarta, and J. Corbalan. Hints to improve automatic load balancing with lewi for hybrid applications. *Journal of Parallel and Distributed Computing*, 74(9):2781–2794, 2014.

[56] R.-P. Mundani, A. Düster, J. Knežević, A. Niggl, and E. Rank. Dynamic load balancing strategies for hierarchical p-fem solvers. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pages 305–312. Springer, 2009.

[57] A. Legrand, H. Renard, Y. Robert, and F. Vivien. Mapping and load-balancing iterative computations. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):546–558, 2004.

[58] S. Sharma, S. Singh, and M. Sharma. Performance analysis of load balancing algorithms. *World academy of science, engineering and technology*, 38(3):269–272, 2008.

[59] S. B. Shaw and A. Singh. A survey on scheduling and load balancing techniques in cloud computing environment. In *2014 international conference on computer and communication technology (ICCCT)*, pages 87–95. IEEE, 2014.

[60] P. Samal and P. Mishra. Analysis of variants in round robin algorithms for load balancing in cloud computing. *International Journal of computer science and Information Technologies*, 4(3):416–419, 2013.

[61] D. C. Devi and V. R. Uthariaraj. Load balancing in cloud computing environment using improved weighted round robin algorithm for nonpreemptive dependent tasks. *The scientific world journal*, 2016, 2016.

[62] K. D. Devine, E. G. Boman, R. T. Heaphy, B. A. Hendrickson, J. D. Teresco, J. Faik, J. E. Flaherty, and L. G. Gervasio. New challenges in dynamic load balancing. *Applied Numerical Mathematics*, 52(2-3):133–152, 2005.

[63] M. H. Willebeek-LeMair and A. P. Reeves. Strategies for dynamic load balancing on highly parallel computers. *IEEE Transactions on parallel and distributed systems*, 4(9):979–993, 1993.

[64] J. Makino. A Fast Parallel Treecode with GRAPE. *Publications of the Astronomical Society of Japan*, 56(3):521–531, 06 2004. URL: `https://doi.org/10.1093/pasj/56.3.521`, `doi:10.1093/pasj/56.3.521`.

[65] M. Bader. *Space-filling curves: an introduction with applications in scientific computing*, volume 9. Springer Science & Business Media, 2012.

[66] H. Sagan. *Space-filling curves*. Springer Science & Business Media, 2012.

[67] B. Moon, H. V. Jagadish, C. Faloutsos, and J. H. Saltz. Analysis of the clustering properties of the hilbert space-filling curve. *IEEE Transactions on knowledge and data engineering*, 13(1):124–141, 2001.

[68] P. M. Campbell, K. D. Devine, J. E. Flaherty, L. G. Gervasio, and J. D. Teresco. Dynamic octree load balancing using space-filling curves. *Williams College Department of Computer Science, Tech. Rep. CS-03-01*, 2003.

[69] S. Aluru and F. E. Sevilgen. Parallel domain decomposition and load balancing using space-filling curves. In *Proceedings Fourth International Conference on High-Performance Computing*, pages 230–235. IEEE, 1997.

[70] M. Lieber, K. Gößner, and W. E. Nagel. The potential of diffusive load balancing at large scale. In *Proceedings of the 23rd European MPI Users' Group Meeting*, pages 154–157, 2016.

[71] K. Schloegel, G. Karypis, V. Kumar, R. Biswas, and L. Oliker. *A performance study of diffusive vs. remapped load-balancing schemes*. NASA Ames Research Center, Research Institute for Advanced Computer Science, 1998.

[72] G. Sutmann. Multi-level load balancing for parallel particle simulations. In *VI International Conference on Particle-based Methods–Fundamentals and Applications*. Jülich Supercomputing Center, 2019.

[73] D. E. Keyes and W. D. Gropp. A comparison of domain decomposition techniques for elliptic partial differential equations and their parallel implementation. *SIAM Journal on Scientific and Statistical Computing*, 8(2):s166–s202, 1987.

[74] B. Hendrickson and K. Devine. Dynamic load balancing in computational mechanics. *Computer methods in applied mechanics and engineering*, 184(2-4):485–500, 2000.

[75] S. Plimpton. Fast parallel algorithms for short-range molecular dynamics. *Journal of computational physics*, 117(1):1–19, 1995.

[76] S. Plimpton and B. Hendrickson. A new parallel method for molecular dynamics simulation of macromolecular systems. *Journal of Computational Chemistry*, 17(3):326–337, 1996.

[77] S. Hirschmann, D. Pflüger, and C. W. Glass. Towards understanding optimal load-balancing of heterogeneous short-range molecular dynamics. In *2016 IEEE 23rd International Conference on High Performance Computing Workshops (HiPCW)*, pages 130–141, 2016. doi:10.1109/HiPCW.2016.027.

[78] K. J. Bowers, R. O. Dror, and D. E. Shaw. Zonal methods for the parallel execution of range-limited n-body simulations. *Journal of Computational Physics*, 221(1):303–329, 2007.

[79] K. J. Bowers, R. O. Dror, and D. E. Shaw. The midpoint method for parallelization of particle simulations. *The Journal of chemical physics*, 124(18):184109, 2006.

[80] D. E. Shaw. A fast, scalable method for the parallel evaluation of distance-limited pairwise particle interactions. *Journal of computational chemistry*, 26(13):1318–1328, 2005.

[81] G. Mie. Zur kinetischen theorie der einatomigen körper. *Annalen der Physik*, 316(8):657–697, 1903. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/andp.19033160802, doi:10.1002/andp.19033160802.

[82] S. Werth, K. Stöbener, M. Horsch, and H. Hasse. Simultaneous description of bulk and interfacial properties of fluids by the mie potential. *Molecular Physics*, 115(9-12):1017–1030, 2017.

[83] M. Horsch, J. Vrabec, and H. Hasse. Modification of the classical nucleation theory based on molecular simulation data for surface tension, critical nucleus size, and nucleation rate. *Physical Review E*, 78(1):011603, 2008.

[84] J. Vrabec, M. Horsch, and H. Hasse. Molecular dynamics based analysis of nucleation and surface energy of droplets in supersaturated vapors of methane and ethane. *Journal of heat transfer*, 131(4), 2009.

[85] M. Horsch, Z. Lin, T. Windmann, H. Hasse, and J. Vrabec. The air pressure effect on the homogeneous nucleation of carbon dioxide by molecular simulation. *Atmospheric research*, 101(3):519–526, 2011.

[86] M. Horsch, J. Vrabec, M. Bernreuther, S. Grottel, G. Reina, A. Wix, K. Schaber, and H. Hasse. Homogeneous nucleation in supersaturated vapors of methane, ethane, and carbon dioxide predicted by brute force molecular dynamics. *The Journal of Chemical Physics*, 128(16):164510, 2008.

[87] S. Grottel, G. Reina, J. Vrabec, and T. Ertl. Visual verification and analysis of cluster detection for molecular dynamics. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1624–1631, 2007.

[88] M. Horsch, S. Miroshnichenko, and J. Vrabec. Steady-state molecular dynamics simulation of vapour to liquid nucleation with mcdonald's dæmon. *arXiv preprint arXiv:0911.5485*, 2009.

[89] K. Langenbach, M. Heilig, M. Horsch, and H. Hasse. Study of homogeneous bubble nucleation in liquid carbon dioxide by a hybrid approach combining molecular dynamics simulation and density gradient theory. *The Journal of chemical physics*, 148(12):124702, 2018.

[90] M. Horsch, H. Hasse, A. K. Shchekin, A. Agarwal, S. Eckelsbach, J. Vrabec, E. A. Müller, and G. Jackson. Excess equimolar radius of liquid drops. *Physical Review E*, 85(3):031605, 2012.

[91] S. Werth, S. V. Lishchuk, M. Horsch, and H. Hasse. The influence of the liquid slab thickness on the planar vapor–liquid interfacial tension. *Physica A: Statistical Mechanics and its Applications*, 392(10):2359–2367, 2013.

[92] M. Horsch and H. Hasse. Molecular simulation of nano-dispersed fluid phases. *Chemical Engineering Science*, 107:235–244, 2014.

[93] S. Werth, M. Kohns, K. Langenbach, M. Heilig, M. Horsch, and H. Hasse. Interfacial and bulk properties of vapor-liquid equilibria in the system toluene+ hydrogen chloride+ carbon dioxide by molecular simulation and density gradient theory+ pc-saft. *Fluid Phase Equilibria*, 427:219–230, 2016.

[94] S. Werth, G. Rutkai, J. Vrabec, M. Horsch, and H. Hasse. Long-range correction for multi-site lennard-jones models and planar interfaces. *Molecular Physics*, 112(17):2227–2234, 2014.

[95] S. Werth, M. Horsch, and H. Hasse. Surface tension of the two center lennard-jones plus quadrupole model fluid. *Fluid Phase Equilibria*, 392:12–18, 2015.

[96] S. Werth, M. Horsch, and H. Hasse. Surface tension of the two center lennard-jones plus point dipole fluid. *The Journal of chemical physics*, 144(5):054702, 2016.

[97] M. Horsch, M. Heitzig, C. Dan, J. Harting, H. Hasse, and J. Vrabec. Contact angle dependence on the fluid- wall dispersive energy. *Langmuir*, 26(13):10913–10917, 2010.

[98] M. Horsch, C. Niethammer, J. Vrabec, and H. Hasse. Computational molecular engineering as an emerging technology in process engineering. *arXiv preprint arXiv:1305.4781*, 2013.

[99] M. Horsch, J. Vrabec, M. Bernreuther, and H. Hasse. Poiseuille flow of liquid-methane in nanoscopic graphite channels by molecular dynamics simulation. In *Turbulence Heat and Mass Transfer 6. Proceedings of the Sixth International Symposium On Turbulence Heat and Mass Transfer*. Begel House Inc., 2009.

[100] S. Werth, M. Horsch, and H. Hasse. Long-range correction for dipolar fluids at planar interfaces. *Molecular Physics*, 113(23):3750–3756, 2015.

[101] P. Neumann, H. Flohr, R. Arora, P. Jarmatz, N. Tchipev, and H.-J. Bungartz. Mamico: Software design for parallel molecular-continuum flow simulations. *Computer Physics Communications*, 200:324–335, 2016.

[102] N. Tchipev, A. Wafai, C. W. Glass, W. Eckhardt, A. Heinecke, H.-J. Bungartz, and P. Neumann. Optimized force calculation in molecular dynamics simulations for the intel xeon phi. In *European Conference on Parallel Processing*, pages 774–785. Springer, 2015.

[103] R. Salomon-Ferrer, A. W. Götz, D. Poole, S. Le Grand, and R. C. Walker. Routine microsecond molecular dynamics simulations with amber on gpus. 2. explicit solvent particle mesh ewald. *Journal of chemical theory and computation*, 9(9):3878–3888, 2013.

[104] A. W. Gotz, M. J. Williamson, D. Xu, D. Poole, S. Le Grand, and R. C. Walker. Routine microsecond molecular dynamics simulations with amber on gpus. 1. generalized born. *Journal of chemical theory and computation*, 8(5):1542–1555, 2012.

[105] F. A. Gratl. Implementation and Evaluation of Task-based Approaches for Molecular Dynamics Simulations. Student research project, Technical University of Munich, Apr 2017.

[106] S. Stephan, S. Becker, K. Langenbach, and H. Hasse. Vapor-liquid interfacial properties of the system cyclohexane+ co2: Experiments, molecular simulation and density gradient theory. *Fluid Phase Equilibria*, 518:112583, 2020.

[107] J. Vrabec, M. Bernreuther, H.-J. Bungartz, W.-L. Chen, W. Cordes, R. Fingerhut, C. W. Glass, J. Gmehling, R. Hamburger, M. Heilig, et al. Skasim–scalable hpc software for molecular simulation in the chemical industry. *Chemie Ingenieur Technik*, 90(3):295–306, 2018.

[108] C. K. Birdsall and A. B. Langdon. *Plasma physics via computer simulation*. CRC press, 2004.

[109] D. Fincham. Leapfrog rotational algorithms. *Molecular Simulation*, 8(3-5):165–178, 1992.

[110] S. Seckler, F. Gratl, N. Tchipev, M. Heinen, J. Vrabec, H.-J. Bungartz, and P. Neumann. Load Balancing and Auto-Tuning for Heterogeneous Particle Systems using ls1 mardyn. Technical report, High-Performance Computing Center Stuttgart (HLRS), Jun 2019.

[111] J. C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kale, and K. Schulten. Scalable molecular dynamics with namd. *Journal of computational chemistry*, 26(16):1781–1802, 2005.

[112] S. Kumar, C. Huang, G. Almasi, and L. V. Kalé. Achieving strong scaling with namd on blue gene/l. In *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*, pages 10–pp. IEEE, 2006.

[113] S. Kumar, C. Huang, G. Zheng, E. Bohm, A. Bhatele, J. C. Phillips, H. Yu, and L. V. Kalé. Scalable molecular dynamics with namd on the ibm blue gene/l system. *IBM Journal of Research and Development*, 52(1.2):177–188, 2008.

[114] A. Bhatelé, L. V. Kalé, and S. Kumar. Dynamic topology aware load balancing algorithms for molecular dynamics applications. In *Proceedings of the 23rd international conference on Supercomputing*, pages 110–116, 2009.

[115] D. Van Der Spoel, E. Lindahl, B. Hess, G. Groenhof, A. E. Mark, and H. J. Berendsen. Gromacs: fast, flexible, and free. *Journal of computational chemistry*, 26(16):1701–1718, 2005.

[116] R. Halver, S. Schulz, and G. Sutmann. ALL - A loadbalancing library, C++ / Fortran library. URL: `https://gitlab.version.fz-juelich.de/SLMS/loadbalancing/-/releases`.

[117] M. Iwasawa, A. Tanikawa, N. Hosono, K. Nitadori, T. Muranushi, and J. Makino. Implementation and performance of fdps: a framework for developing parallel particle simulation codes. *Publications of the Astronomical Society of Japan*, 68(4):54, 2016.

[118] H.-J. Limbach, A. Arnold, B. A. Mann, and C. Holm. Espresso—an extensible simulation package for research on soft matter systems. *Computer Physics Communications*, 174(9):704–727, 2006.

[119] S. Hirschmann, M. Lahnert, C. Schober, M. Brunn, M. Mehl, and D. Pflüger. Load-balancing and spatial adaptivity for coarse-grained molecular dynamics applications. In *High Performance Computing in Science and Engineering'18*, pages 409–423. Springer, 2019.

[120] S. Deublein, B. Eckl, J. Stoll, S. V. Lishchuk, G. Guevara-Carrion, C. W. Glass, T. Merker, M. Bernreuther, H. Hasse, and J. Vrabec. ms2: A molecular simulation tool for thermodynamic properties. *Computer Physics Communications*, 182(11):2350–2367, 2011.

[121] C. W. Glass, S. Reiser, G. Rutkai, S. Deublein, A. Köster, G. Guevara-Carrion, A. Wafai, M. Horsch, M. Bernreuther, T. Windmann, et al. ms2: A molecular simulation tool for thermodynamic properties, new version release. *Computer Physics Communications*, 185(12):3302–3306, 2014.

[122] G. Rutkai, A. Köster, G. Guevara-Carrion, T. Janzen, M. Schappals, C. W. Glass, M. Bernreuther, A. Wafai, S. Stephan, M. Kohns, et al. ms2: A molecular simulation tool for thermodynamic properties, release 3.0. *Computer Physics Communications*, 221:343–351, 2017.

[123] R. Fingerhut, G. Guevara-Carrion, I. Nitzke, D. Saric, J. Marx, K. Langenbach, S. Prokopev, D. Celnỳ, M. Bernreuther, S. Stephan, et al. ms2: A molecular simulation tool for thermodynamic properties, release 4.0. *Computer Physics Communications*, 262:107860, 2021.

[124] L. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of OOPSLA'93*, pages 91–108. ACM Press, September 1993.

[125] S. Seckler, N. Tchipev, H.-J. Bungartz, and P. Neumann. Load balancing for molecular dynamics simulations on heterogeneous architectures. In *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*, pages 101–110. IEEE, 2016.

[126] C. L. Lawson and R. J. Hanson. *Solving least squares problems*. SIAM, 1995.

[127] V. Franc, V. Hlaváč, and M. Navara. Sequential coordinate-wise algorithm for the non-negative least squares problem. In *International Conference on Computer Analysis of Images and Patterns*, pages 407–414. Springer, 2005.

[128] J. Vrabec, J. Stoll, and H. Hasse. A set of molecular models for symmetric quadrupolar fluids. *The Journal of Physical Chemistry B*, 105(48):12126–12133, 2001.

[129] S. Griebel. Optimizing load balancing of heterogeneous particle simulations on heterogeneous systems. Bachelor's thesis, Technical University of Munich, Jul 2017.

[130] S. Seckler, F. Gratl, M. Heinen, J. Vrabec, H.-J. Bungartz, and P. Neumann. Autopas in ls1 mardyn: Massively parallel particle simulations with node-level auto-tuning. *Journal of Computational Science*, 50:101296, 2021. doi:10.1016/j.jocs.2020.101296.

[131] F. A. Gratl, S. Seckler, N. Tchipev, H.-J. Bungartz, and P. Neumann. Autopas: Auto-tuning for particle simulations. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 748–757. IEEE, 2019.

[132] F. A. Gratl, S. Seckler, H.-J. Bungartz, and P. Neumann. N Ways to Simulate Short-Range Particle Systems: Automated Algorithm Selection with the Node-Level Library AutoPas. *Computer Physics Communications*, submitted (as of June 2021).

[133] M. J. Abraham, T. Murtola, R. Schulz, S. Páll, J. C. Smith, B. Hess, and E. Lindahl. Gromacs: High performance molecular simulations through multi-level parallelism from laptops to supercomputers. *SoftwareX*, 1:19–25, 2015.

[134] S. Páll, A. Zhmurov, P. Bauer, M. Abraham, M. Lundborg, A. Gray, B. Hess, and E. Lindahl. Heterogeneous parallelization and acceleration of molecular dynamics simulations in gromacs. *The Journal of Chemical Physics*, 153(13):134110, 2020.

[135] P. Gonnet, A. B. Chalk, and M. Schaller. Quicksched: Task-based parallelism with dependencies and conflicts. *arXiv preprint arXiv:1601.05384*, 2016.

[136] F. A. Gratl. Implementation and evaluation of task-based approaches for molecular dynamics simulations. Studienarbeit, Technical University of Munich, Apr 2017.

[137] C. Tapus, I.-H. Chung, and J. K. Hollingsworth. Active harmony: Towards automated performance tuning. In *SC'02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, pages 44–44. IEEE, 2002.

[138] M. Iwasawa, D. Namekata, K. Nitadori, K. Nomura, L. Wang, M. Tsubouchi, and J. Makino. Accelerated fdps: Algorithms to use accelerators with fdps. *Publications of the Astronomical Society of Japan*, 72(1):13, 2020.

[139] M. Iwasawa, D. Namekata, R. Sakamoto, T. Nakamura, Y. Kimura, K. Nitadori, L. Wang, M. Tsubouchi, J. Makino, Z. Liu, et al. Implementation and performance of barnes-hut n-body algorithm on extreme-scale heterogeneous many-core architectures. *The International Journal of High Performance Computing Applications*, 34(6):615–628, 2020.

[140] M. Iwasawa, D. Namekata, K. Nomura, M. Tsubouchi, and J. Makino. Extreme-scale particle-based simulations on advanced hpc platforms. *CCF Transactions on High Performance Computing*, pages 1–13, 2020.

[141] T. Wilde, M. Ott, A. Auweter, I. Meijer, P. Ruch, M. Hilger, S. Kühnert, and H. Huber. Coolmuc-2: A supercomputing cluster with heat recovery for adsorption cooling. In *2017 33rd Thermal Measurement, Modeling & Management Symposium (SEMI-THERM)*, pages 115–121. IEEE, 2017.

[142] V. Weinberg and M. Allalen. The new intel xeon phi based system supermic at lrz.

[143] V. Weinberg and M. Allalen. First experiences with the intel mic architecture at lrz. *arXiv preprint arXiv:1308.3123*, 2013.

A Derivations

A.1 Lower Bound for Deviation Homogeneous Speed

In this section, a lower estimate for the deviation of an arbitrary node is derived, assuming homogeneous speed of all processes. For a leaf node, this deviation is defined as the quadratic deviation from the optimal load C_{opt} assigned to a process.

$$D_i = (C_i - C_{\text{opt}})^2 \tag{A.1}$$

For a non-leaf node, the deviation is defined as the sum of the deviations of its children. If recursion is applied, it is the sum of the deviations of all leaf nodes of which the original node is an ancestor.

$$D = \sum_{i \text{ is descendant leaf node}} (C_i - C_{\text{opt}})^2 \tag{A.2}$$

Adding and subtracting the expectation of the cost for the node, i.e., $\mathbb{E}(C_i) = \sum_{i \text{ is descendant leaf node}} C_i$ leads to (note that C_{opt} is the optimal load considering all subdomains/nodes).

$$= \sum_{i \text{ is descendant leaf node}} ((C_i - \mathbb{E}(C_i)) - (C_{\text{opt}} - \mathbb{E}(C_i)))^2 . \tag{A.3}$$

$$= \sum_i (C_i - \mathbb{E}(C_i))^2 - 2(C_{\text{opt}} - \mathbb{E}(C_i)) \underbrace{\sum_i (C_i - \mathbb{E}(C_i))}_{=0} + \sum_i (C_{\text{opt}} - \mathbb{E}(C_i))^2 \tag{A.4}$$

$$= \sum_i (C_i - \mathbb{E}(C_i))^2 + \sum_i (C_{\text{opt}} - \mathbb{E}(C_i))^2 \tag{A.5}$$

The minimum of the deviation of one term is thus

$$\inf D = n_{\text{descendant leaves}} (C_{\text{opt}} - \mathbb{E}(C_i))^2 , \tag{A.6}$$

where n is the number of descendant leaf nodes of the regarded node. Note that for the root note $C_{\text{opt}} = \mathbb{E}(C_i)$ and $\inf D$ is therefore zero.

The above formula can be used during the creation of the tree, i.e., for the splitting of a domain (with total load C and n processes) into two subdomains (with loads C_1 and

C_2 and process counts n_1 and n_2). The minimal deviation for such a splitting is given as

$$\inf D = n_1 \left(C_{\text{opt}} - \mathbb{E}\left(C_1\right)\right)^2 + n_2 \left(C_{\text{opt}} - \mathbb{E}\left(C_2\right)\right)^2 \tag{A.7}$$

$$= n_1 \left(C_{\text{opt}} - \frac{C_1}{n_1}\right)^2 + n_2 \left(C_{\text{opt}} - \frac{C_2}{n_2}\right)^2. \tag{A.8}$$

A.2 Lower Bound for Deviation Heterogeneous Speed

Analogously to section A.1, a lower bound can be found for the deviation of a non-leaf node if performance is considered. Here, the deviation of a leaf node is

$$D_i = \left(C_i - \frac{P_i}{P_{\text{avg}}} \cdot C_{\text{opt}} \right)^2, \tag{A.9}$$

where C_i is the load assigned to the leaf and P_i is the performance of the associated process. P_{avg} is the total average performance of all processes and C_{opt} the optimal load assigned to a process, assuming all processes have the same performance, i.e., $C_{\text{opt}} = \frac{C_{\text{total}}}{n_{\text{total}}}$, where n_{total} is the total number of processes. The deviation of a non-leaf node is thus

$$D = \sum_{i \text{ is descendant leaf node}} \left(C_i - \frac{P_i}{P_{\text{avg}}} \cdot C_{\text{opt}} \right)^2. \tag{A.10}$$

using $c_i \coloneqq C_i - \mathbb{E}(C_i)$ and $p_i \coloneqq P_i - \mathbb{E}(P_i)$

$$D = \sum_i \left(\mathbb{E}(C_i) + c_i - \frac{\mathbb{E}(P_i) + p_i}{P_{\text{avg}}} \cdot C_{\text{opt}} \right)^2 \tag{A.11}$$

$$= \sum_i \left(\left(\mathbb{E}(C_i) - \frac{\mathbb{E}(P_i)}{P_{\text{avg}}} \cdot C_{\text{opt}} \right) + \left(c_i - \frac{p_i}{P_{\text{avg}}} \cdot C_{\text{opt}} \right) \right)^2 \tag{A.12}$$

$$\begin{aligned} = &\sum_i \left(\mathbb{E}(C_i) - \frac{\mathbb{E}(P_i)}{P_{\text{avg}}} \cdot C_{\text{opt}} \right)^2 + \sum_i \left(c_i - \frac{p_i}{P_{\text{avg}}} \cdot C_{\text{opt}} \right)^2 \\ &+ 2 \left(\mathbb{E}(C_i) - \frac{\mathbb{E}(P_i)}{P_{\text{avg}}} \cdot C_{\text{opt}} \right) \cdot \sum_i \left(c_i - \frac{p_i}{P_{\text{avg}}} \cdot C_{\text{opt}} \right) \end{aligned} \tag{A.13}$$

with $\mathbb{E}(c_i) = 0$ and $\mathbb{E}(p_i) = 0$ it follows that

$$D = \sum_i \left(\mathbb{E}(C_i) - \frac{\mathbb{E}(P_i)}{P_{\text{avg}}} \cdot C_{\text{opt}} \right)^2 + \sum_i \left(c_i - \frac{p_i}{P_{\text{avg}}} \cdot C_{\text{opt}} \right)^2 \tag{A.14}$$

$$\geq \sum_i \left(\mathbb{E}(C_i) - \frac{\mathbb{E}(P_i)}{P_{\text{avg}}} \cdot C_{\text{opt}} \right)^2 \tag{A.15}$$

$$= n_{\text{descendant leaves}} \cdot \left(\mathbb{E}(C_i) - \frac{\mathbb{E}(P_i)}{P_{\text{avg}}} \cdot C_{\text{opt}} \right)^2. \tag{A.16}$$

When splitting a node in two subnodes with process counts n_1 and n_2 and average performances $P_{\text{avg},1}$ and $P_{\text{avg},2}$, as well as average loads $C_{\text{avg},1}$ and $C_{\text{avg},2}$, the lower bound for D is given as

$$\inf D = n_1 \left(C_{\text{opt}} \cdot \frac{P_{\text{avg},1}}{P_{\text{avg}}} - C_{\text{avg},1} \right)^2 + n_2 \left(C_{\text{opt}} \cdot \frac{P_{\text{avg},2}}{P_{\text{avg}}} - C_{\text{avg},2} \right)^2 \tag{A.17}$$

$$= n_1 \left(\frac{C_{\text{opt}} \cdot P_1}{n_1 \cdot P_{\text{avg}}} - \frac{C_1}{n_1} \right)^2 + n_2 \left(\frac{C_{\text{opt}} \cdot P_2}{n_2 \cdot P_{\text{avg}}} - \frac{C_2}{n_2} \right)^2. \tag{A.18}$$

B Cluster Descriptions

B.1 CoolMUC2

CoolMUC2 is one segment of the Linux Cluster [1] which is hosted at the Leibniz Supercomputing Center (LRZ) in Munich, Germany [2] [141]. It provides 812 dual-socket nodes and uses Intel Xeon E5-2697 v3 14-core Haswell CPUs. The nodes are water-cooled and connected with FDR Infiniband.

B.2 CoolMAC

CoolMAC, also called the MAC cluster, was a cluster hosted by LRZ and financed through the Munich Centre of Advanced Computing [3]. The cluster was a multi-purpose cluster that consisted of five partitions, each hosting different hardware. There were two large partitions *snb* and *bdz* that used Intel SandyBridge, resp. AMD Bulldozer nodes and two smaller partitions *nvd* and *ati* that allowed the usage of Nvidia, resp. AMD GPUs. Additionally, there was a small partition using Intel Westmere CPUs. An overview of the different partitions is given below.

Name	CPU	Nodes	Sockets x Cores	GPUs per Node
snb	Intel SandyBridge-EP Xeon E5-2670	28	2 x 8	-
bdz	AMD Bulldozer Opteron 6274	19	4 x 16	-
nvd	Intel SandyBridge-EP Xeon E5-2670	4	2 x 8	2 x NVIDIA M2090
ati	Intel SandyBridge-EP Xeon E5-2670	4	2 x 8	2 x AMD FirePro W8000
wsm	Intel Westmere-EX Xeon E7-4830	2	4 x 8	-

B.3 SuperMUC Phase 1

SuperMUC Phase 1 was hosted at LRZ and consisted of 18 islands that host a total of 9216 nodes. Each node housed two Sandy Bridge-EP Xeon E5-2680 eight-core processors

[1] `https://doku.lrz.de/display/PUBLIC/Linux+Cluster`
[2] `https://www.lrz.de`
[3] `http://www.mac.tum.de/`

resulting in a total of 147456 cores. The nodes were connected using Infiniband FDR10 in a fat-tree topology.

B.4 SuperMIC

SuperMIC was part of the SuperMUC system and contained 32 nodes. These nodes were dual-socket systems using Ivy-Bridge E5-2650 v2 processors [142, 143]. SuperMIC was meant for testing Intel Xeon Phi coprocessors cards and hosted two Intel Xeon Phi 5110P coprocessors per node. The coprocessors were connected using PCIe 2.0. For the node inter-connect, Infiniband FDR14 was used.

B.5 SuperMUC-NG

SuperMUC-NG is the current HPC system (as of June 2021) at LRZ [4]. SuperMUC-NG's main compute partition consists of 6336 dual-socket nodes partitioned into 8 islands. The system uses Intel Skylake Xeon Platinum 8174 CPUs (24 cores), providing a total of 304128 cores.

B.6 Hazel Hen

Hazel Hen was an HPC system at HLRS [5]. It consisted of 41 cabinets housing a total of 7712 nodes. Each node hosted two Intel Xeon E5-2680 v3 12-core CPUs. Hazel Hen's nodes were connected using Cray's Aries interconnect.

[4] https://doku.lrz.de/display/PUBLIC/SuperMUC-NG
[5] https://www.hlrs.de/systems/cray-xc40-hazel-hen/