Technische Universität München
Lehrstuhl für Informatik mit Schwerpunkt Wissenschaftliches Rechnen

Algorithmic and Implementational Optimizations of Molecular Dynamics Simulations for Process Engineering

Nikola Plamenov Tchipev

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

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzende(r):	Prof. Dr. Rüdiger Westermann
Prüfer der Dissertation:	1. Prof. Dr. Hans-Joachim Bungartz
	2. Prof. Dr. Jadran Vrabec
	3. Prof. Dr. Philipp Neumann

Die Dissertation wurde am 02.01.2020 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 07.04.2020 angenommen.

Contents

I Introduction **11**

II Foundations **15**

1 Molecular Dynamics with `ls1 mardyn` **17**
1.1 Molecular dynamics as an N-body problem . . . 17
1.2 Rigid body dynamics . . . 19
1.3 Time integration . . . 20
1.4 Forces and potential functions . . . 20
1.5 Short- and long-range forces . . . 22
1.6 Algorithms for short-range force calculation . . . 23
1.7 Further simulation packages for molecular dynamics . . . 26

2 Node-Level Performance for MD **29**
2.1 Target platforms . . . 29
2.2 HPC from a bottom-up perspective . . . 30
2.3 Performance characteristics of MD . . . 39
2.4 Performance metrics . . . 40

3 Fast Multipole Method **45**
3.1 General description . . . 45
3.2 Types of clusters . . . 45
3.3 Mathematical formulation . . . 47
3.4 Tree structure . . . 50
3.5 Separation criteria . . . 51
3.6 Tree traversals . . . 52
3.7 Algorithm . . . 54
3.8 Factors affecting runtime of FMM . . . 55
3.9 Further remarks . . . 56
3.10 Choices and difficulties upon implementation . . . 57

III Node-Level Performance for Molecular Dynamics **59**

4 Scalar-Level Performance **61**
4.1 Data structures and layout . . . 61
4.2 Floating-point precision . . . 69
4.3 Reduced memory mode . . . 75
4.4 Summary . . . 77

5 Vector-Level Performance **79**
5.1 Vectorization of nested loops . . . 81
5.2 SIMD divergence handling . . . 82
5.3 Vectorization for multicentered molecules . . . 83
5.4 Intrinsics wrappers . . . 85
5.5 Results . . . 87

5.6 Conclusion 93

6 Core-Level Performance 95
6.1 Force calculation study on KNC 96
6.2 Entire simulation 99
6.3 Conclusion 100

7 Multi-Core-Level Performance 103
7.1 OpenMP parallelization of MD simulations 104
7.2 Related work 105
7.3 First scalable schemes: `no3`, `c08`, `c18` 106
7.4 Parallelization of small systems: `cfb`, `tfb` 112
7.5 Tasks with dependencies: Quicksched, OpenMP 117

8 Multi-Socket-Level Performance 123
8.1 Low-synchronization schemes: `sli` and `c04` 123
8.2 Results 126
8.3 Conclusion 129
8.4 Outlook 129

9 Putting it all Together 131
9.1 Hybrid MPI-OpenMP performance 132
9.2 Comparison to LAMMPS and GROMACS 134
9.3 Comparison to WR13 on eight nodes 135
9.4 Petaflop molecular dynamics: a new world record 136
9.5 Foundations for a new, autotuning library 140
9.6 Summary 141

IV Fast Multipole Method 143

10 Basic implementation 145
10.1 Implementation 145
10.2 Convergence 146
10.3 Complexity 149

11 Acceleration of the M2L Phase 153
11.1 Choice of basis functions 153
11.2 Acceleration techniques 154
11.3 Results 156

12 Parallelizing the Fast Multipole Method 159
12.1 Sources of parallelism in FMM 159
12.2 Shared-memory parallelization 161
12.3 Distributed-memory parallelization 164
12.4 Conclusion 172

V Summary 175

A Lennard-Jones Kernel in Intrinsics Wrappers 177

B Full Metrics of VTune Analysis **179**

Acknowledgements

This work would not have been possible without the support of many people throughout the years. I would like to express my gratitude to my family and friends for being by my side all this time.

I would like to thank Prof. Dr. Hans-Joachim Bungartz for the opportunity to work at the chair of Scientific Computing and Computational Science at the Technical University of Munich. His supervision, support and personal example have been invaluable for me. I would like to also thank Prof. Dr. Philipp Neumann for his guidance and supervision since the beginning of my Master's studies. He has always been an inspiration for me with his organized hard work and persistence.

I am grateful to all colleagues from the chair and especially Steffen Seckler, Fabio Gratl, Jean-Matthieu Gallard, Michael Obersteiner and Wolfgang Eckhardt. Their help, advice and feedback made the project go much easier.

I thank all partners, with whom we have worked together on the projects SkaSim, TaLPas and as part of the Intel Parallel Computing Center at TU Munich and the Leibnitz Supercomputing Centre. In particular, I am very grateful to Prof. Dr. Jadran Vrabec, Matthias Heinen and Prof. Dr. Martin Horsch for their continual support on the practical applications of the code we developed. Also Dr. Martin Bernreuther and Dr. Colin Glass have provided insightful suggestions and feedback to me.

Last, but not least, I would like to thank the many students, who worked under my supervision for their hard work and patience. I thank Andrei Costinescu, Uwe Ehmann, Marat Faizov, Hanno Flohr, Wolfgang Hölzl, Benedikt Jaeger, Moritz Krügener, Micha Müller, Ludwig Peuckert and Thomas Schilling for their contributions to this project.

Abstract

Molecular dynamics (MD) simulations have become an important tool in a wide range of areas, such as life sciences, structural mechanics or process engineering and thermodynamics. Each of these areas has its own specific requirements, often addressed with specialized codes. In this work we focus on process engineering and thermodynamics with the program `ls1 mardyn`. A typical representative of N-body problems, MD tackles heavy computational loads, which mandate both implementational and algorithmic optimizations. In this work we focus on improving the node-level performance of `ls1 mardyn` and the Linked Cells algorithm in general and develop an implementation of the Fast Multipole Method (FMM) for the program.

The first part of the work addresses implementational improvements such as SIMD vectorization and shared-memory parallelization of the Linked Cells algorithm. The vectorization of `ls1 mardyn` is brought to maturity, while multiple new strategies are developed for the shared-memory parallelization. Several of these strategies were demonstrated to be very efficient, providing excellent scalability on up to 256 threads. The high node-level performance allowed the simulation of $2 \cdot 10^{13}$ molecules, representing a fivefold increase in the number of molecules simulated to date.

In the second part of this work, an algorithmic improvement to `ls1 mardyn` is made by developing a high performing version of FMM. An FFT acceleration is used to improve scalar performance of FMM, while a scheduling library is used for shared-memory parallelization. Several improvements for the distributed memory parallelization of FMM are presented, leading to an excellent performance of the newly introduced long-range interactions of `ls1 mardyn`.

Overall, our work serves to show how to improve the node-level performance of the Linked Cells algorithm and, in particular, introduces multiple scalable approaches for its shared-memory parallelization. Extensive improvements to the `ls1 mardyn` program are made, including a high-performing FMM implementation.

PART I

INTRODUCTION

With the increase in available computational power over the last decades, numerical simulations have become an indispensable tool in science and engineering. The advances in computer technology and the widespread availability of computing resources have led to an increase in their use for all kinds of tasks. The safety and low costs of computing power have made them a viable alternative to experiments. This is especially the case in areas where experiments are costly or hazardous, such as in reactor safety [52]. Moreover, there are many areas where experiments are simply not possible, such as in astrophysics [104]. In such areas, numerical simulations are the primary tool to test models and theories to explain the observed phenomena.

The various application fields of numerical simulation imply that the simulated processes differ greatly in the time and length scales coming into question. Figure 1 gives an overview on some of the different modelling and simulation approaches. When the problem in question is sufficiently large, one may assume that matter is continuous and solve differential equations for the quantities of interest, arising from the respective conservation laws. For example, for fluid flow problems in aerospace or automotive engineering, one would solve the Navier-Stokes equations for the distribution of the velocity and pressure around an airplane or a car. As the length scale of the considered problem decreases, however, intermolecular interactions play an ever increasing role and the continuum assumption begins to break down, leading to poorer fits between simulations and observations of real-world processes. An example again from fluid dynamics would be flow in porous media or at the interfaces between fluid phases. At such scales, so-called mesoscopic models, such as the Lattice-Boltzmann Method, might be more appropriate. Such methods aim to capture the behaviour of ensembles of particles, thereby coming closer to the actual molecular structure of matter. The increase in model quality, however, comes at the cost of higher computational effort, which places practical limitations on the simulated time scales. Going further down to even smaller problems, we arrive at molecular dynamics (MD), which is the topic of this thesis. For instance, if one wants to compute the flow in nanofilters, where channels would be wide enough to let only tens to hundreds of molecules through, interactions of individual molecules would need to be computed. The computational effort increases yet again, further restricting the time scales that can be simulated in a reasonable amount of time. Finally, at even smaller scales, subatomic interactions of electrons, protons and neutrons within a molecule require problems in Quantum Mechanics to be solved. The computational effort in these simulations is very high, restricting the systems to a few hundred atoms and very short timescales.

Molecular dynamics for process engineering with `ls1 mardyn` Molecular dynamics has a broad scope of applications ranging from life sciences [40] and structural mechanics [16] to process engineering and thermodynamics [60,112] among others. The particular application of MD in this work is process engineering. In this area, the goal is to predict various thermodynamic properties of either pure chemical species or mixtures under different conditions. A list of some of the multiple concrete applications can be found in [60,112]. Some topics include simulation of droplet formation during the condensation of a gas, bubble formation during boil-

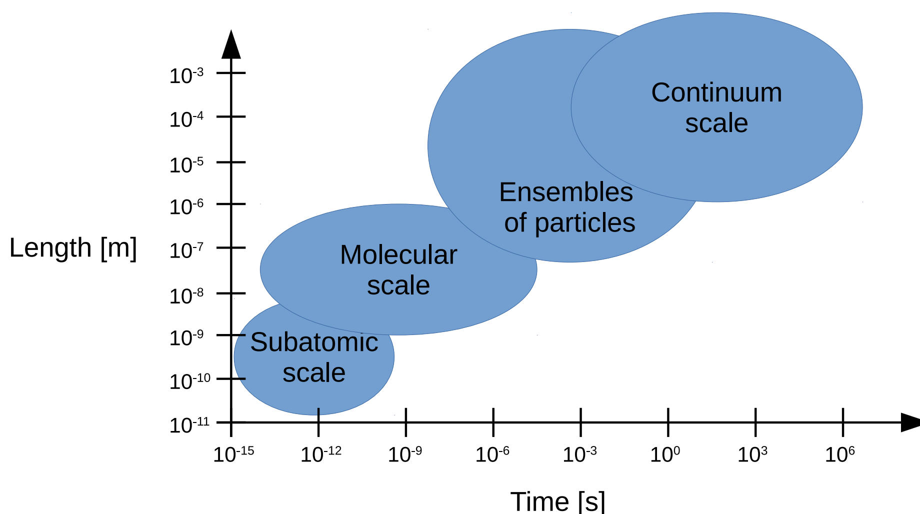


Figure 1: Time and length scales for different simulation methods.

ing of a liquid, gas separation and fluids in contact with surfaces among others. Further goals are computation of surface tension and phase-equilibrium properties, multicriteria optimization of molecular models, investigation of finite-size effects for vapour-liquid interfaces and more. The insight provided by these simulations can be used for optimizing industrial processes and improving the predictive quality of models.

The simulation software for these processes is, in itself of interest for the area of Scientific Computing. Multiple simulation packages have emerged through the years, such as `LAMMPS` [90], `GROMACS` [2] or `ls1 mardyn` [81]. These codes are developed continuously over the years by many contributors, resulting not only in great capabilities of the software, but also in great complexity of the code. Learning to use any of those tools requires effort on the side of application scientists, which places a need for a continuous improvement and support of the code over decades. As the computing hardware changes, the codes need to be adapted to it, so that the performance of the new hardware can be properly utilised. Such changes to the code fall in the area of High-Performance Computing (HPC) optimization. Moreover, the never ending interest of researchers to simulate new phenomena requires also new models to be added to the codes. In the current thesis, we continue the development of the simulation program `ls1 mardyn` aiming at MD applications in process engineering.

`ls1 mardyn` (**l**arge **s**ystems **1**: **m**olecul**ar dyn**amics) is a relatively young simulation program developed since 2005 [15,25,57,81]. The development takes place jointly by several German institutions, primarily the Technical University of Munich, the High Performance Computing Center Stuttgart, the Technical University of Kaiserslautern and the Technical University of Berlin. The code specializes in simulations of small, rigid multicentered molecules. It is written in C++ and features a modular, object-oriented structure throughout its entire code base. The program places an emphasis on memory-efficiency and high performance and was used to set a world record for the largest molecular dynamics simulation in 2013 [27]. The code is open-source and available under the BSD 2-clause license from `ls1-mardyn.de`.

Goals of this thesis In this thesis, we consider "algorithmic" as well as "implementational" improvements to the program `ls1 mardyn`. Under "implementational" optimizations we understand HPC optimizations to the code of the simulation. We are, thus, aiming to improve the utilisation of the hardware, without changing the mathematical description of the program. This

predominantly means that we want to make the same program run faster on the same hardware, though improving or maintaining memory-efficiency is also of interest. The implementational optimizations that we focus on, fall in the category of node-level performance optimization. That means that we focus primarily on data structures, single instruction, multiple data (SIMD) vectorization and shared-memory parallelization. Distributed-memory parallelization falls beyond the scope of this work.

Under "algorithmic" optimizations, on the other hand, we understand changes to the mathematical formulation of the code. In simple words, we want to add new functionality to the program. The algorithmic improvement considered in this thesis, is the development of a Fast Multipole Method (FMM) [48] implementation for `ls1 mardyn`. Due to reasons detailed in later chapters, the algorithm is developed anew within `ls1 mardyn`, which also implies that it needs to be HPC optimized and parallelized as well, in order to provide a competitive performance.

Structure of this work This thesis is structured as follows. Part II presents the foundational knowledge necessary for understanding the contributions of this work. Basic introductions to molecular dynamics, node-level performance and the Fast Multipole Method are given. Part III presents the implementational work on `ls1 mardyn`. Optimizations of the program are presented in a bottom-up fashion, starting from the core algorithms and data structures and going up to parallelization for shared-memory execution in inter-socket CPU node configurations. In Part IV, we present the algorithmic improvement to `ls1 mardyn`, a newly developed FMM implementation. Sequential optimizations and parallelization of the FMM algorithm are presented. The balance of this thesis between algorithmic and implementational improvements is shifted towards implementational ones, as also reflected by the length of the respective parts. Finally, in Part V we summarize the contributions of this work.

Publications The following publications present some of the primary contributions of the author of this thesis: [108,109]. The work of [109] served to show that memory-efficient OpenMP parallelizations for the core algorithms of `ls1 mardyn` can be very efficient and paved the way for the rest of the (subsequently) discovered parallelization schemes. The work of [108] represents the culmination of this work, in which the new schemes enabled the setting of a new world record for the largest molecular simulation to date.

The author of this work also contributed to a significant degree to the following publications on HPC optimizations of MD and `ls1 mardyn`: [46,98]. In the works [92,94,95], the lessons learned by the author were applied to other areas, which fall outside the scope of MD.

The following student projects were supervised as part of this work: [32,35,38,39,44,45,59, 62,78,82,83,88,97]. Multiple of these projects resulted in important insights and results from them were, hence, included in this document.

PART II

FOUNDATIONS

In this part, we introduce the foundational knowledge, needed to present our contributions. A basic introduction to molecular dynamics is given, together with the governing mathematical and physical models. The numerical approximations to those models are presented as well as the most popular algorithms for implementing them. Some of the available molecular dynamics packages are introduced. Next, we give an introduction to the area of node-level performance. Basic features of the current hardware, relevant to the discussion in later chapters, are introduced. Different approaches and strategies for utilising those features are discussed, together with metrics for evaluating performance. Finally, in the third chapter of this part, a general introduction to the Fast Multipole Method is presented.

1

Molecular Dynamics with `ls1 mardyn`

In this chapter we present some of the basic theory of molecular dynamics simulations. We do not aim to give an exhaustive overview, rather only the main aspects, which are relevant to the later chapters and the simulation program `ls1 mardyn`. Exhaustive overviews can be found in [51,93]. Some aspects of the theory, which need to be discussed in greater depth in order to present the new optimizations and results, are delegated to Parts III and IV.

1.1 Molecular dynamics as an N-body problem

Molecular dynamics falls in the category of simulations of the N-body problem, together with Astrophysics [104], Smooth-Particle Hydrodynamics [73] and Vortex Dynamics [18,59] among others. The N-body problem can be stated as follows: given are N bodies with positions $\vec{x}_i$, velocities $\vec{v}_i$ and masses m_i for $i = 1, \ldots, N$ at some initial time t_0. Considering the mutual interactions, the goal is to compute the positions and velocities of the system at subsequent times.

In order to advance the system in time, accelerations $\vec{a}_i$ are introduced and Newton's equations of motion are solved:

$$\frac{d\vec{x}_i}{dt} = \vec{v}_i, \tag{1.1}$$

$$\frac{d\vec{v}_i}{dt} = \vec{a}_i \tag{1.2}$$

$$\tag{1.3}$$

The accelerations $\vec{a}_i$ are determined from the forces acting on each particle $\vec{F}_i$ via Newton's second law:

$$\vec{a}_i = \frac{\vec{F}_i}{m_i}. \tag{1.4}$$

In turn, each of the forces $\vec{F}_i$ is computed by summing the force contributions of all pairs of forces $\vec{F}_{ij}$ acting between particles i and j:

$$\vec{F}_i = \sum_{j=1;j\neq i}^{N} \vec{F}_{ij}. \tag{1.5}$$

For the exposition in this document, pairwise forces $\vec{F}_{ij}$ are sufficient. In general, however, it is possible to include also other force contributions in Equation (1.5). These could be external forces acting on every body in the system (e.g. gravity in molecular dynamics simulations) or contributions involving more than two bodies (e.g. bond angles or torsions).

Listing 1.1: Basic Algorithm of N-body simulations

```
init(); // initialise positions, velocities, time parameters t=t0, dt, T_end
while(t + dt < T_end) {
	// advance time
	t += dt;

	// compute new positions
	for (i = 0; i < N-1; ++i)
		x[i] = advance_position(x[i], v[i], f[i]);

	// compute new forces
	for (i = 0; i < N-1; ++i) {
		f[i] = 0.0;
		for (j=0; j < N-1; ++j) {
			if (i != j)
				f[i] += force(Bodies[i], Bodies[j]);
		}
	}

	// compute new velocities
	for (i = 0; i < N-1; ++i)
		v[i] = advance_velocity(v[i], f[i]);
}
```

Basic algorithm Implementations of N-body simulations and molecular dynamics in particular, usually follow the form outlined in Listing 1.1. The precise forms of the `advance_positions()` and `advance_velocities()` and `force()` functions depend on the chosen time integration scheme and the forces acting between the bodies. Time integration schemes are discussed below in Section 1.3, while forces are discussed in Section 1.4.

As suggested already by Listing 1.1, the computation of positions and velocities are $\mathcal{O}(N)$ operations, as the processing of one molecule does not depend on other molecules. The force calculation for one molecule, however, involves, generally, all $N - 1$ other molecules. It is, thus, an $\mathcal{O}(N^2)$ operation and often the most time-consuming part of the simulation. For this reason, most of the remainder of this work will be dedicated to the (pairwise non-bonded) force calculation. It is the primary concern of Part III.

Without further assumptions, the calculation of the forces is an expensive $\mathcal{O}(N^2)$ operation, since every particle interacts with every other particle. Many of the employed force kernels obey Newton's third law, however:

$$\vec{F}_{ij} = -\vec{F}_{ji}. \tag{1.6}$$

If it is made use of, the number of force calculations can be halved, which is a considerable gain. This is achieved in the following way: after the force contribution on particle i due to particle j has been computed — F_{ij} — it is added to the force buffer of particle i and subtracted from the force buffer of particle j. This optimization comes with a significant cost, however, because it makes the parallel implementation of the force calculation challenging. Hence, throughout this work, Newton's third law optimization plays an important role. We will denote it as *Newton3* for the remainder of this work.

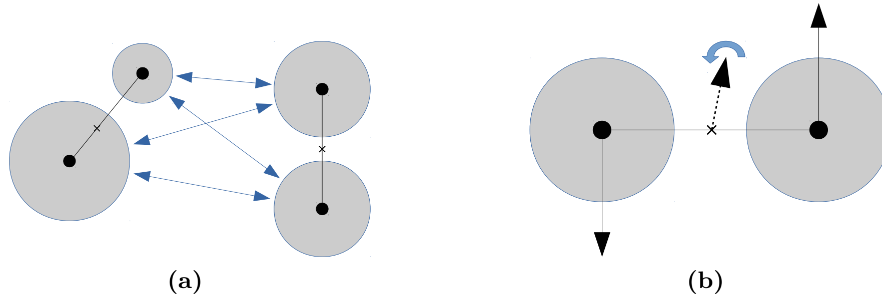


Figure 1.1: (a) Force calculation between two two-centered rigid bodies. All interaction sites interact with each other, resulting in a calculation of a total of four force pairs. (b) Forces on sites of a multicentered body may induce rotational motion. Hence, torques and angular momenta need to be accounted for.

1.2 Rigid body dynamics

In the case of `ls1 mardyn` (and often in N-body simulation in general) it is useful to consider two or more point-bodies, whose position relative to each other is fixed. Such a body is called a "rigid body" and can be used to represent small multicentered molecules, comprising different atoms. The assumption one makes in doing this is that all bond lengths and bond angles of a molecule are fixed. We will refer to the point bodies constituting a single rigid body as "interaction centers" or "interaction sites" for the remainder of this work. Since `ls1 mardyn` considers rigid multicentered molecules, we extend the discussion of Section 1.1 to rigid bodies. Figure 1.1(a) illustrates the calculation of forces between two two-centered bodies.

In general, let S_i denote the set of interaction sites of molecule i. Then, for each molecule i, the force contribution between each interaction site of i and each interaction site of every other molecule needs to be computed. The force calculation of Equation (1.5) then becomes:

$$\vec{F}_i = \sum_{\substack{j=1 \\ j \neq i}}^{N} \sum_{k \in S_i} \sum_{l \in S_j} \vec{F}_{kl}. \tag{1.7}$$

Thus, the number of operations, necessary to carry out the force calculation, grows quadratically with the number of interaction sites per rigid body. Moreover, multicentered bodies have additional degrees of freedom as opposed to single-centered spherical bodies. Figure 1.1(b) illustrates this fact. If two sites of a molecule experience forces orthogonal to the axis of the molecule and in opposite direction, then a rotatory motion is induced. Thus, the respective torques need to be computed:

$$\vec{\tau}_i = \sum_{k \in S_i} \vec{d}_k \times \vec{F}_k, \tag{1.8}$$

where $\vec{F}_k$ is the force on the k-th site, $\vec{d}_k$ is the vector pointing from the center-of-mass of the molecule to the k-th site and $\times$ denotes the cross product. From these torques, the rotational equations of motion arise:

$$\frac{d\vec{\omega}_i}{dt} = I_i^{-1} \vec{\tau}_i, \tag{1.9}$$

where $\vec{\omega}_i$ is the angular velocity and I_i is the moment of inertia of the i-th rigid-body. Moreover, apart from translational kinetic energy, the body, thus, also has rotational kinetic energy, which needs to be taken into account by the physical models accordingly.

Different possibilities for storing molecular data for multicentered molecules exist. In `ls1 mardyn`, the position of the center of mass of a molecule is stored together with its orientation in terms of a quaternion [68], instead of the position of each individual site. The positions of the individual sites relative to the center of mass are stored once per chemical species for a molecule with a certain orientation. From this data, the absolute positions of each molecular site can be computed when needed.

1.3 Time integration

The time integration scheme concerns itself with the solution of the system of ordinary differential equations of Equations (1.1) and (1.2). In `ls1 mardyn`, a variant of the Leapfrog algorithm (see e.g. [51]) is being used:

$$\vec{v}\left(t+\frac{\Delta t}{2}\right) = \vec{v}\left(t-\frac{\Delta t}{2}\right) + \Delta t\vec{a}(t), \tag{1.10}$$

$$\vec{r}(t+\Delta t) = \vec{r}(t) + \Delta t\vec{v}\left(t+\frac{\Delta t}{2}\right). \tag{1.11}$$

$$\tag{1.12}$$

Note that the velocity is shifted by one half timestep. In order to evaluate kinetic and potential energy at the same physical time, however, the velocity update is often split up into two half-steps ([93]):

$$\vec{v}\left(t+\frac{\Delta t}{2}\right) = \vec{v}\,(t) + \frac{\Delta t}{2}\vec{a}(t), \tag{1.13}$$

$$\vec{r}(t+\Delta t) = \vec{r}(t) + \Delta t\vec{v}\left(t+\frac{\Delta t}{2}\right), \tag{1.14}$$

$$\vec{v}\,(t+\Delta t) = \vec{v}\left(t+\frac{\Delta t}{2}\right) + \frac{\Delta t}{2}\vec{a}(t). \tag{1.15}$$

$$\tag{1.16}$$

The particular Leapfrog flavour, implemented in `ls1 mardyn`, is the rotational variant [34]. Other schemes for time integration are also possible, see e.g. [51,93].

1.4 Forces and potential functions

Molecular dynamics employs *conservative forces* for the pairwise force contributions. This means that a scalar potential function $U : \mathbb{R} \to \mathbb{R}$ can be defined, from which the forces can be generated by computing the gradient of the potential:

$$\vec{F} = -\nabla U. \tag{1.17}$$

In this section we present the interaction potentials supported by `ls1 mardyn`. These are the Lennard-Jones 12-6 potential and the electrostatic potentials.

Lennard-Jones potential A potential, used very frequently in molecular dynamics, is the 12-6 Lennard-Jones potential [72]:

$$U_{ij}^{LJ} = 4\varepsilon_{ij}\left(\left(\frac{\sigma_{ij}}{r_{ij}}\right)^{12} - \left(\frac{\sigma_{ij}}{r_{ij}}\right)^{6}\right), \tag{1.18}$$

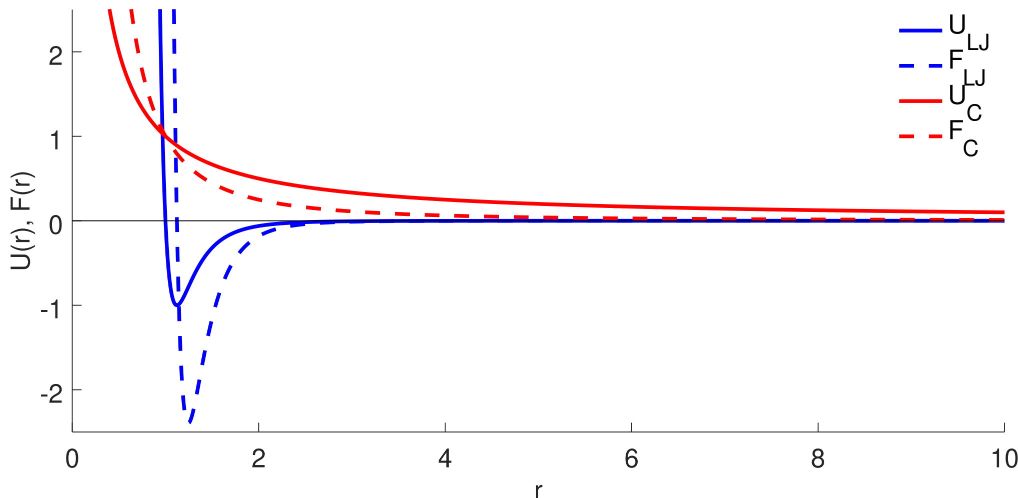


Figure 1.2: Dimensionless potential and force for the Lennard-Jones potential with $\varepsilon = 1$ and $\sigma = 1$ and the Coulomb potential for $q = 1$ and $\frac{1}{4\pi\epsilon_0} = 1$.

where $r_{ij} = |\vec{r}_i - \vec{r}_j|$ denotes the distance between the two particles, σ_{ij} and ε_{ij} denote parameters of the potential, dependent on the type of particles i and j. For molecules (or sites) of the same chemical species, the parameters become independent of the individual molecules and are constant: $\varepsilon_{ij} = \varepsilon$ and $\sigma_{ij} = \sigma$. For example, when simulating Argon values of $\varepsilon = k_B \cdot 125.7$ K, $\sigma = 0.3345$ nm can be used [120], where $k_B = 1.38 \cdot 10^{-23}$ J/K is the Boltzmann constant.

Figure 1.2 plots the 12-6 Lennard-Jones potential and its force for $\varepsilon = 1$ and $\sigma = 1$. The parameter ε determines the depth of the potential well, which is at $r' = 2^{\frac{1}{6}}\sigma$. The value of σ determines the zero-crossing of the potential and is also a measure of the diameter of the particles.

The Lennard-Jones potential models Pauli repulsion at close distances (the r^{-12} term) and Van der Waals attraction at long distances (the $-r^{-6}$ term). The repulsion term prevents molecules from coming unphysically close to each other. In the MD simulations we consider, all molecules have at least one Lennard-Jones interaction site, which prevents them from coming too close to each other.

Finally, if the molecules are not of the same type, mixing rules determine the ε_{ij} and σ_{ij} from the ε and σ parameters of the individual species. Some of the most commonly used mixing rules are the Lorentz-Berthelot rules [11,74]:

$$\varepsilon_{ij} = \sqrt{\varepsilon_i \varepsilon_j}, \quad \sigma_{ij} = \frac{\sigma_i + \sigma_j}{2}. \tag{1.19}$$

These rules simply average the parameters of the pure substances in an appropriate fashion.

Electrostatic potentials Apart from the Lennard-Jones potential, electrostatically charged molecules interact also through the Coulomb potential. Two point charges give rise to the following potential energy:

$$U^{qq}(r_{ij}) = \frac{1}{4\pi\epsilon_0} \frac{q_i q_j}{r_{ij}}, \tag{1.20}$$

where ϵ_0 is the vacuum permittivity constant, q_i and q_j are the charges of the two point charges and r_{ij} is the distance between them. Figure 1.2 also shows the electrostatic potential. As can

be observed already from the figure, it decays much slower than the Lennard-Jones potential. This will be discussed in more detail in the next section.

If two charges of equal magnitude and opposite sign are only a small distance apart, their combined electrostatic potential can be approximated at large distances as a point-dipole. A point dipole interacts with a charge or with other dipoles, which gives rise to the following potentials [25,47,116]:

$$U^{q\mu}(r_{ij},\omega_j) = \frac{1}{4\pi\epsilon_0}\frac{q_i\mu_j}{r_{ij}^2}\cdot f_1(\omega_j), \tag{1.21}$$

$$U^{\mu\mu}(r_{ij},\omega_i,\omega_j) = \frac{1}{4\pi\epsilon_0}\frac{\mu_i\mu_j}{r_{ij}^3}\cdot f_2(\omega_i,\omega_j), \tag{1.22}$$

where r_{ij} is the distance and f_1 and f_2 are dimensionless functions of the orientation angles ω_i, ω_j of the dipoles.

In the same fashion, if two point dipoles of equal moment and opposite direction are only a small distance apart, their combined electrostatic potential can be approximated at large distances as a point-quadrupole. The interaction of point quadrupoles with charges, dipoles or other quadrupoles gives rise to the following potentials [25,47,116]:

$$U^{qQ}(r_{ij},\omega_j) = \frac{1}{4\pi\epsilon_0}\frac{q_iQ_j}{r_{ij}^3}\cdot f_3(\omega_j), \tag{1.23}$$

$$U^{\mu Q}(r_{ij},\omega_i,\omega_j) = \frac{1}{4\pi\epsilon_0}\frac{\mu_iQ_j}{r_{ij}^4}\cdot f_4(\omega_i,\omega_j), \tag{1.24}$$

$$U^{QQ}(r_{ij},\omega_i,\omega_j) = \frac{1}{4\pi\epsilon_0}\frac{Q_iQ_j}{r_{ij}^5}\cdot f_5(\omega_i,\omega_j), \tag{1.25}$$

where r_{ij} is the distance and f_3, f_4 and f_5 are again dimensionless functions of the orientation angles ω_i and ω_j.

While the point-dipole and point-quadrupole potentials are more complex to compute, they reduce the number of distance calculations between different molecules, thereby reducing the computational complexity of the program. The dipole and quadrupole potentials are also special cases of the multipole expansion, which will be elaborated upon in the context of the Fast Multipole Method in Chapter 3.

Molecular model notation Following [15], we use the notation `l`CLJ`c`C`d`D`q`Q notation for multicentered models of molecules. The numbers `l,c,d` and `q` denote, respectively, the number of Lennard-Jones-, Coulomb-, dipole- and quadrupole sites. A value of zero is omitted. Thus, for example, 1CLJ denotes single-centered molecules with one Lennard-Jones site; 1CLJ3C denotes a molecule with one Lennard-Jones site and three Coulomb sites and so on.

1.5 Short- and long-range forces

As mentioned in Section 1.4, and illustrated in Figure 1.2, some potential functions decay rapidly with increasing distance from the source, while others do not. Depending on how fast a potential function $U(r)$ decays, one distinguishes between *short-range* and *long-range* potentials. One criterion for distinguishing between them is given in [51]: functions decaying faster than r^{-D} (in D spatial dimensions) are short-range, while ones decaying slower are long-range functions. Thus, the Lennard-Jones potential is short-range, because it decays like r^{-6}, while the Coulomb potential is long-range, because it decays like r^{-1} in three dimensions.

Short-range potentials If the potential decays rapidly one can evaluate all interactions up to a certain cutoff radius r_c and truncate all remaining interactions. In doing so, one implicitly assumes that interactions beyond the cutoff-radius are negligible or cancel each other out. This is the case if particles are distributed homogeneously. If particles are distributed inhomogeneously, one can still end up with unphysical results and further corrections might be necessary. For example, if beyond the cutoff radius there are many more particles to one side of a particle than on the other, it would be attracted to the large group of particles due to the accumulation of many attractive forces, no matter how small they may be. Truncating these interactions would not be able to reproduce this effect.

Truncating interactions at distances larger than r_c solves the problem of the $\mathcal{O}(N^2)$ complexity, however. This is because only a fixed number of molecules can fit within a ball with radius r_c. As pointed out in Section 1.4, molecules and atoms in MD do not come arbitrarily close to each other due to the repulsive part of the Lennard-Jones potential and have a certain diameter. Thus, any ball of a fixed radius can contain only a fixed amount of molecules. An upper bound C of the number of interactions per molecule can be easily obtained from the volumes of the cutoff ball and of the molecules. For a monatomic molecule, modelled with one Lennard-Jones site, the upper bound C is given by $C = \frac{8r_c^3}{\sigma^3}$. Thus, every molecule can interact at most with C molecules, which leads to the desired linear $\mathcal{O}(N \cdot C)$ complexity. The most popular algorithms for realizing short-range force calculation are introduced in Section 1.6.

Throughout the remainder of this work, we will often refer to the molecules, which fall within the cutoff sphere of a molecule as its interaction neighbours or interaction partners. The process of determining which molecules are neighbours will often be referred to as neighbour finding or neighbour search.

Long-range potentials If the potential function does not decay fast enough, then introducing cutoffs and truncation generally leads to inaccurate results [33,124]. In such cases, some method of accounting for the interactions between all pairs of molecules is needed. In order to circumvent the $\mathcal{O}(N^2)$ complexity, fast summation methods such as Ewald summations [31], PME [22], P3M [23], the Fast Mutlipole Method [48] or, more recently, Multilevel Summation Method [54] can be used. Our choice is the Fast Multipole Method, which will be presented in more detail in Chapter 3.

For molecules, whose total net charge is zero, however, the lowest-order interactions become dipole-dipole interactions, whose potential function decays like r^{-3}, as seen from Equation (1.22). While this is still not a short-range interaction in three dimensions as per the definition of [51], the reaction field method [4,8] can be used in this case, which is also a cutoff-based method. This is `ls1 mardyn`'s default method for handling electrostatic interactions.

1.6 Algorithms for short-range force calculation

1.6.1 Direct N^2 calculation

Figure 1.3(a) shows the naive approach for computing short-range forces. If no auxiliary structures are used, then every molecule interacts only with its neighbours, but the neighbour search needs to check the distance to all other molecules, in order to determine whether they are neighbours or not. Although this avoids the $\mathcal{O}(N^2)$ evaluations of the force kernel, it still requires $\mathcal{O}(N^2)$ operations for distance calculations. This renders the method generally unfeasible,

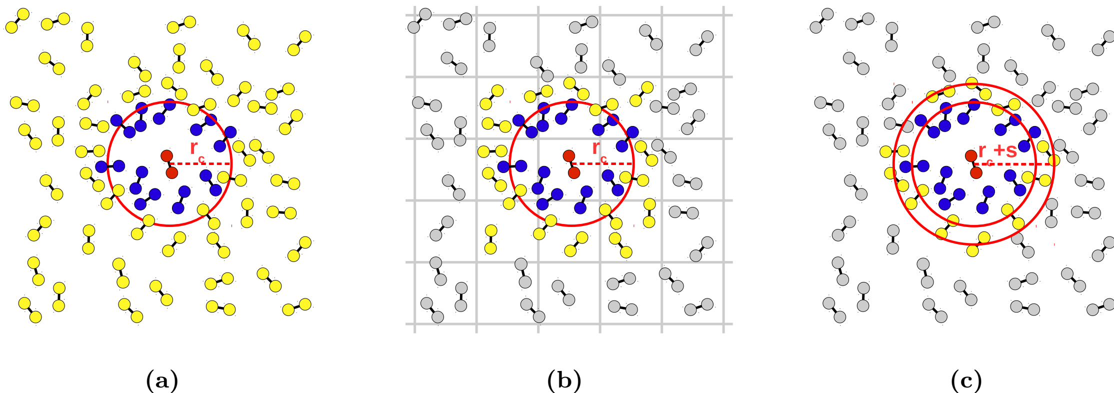


Figure 1.3: Methods for short-range force calculation for the molecule in red. The molecules in blue are the interaction neighbours of the molecule in red. The distance to molecules in yellow is checked by the respective algorithms, in order to determine whether they are neighbours of the molecule in blue or not. Distances to molecules in grey do not need to be checked by the respective algorithms. (a) Direct N^2 calculation. (b) Linked Cells method. (c) Verlet Lists method.

except in special cases[1].

1.6.2 Linked Cells method

Figure 1.3(b) illustrates the Linked Cells method [51,93], which is the method used by `ls1 mardyn`. The difference to the Direct N^2 method is that molecules get sorted into cubic bins according to their positions. If the side length of the bins l is $l \geq r_c$, then all neighbours of a molecule can be found by searching only the cell containing the molecule and its $3^D - 1$ adjacent cells. Again, due to the nonzero volume of each molecule, only a certain number of molecules may fit in those cells, leading to an $\mathcal{O}(N)$ complexity of the neighbour search. Some book-keeping is required to resort the molecules in the correct bins as they move around during the simulation, but it requires $\mathcal{O}(1)$ operations per particle and is, hence, not a particular bottleneck.

A drawback of the Linked Cells method is that it still suffers from a relatively high amount of cutoff-condition checks. The probability that a checked molecule is inside the cutoff radius is given by the ratio of the volumes of a D-dimensional sphere with radius r_c and a D-dimensional hypercube of side-length $3r_c$. In $D = 3$ dimensions, this gives a "hit-rate" of:

$$\frac{\frac{4\pi r_c^3}{3}}{(3r_c)^3} = \frac{4\pi}{3^4} \approx 15.5\%. \tag{1.26}$$

As we shall see in Section 1.6.3, the Verlet Lists method can achieve substantially higher values. For this reason, effort has gone in increasing the hit-rate. One possibility is to use bins of side-lengths $\leq r_c$, at the cost of checking more bins. Carrying out the same calculation as in Equation (1.26) for $l = \frac{r_c}{2}$ gives $\approx 26.8\%$, which, however, comes at the cost of traversing $5^3 = 125$ cells instead of $3^3 = 27$ cells to find all neighbours. This additional overhead of accessing $5\times$ more bins limits the gains of the higher hit-rate. For yet smaller side-lengths the

[1] For example in the limit of strong scaling with MPI parallelization, where the domain per MPI process becomes ever smaller with increasing the number of MPI ranks.

cell configuration	hit-rate	# occurrences per cell
within cell	90%	1
common face	33%	6
common edge	9%	12
common corner	2%	8

Table 1.1: Hit-rate and number of calls per cell for different cell-pair configurations.

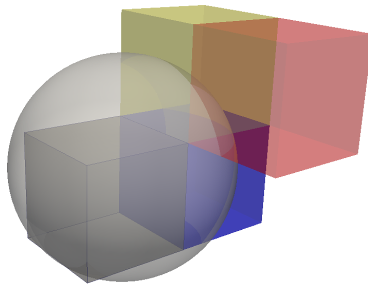


Figure 1.4: The hit-rate of the neighbour search is highest within the cell itself (grey), and decreases for cell pairs which share a face (grey and blue), edge (grey and yellow), or corner (grey and red).

costs of accessing numerous bins, containing ever fewer molecules per bin, outweigh the gains of the increasing hit-rate [15]. For this reason, codes typically use $l = r_c$ or $l = \frac{r_c}{2}$. In `ls1 mardyn` the $l = r_c$ variant is the default one, although a version with an adaptive choice of the side-length was present in the code for a while [15,81]. Other approaches for increasing the hit-rate of Linked Cells also exist ([41]), but again involve other sources of overhead.

For the sake of the discussion in Section 5.5.1, we point out that the hit-rate varies strongly between pairs of cells, depending on the spatial configuration of the two cells. Figure 1.4 illustrates the different spatial configurations of pairs of cells for the $l = r_c$ case. A simple Monte Carlo estimate of the hit-rate is given in Table 1.1. As can be seen from Table 1.1, the hit-rate within a cell is about 90%, but drops down to 2% for cells sharing a common corner. The overall hit-rate of the Linked Cells algorithm is, then, an average of the hit-rates of the different configurations, weighted by the number of occurrences of each configuration.

1.6.3 Verlet Lists Method

Apart from the Linked Cells method, another popular method for short-range algorithms is the Verlet neighbourhood list method [93,110]. Figure 1.3(c) illustrates the basic idea. This algorithm makes use of the observation that molecules move relatively slowly in MD. Thus, the set of neighbours of a molecule in timestep t_n is nearly the same as the set of neighbours in timestep t_{n+1}. It is, thus, worthwhile to “remember” the set of neighbours that a molecule has interacted with at time t_n and somehow reuse this set at time t_{n+1}. What is left is to account for molecules, which may enter or exit the cutoff-sphere between t_n and t_{n+1}. This is done by building the set of neighbours for a slightly larger sphere with radius $r_c + s$, where s is referred to as “skin thickness”. The molecules in this list of neighbours (and neighbour-candidates) then, again need to be checked whether they are in the cutoff radius or not. But the gains in hit-rate

are significant: for a skin length of $s = 0.15r_c$, this results in a hit rate of

$$\frac{\frac{4}{3}\pi r_c^3}{\frac{4}{3}\pi(r_c + s)^3} = \frac{r_c^3}{(1.15r_c)^3} = \frac{1}{1.15^3} \approx 0.66. \tag{1.27}$$

This almost 4× higher hit-rate carries a price, however. A considerable memory overhead is incurred, which often dominates the memory requirements of the entire program, as it is on the order of $\mathcal{O}(N \cdot C)$, where C is the number of molecules that fall in the cutoff-sphere, mentioned earlier. As MD is typically not memory intensive, this is not a problem for many applications.

Of course, the neighbour list needs to be constructed in the first place. Usually one resorts to a version of the Linked Cells algorithm for that purpose [90]. The list is then rebuilt every 5-20 timesteps, depending on the velocity of the particles and the skin radius s.

If the calculation of the force between two molecules is cheap (e.g. single-center Lennard-Jones or Coulomb interaction), then the very high hit-rate of Verlet Lists makes them more appropriate. If, however, the force calculation is more expensive (e.g. multicentered molecules or dipole/quadrupole interactions), or memory-efficiency is desired, then Linked Cells might be the better choice.

1.7 Further simulation packages for molecular dynamics

Gromacs (formerly GROningen MAchine for Chemical Simulations) is a molecular dynamics simulation package used for simulation of biochemical molecules such as proteins, lipids and nucleic acids among others[2]. Its development began around 1991 at the University of Groningen. More recently, the development of the code has shifted to the Royal Institute of Technology in Stockholm, Sweden and the Uppsala University, Sweden. It is an open-source project, available under the GNU Lesser General Public License (LGPL) from `gromacs.org`. GROMACS is written in C++ and features MPI parallelization via spatial decomposition and threading parallelization. It has multiple, processor-specific SIMD intrinsic kernels and GPU support [2,85], which make it one of the fastest MD simulation packages. It also features built-in ensemble parallelization, in which different replicas of the simulation can be run in parallel, providing a very loosely coupled parallelization on the level of simulations, which is well suited to massively parallel architectures. It is based on Verlet Lists and supports different methods for long-range calculations.

Lammps (Large-scale Atomic/Molecular Massively Parallel Simulator) is one of the most mature and widely used simulation packages for molecular dynamics[3]. The development of the code began around 1995 [90] at the Sandia National Labs facility of the US Department of Energy[4]. Since then the code has grown considerably, with multiple users from around the globe contributing to this open-source project. It is used for atomic, polymeric, biological, solid-state, granular, coarse-grained, or macroscopic systems and supports a variety of interatomic potentials, among which the Lennard-Jones, charge and dipole ones[5]. The code is available under the GNU General Public License from `https://github.com/lammps/lammps`. It is written in C++ and features an MPI parallelization based on spatial decomposition with both static or dynamic load-balancing. Multiple acceleration and optimization packages have been contributed, adding support for SIMD vectorization, OpenMP parallelization and accelerator architectures such as GPUs and Xeon Phi [14]. It is also based on Verlet Lists, which are constructed with the help of Linked Cells. Support for long-range interactions is also available through different methods.

[2] `http://www.gromacs.org/`
[3] `https://lammps.sandia.gov/`
[4] http://www.sandia.gov/
[5] `https://lammps.sandia.gov/doc/Manual.html`

Further simulation codes Many other high-performing and advanced molecular dynamics simulation packages exist, targeting one or more applications of MD. Among them are NAMD [54,89], AMBER [17,71], ESPResSo [7] and Desmond [12,101,102].

2

Node-Level Performance for MD

In this chapter, we review the foundations, necessary for understanding and leveraging node-level performance in general and more specifically in MD.

2.1 Target platforms

This work focuses primarily on Intel Xeon architectures from Intel SandyBridge[1] up to Intel Broadwell[2] and with an outlook for Intel Skylake[3]. Figure 2.1 shows a schematic of a supercomputer node from these hardware families as found in the LRZ SuperMUC[4] and HLRS Hazel Hen[5] supercomputers. Intel Xeon Phi architectures from Intel Knights Corner[6] up to Intel Knights Landing[7] are considered as well. For the purposes of the discussions in the following chapters, we give a brief overview of the architecture.

Core architecture Each core has several execution units, including two floating-point units (FPU), see Figure 2.1. Each FPU is capable of executing operations on mathematical vectors of a certain length. The vector length ranges from two for double precision floating-point operations in earlier models such as Westmere (128-bit width) to thirty-two for single precision floating-point operations on Knights Landing or Skylake Xeon (512-bit width). Every operation is superscalar, meaning that it consists of different substages executed by different subunits in a pipeline-like fashion. While one subunit executes one substage of one operation, an idle subunit can execute the respective substage of another operation.

Inside the core also an increasing number of SIMD vector registers are available for storing data. The number of registers ranges from eight in the earlier implementations to thirty-two in the latest Skylake and Knights Landing models. Larger storage space is available in the different level caches. The L1 cache is split into Instruction and Data partitions, typically about 32KB each. The next larger cache is the L2 cache of size about 256KB on Xeon architectures and 512KB on Xeon Phi architectures.

Each core supports two hyperthreads, which essentially share almost all physical resources such as execution units and memory. The gains due to hyperthreading are, hence, limited and consist mostly of improved utilisation of the individual resources. For example, while one thread is utilising one of the FPUs, the second one could utilise the other or perform load or store operations on other ports.

[1] https://ark.intel.com/products/codename/29900/Sandy-Bridge
[2] https://ark.intel.com/products/codename/38530/Broadwell
[3] https://ark.intel.com/products/codename/37572/Skylake
[4] https://www.lrz.de/services/compute/supermuc/systemdescription/
[5] https://www.hlrs.de/systems/cray-xc40-hazel-hen/
[6] https://ark.intel.com/products/codename/57721/Knights-Corner
[7] https://ark.intel.com/products/codename/48999/Knights-Landing

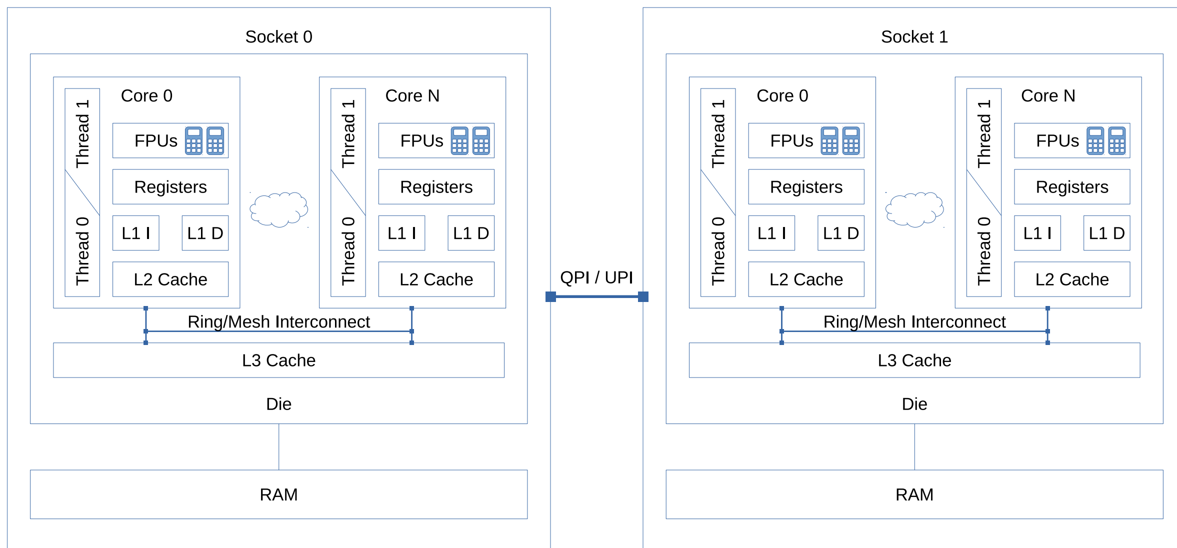


Figure 2.1: Schematic diagram of a supercomputer node on SuperMUC or Hazel Hen.

Die architecture In the considered hardware platforms multiple cores are bundled together on the same die. This is also where the L3 cache resides. Up to 72 cores can be on the same chip in e.g. Knights Landing. In the older models, the cores are connected to each other and to different regions of the L3 cache via a ring interconnect. As the core count began to grow considerably, however, the one-dimensional ring topology became inefficient. Hence, starting with Knights Landing and Skylake the ring topology was replaced by a two-dimensional mesh interconnect topology.

The L3 cache is on the order of tens of Megabytes and varies more greatly between models. The Xeon Phi architecture has no L3 cache; instead, its next level memory is a high bandwidth one consisting of up to 16 GB GDDR5 for Knights Corner and 16 GB MCDRAM for Knights Landing. In addition, the Knights Landing architecture also supports up to 384 GB DDR4 RAM.

Node architecture On the supercomputers we consider, nodes typically bundle two Xeon sockets together, with each having a separate RAM partition attached to it. This means that the memory of one socket is accessible from the other socket in a shared-memory fashion. Access to the other socket's memory incurs some penalty, however, which makes this a non-uniform memory access (NUMA) architecture. The two sockets are connected through an Intel QuickPath Interconnect (QPI) or an Intel UltraPath Interconnect (UPI) from Skylake onwards. Supercomputers equipped with Xeon Phi cards usually come in single-socket configurations.

2.2 HPC from a bottom-up perspective

In this work we take a bottom-up approach to HPC programming, in order to gain a better understanding of where performance is potentially lost. Figure 2.2 illustrates some of the different levels of concern, when performing an HPC optimization of a program. As can be seen, up to six different layers can be distinguished, each coming with its own considerations and relying upon the efficient execution of lower layers. Most of the time the lower-level optimizations and decisions do not depend on the higher-level ones, but this is not necessarily the case. For example, one algorithm can be superior to another in serial execution, but have a poorer par-

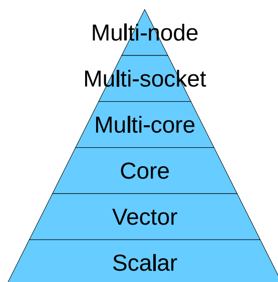


Figure 2.2: Different levels of concern in HPC optimization.

allel scalability, ultimately leading to lower overall performance. For instance the Gauss-Seidel method for solving linear systems of equations is inherently serial and needs to be modified in order to be parallelized at all. We now present the different levels in more detail.

2.2.1 Scalar-level performance

The first question is which algorithm should be used — what algorithms are available and how do they perform for the different target scenarios. The question of whether to use the Linked Cells method or the Verlet Lists method falls in this level. Another aspect, discussed in this work, is on the choice of the floating-point precision: does the application necessitate the usage of double precision or does single precision suffice? Is maybe a combination of the two precisions (referred to as mixed precision) applicable and, if so, is it beneficial?

The next question at this level would be whether the code is good for scalar and superscalar execution. One would investigate how the data is organized into data structures and how the data structures are accessed — are they traversed in a linear fashion or are there unpredictable "jumps" in the accessed memory addresses. Does the code contain a lot of small function calls or conditional statements, which lead to excessive branching and so on.

Clockticks-per-instruction metric An important metric produced by profiling tools for analyzing code at this level is the clockticks-per-instruction retired (CPI) metric. The theoretically best value of the CPI metric is 0.25, meaning that the processor retires four instructions in every CPU cycle. In practice, however, various reasons cause the value to be higher, such as waiting on data to be fetched from main memory, long-latency instructions such as division or square root or branch mispredictions. A value of CPI < 1 is typically considered good for HPC applications[20].

Roofline model An important model, which is often used to analyze codes at the scalar and vector levels, is the roofline model, illustrated in Figure 2.3. Depending on how many operations are performed per one byte of data (arithmetic intensity, AI) one distinguishes between compute-bound and memory-bound code. If the limiting factor of execution is the actual time, taken to perform (floating-point) calculations, the code is said to be compute-bound. For example, algorithms, which do a lot of computational operations on a small amount of data, e.g. $\mathcal{O}(N^2)$ or $\mathcal{O}(N^3)$ algorithms, are often compute-bound. In contrast, if a lot of time goes by waiting for data to be fetched from memory, only to perform a few operations on the data, (e.g. multiply an array by a scalar $a[i] := s * a[i]$), then it is said that the code is memory-bound. Algorithms which scale like $\mathcal{O}(N)$ or $\mathcal{O}(\log N)$, are often memory-bound. The questions at this level and

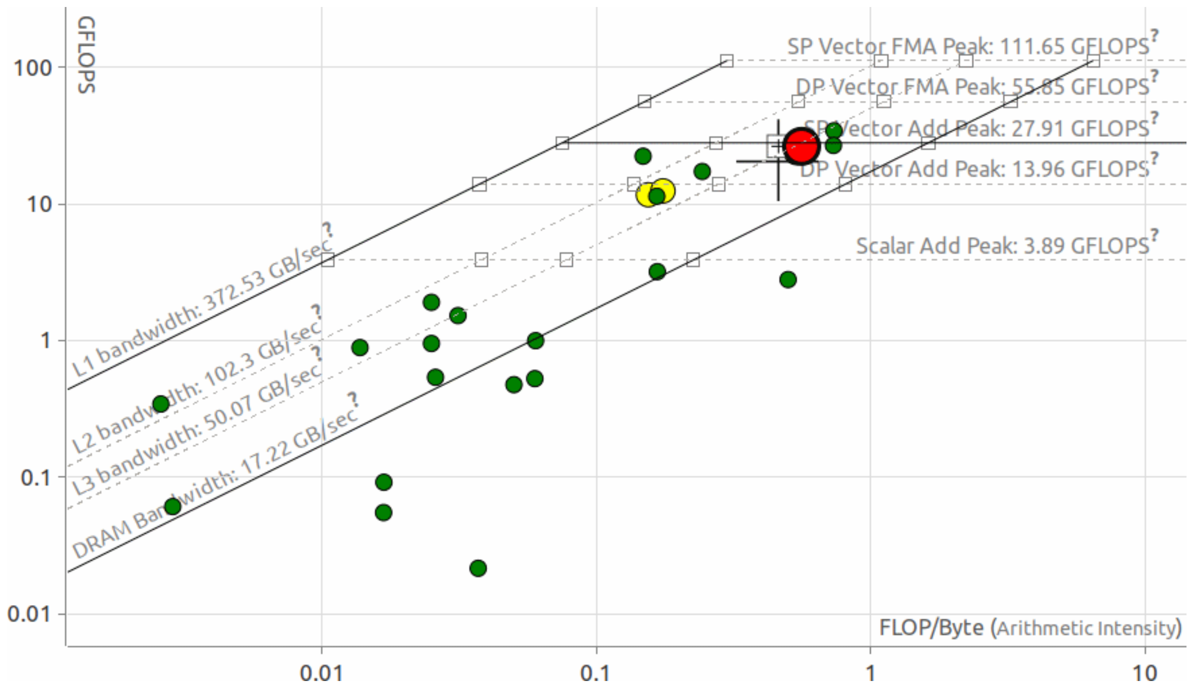


Figure 2.3: Roofline model for analyzing programs at the scalar and vector level. The attainable floating-point operations per second are plotted as a function of the arithmetic intensity. Screenshot produced from the Intel Advisor tool for `ls1 mardyn` running on a Intel Core i7-4770 CPU @ 3.4 GHz desktop Haswell machine. Circles represent the execution of different functions within the program. The colour of the circles represents how long each function is running, with green denoting low runtime, yellow denoting intermediate runtime and red denoting long runtime (red is the short-range force calculation in this case).

the respective optimizations they mandate are, clearly, highly application- and scenario-specific. In `ls1 mardyn`, the short-range force calculation is often compute-bound.

As can be seen from Figure 2.3, different “roofs” come into play, depending on the different possible limiting factors. If, for example, the data fits into the L3 cache, but not the L2 cache, then the bandwidth to the L3 cache is the one, which determines performance, and not the L2 bandwidth or the RAM one. In a similar fashion, different limitations on compute throughput may apply. Calculations in `double` precision proceed at a lower rate than calculations in single precision, due to the different vector length (e.g. four vs eight in AVX). The throughput is also lower, if the code performs only additions or multiplications, which cannot be fused into “fused multiply-add operations” (FMA).

2.2.2 Vector-level performance

When optimizing vector-level performance, we aim to improve the utilisation of the SIMD capabilities of the hardware. Since around 1997, Intel CPUs feature extensions to the instruction set, which allow the processing of short arrays of data in a vector fashion. The first extension was called MMX and featured mostly integer arithmetic on 64-bit long vectors consisting of 8-, 16- or 32-bit integers. Since then, the SIMD capabilities have been incrementally extended over the years via the instruction sets SSE, AVX, AVX2, KNC and AVX512 to also floating-point arithmetic and a total length of 512-bits. The potential gains are significant: vectorizing floating-point arithmetic could, theoretically, give a speed-up of a factor of 16, which is certainly

worth pursuing for any HPC application. Making efficient use of the SIMD capabilities, however, is not an easy task. Attempts to achieve autovectorization by the compiler usually produce limited gains and, hence, intervention by the programmer is necessary. Furthermore, an appropriate data layout is a prerequisite, which often requires a considerable effort in redesigning the code.

Data Layout: Array-of-Structures and Structure-of-Arrays We illustrate the different possibilities for data layout with an example from `ls1 mardyn`. As the code follows a traditional, object-oriented programming style, molecules are represented by a specific class `Molecule`, storing its various attributes, such as position, velocity, etc. The molecules are then stored in containers, be it an `std::vector` or a `std::list` or something else. This is known as an Array-of-Structures (AoS) layout, cf. Listing 2.1. A code using an AoS layout can be vectorized, though the gains are limited. It is possible to vectorize across the dimension $D = 3$ (as is done in [86]), but this is cumbersome and requires considerable effort, considering that the vectorization length can reach 16.

A more vectorization-friendly layout is the so-called Structure-of-Arrays (SoA) layout, illustrated in Listing 2.2. In that layout, the positions of the molecules are "transposed" so that they become contiguously stored in memory. All x-coordinates are stored one after the other, all y-coordinates, etc., which makes loading of data into vector registers much easier. Aligning the data to specific boundaries (e.g. 32 Byte-boundaries for AVX, 64 Byte-boundaries for KNC/AVX512) allows to use aligned loads, which can further increase throughput.

Depending on the use case, sometimes the AoS layout is more convenient (e.g. sending molecules via MPI), sometimes the SoA one (force-calculation). For this reason, both layouts are made use of in `ls1 mardyn` in different stages of the program, as will be detailed in Section 4.1.

Listing 2.1: Positions in AoS layout

```
1 class Molecule {
2   double _R[3];
3 };
4 vector<Molecule> _cell_mols;
```

Listing 2.2: Positions in SoA layout

```
1 class SoA {
2   vector<double> _R_x, _R_y, _R_z;
3 };
4 SoA _cell_mols;
```

Autovectorization The first possibility to leverage the vector capabilities of the CPU is to rely on autovectorization, i.e. to leave the generation of vector code to the compiler. Vectorization, however, can be interpreted as parallelization on a very fine-grained scale. Considering that parallelization on finer scales is generally more difficult than parallelization on coarse scales, it should not come as a surprise that the compiler often fails to autovectorize. The main hurdle is that the compiler needs to *prove* that introducing optimizations to the code, in the form of vector operations, will still generate correct code. Dependencies — or their lack — while known to the programmer, are often difficult to prove. Here is a list of some of the considerations faced by the compiler when considering a `for`-loop for vectorization:

- dependencies between iterations of the loop,
- hidden dependencies between different variables,
- contiguous storage of variables in memory,
- consecutive access of memory locations,
- complex control flow (conditional branching, function calls),

- sufficient amount of work to amortize the often more costly vector operations,
- non-associativity of floating-point addition.

For these reasons, intervention by the programmer is necessary in most cases, in order to utilise the SIMD capabilities to a sufficient degree.

Pragma-based annotation The approach to vectorization, requiring the minimal effort by the programmer, is to annotate the code with additional `#pragma` directives, telling the compiler that it may safely assume additional properties of the code in certain regions. The offered pragmas, however, differ depending on the compiler used and the compiler may still choose to ignore them. Another question is whether the solution provided by the compiler is as efficient as hand-written vectorized code.

An emerging approach to vectorization are OpenMP SIMD extensions, introduced with version 4.0 of the standard [106]. Although relatively recent, this approach looks quite promising for several reasons. First, it generalizes the pragmas supported by one or the other compiler and adds more. Second, OpenMP is an established and flexible community standard, which has proven its capabilities for parallel programming. It is, hence, familiar to the existing HPC programmers, which makes its adoption easier. Finally, it resolves differences between compilers and guarantees portability and support in the future. It is, thus, reasonable to expect that it will become a go-to solution in the future. The question about the efficiency of the solution found by the compiler remains, however.

Intrinsic instructions A more low-level approach to vectorization is using intrinsic instructions [19]. As with many low-level solutions, this approach has a higher potential, but it comes at the cost of more programming effort. This approach consists of additional function calls and data types added to the `C` and assembly languages. The user is responsible not only for the arithmetic operations, but for all manipulations of the SIMD vector registers, including loading and storing data from memory into registers, permutations within registers, masking, etc. Solutions are tailored to the specific instruction set and CPU architecture, respectively. As the hardware evolved and the different sets of instructions were introduced, oftentimes the code needed to be revised for each new instruction extension. Moreover, changes to the simulation software sometimes became difficult to propagate. For example, changing the floating-point precision from `float` to `double` in AVX involves changing the declaration of the variables from `__m256` to `__m256d` as well as the arithmetic operations from `_mm256_mul_ps` to `_mm256_mul_pd` and — most significantly — the vectorization width from eight to four. For these reasons, with time, many programmers wrote their own wrappers around the instruction sets. One such example is Agner Fog's vectorclass library[8]. These libraries, however, usually evolved around different needs and placed emphasis on one or the other functionality. Moreover, as with any library, the question remains of whether being sufficiently general enough for all users can be specific enough for the individual users, in order to provide the additional edge over other methods of vectorization.

Another issue arises when dealing with some special instructions. For example, the IEEE-compliant versions of the floating-point division or square-root operations are vectorized only for 128-bit width on some architectures, which support 256-bit AVX. Hence, maintaining SIMD efficiency might mandate considering special instructions such as `_mm256_rcp_ps` or `_mm256_rsqrt_ps`. While faster, these instructions deliver lower precision. If higher precision is desired, Newton-Raphson iterations need to be used [78].

[8]`https://www.agner.org/optimize/#vectorclass`

Listing 2.3: Short-range force calculation

```
1  // compute new forces
2  for (i = 0; i < N-1; ++i) {
3          f[i] = 0.0;
4  }
5  for (i = 0; i < N-1; ++i) {
6          for (j=0; j < N-1; ++j) {
7                  if (i != j and dist(x[i], x[j]) < r_c)
8                          f[i] += f_ij(Bodies[i], Bodies[j]);
9          }
10 }
```

Coming back to molecular dynamics, the handling of the cutoff condition in the short-range force calculation needs considerable attention. Listing 2.3 illustrates this issue. The question is what to do when some elements in the contiguous force storage need to be accessed, while others must not, due to the cutoff-condition being false. One approach is to load the data in contiguous chunks and `mask` the respective entries (multiply them by zero) so that they do not contribute. Another approach is to compute the indices of the entries, which need to be modified and to `gather` them in one vector register. The contributions are computed and, if *Newton3* is applied, the resulting force contributions get `scatter`-ed back. Further details are discussed in Chapter 5.

Vectorization efforts for `ls1 mardyn` began to be developed already in [26]. An intrinsics-based implementation was developed and maintained ever since. In this work it was extended and brought to maturity by writing our own set of intrinsics wrappers.

Other possibilities Further possibilities for code vectorization exist. One approach is Intel Cilk Plus[9] which adds array-like notation and mathematical operations on vectors and matrices or directly writing parts of the code in assembly language.

Limitations to vectorization speed-ups Unfortunately, even after considerable effort on the side of HPC programmers, the final gains due to vectorization may still be limited. In practice, it turns out that there are several stumbling blocks, which are not obvious a priori. We now discuss some of them.

First, if one considers vectorization of increasing vector lengths from one, to two, to four and so on, as the compute "roof" rises, its intersection with the bandwidth lines moves ever further to the right, cf. Figure 2.3. This means, that when increasing the vectorization length, the cross-over value of arithmetic intensity increases. As one can expect to attain the theoretical speed-up only in the compute-bound region, this means that the algorithm itself needs to be more and more computationally intense, in order to gain the full speed-up. Otherwise, a lower gain will be observed.

Second, some instructions for long vector-lengths have a higher power usage. Since Haswell, the CPU downscales the clock frequency by up to 17%, when it detects AVX2 instructions. This is done in order to avoid overheating and keep power usage within specifications. For an Intel Xeon Gold 6148 Skylake CPU, the frequencies are 3.1, 2.6 and 2.1 GHz when running, respectively, non-AVX2, AVX2 and AVX512 workloads, which represents more dramatic decreases. The power usage, however, remains constant, which means that while the speed-up is not as good, the calculation consumes less energy.

Finally, Amdahl's law also places limits on the gains, which is discussed in Section 2.4.1.

[9] `https://www.cilkplus.org/`

2.2.3 Core-level performance

In this work, under core-level performance, we understand the use of all threads that a core supports. This is known as hyperthreading on Intel architectures (official name Intel Hyper-Threading Technology[10]). Here we introduce the concept, while some further details are given in Chapter 6.

Hyperthreading is a technology by Intel, which exposes two logical cores to the operating system for each physical core on Xeon and four logical cores on Xeon Phi. This means that not only main memory is shared, as is typical for shared-memory contexts, but also the caches and execution units, including floating-point units. Only some hardware units are duplicated, necessary for storing the state of the program, such as registers.

Hyperthreading is especially important on the Intel Knights Corner architecture, because every thread can issue instructions only every second cycle. This means that if a program does not make use of this feature, it is upfront limited to 50% of the theoretical performance of the architecture. For this reason, we pay considerable attention to this hardware feature in this work.

The gains due to hyperthreading vary strongly with the chosen application. The expected benefits are due to a better utilisation of the resources shared by the hyperthreads. In this work, the observed benefits are two-fold. On the one hand, it helps to hide memory latency. For example, if a thread spends a lot of time waiting for data from RAM (or even waiting on input/output operations to the hard drive), then the second thread can utilise the floating-point units. The second benefit comes from mitigating problems with instruction-level parallelism. Particular kernels can exhibit poor properties for superscalar execution. The 12-6 Lennard-Jones kernel is one such example. Listing 2.4 shows a schematic of the problematic part.

Listing 2.4: 12-6 Lennard-Jones Kernel Calculation

```
1  // (r1_x, r1_y, r1_z) and (r2_x, r2_y, r2_z)
2  //  are the positions of the two particles
3  double c_dx = r1_x - r2_x;
4  double c_dy = r1_y - r2_y;
5  double c_dz = r1_z - r2_z;
6
7  double c_r2 = c_dx * c_dx + c_dy * c_dy + c_dz * c_dz;
8
9  double r2_inv = 1.0 / c_r2;
10
11 double lj2 = sig2 * r2_inv;
12 double lj4 = lj2 * lj2;
13 double lj6 = lj4 * lj2;
14 double lj12 = lj6 * lj6;
15 double lj12m6 = lj12 - lj6;
```

The repeated multiplications, necessary to compute $\left(\frac{\sigma}{r}\right)^{12}$ from σ^2 and $\left(\frac{1}{r}\right)^2$, all stress the multiplication unit, but not the addition unit. What makes matters worse, the result of each multiplication depends on the result of the fully completed previous multiplication. Thus, the subsequent operation cannot begin executing and pipelining cannot be utilised, leading to idling hardware. This is where hyperthreading can be of use again: the stream of instructions of the second hyperthread, being independent of the first stream, can provide additional instructions to fill the superscalar pipeline, leading to noticeable performance improvements.

[10] https://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html

The core-level of performance can be addressed with the same programming approach as the shared-memory parallelization for the multi-core level, provided that it can scale well to so many threads. This is because the operating system simply presents two/four times more cores to the program. There are some differences between the two levels, however. On the core-level, synchronization between hyperthreads is cheap and resources are shared, while on the multi-core-level synchronization is expensive and resources are (mostly) dedicated. These differences could prompt the programmer to develop a separate strategy for the hyperthreading-level alone, leading to a form of "nested" parallelism. As we shall see in later chapters, however, it turned out that this is not necessary in this work, as our "flat" approaches perform very well for very large numbers of threads.

2.2.4 Multi-core-level performance

In this work, under multi-core-level performance we understand shared-memory parallelization aiming at an efficient execution on cores residing on the same die. As this type of parallelism has been around for a long time, multiple mature and high-level solutions exist. Nevertheless, this level of code optimization is typically quite difficult.

First and most important, verifying program correctness is not easy. Race-conditions can occur if two or more threads attempt to read and write to the same memory location. What makes matters worse is that race-conditions may be difficult to detect and can even mean that the parallelization approach is wrong. Second, maintaining data locality can be an issue, since the programmer has no explicit control over RAM or the caches. False-sharing may occur, in which case two threads write to the same cache-line, resulting in a frequent invalidation of the line and a loss of performance. Next, and one of the most difficult aspects of shared-memory programming, achieving an appropriate granularity of the parallelized work, can be quite difficult. If the parallelization approach is too fine-grained, synchronization overhead may kill performance. On the other hand, if the approach is too coarse-grained, issues with load-balancing and idling threads may arise, again negating the gains of parallelization. Last, but not least, the use of system calls or external libraries must be kept in mind. For example, memory allocation and deallocation or random number generation may involve system calls, which do not scale as well as the own application and result in bottlenecks.

In this work we focus on solutions, which make use of the OpenMP standard [106]. We now briefly present two of the most common approaches for parallelization on shared-memory architectures: loop-based and task-based.

Loop-based parallelism The classic and most mature approach for parallelization on the level of threads is `for`-loop parallelization, in which the iteration space of the loop is distributed among threads. It offers excellent performance and features advanced load-balanced scheduling possibilities via `schedule(dynamic)` or `schedule(guided)`. Some of the best shared-memory parallelization schemes in this work make use of such a parallelization scheme.

Task-based parallelism Task-based parallelism is an alternative to loop-based parallelism. In task-based parallelism, the code spawns tasks and puts them in a queue. The threads then take tasks from the queue and process them one after the other until the work in the queue is done. An advantage of this parallelization model is that the tasks may be completely unrelated and spawned from various subfunctions of the program. This is hard to achieve with the more traditional loop-based parallelism.

OpenMP adopted tasks with the 3.0 standard [105] and introduced dependencies among them with the 4.0 standard [106]. Nevertheless, even with the introduced dependencies, the

OpenMP variant lacks support for some operations like reductions, which are necessary for the efficient parallelization of the short-range force calculation. For this reason, solutions via the Quicksched library [42] were also investigated.

2.2.5 Multi-socket-level performance

Under multi-socket-level performance we understand shared-memory parallelism spanning more than one socket. Programming for multi-socket environments (e.g. dual-socket systems on a cluster) introduces an additional level of concern, because each socket has its separate RAM partition. This leads to NUMA-accesses, since accessing the RAM of the two different partitions comes at different costs. If a single program is run on the entire node with threads across both sockets, this non-uniformity may need to be taken into account by the programmer. This can be done by ensuring that a thread always accesses the same memory locations and ensuring that those locations are placed "close" to the thread itself. Placing the data "close" is ensured by the operating system via the so-called "first-touch" policy: the data is physically allocated as "close" as possible to the thread touching it for the first time.

This can render schemes with load-balancing, which rely on features such as OpenMP's `schedule(dynamic)` less efficient for multi-socket execution, as the programmer has no control over which regions of the iteration space (and, respectively, the data) are accessed by which thread.

Apart from NUMA-awareness, the programmer must also keep in mind that synchronization between threads (for example via an `omp barrier`) becomes more expensive and aim for parallelization approaches with low synchronization costs.

2.2.6 Multi-node-level performance

Although beyond the scope of this work, we introduce some concepts on multi-node parallelization (distributed-memory parallelization), necessary for understanding the following chapters.

Across nodes, the method of choice in the area of HPC is the Message Passing Interface (MPI, [77]). It allows processes running on different nodes to communicate with each other by sending messages. A well established scheme, which is used also in this work, is the so-called Domain Decomposition approach. It is often chosen because it minimizes the length and count of messages to be sent, which, in turn, minimizes the overhead of the parallelization.

Figure 2.4 illustrates the scheme for the Linked Cells approach. The simulation domain is partitioned geometrically in regions with equal workload. Around the cells local to a processor, one layer of cells is introduced, usually called "halo" or "ghost" cells. If the length of a cell is smaller than the cutoff-radius, the halo-layer is thicker than one cell. In the halo layer, the positions of the molecules are copied from their owner processor and communicated to the neighbour just before the force calculation. Communication is also required after the position update, since molecules may leave the domain of one processor and enter the domain of another. Various extensions to the method exist [13], as well as other parallelization strategies [15,90], but fall beyond the scope of this work. The calculation of macroscopic quantities, such as temperature or pressure, requires also collective communication.

In `ls1 mardyn`, the basic strategy outlined here is implemented. It is referred to as "full shell," since all calculations in the halo layer are essentially computed twice. Implementing more advanced strategies, such as half shell, eighth shell or the neutral territory method [13,100] is work in progress, as well as more advanced load-balancing schemes [96,98,99].

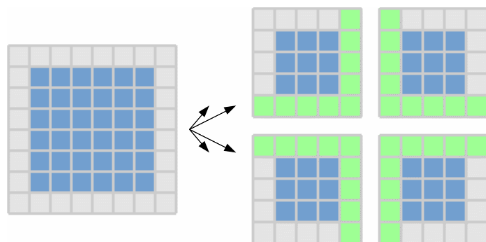


Figure 2.4: Illustration of the Domain Decomposition method. Left: execution on one processor, right: execution on four processors. Blue cells represent inner parts of the domain; grey cells represent areas where boundary conditions are applied. Green cells represent newly introduced parallel boundary cells, which store replicas of cells owned by another processor.

2.3 Performance characteristics of MD

The short-range force calculation is the primary focus of this work. It exhibits a complex behaviour, which varies considerably depending on four primary factors. One of the most significant factors is the cutoff radius r_c. If the cutoff radius is increased by a factor of two, the volume of the interaction sphere of each molecule increases by a factor of eight, which leads to an increase in the total calculations performed by the same factor. The dependence of the number of operations on the cutoff-radius is, thus, cubic. Another factor is the density of the system ρ. When increasing the density by a factor of two, the number of molecules in each cell doubles, meaning that around four times more calculations are performed. This gives a quadratic dependence. The next factor is the number of interaction sites of each molecule, also with a quadratic dependence. The last factor, which also significantly affects the number of operations is the type of interaction sites — Lennard-Jones, charge, dipole or quadrupole. Changing from one type to another gives a constant dependence, which can also be quite strong, however. Evaluating quadrupole-quadrupole interactions, for example, takes almost five times more operations than evaluating Lennard-Jones interactions.

Due to these factors, the short-range force calculation sometimes exhibits behaviour typical of compute-bound programs, while other times of memory-bound programs. For example, we can expect that single-center Lennard-Jones systems at a low density or cutoff-radius perform few operations per cell and are memory-bound. Because of that, they are less affected by compute-bound optimizations, such as SIMD vectorization, and are more affected by memory-bound optimizations, such as improvements to the memory-layout. On the other hand, systems with a high density, larger cutoff radius and multiple sites per molecule exhibit typical compute-bound behaviour and benefit more from SIMD vectorization. For these reasons, optimization work on the scalar- and vector-level is not easy and might even sometimes mandate trade-offs.

Regarding the remaining levels, on the core-level, gains due to hiding memory-latency and kernel deficiencies are expected. The core-, multi-core- and multi-socket-levels — as described here — are intended to be addressed via shared-memory parallelization. MD inherently contains a lot of parallelism, as many operations need to be performed on all molecules. However, the *Newton3* optimization complicates the parallel implementation of the force-calculation, which is the most time-intensive task. If the force-calculation cannot be parallelized efficiently, one cannot hope to achieve a good overall performance. Developing a high-performing OpenMP parallelization is, hence, one of the main goals of this work. As for multi-node performance, MD is known to scale very well up to thousands of nodes [27].

2.4 Performance metrics

2.4.1 Parallel speed-up and efficiency

Shared-memory parallelization and SIMD vectorization form a large part of the optimizations performed in this work. As both can be considered a form of parallelization, the easiest metrics to evaluate them are speed-up and parallel efficiency. Speed-up is defined via

$$S(n) = \frac{T(1)}{T(n)}, \tag{2.1}$$

where $T(n)$ is the time the program takes on n processing units and $S(n)$ is the speed-up for n processing units. Parallel efficiency $E(n)$ is defined as:

$$E(n) = \frac{S(n)}{n}, \tag{2.2}$$

and should, ideally, be as close to one as possible. Before presenting further metrics, we briefly explore a theoretical limitation to the attainable parallel speed-ups: Amdahl's law.

Amdahl's law and its implications for SIMD vectorization Amdahl's law gives an upper bound on the speed-up that can be obtained, depending on how much of the total program can be parallelized. This is because oftentimes one cannot parallelize the program in its entirety due to the presence of inherently sequential parts in the code.

Let the fraction of the time on one processor $T(1)$, which can be parallelized be p. The fraction, which cannot be parallelized is then $1 - p$. For the times $T(1)$ and $T(n)$ we, then, have:

$$T(1) = (1 - p)T(1) + pT(1), \tag{2.3}$$

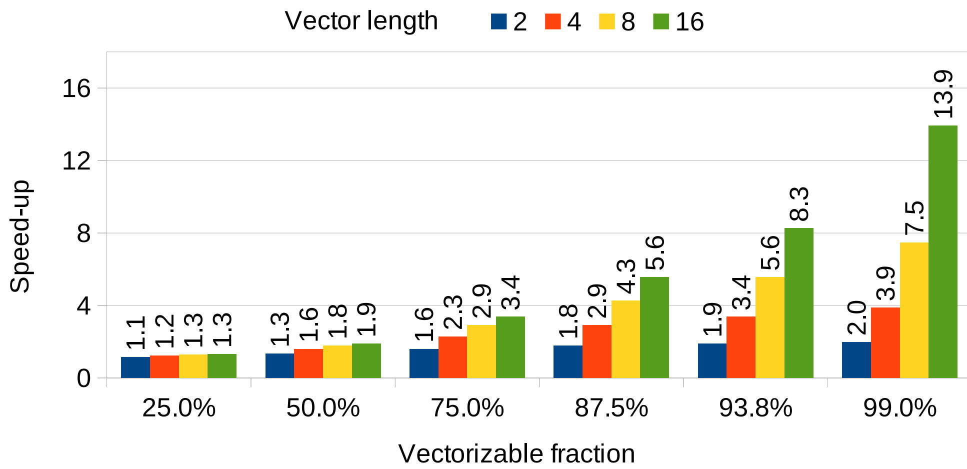
$$T(n) = (1 - p)T(1) + \frac{pT(1)}{n}, \tag{2.4}$$

which gives the Amdahl limitation on the attainable total speed-up $S_A(n)$:

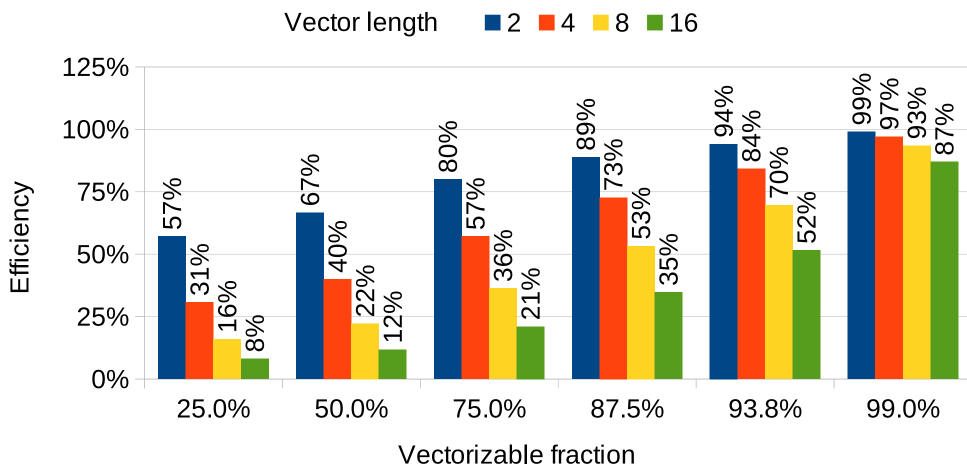
$$S_A(n) = \frac{1 - p + p}{1 - p + \frac{p}{n}} = \frac{1}{1 - p + \frac{p}{n}}. \tag{2.5}$$

While this is a well-known limitation in parallel programming, we point out that it has implications for SIMD vectorization as well. On the vectorization level, the "parallelizable fraction" p translates to "vectorizable fraction". Clearly, not all parts of a program can be vectorized, hence, such a fraction always exists and this law is applicable here as well. Moreover, not all parts of the code, which can be vectorized, will actually be sped up. As noted earlier, memory-bound portions of the code do not profit much from vectorization, since performance is limited by memory bandwidth or latency. Hence, the question becomes how much of the program is vectorizable *and* compute-bound.

In Figure 2.5 we plot some values for the Amdahl's law limitations on speed-up and efficiency for different vector lengths and different vectorizable fractions. As can be seen from the figure, when increasing the vector length for the same program, efficiency drops considerably. Moreover, achieving very high speed-ups from long vector lengths mandates very high fractions of the program to be compute-bound and vectorizable.



(a)



(b)

Figure 2.5: Limitations due to Amdahl's law for SIMD vectorization or parallelization. (a) Speed-up. (b) Parallel efficiency.

Need for further metrics While speed-up and parallel efficiency are quite useful, they are, however, relative to the performance of the program itself and do not contain information about the absolute performance of a program. Thus, they cannot be used to compare performance to alternative programs. For example, `ls1 mardyn` uses the Linked Cells method, while LAMMPS uses the Verlet Lists method, which may run at different speeds, depending on the molecular system being simulated. For this reason, we consider further metrics as well in the following sections.

2.4.2 Floating-point operations per second

Another possibility to evaluate performance of a program is to consider the floating-point operations throughput of the program for compute-bound applications or Bytes-throughput for memory-bound ones. The respective metrics are then simply the number of floating-point operations performed per second (FLOP/sec) or Bytes, which have been streamed through the CPU per second (Byte/sec). Another measure, which is convenient in some cases, is FLOP/sec per computing unit:

$$\frac{FLOP/sec(n)}{n}, \tag{2.6}$$

where n is the number of computing units and $FLOP/sec(n)$ is the number of FLOP/sec on n computing units. This value should, ideally, remain constant when increasing n.

The additional advantage of this metric is that FLOP/sec and Byte/sec can be compared directly to the theoretical throughput values of the architecture, giving an estimate of the utilisation of the hardware. As they are also absolute values, they can be compared across different simulation codes, provided that they use comparable underlying algorithms.

This metric has some disadvantages, however. The first one is that floating-point operations "were not all created equal". For example, on the investigated hardware an addition operation requires 3 CPU cycles, while a multiplication operation requires 5 cycles. Other operations like division or square root can take up to 30-40 cycles or more [19] and may depend on the floating-point precision. Thus, if an algorithm contains divisions (such as the Lennard-Jones kernel), square roots (the Coulomb kernel) or trigonometric operations, a simple count of the FLOP operations would be unfair versus an algorithm which contains only additions and multiplications.

A second problem may arise, when comparing FLOP/sec or Byte/sec throughput of different programs solving the same problem, but via different algorithms. For example, comparing MD codes, in which one works via the Linked Cells method, while the other makes use of the Verlet Lists method. Although the programs may exhibit similar FLOP/sec performance, the Linked-cell method may perform much more FLOPs for cutoff-checks, due to the lower hit-rate. For this reason, FLOP/sec and Byte/sec are good for measuring hardware utilisation, but need to be used with care.

In this work, we use this metric to evaluate different versions of `ls1 mardyn`, which perform the FLOP counts in the same way. Except where noted, the number of FLOPs presented for `ls1 mardyn` is counted in software, i.e. by writing our own functions to estimate the counts. These functions count only FLOPs performed in the short-range force calculation and count all FLOPs conservatively, i.e. with the same weight of one, including divisions and square-roots. In some chapters we present FLOP estimates provided by Intel profiling tools such as Amplifier and Advisor. These tools count all operations and account for divisions and square-roots in a different fashion and, hence, deviate from the own FLOP counts.

2.4.3 Application-specific metrics

We now discuss some application-specific performance measures, which can be used to compare also different MD programs.

Updates per second The first possibility would be to consider the number of updates (iterations) per second:

$$\text{UP/sec} = \frac{No.iterations}{Time[sec]}. \tag{2.7}$$

This metric can be used to compare different programs solving the same physical problem. It has the drawback of being heavily dependent on particular system settings, however, and especially the number of molecules in the system.

Molecule updates per second Another metric, which we make heavy use of here, is molecule-updates per second [11]:

$$\text{MUP/sec} = \frac{No.iterations \cdot No.molecules}{Time[sec]}. \tag{2.8}$$

Unlike the previous metric, this one is now also weighted by system size and, thus, can be used to determine how efficiently systems of different numbers of molecules are being solved. Knowing this is important to determine a good choice for how many processing units to use, for example. Due to the large numbers involved, this metric is often given in millions (MMUP/sec).

[11] Analogously to the MLUPS metric in Lattice-Boltzmann simulations, which measures million lattice updates per second.

3
Fast Multipole Method

3.1 General description

As mentioned in Section 1.4, some potential functions, such as the Coulomb potential, do not decay fast enough to be truncated without a significant loss of accuracy. This mandates that all particles in the system interact with each other in some fashion. Figure 3.1 illustrates four different possibilities for carrying out the calculations. For the sake of clarity, in this section we distinguish sources (electric field generating points) and targets (points at which the electric field is evaluated). In MD, however, all sources are also targets and vice versa.

Direct calculation Figure 3.1(a) illustrates that the direct calculation between M point-sources (dark blue) and N point-targets (dark red) leads to an $\mathcal{O}(N \cdot M)$ complexity. While easy to implement, this is a severe limitation on the number of particles which can be simulated, as the overall complexity of the program is $\mathcal{O}(N^2)$.

Barnes-Hut algorithm The first major advancement in the area of N-body simulations was the invention of the Barnes-Hut algorithm [9]. It introduces an octree-based clustering of the source particles, see Figure 3.1(b). Once a hierarchical tree is constructed, the tree is traversed for each particle, in order to compute the potential. This leads to a $\mathcal{O}(\log N)$ complexity per particle, and a total complexity of $\mathcal{O}(N \log N)$ of the algorithm.

Analogue to Barnes-Hut algorithm Figure 3.1(c) shows a theoretically possible alternative of the Barnes-Hut method, in which the clustering is done not for the source, but for the target particles. Although rarely, if ever, used in practice, it is also an $\mathcal{O}(N \log N)$ possibility.

Fast Multipole Method Figure 3.1(d) illustrates the basic idea of the Fast Multipole Method as introduced in [48]. The clustering is done both for the source and for the target particles, again in a hierarchical fashion. This allows the reduction of the per-particle tree traversals of the Barnes-Hut method down to a single one. Thus, the complexity is reduced down to $\mathcal{O}(N)$.

3.2 Types of clusters

Point-sources are grouped into source pseudoparticles, traditionally known as "multipole" particles, see Figure 3.2(a). The purpose of a multipole particle is to compute the aggregated influence of several sources, to be evaluated on far-away targets. Multiple sources can be grouped together, as well as multiple multipole particles. The influence of the entire multipole particle on point-targets and local pseudoparticles can be computed, as long as they are within

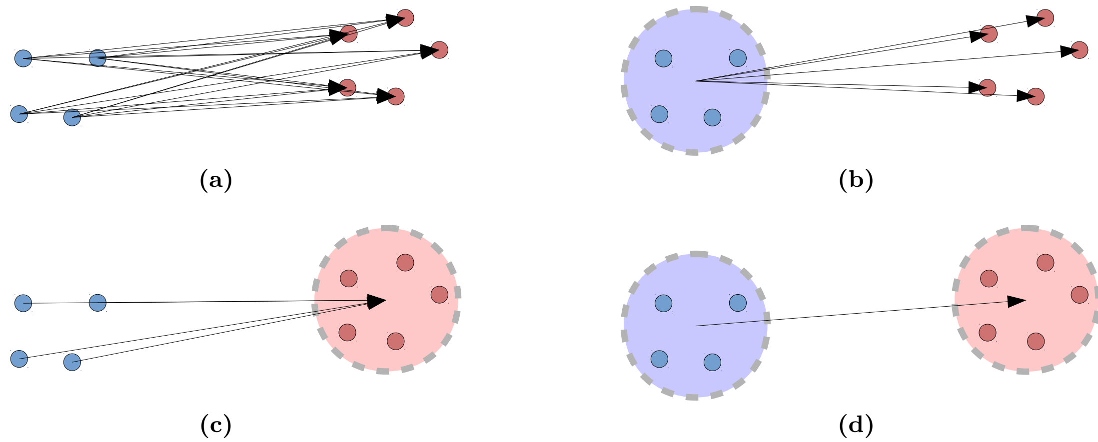


Figure 3.1: Interaction between source particles (blue) and target particles (red) via different algorithms. (a) Direct calculation: all sources act on all targets. (b) Barnes-Hut algorithm: sources are grouped into pseudoparticles. Each pseudoparticle acts on certain targets. (c) Barnes-Hut analogue: targets are grouped into pseudoparticles. Each source acts on certain pseudoparticles. (d) FMM: both sources and targets are grouped into pseudoparticles. Certain pseudoparticles act on certain pseudoparticles.

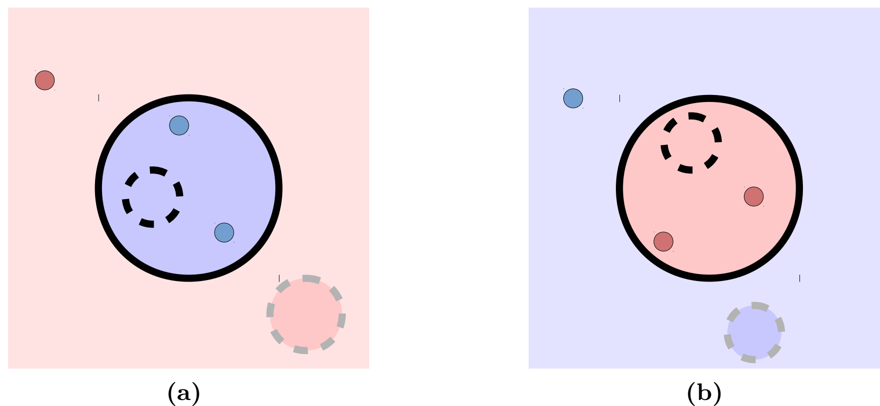


Figure 3.2: Illustration of multipole pseudoparticles (a) and local pseudoparticles (b). Small circles represent point-sources (blue) and point-targets (red). Larger circles with solid or dashed boundaries represent multipole pseudoparticles (blue) and local pseudoparticles (red). See text for further description.

the region of convergence of the expansion. The region of convergence is the exterior of a sphere with center the expansion origin and radius the distance to the furthest included pole. Mathematically, the expansion is a series expansion in terms of the distance from the center r, in negative powers:

$$\frac{a_0}{r} + \frac{a_1}{r^2} + \frac{a_2}{r^3} + \dots, \tag{3.1}$$

where $\{a_i\}_{i=0}^{\infty}$ are the coefficients to be computed from aggregating the influence of the sources. The expansion does not converge on the boundary of the sphere. It becomes ever more accurate with increasing distance from the center r. It is similar to the principal part of a Laurent expansion.

Point-targets are grouped into target pseudoparticles, traditionally known as "local" particles, see Figure 3.2(b). The purpose of a local particle is to create a local representation of far-away sources, which is valid within a certain region. The influence of the included sources can then be computed rapidly on any target particle or any other local particle, fully enclosed in the region of convergence. The region of convergence is the interior of a sphere with center the expansion origin and radius the distance to the closest included source. Mathematically, the expansion is a series expansion in terms of the distance from the center r, in positive powers:

$$b_0 + b_1 r + b_2 r^2 + \dots, \tag{3.2}$$

where $\{b_i\}_{i=0}^{\infty}$ are the coefficients to be computed from aggregating the influence of the sources. The expansion becomes ever more accurate with decreasing r, similar to a classic Taylor expansion.

Both the multipole and local expansions have, theoretically, an infinite number of terms. In practice, however, they need to be truncated after some term p. Increasing p increases the precision of the FMM approximation, at the cost of a higher computational load. The necessary truncation order p is dictated by the precision requirements of the application. The complexity in p is very high: some variants, which are used in practice, use $\mathcal{O}(p^6)$ implementations [121], while the lowest complexity is $\mathcal{O}(p^2 \log p)$ [30]. For this reason, the user wants to keep p as low as allowed by the application and consider mathematical optimizations to reduce the complexity. The unoptimized implementation presented here is $\mathcal{O}(p^4)$, which will be improved to $\mathcal{O}(p^2 \log p)$ via Fast Fourier Transforms in Chapter 11.

3.3 Mathematical formulation

In this section we present some of the mathematical formulae upon which our FMM implementation is based. Multiple possibilities exist for the precise form of the coefficients in Equations (3.1) and (3.2). These include solid harmonics [93], spherical harmonics [48], Cartesian Taylor expansions [111] or based on the method of equivalent charges [5]. They differ in the storage, complexity and available mathematical optimizations. A comparison can be found in [121] and will also be discussed in Chapter 11.

The FMM implementation, which was implemented as part of this work, uses solid harmonics expansions. In presenting the expansions we follow the derivation of [93], which, in turn, followed [87]. This derivation was selected because it implements solid harmonics, without having to resort to the computationally expensive `sin` and `cos` functions, and requires only a modest use of the `sqrt` function, which reduces runtime considerably [87,113].

3.3.1 Expanding the $\frac{1}{d}$ function

The multipole expansion for the $\frac{1}{d}$ function, where $d = |r - r'|$, $\vec{r} = (r, \theta, \phi)$ and $\vec{r'} = (r', \theta', \phi')$ with $r > r'$ can be written as:

$$\frac{1}{|\vec{r} - \vec{r'}|} = \sum_{l \geq 0} \sum_{m=-l}^{l} \frac{r'^l}{r^{l+1}} \frac{(l-m)!}{(l+m)!} P_l^m(\cos\theta) P_l^m(\cos\theta') e^{im(\phi - \phi')}. \tag{3.3}$$

$P_l^m(u)$ are the Associated Legendre Polynomials, [3]:

$$P_l^m(u) = \frac{(-1)^m}{2^l l!}(1-u^2)^{m/2} \frac{d^{l+m}}{du^{l+m}}(u^2-1)^l, \tag{3.4}$$

where $l \in \mathbb{N}_0, m \in \mathbb{Z}, u \in \mathbb{R}, u \in [-1, 1]$, and $P_l^m(u) \in \mathbb{R}$.

3.3.2 Defining the M and L functions

Proceeding, the summand of Equation (3.3) is split into the product:

$$\left(\frac{(l-m)!}{r^{l+1}} P_l^m(\cos\theta) e^{im\phi}\right) \left(\frac{r'^l}{(l+m)!} P_l^m(\cos\theta') e^{im\phi'}\right).$$

Note now that the first term depends only on the first position $\vec{r} = (r, \theta, \phi)$, while the second term depends only on the second one $\vec{r'} = (r', \theta', \phi')$. The two position variables have thus been "separated". Kernels, which can be separated in this fashion, are called "separable" or "degenerate" [65].

With the following definitions

$$\mathbf{M}_{l,m}(\vec{r}) = \frac{(l-m)!}{r^{l+1}} P_l^m(\cos\theta) e^{im\phi}, \tag{3.5}$$

$$\mathbf{L}_{l,m}(\vec{r}) = \frac{r^l}{(l+m)!} P_l^m(\cos\theta) e^{-im\phi}, \tag{3.6}$$

Equation (3.3) then becomes

$$\frac{1}{|\vec{r} - \vec{r'}|} = \sum_{l \geq 0} \sum_{m=-l}^{l} \mathbf{M}_{l,m}(\vec{r}) \mathbf{L}_{l,m}(\vec{r'}). \tag{3.7}$$

This is the basic identity, which defines both the multipole and local expansions. To form a multipole expansion, the $\mathbf{L}_{l,m}(\vec{r'})$ coefficients are evaluated at the positions of the sources, multiplied by the respective charges and summed up. The potential at a point r can then be computed by evaluating the $\mathbf{M}_{l,m}(\vec{r})$ coefficients and, at the end, multiplying by the charge at that point. Conversely, to form a local expansion, the $\mathbf{M}_{l,m}(\vec{r})$ coefficients are evaluated at the positions of the sources, multiplied by the respective charges and summed up. The potential at a point r' can then be computed by evaluating the $\mathbf{L}_{l,m}(\vec{r'})$ coefficients and, at the end, multiplying by the charge at that point.

For the operations between pseudoparticles, three more identities are necessary. We will discuss here how to "shift" the origin of multipole and local expansions. How this comes into play with aggregating pseudoparticles within FMM will be clarified in Section 3.7.

In order to shift the center of a multipole expansion, the following relation is needed:

$$\mathbf{L}_{l,m}(\vec{r'} - \vec{d}) = \sum_{l'm'} \mathbf{L}_{l',m'}(r') \mathbf{L}_{l-l',m-m'}(-\vec{d}), \tag{3.8}$$

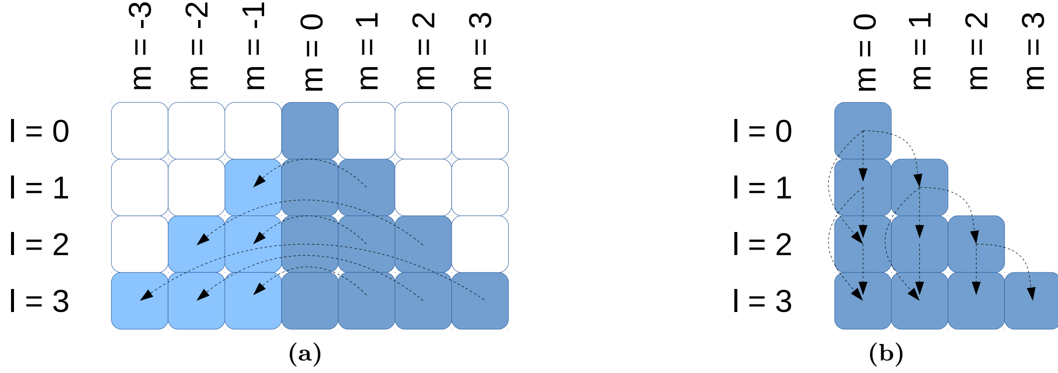


Figure 3.3: (a) Schematic representation of storage for the coefficients of the **M** and **L** expansions in dependence of $l = 0, 1, \dots$ and $m = -l, \dots, l$ up to truncation order $p = 3$. Entries in white squares are outside the scope of the summations, but could formally be defined to be zero. Light blue entries formally participate in the sums, but can always be computed from the respective dark-blue entries by symmetry relations. Explicitly stored are only the dark blue squares. (b) Schematic representation of the dependencies between elements in the recursive evaluation of $\mathbf{M}(\vec{r})$ and $\mathbf{L}(\vec{r})$ up to truncation order $p = 3$.

where $\vec{d}$ is the shift vector of the translation. In order to shift a multipole expansion to a local one about a different origin, the following relation is used:

$$\mathbf{M}_{l,m}(\vec{r'} - \vec{d}) = \sum_{l'm'} (-1)^{l'} \mathbf{L}_{l',m'}(r') \mathbf{M}_{l+l',m+m'}(-\vec{d}), \tag{3.9}$$

where, again, $\vec{d}$ is the vector of the translation, connecting the two origins. The last shift necessary is the shift of the center of a local expansion. This is done via the following relation:

$$\mathbf{M}_{l,m}(\vec{r'} - \vec{d}) = \sum_{l'm'} \mathbf{M}_{l+l',m+m'}(\vec{r'}) \mathbf{L}_{l',m'}(\vec{d}). \tag{3.10}$$

The shifting operations of Equations (3.8) to (3.10) are forms of convolutions between **M** and **L** expansions. They have been discussed in greater detail in [39] and will also be discussed in Chapter 11.

All of these functions give the calculation of the potential function. Since the force is defined via the gradient of the potential, $\vec{F} = -\nabla U$, it is necessary to compute $\nabla \mathbf{L}$ (and also $\nabla \mathbf{M}$ in some adaptive extensions). The formulas for that can be found in [93] and [87].

3.3.3 Some notes on the implementation and storage

Figure 3.3(a) visualizes the storage pattern for the **M** and **L** coefficients up to truncation order $p = 3$. The range of the m index from $-l$ to l mandates a specific triangular pattern. The entries $m > |l|$ are identically zero and do not need to be allocated. Due to the following symmetry relations

$$\mathbf{M}_{l,-m}(\vec{r}) = (-1)^m \mathbf{M}^*_{l,m}(\vec{r}), \tag{3.11}$$

$$\mathbf{L}_{l,-m}(\vec{r}) = (-1)^m \mathbf{L}^*_{l,m}(\vec{r}), \tag{3.12}$$

(where * denotes the complex conjugate), one typically only stores the $m \geq 0$ entries. The $m < 0$ entries also participate in the different shifts and summations, but are computed from the $-m$ entries on the fly.

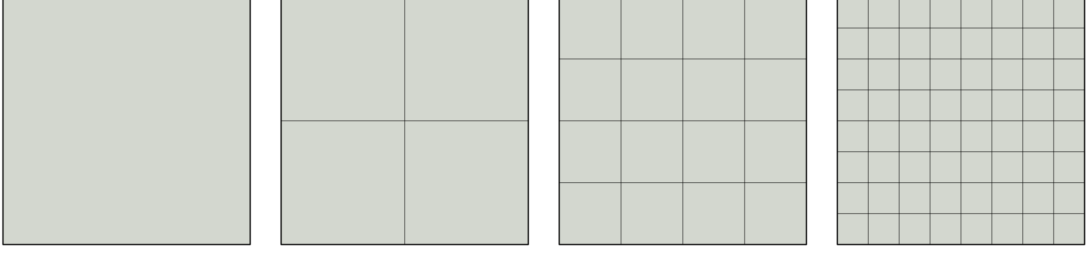


Figure 3.4: Quadtree structure for a two-dimensional square domain in FMM.

For implementational reasons, **M** and **L** are split into real and imaginary parts:

$$\mathbf{M}_{l,m}(\vec{r}) = \mathbf{M}^c_{l,m}(\vec{r}) + i\mathbf{M}^s_{l,m}(\vec{r}), \tag{3.13}$$
$$\mathbf{L}_{l,m}(\vec{r}) = \mathbf{L}^c_{l,m}(\vec{r}) - i\mathbf{L}^s_{l,m}(\vec{r}), \tag{3.14}$$

where ^{c} stands for "cosine-like" and ^{s} stands for "sine-like". Taking into account the usual conversion between Cartesian coordinates (x, y, z) and spherical coordinates (r, θ, ϕ) coordinates

$$x = r \sin\theta \cos\phi, \tag{3.15}$$
$$y = r \sin\theta \sin\phi, \tag{3.16}$$
$$z = r \cos\theta, \tag{3.17}$$

the following recursion relations hold [93]:

$$\mathbf{M}^{c,s}_{l,m} = \frac{1}{r^2}\left((2l-1)z\mathbf{M}^{c,s}_{l-1,m} - (l-1+m)(l-1-m)\mathbf{M}^{c,s}_{l-2,m}\right), \qquad m < l \tag{3.18}$$
$$\mathbf{L}^{c,s}_{l,m} = \frac{1}{(l+m)(l-m)}\left((2l-1)z\mathbf{L}^{c,s}_{l-1,m} - r^2\mathbf{L}^{c,s}_{l-2,m}\right), \qquad m < l \tag{3.19}$$
$$\mathbf{M}^c_{m,m} = -\frac{2m-1}{r^2}\left(x\mathbf{M}^c_{m-1,m-1} - y\mathbf{M}^s_{m-1,m-1}\right), \qquad m = l \tag{3.20}$$
$$\mathbf{M}^s_{m,m} = -\frac{2m-1}{r^2}\left(y\mathbf{M}^c_{m-1,m-1} + x\mathbf{M}^s_{m-1,m-1}\right), \qquad m = l \tag{3.21}$$
$$\mathbf{L}^c_{m,m} = -\frac{1}{2m}\left(x\mathbf{L}^c_{m-1,m-1} - y\mathbf{L}^s_{m-1,m-1}\right), \qquad m = l \tag{3.22}$$
$$\mathbf{L}^s_{m,m} = -\frac{1}{2m}\left(y\mathbf{L}^c_{m-1,m-1} + x\mathbf{L}^s_{m-1,m-1}\right), \qquad m = l \tag{3.23}$$

with the starting conditions

$$\mathbf{M}^c_{0,0} = \frac{1}{r},\ \mathbf{M}^s_{0,0} = 0,\ \mathbf{L}^c_{0,0} = 1,\ \mathbf{L}^s_{0,0} = 0. \tag{3.24}$$

Figure 3.3(b) illustrates the recursion dependencies between the elements.

3.4 Tree structure

Similarly to the Barnes-Hut algorithm, FMM allocates a hierarchy of grids for the computational domain in an octree-fashion (quadtree in two dimensions). A multipole and a local expansion are then allocated for every cell, which represent all particles contained in the geometric region spanned by the cell. Figure 3.4 illustrates the construction of a quadtree for a square simulation domain. One means of constructing the tree is to place the whole simulation box at the root

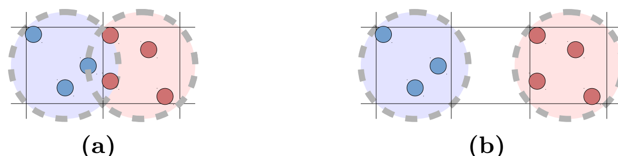


Figure 3.5: The two cells in (a) cannot interact, because their bounding spheres intersect. The two cells in (b) can interact, because their bounding spheres do not intersect.

of the tree and to subdivide each cell recursively, until only “a few” particles reside in each leaf cell. The number of particles per cell and, correspondingly, the depth of the tree, need to be chosen appropriately for the scenario at hand, as discussed further in Section 3.8. Another means of constructing the tree is to do so in a bottom-up fashion, by estimating the number of particles per cell on the finest level and allocating coarser grids until the root of the tree is reached.

In this work, the discussion is mostly restricted to uniform trees for homogeneously distributed particles with 2^k cells per dimension. Extensions to arbitrary numbers of cells per dimension can be done via the method of [118], while extensions to increase adaptivity are briefly discussed in Section 3.9.

3.5 Separation criteria

The feature which distinguishes FMM from the Barnes-Hut method is the ability to compute the influence of a multipole cluster on a local cluster. This is referred to as a Multipole-To-Local (M2L) operation. Performing this operation requires that the regions of convergence of both clusters are taken into account, which results in a need for a certain separation between them. Thus, a “separation criterion” for when two clusters can interact in this way is needed (referred to as “multipole acceptance criterion” in some works [121]).

As mentioned in Section 3.2, the radius of convergence of a multipole expansion is defined by the distance to the furthest included point-source. Consider Figure 3.5(a). When constructing a multipole expansion for a square/cubic cell, a particle may, in general, be located in the corner of a cell. As usually the center of the expansion of a cell is placed at the center of the cell, this implies that the region of convergence is the exterior of the bounding sphere of the cell. Similarly, when constructing a local expansion for a cell, a particle may be located in a corner of the cell. If the expansion is made about the center of the cubic cell, then the minimal region of convergence, which guarantees convergence for the entire interior of the cell, is the interior of the bounding sphere of the cell. Thus, if two cells are direct neighbours, they cannot be interacted via pseudoparticle interactions, because their bounding spheres intersect, meaning that the expansion series will not converge in the intersection. If they touch in a corner, they still cannot be interacted, because in that corner, being on the boundary of both bounding spheres, the expansions again do not converge. For this reason, there needs to be some separation distance between two cells, if they are to be interacted via pseudoparticle interactions. If the distance is sufficiently large, the cells are said to be “well-separated”.

Cell-based separation In this work, we say that if two cells are separated by at least one cell, then they are well-separated. Thus, for a regular grid of cells, all cells, except for the $3^D - 1$ cells, which touch it, are well-separated. The $3^D - 1$ cells, which touch one cell, are referred to as its “nearest neighbours”. In the current work and code implementation, a separation

of at least one cell is used and only cells of the same size are considered. It is possible to increase the separation requirement by requiring that the interacting clusters are at least two cells apart or even more [70]. The discussion presented here extends to that case in a relatively straight-forward fashion.

Separation for arbitrary cell positions and sizes Alternative definitions for well-separatedness also exist, which also take into account the size of the cells, in the context of the Dual Tree Traversal-based FMM [121]. One example is the following criterion:

$$\frac{B_i + B_j}{r_{ij}} < \theta. \tag{3.25}$$

In Equation (3.25) B_i and B_j are (measures of) the radii of the bounding spheres of the two cells and r_{ij} is the distance between the centers of the expansions of the two cells. The parameter θ is a value specified at runtime and can be used to control the accuracy of the calculation.

The criterion of Equation (3.25) is more flexible than the cell-based one and allows a tighter control of the FMM accuracy. However, it needs to determine which cells are well-separated during the calculation, while the cell-based criterion can be hardcoded, leading to less conditional statements and fewer arithmetic operations.

3.6 Tree traversals

The next building block, necessary for the construction of an FMM algorithm, is a means of traversing all pairs of cells and selecting which ones should be interacted via M2L interactions. This is the task of the tree-traversal algorithm. Its goal is two-fold:

- on every level, a cell should interact with as many well-separated cells as possible, so that less work remains for the finer levels,
- when "summing up" across levels, no region of the domain should be contributed twice.

The two different separation criteria are used with two different tree-traversal algorithms.

List-Based Traversal The cell-based separation criterion is often combined with the List-Based Traversal (LBT) and applied to uniform trees. A set of cells, which fulfills this criterion is called a cell's "interaction list". Since a cell cannot interact with its nearest neighbours, these regions need to be interacted on the next finer level. This results in the following definition:

> **Interaction List** *The interaction list of a cell is the set of children of its parent's nearest neighbours, which are well-separated from it.*

Figure 3.6 illustrates the interaction lists of the cells in red. In this fashion cells which are not close to the boundary have $6^D - 3^D$ cells in their interaction list. This gives 3, 27 and 189 cells in one, two and three dimensions, respectively. In later chapters we will sometimes refer to this particular interaction pattern as the "M2L stencil". Figures 3.6(a) and 3.6(b) shows how the interaction lists of a cell and its parent fit together to cover ever larger portions of the simulation domain. Finally, on the finest level of the grid the interactions between a cell and its nearest neighbours are computed via a direct $\mathcal{O}(N^2)$ calculation. Interactions, computed directly, are referred to as Particle-to-Particle (P2P) operations.

An advantage of this tree traversal is that the interaction lists are known apriori and can be hardcoded, resulting in little overhead. The method could also be extended from uniform to adaptive trees, provided that cells are uniformly subdivided. The LBT algorithm cannot work with arbitrary positions or sizes of the cells, however.

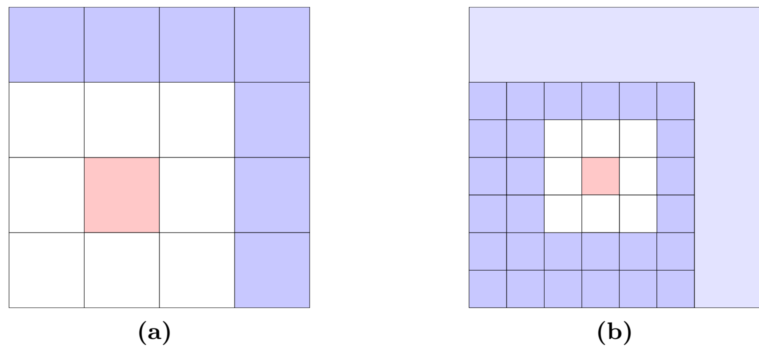


Figure 3.6: (a) Cells in blue are the interaction list of the cell in red. The interaction lists on levels of the tree, coarser than the illustrated one, are empty. (b) Cells in dark blue are the interaction list of the cell in red. Region in light blue has been accounted for on the coarser level by the parent cell of the red cell.

Dual Tree Traversal In this work we use LBT. However, in later chapters we draw comparisons to the `ExaFMM` code, which uses the alternative, Dual Tree Traversal (DTT) variant. Hence, we extend the discussion to include it here as well.

DTT is reminiscent of the classic depth-first-search and breadth-first-search algorithms for visiting the nodes of a graph, with the exception that it works on pairs of cells. Listing 3.1 illustrates this tree traversal. The traversal is based on a stack, which is initialized to contain the pair (root, root). While the stack is not empty, a pair is popped from the stack, the larger of the two cells is split and new pairs are formed by pairing the smaller cell with the children of the larger one. The new pairs are then considered. If both cells of a pair are leaves, they are interacted via a direct $\mathcal{O}(N^2)$ calculation. Otherwise the separation criterion is evaluated on the cells. If the separation criterion is fulfilled, then an M2L operation is performed and, if not, then the pair is pushed to the stack for further examination.

Listing 3.1: Dual Tree Traversal

```
Stack<pair<Source, Target>> st;
st.push(pair(root, root));
while (st.isNotEmpty()) {
    Pair (source_A, target_B) = st.pop();
    if (source_A.radius() > target_B.radius()) {
        for_each (a = source_A.children()) {
            Interact(a, target_B, st);
        }
    } else {
        for_each (b = target_B.children()) {
            Interact(source_A,b, st);
        }
    }
}
void Interact(Source & S, Target & T, Stack & st) {
if (S.isLeaf() and T.isLeaf()) {
    P2P(S,T);
} else if (separation_criterion(S,T) == true) {
    M2L(S,T)
} else {
    st.push(pair(S,T));
}
```

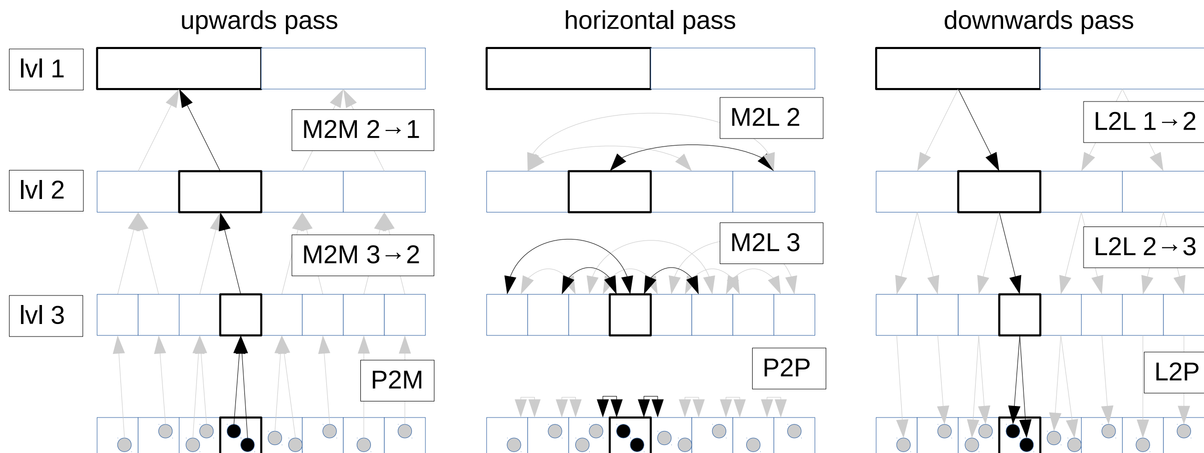


Figure 3.7: Tree operations and dependencies in FMM. In the absence of boundary conditions, work can stop at level 2, since there is no M2L work on levels 1 or 0. Black arrows and cells indicate the flow of information for the molecules coloured in black.

An advantage of this variant is that it can work with cells of arbitrary positions, shapes and sizes. This can be used to optimize the FMM calculation by shrinking cells so that their bounding sphere is of minimal size. Doing so can improve the accuracy and runtime of FMM [121].

3.7 Algorithm

Having explored all the principal building blocks, we are now ready to present the full FMM algorithm. Most of the FMM literature recognizes two "passes": "upwards pass" and "downwards pass". In the following, we will deviate from the traditional notation, by distinguishing a separate "horizontal pass" from the downward pass, as also some other authors do [1]. We believe that the presented description better reflects the symmetry between the upward- and downward passes and also better reflects the mathematical properties of the involved operations. We now describe the individual steps, illustrated by Figure 3.7.

0. **Initialization**
   During the initialization phase, the tree structure is constructed. Multipole and local expansions are allocated for each cell and their coefficients are set to zero.

1. **Upwards pass**
   In the upwards pass, operations take place from each fine level to the next coarser level. Thus, clusters are formed from the finest level "upwards" to the root of the tree. For a cell on a particular level, the operations on all children cells need to be completed before the current cell can be processed. Source pseudoparticles are formed in a hierarchical fashion, by computing a multipole expansion for each cell, which represents all sources contained in the volume spanned by the cell. At the finest (lowest) level, Particle-To-Multipole (P2M) operations take place by evaluating the **L** coefficients for the (x, y, z) vector giving the displacement of the particle from the center of the cell. These coefficients are then multiplied by the charge of the particle and added to the **L** coefficients of the respective multipole particle. At the coarser levels, Multipole-To-Multipole (M2M) operations take place by shifting the expansions of the next finer level via Equation (3.8) and adding them

to the coefficients of the respective multipole cell. The process proceeds, in general, until a multipole cell has been formed for the root of the tree, containing the influence of all particles.

2. **Horizontal pass**
During the horizontal pass the tree traversal is performed, either via LBT or DTT. Operations take place generally within each level, although the DTT variant may also allow operations between different levels if the separation criterion is fulfilled. M2L operations are performed on coarser levels via Equation (3.9), which have the effect of computing the influence of the respective source pseudoparticle on the respective target pseudoparticle. The contributions are added to the — hitherto zero-valued — **M** coefficients of the local expansion. Finally, P2P operations are performed on leaf cells. All operations in this pass are mathematical additions and, hence, are associative and can be executed in any order. This exposes a very high degree of parallelism.

3. **Downwards pass**
In the downwards pass, operations go from each coarse level to the next finer level. For a cell on a particular level, the operations on all parent cells need to be completed before the current cell can be processed. Local-To-Local (L2L) operations are performed for each cell via Equation (3.10). The effect is that local expansions on one level receive the aggregated influence from M2L operations on all parent levels. As one descends deeper into the tree, the aggregated influence of ever larger regions of the simulation domain get accumulated in the local expansions. This proceeds all the way down to the finest level, where the local expansions now store information from all cells, except for the $3^D - 1$ directly neighbouring cells. Local-To-Particle (L2P) operations then take place, adding potentials and forces onto each particle. As the $3^D - 1$ direct cell neighbours have been taken into account via the P2P operations of the horizontal pass, potential and force contributions from the entire simulation box have been computed for each particle.

4. **Next iteration**
Before beginning the next iteration (with updated positions of the particles), all expansion coefficients are zeroed and any particles which have moved from one cell to another are resorted. The next upwards pass can then begin.

In the traditional FMM literature, the horizontal pass is part of the downward pass. Starting at the top, on each level, one performs M2L operations and, after they are completed, performs L2L operations to propagate the accumulated values to the children cells. Then, on the next coarser level, one again performs all M2L operations followed by L2L operations, and so on.

3.8 Factors affecting runtime of FMM

In this section we comment on some of the different factors affecting the overall FMM runtime.

Runtime of the different passes The upwards pass is relatively cheap: one P2M operation is performed per particle, and one M2M operation per cell. Similarly, the downwards pass is also cheap: one L2L operation is performed per cell, and one L2P operation is performed per particle. The horizontal pass is the most computationally intensive one by a large margin. First, $6^D - 3^D$ M2L operations are performed per cell, which outweighs the count of M2M or L2L operations by far. Moreover, the P2P operations involve locally $\mathcal{O}(n^2)$ operations for n particles per cell, which is usually more expensive than the P2M or L2P calls (except if p is very high

and n is very low). For these reasons, the horizontal pass is the most expensive one and, hence, the greatest optimization effort is dedicated to it [70].

Distribution of M2L work among tree levels Next, we point out that most of the M2L work is on the finest level of the tree. To see this, consider that the number of M2L operations scales linearly with the number of cells. If the number of cells on the finest level is $N_{cells,F}$, then the number of cells in all higher levels can be computed via the simple geometric series

$$\frac{N_{cells,F}}{8} + \frac{N_{cells,F}}{64} + \frac{N_{cells,F}}{512} + \ldots = \frac{\frac{N_{cells,F}}{8}}{1 - \frac{1}{8}} = \frac{N_{cells,F}}{7}. \tag{3.26}$$

This means that the M2L workload in all higher levels combined is about seven times lower than on the finest level.

The need for balancing P2P and M2L We now point out that minimizing runtime of the algorithm depends on a delicate balance between the M2L and P2P workloads. To see this, consider an increase of the depth of the tree by one level. This would imply that the volume of cells at the finest level shrinks by a factor of eight. Since P2P is $\mathcal{O}(n^2)$, this means that the number of particle-to-particle interactions shrinks by a factor of 64. Doing so, however, adds a number of cells eight times larger than before, which increases the M2L workload by roughly a factor of eight. Thus, minimizing runtime mandates a careful choice of the depth of the tree in order to find the minimum between the sum of the P2P and the M2L runtime. The need to balance between the M2L and P2P workloads is illustrated in Section 10.3.

Optimizing M2L The next thing to note is that if one could optimize the M2L runtime by large factors, one could go to a tree, which is one level deeper, allowing for potentially large reductions of runtime, due to the strong reduction of P2P time. For this reason, a great deal of effort has gone into optimizing the M2L phase. The possibilities are multiple and extensively studied in the FMM literature [70]. Some of the possibilities are:

- low-level optimizations, such as loop unrolling or template meta programming [10,62,121],
- Wigner rotation accelerations [119],
- Fast Fourier Transform (FFT) accelerations [30,39,69],
- plane wave acceleration [50].

Optimizations of the complexity from $\mathcal{O}(p^6)$ down to $\mathcal{O}(p^2 \log p)$ are possible, yielding the desired large factors. Our choice of acceleration has been FFT with $\mathcal{O}(p^2 \log p)$ scaling and will be detailed in Chapter 11.

3.9 Further remarks

Boundary conditions Up to this point, boundary conditions for FMM have not been discussed. Performing the outlined algorithm with no special treatment of cells touching the boundary would result in "open" or "free-space" boundary conditions. Adding periodic boundary conditions, however, is relatively straightforward [65,67] and will be briefly discussed in Section 10.2.2.

Adaptivity The outlined algorithm assumes a uniform tree structure and particle distribution. If that is not the case, the cell structure can be made more adaptive, by building the tree by subdivision in a top-down fashion and subdividing it if a predetermined threshold for number of particles per cell is exceeded. Another possibility is to define and implement two further operations: Multipole-To-Particle (M2P) and Particle-To-Local (P2L) [122]. These could be used to increase adaptivity, as, for example, it could be more efficient to deal with individual particles, instead of forming (potentially expensive) multipole or local expansions for cells with very few particles.

3.10 Choices and difficulties upon implementation

A new FMM implementation for `ls1 mardyn` Different application scenarios (in MD) impose different requirements on the FMM implementation. This includes precision of the calculation (primarily determined by the truncation order p), supported electrostatic potentials, adaptivity or boundary conditions. This makes it difficult to use an FMM implementation, which was developed for a different application.

In the case of `ls1 mardyn`, support for the charge, dipole and quadrupole potentials is desired, as well as treatment of multicentered molecules. Moreover, inhomogeneous particle distributions and the ability to switch periodicity on or off for one or multiple dimensions are desired.

Finally and most importantly, `ls1 mardyn` aims for a very good and memory-efficient HPC execution, so that simulations on entire supercomputers using all of the available RAM can be performed [27,108]. Achieving this would be very difficult or even impossible, if an external FMM library was to be used. For these reasons a new FMM implementation for `ls1 mardyn` was begun.

Choices and difficulties in comparing FMM A critical evaluation of a method and its implementation requires a comparison to alternative methods and implementations. In this regard, the task of comparing FMM is made difficult by the variety of choices that need to be made during implementation:

- mathematical expansions and basis functions:
  - Solid Harmonics,
  - Spherical Harmonics,
  - Cartesian Taylor,
  - Equivalent charges,
- accelerations of the M2L phase:
  - low-level optimizations,
  - FFT acceleration,
  - Wigner rotation acceleration,
  - plane wave acceleration,
- tree traversal:
  - LBT,
  - DTT.

All of these choices affect FMM performance and some of them affect also accuracy. Moreover, implementational optimizations, such as SIMD, shared- and distributed memory parallelization, also affect performance drastically. Furthermore, parameters chosen at runtime, such as truncation order or tree depth, also influence performance considerably. For this reason, FMM is informally often referred to as “fragile”, meaning that if one component is not optimized sufficiently, overall performance can suffer considerably. This also makes it very difficult to compare FMM to alternatives, which we believe is one of the reasons why contradictory statements are found about FMM in the literature. Some authors find FMM inferior to other methods [91], while other authors praise it [6,70].

Keeping the aforementioned arguments in mind, in the current work we make a best attempt at comparing the performance of our FMM implementation to `ExaFMM`[1]. `ExaFMM` is a high-performing, open-source implementation, which is claimed to be among the fastest FMM codes [121]. In this fashion we evaluate the quality of our implementation relative to an established FMM code. We leave the evaluation of FMM against other methods to other works, such as [6].

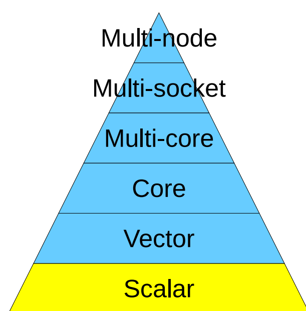
[1] `https://github.com/exafmm/exafmm`

PART III

NODE-LEVEL PERFORMANCE FOR MOLECULAR DYNAMICS

In this part, we explore the node-level performance of the program `ls1 mardyn` with the specifics required for its target applications in chemical engineering. The implementational optimizations performed in this work are presented here. We focus primarily on the short-range force calculation and describe the changes introduced to leverage its node-level performance.

The optimizations are presented in the bottom-up fashion outlined in Section 2.2. Chapters 4 to 8 present, respectively, our work on scalar-level-, vector-level-, core-level-, multicore-level- and multisocket-level performance optimization. Chapter 9 puts everything together and presents an evaluation of the overall achieved performance. Comparisons to a reference version of `ls1 mardyn` and to other high performing MD codes are included. Finally, the improvements are leveraged to perform the world's first simulation with up to $2 \cdot 10^{13}$ molecules.



4 Scalar-Level Performance

We begin our discussion with scalar-level performance analysis and optimizations as they lay out the foundations for all further optimization work. In Section 4.1, we discuss the data structures and layout of the production trunk of `ls1 mardyn` and the improvements introduced as part of this work. In Section 4.2, a newly introduced functionality to change the floating-point precision of `ls1 mardyn` is presented and evaluated. Section 4.3 presents the "Reduced Memory Mode" (`RMM`) feature of `ls1 mardyn`, which was recently introduced to the production trunk. Finally, Section 4.4 summarizes the lessons learned in this chapter.

4.1 Data structures and layout

4.1.1 Former data structures

Prior to the start of this work, the data structures of `ls1 mardyn` were targeting primarily a pure-MPI execution, and — while SIMD vectorization was beginning to be introduced — the data structures were still more oriented towards the legacy AoS force calculation. When transitioning to an implementation, which aims to leverage SIMD and multi-threaded OpenMP execution, several of the features of the former design proved to be obstacles, which needed to be addressed.

Global `std::list<Molecule>` storage All molecules of a process were stored in a global `std::list`. This way of storing the molecules makes iterating over them in application-specific code easy and convenient. However, an `std::list` is difficult to parallelize via OpenMP `for`-loop parallelization, since OpenMP mandates loops to be in the canonical form [106]. While workarounds are possible (e.g. via using OpenMP tasks, or via a separately maintained `std::vector` of pointers to each molecule), they impose an undesirable overhead.

Linked Cells implemented via `std::vector<Molecule*>` The Linked Cells algorithm used to store a `std::vector<Molecule*>` structure for each cell. Implementing the algorithm in this way makes the resorting of molecules in cells cheap, as only pointers need to be moved from one cell to another. This allows to avoid copying full `Molecule` objects, which can reach up to 288 Bytes in size [108]. Even more problematic to copy are the molecule "Caches", containing pointers to interaction site data (discussed in the following paragraph). This implementational choice, however, implies indirect accesses to memory during the force calculation, which — being locally a $\mathcal{O}(N^2)$ operation — needs to access molecules many more times, than resorting which is an $\mathcal{O}(N)$ operation. Moreover, MD simulations can run for millions of timesteps. After such long periods, molecules can move considerably in space, which means that the per-cell vector of pointers can point to locations, which are distant in memory, leading to a poor locality of the memory accesses.

Legacy AoS-like "Caches" for handling multicentered molecules As discussed in Section 1.2, multicentered molecules store only the position and orientation, but not all sites of a molecule explicitly. For the purposes of application-specific functionality which may need the position of individual sites (as well as the legacy, unvectorized force calculation), molecules had additional class fields for site data, see Listing 4.1. As the number of sites — and type of sites — is variable, however, these fields cannot be statically allocated to arrays of fixed size. This means that they need to be dynamically allocated, which involves costly system calls to `new` and `delete`. Copying or moving molecules, thus, needs the additional care of managing this dynamically allocated memory. Moreover, the dynamically allocated storage for sites can lie arbitrarily far in RAM from the storage allocated for the molecule itself, leading again to poor locality of memory accesses. Last but not least, these "Caches" are ultimately part of the `Molecule` class and should, thus, be regarded as part of the AoS layout, which is suboptimal for vectorization.

Listing 4.1: Legacy Molecule "Caches" for site positions, forces and torques

```
1  class Molecule {
2      ...
3      // site positions
4      double *_sites_d;
5      double *_ljcenters_d, *_charges_d, *_dipoles_d, *_quadrupoles_d;
6      // site orientation for dipoles, quadrupoles
7      double *_osites_e;
8      double *_dipoles_e, *_quadrupoles_e;
9      // site Forces
10     double *_sites_F;
11     double *_ljcenters_F, *_charges_F, *_dipoles_F, *_quadrupoles_F;
12     ...
13 };
```

Sliding window and SoA management Different memory representations of molecules were investigated in [29]. The "sliding window" approach was adapted, see Figure 4.1. For the purposes of vectorization, SoA storage was introduced in [26] and constructed on-the-fly only for cells, currently in the sliding window. This means that SoA storage was constructed for roughly one $D-1$-dimensional slice of the D-dimensional domain, thus keeping memory overhead for SoA storage low. While this approach worked quite well for one or two threads per MPI process ([25] or, respectively [27]), it becomes problematic for running tens or hundreds of OpenMP threads. This is because either a lot of synchronization would be needed to work within the same window, or multiple windows would need to be opened. The latter would require to keep track of potential intersections to avoid race-conditions, which complicates the implementation. Last but not least, we point out that for multicentered molecules, the SoA storage also stores site information, similar to the molecule "Caches" [28], implying a duplication of the storage.

Hand-written intrinsics force-calculation kernels The force calculation kernels were implemented directly using intrinsics and conditional compilation. This had the disadvantage of code-duplication and difficult maintenance, as each kernel was implemented once for each instruction set. This meant that up to seven different versions of the kernels needed to be maintained: AoS, SoA, SoA with SSE, SoA with AVX, SoA with AVX2, SoA with KNC and SoA with AVX512. This nearly sevenfold duplication would have been very difficult to maintain, should the need for changes or bugfixes arise.

24	25	26	27	28	29	30	31
16	17	18	19	20	21	22	23
8	9	10	11	12	13	14	15
0	1	2	3	4	5	6	7

Figure 4.1: Illustration of the sliding window approach for the force calculation with *Newton3*. The currently processed cell is in dark grey (cell 11). Cells in light grey are within the sliding window. When processing cell 11, cell 20 is touched for the first time and added to the window. After the work on cell 11 is finished, it is removed from the window and cell 21 is added to it. Cell 12 is processed next. If the *Newton3* optimization would not be used, the window would extend for one more layer, i.e. to cell 2.

4.1.2 New requirements for `ls1 mardyn`

While working on extending `ls1 mardyn` for execution on the KNC architecture [57,109], two new requirements for the efficient node-level execution of `ls1 mardyn` became acute:

- **Massive node-level parallelism**
  The KNC architecture features over 60 cores, each capable of four-way hyperthreading. In order to leverage its full performance, however, at least two threads per core should be started, raising the total number of threads and/or processes to 120. As we shall see in subsequent chapters, gains can be expected from running on up to all four hyperthreads per core, raising the number to 240. Furthermore, even Xeon architectures can support up to 112 threads in a dual-socket Skylake configuration, highlighting the need for good node-level parallelism further.

- **Memory efficient execution**
  As is usually expected from accelerator architectures like KNC, a sufficiently large computational load is required, in order to leverage their full potential. On the other hand, only 8 to 16 GBytes of RAM are available for the card, rendering a need for memory efficiency. Initial experiments with `ls1 mardyn` on KNC showed that running 120–240 MPI ranks per card can either exceed the memory due to extensive memory overhead for halo layers, or be inefficient, due to duplication of force calculations within the halo layers [57]. This illustrates the need and potential for advanced shared-memory schemes, which can avoid the halo-storage needed by MPI altogether.

In order to meet these requirements, great effort was invested in devising memory-efficient OpenMP schemes for `ls1 mardyn`, which meant considerable changes to the program's data structures.

4.1.3 New data structures

Work in this direction began already in [57] and was brought to maturity with the present work. At first, a specialized branch of `ls1 mardyn` was developed to experiment with new data structures. The changes were then brought to the production trunk with [108]. We now list the most significant changes.

Global `std::list<Molecule>` storage removed The former global list-based storage of molecules residing on one process was removed. Instead, molecules are stored directly via objects in `std::vector<Molecule>` per cell. New classes for allowing traversals of all molecules through a convenient `ParticleIterator` interface were developed. The `ParticleIterator` provides classic C++ `std::iterator` functionality for use in the application-specific code. Internally, the `operator++()` handles access to the `std::vector<Molecule>` of a cell and the induced jumps from one cell's `std::vector` to the next.

Linked Cells implemented via `std::vector<Molecule>` Despite the outlined advantages of the former design, this design has the advantage of better memory locality. Moreover, this design makes it easier to iterate over subsets of the simulation domain to e.g. apply boundary conditions or application-specific functionality. Furthermore, resorting of particles can make use of additional assumptions or optimizations — for example that particles move only from one cell to a neighbouring one. Overall, this change was found to be beneficial, in contrast to the findings of [29]. This is in part also because it was noticed, that an observed initial overhead for (re-) allocating the vector of Molecule objects would amortize after a few iterations. Moreover, it was decided to drop support for running with cell length $l < r_c$ (due to limited gains and design complications), which was the more underperforming case in this comparison in [29].

Molecule Caches merged in SoAs Based on the observation, that molecule Caches store site data, which is also duplicated in SoAs, it was decided to remove the legacy Caches and use the SoAs to provide that functionality instead. This meant that `Molecule` objects are no longer responsible for dynamically allocated data, which simplifies moving molecules from one cell to another. Moreover, the memory is less fragmented, because the total calls to `new` are reduced from $\mathcal{O}(N)$ to $\mathcal{O}(C)$, where N is the number of molecules in the simulation and C is the number of cells.

Sliding window removed Since SoAs are now permanently stored, there is no longer a need for a sliding window. Each cell now has an SoA reference, which avoids additional indirections of memory. In this way, threads can work on the force calculation in all cells of the domain simultaneously, as long as the arising race-conditions are handled. Multiple new OpenMP schemes have been developed for handling the race-conditions and will be discussed in Chapter 7 at length. For switching among these OpenMP schemes — also at runtime — a separate interface, called `CellPairTraversals` was extracted from the `LinkedCells` class. The new OpenMP schemes were implemented by realizing this interface in different ways.

SoA optimizations Some optimizations were performed in the SoA structures as well. In order to prevent false-sharing, SoAs always work on data which is both aligned to the start of cache lines and of length multiples of cache line length (=64 Byte). The memory footprint of SoAs was reduced by appending the y-coordinates of molecules at the end of the x-coordinates and the z-coordinates at the end of the y-coordinates. All three coordinates are, thus, allocated contiguously in one array, but with 64-Byte padding between them. After that, data of sites of different type (Lennard-Jones, Coulomb, ...) were merged together in a new `ConcatenatedSites` class, further reducing the memory footprint. The number of SoA reallocations was also reduced, by allowing them to grow as needed to accommodate incoming molecules, but not shrink them, when molecules leave a cell. This is not a problem, because if the scenario is homogeneous, molecules do not cluster up and the counts of molecules per cell in the entire domain remain

relatively constant. If the scenario is inhomogeneous, the entire Linked Cells container (including cells and SoAs) is periodically rebuilt during MPI rebalancing of the simulation, keeping sizes close to the necessary values.

Intrinsics wrappers Multiple new classes and data structures were introduced for wrapping around the intrinsics code. These allow maintaining only two versions of the code: AoS and (wrapped) SoA. The AoS version is maintained primarily for backwards compatibility. Although internally again managed via conditional compilation, the SoA variant implements the kernels once using special data types and those data types are then — internally — conditionally compiled for the different instruction sets. On top of that, for each of the aforementioned instruction sets, three different floating-point precision modes can be selected — single, mixed or double. See Section 4.2 for a more detailed discussion on the choice of floating-point precision modes and Chapter 5 on the vectorization modes.

ThreadData pattern While working on cell-pair traversals, a particular ThreadData usage pattern emerged for variables, needed by individual threads. Ideally, the threadprivate OpenMP directive would have been used. But, due to the separation of the #pragma omp parallel statement in one class and the work on the cell pairs in (several) other classes, using threadprivate was not feasible. Moreover, sometimes more advanced data structures or functionality on that data was needed. For these reasons, nested ThreadData classes were implemented inside the classes, which work on cells (CellProcessor classes). For each thread, a ThreadData object was dynamically allocated, obeying the first-touch policy, to ensure maximal locality and convenience.

4.1.4 Software structure

Figure 4.2 shows a class diagram illustrating the key features of the node-level core functionality. The software can be logically grouped into multiple packages[1]:

- **SIMD and precision**
  Classes in this group are responsible for providing intrinsics wrappers for vectorizing for the different instruction sets and for changing the floating-point precision. The different intrinsic types, e.g. __m256d, __m512d or __m256, are wrapped in a templated class RealVec<typename T> with specializations RealVec<double> and RealVec<float>, respectively. Functionality for mathematical operations (e.g. _mm256_mul_pd) or loading and storing of data to and from SIMD registers is provided via member functions and operators of those classes (e.g. operator*()). Care is taken that all member functions are inlined, so that no runtime penalties arise. The code for different instruction sets is selected conditionally via preprocessor macros. Changing the floating-point precision is done via conditional typedef definitions (SPSP, SPDP and DPDP) and the classes RealCalcVec, RealAccumVec and RealAccumVecSPDP. How this is done is explained in more detail in Section 4.2. Two different means for handling the cutoff condition are supported via the Chooser (MaskChooser or GatherChooser), as well as MaskVec<typename T> with specializations MaskVec<float> and MaskVec<double>. The MaskChooser handles the cutoff condition by multiplying contributions beyond the cutoff radius by zero (this process is referred to as "masking"). The GatherChooser handles the condition by first computing

[1]Note that in the code some of those packages are not formally organized into e.g. subfolders or namespaces.

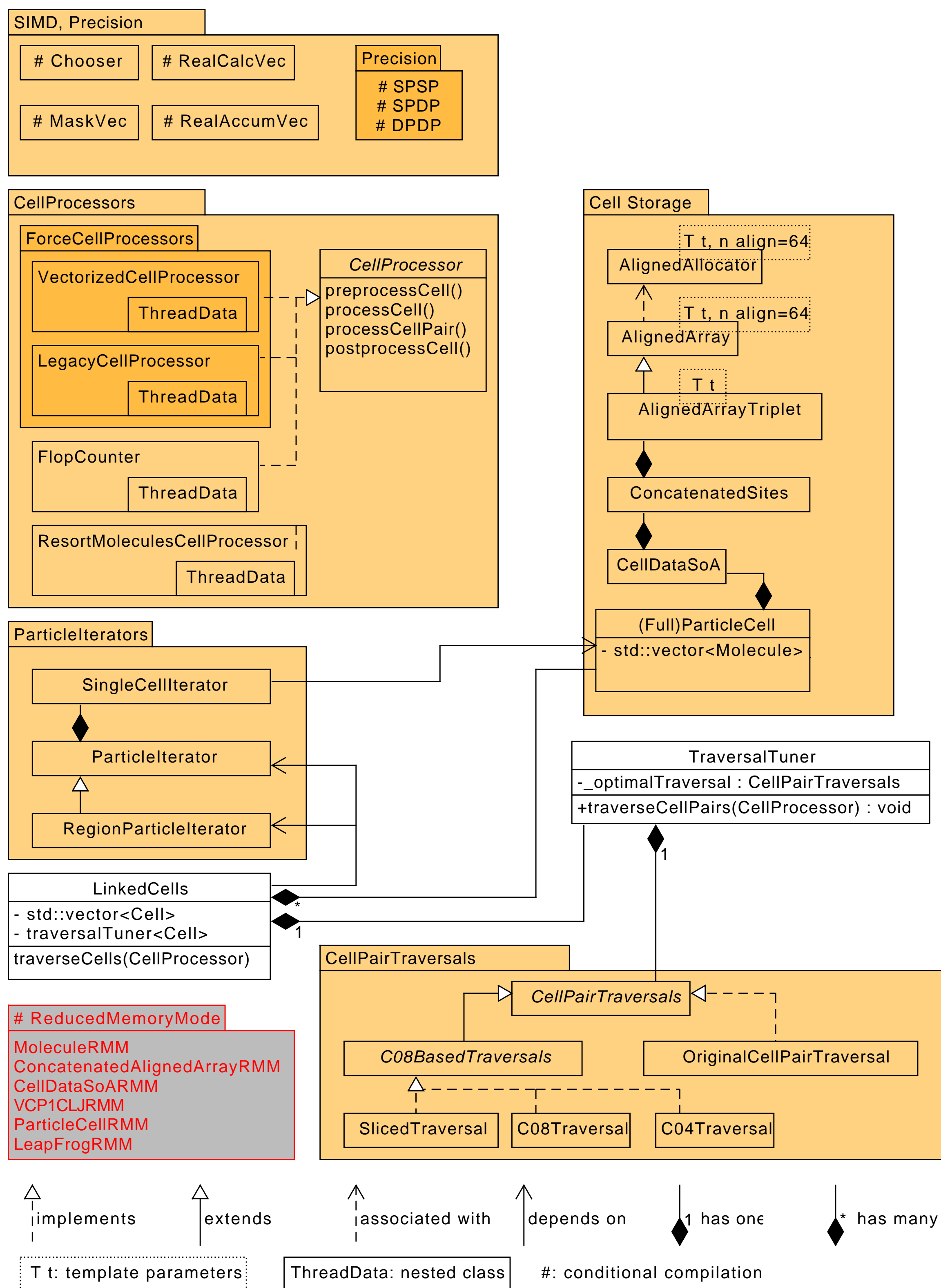


Figure 4.2: Partial class diagram of `ls1 mardyn` illustrating software structure basis for the force calculation and iterating over molecules.

the indices of molecules, which are within the cutoff radius and then “gathering” those molecules into registers and “scattering” computed forces back, if *Newton3* is used. Further details on the vectorization are presented in Chapter 5.

- **Storage**
  This group of classes is responsible for molecule storage and data layout. A custom C++11 allocator was written [78], so that std::vector < T, AlignedAllocator < T, alignment > > could be used for convenience in SoA structures, as opposed to using a plain-C pointer. The allocator is placed in the class AlignedAllocator < typename T, size_t Alignment = CACHE_LINE_SIZE> and ensures alignment of all allocated data, relevant for SIMD operations. The first template argument is the type (e.g. float, double, int, etc.), while the second one is the alignment, which is intended to always be of the size of one cache line. As cache lines on all architectures investigated in this work are of size 64 Bytes, the value is hardcoded, but can easily be changed.

  The allocator is used by the class AlignedArray < typename T, size_t Alignment >, which wraps a member field of the type std::vector < T, AlignedAllocator < T, alignment > >. It provides the necessary interfaces used in the code and also takes care that the allocated storage in Bytes of the std::vector is always a multiple of the cache line size, regardless of the number of Bytes that the type T requires.

  The classes AlignedArrayTriplet<typename T> and ConcatenatedSites<typename T> are used to reduce the memory footprint and improve the locality of SoAs. As the length of the x-, y- and z-coordinates needed in a cell is always equal, they can be placed one after another and allocated in contiguous storage. This is done in the AlignedArrayTriplet class. Care is taken to insert padding after each coordinate, so that the values of the next coordinate also start at a cache-line-aligned boundary. The class ConcatenatedSites places data of one type (e.g. position) of different sites (LJ, Coulomb, dipole, ...) in a single AlignedArrayTriplet.

  The class CellDataSoA aggregates all variables necessary for the multicentered force calculation on a cell. It totals four ConcatenatedSites objects, five AlignedArrayTriplet objects, eight AlignedArray objects and five integers.

  Finally, each cell used by the Linked Cells container ((Full)ParticleCell) has an std::vector<Molecule> for AoS storage of molecules and a CellDataSoA object for SoA storage. The synchronization between the AoS and SoA storage is done in every iteration.

- **Cell processors**
  The CellProcessor abstract class is used for multiple purposes whenever work on entire cells is intended, instead of on individual molecules. Apart from the force calculation, this includes functionalities such as resorting molecules in cells or counting the total number of floating-point operations, which the force calculation will perform. It defines the following pure virtual methods: initTraversal(), preprocessCell(Cell&), processCell(Cell&), processCellPair(Cell&, Cell&), postprocessCell(Cell&), endTraversal(). Depending on the purpose of the different cell processors, they implement one or several of the methods. Processors for the force calculation implement it within the processCell and processCellPair methods.

- **Cell-pair traversals**
  The abstract class CellPairTraversals provides a common interface for schemes for iterating over pairs of cells. It defines (among others) the virtual method traverseCellPairs (CellProcessor *), which is the primary method used for the force calculation. Through

a pointer to an `std::vector` of cells passed from the `LinkedCells` class, cell-pair traversals take care that all pairs of an inner cell and all its neighbouring cells are traversed. Notably, the different OpenMP parallelization schemes explored in Chapter 7 are realised via different traversals extending the `CellPairTraversals` and `C08BasedTraversals` abstract classes.

- **ParticleIterators**
  The purpose of classes within this package is to provide a convenient and OpenMP-parallelizable interface for traversing all molecules from the outside, e.g. from within application-specific codes. As the name suggests, the `SingleCellIterator` traverses molecules within one cell, while the `ParticleIterator` manages the logic for moving from one cell to the next and exposes AoS objects to the outside. Depending on whether the `ParticleIterator` is called within an OpenMP `parallel` region or not, the work is internally OpenMP parallelized over cells, or not. This is achieved via striding based on the thread id and the total number of threads. The `RegionParticleIterator` class is an extension of the `ParticleIterator` class and can additionally specify a geometric box so that only molecules within it are traversed (in parallel or not), instead of traversing the entire simulation domain.

- **Traversal tuner and Linked Cells class**
  The `LinkedCells` class stores an `std::vector` of cells, which are worked on as a three-dimensional array. It has a `TraversalTuner` object, which was intended to autotune among the different OpenMP parallelization strategies, prior to the extraction of this functionality in a separate library, see Section 9.5. The Linked Cells class constructs `ParticleIterator` objects when needed from the outside.

- **`ReducedMemoryMode`** As part of the work in [108], a specific, “Reduced Memory Mode” (**`RMM`**) was developed for `ls1 mardyn` and integrated in the production trunk. Its purpose is to provide memory-optimized functionality for the 1CLJ case by dropping all generalizations needed for the implementation of multicentered molecules. The size of a molecule is reduced from roughly 288 Bytes down to 32 Bytes. To this end, multiple, reduced-memory variants of existing classes needed to be introduced, for molecule objects, cell objects and force calculation, among others. This was done partially through C++ polymorphism and partially through conditional compilation, in order to avoid `virtual` function calls propagating through the entire code base. Classes such as `MoleculeRMM`, `ConcatenatedAlignedArrayRMM`, `ParticleCellRMM`, `VCP1CLJRMM` were introduced. Further details are given in Section 4.3.

4.1.5 Results

Isolated effect of changes to data structures The changes to the data structures and layout explained here were introduced incrementally over a large span of time. Other optimizations of the code were also introduced in the meantime. It is, thus, difficult to isolate the effect of the new data structures and layout alone. Nevertheless, the effects of the new software design were considerable and will be pointed out in Section 8.1.

AoS versus SoA force calculation In Figure 4.3 we present a comparison of the performance of the AoS- and SoA-based force calculations in the default, double precision mode and without vectorization. The simulations were performed on a node of the Hazel Hen supercomputer running a Intel®Xeon®CPU E5-2680 v3. Simulation parameters are given in Table 4.1.

scenario	model	N	cutoff $\frac{r_c}{\sigma}$	density $\rho\sigma^3$	Temperature $\frac{T}{\epsilon}$
Argon	1CLJ	2 048	5.15	0.42	1.71
CO_2	3CLJ3C	2 048	5.99	0.23	7.40
TIP4P	1CLJ3C	512 000	3.17	1.01	4.00
EOX	3CLJD	2 048	5.17	0.35	7.85
C_6H_{12}	6CLJ	16 000	3.00	0.10	4.74

Table 4.1: Simulation parameters for results in Figure 4.3

It should be noted that some optimizations were undertaken, which improve the performance of the SoA-based version, possibly at the cost of lowering the performance of the AoS one. The performance improvements come from the fact that the SoA-based calculation has a better memory locality, as the data of all molecules in a cell lies in contiguous memory locations and does not need to access the `Molecule` objects. Memory accesses are of stride one and only the variables necessary for the force calculation itself are loaded from memory.

Overall, the speed-ups reach up to 2.24× for 1CLJ systems and 1.16× to 1.39× for multi-centered systems. This is because the 1CLJ force calculation is least computationally intensive and, hence, most affected by the improved memory access behaviour.

Analysis through the Intel®Advisor tool[2] for the Argon 1CLJ case reports that a total of 3.95 GFLOP were performed at a 0.028 Flops-per-Byte arithmetic intensity for the AoS version. For the SoA version, the values are 3.74 GFLOP performed at 0.145 Flops-per-Byte arithmetic intensity. The much higher arithmetic intensity used to perform roughly the same number of floating-point operations, thus, results in a high speed-up. The GFLOP values differ slightly, due to a different technical realisation of computation of site-site distances from molecule-molecule distances.

Looking at the EOX case, as a representative of the multicentered cases, Advisor reports 15.85 GFLOP performed at 0.039 Flops-per-Byte intensity for the AoS version and 23.77 GFLOP at 0.16 intensity, respectively. Again, the arithmetic intensity increases nearly three-fold, but the number of floating-point operations has also increased considerably, resulting in a smaller overall speed-up of 1.24×. Indeed, the SoA version performs some additional operations, as compared to the AoS version, in order to be able to vectorize the loop-structure when using multicentered molecules. These extra operations will be discussed in Chapter 5 in more detail.

Overall, switching from the AoS based calculation to the SoA-based one gives speed-ups, which vary strongly, depending on the simulation. For simplicity, in the remainder of this work we will exclude the AoS-based calculation from the analysis, despite that the speed-up over the AoS variant could be considered as part of the vectorization speed-ups, since the change to SoA-based variants was primarily motivated by the vectorization.

4.2 Floating-point precision

At the start of the project, the production trunk of `ls1 mardyn` was hardcoded to using 64-bit double floating-point precision. In one instance, however, [27] a 32-bit single precision branch of the program was developed. In order to enable the runs described in Section 8.1 in an extensible and maintainable manner, it was decided to add the possibility for running in single precision to the production trunk of mardyn. The intrinsics wrappers were extended for that purpose.

[2] `https://software.intel.com/en-us/advisor`

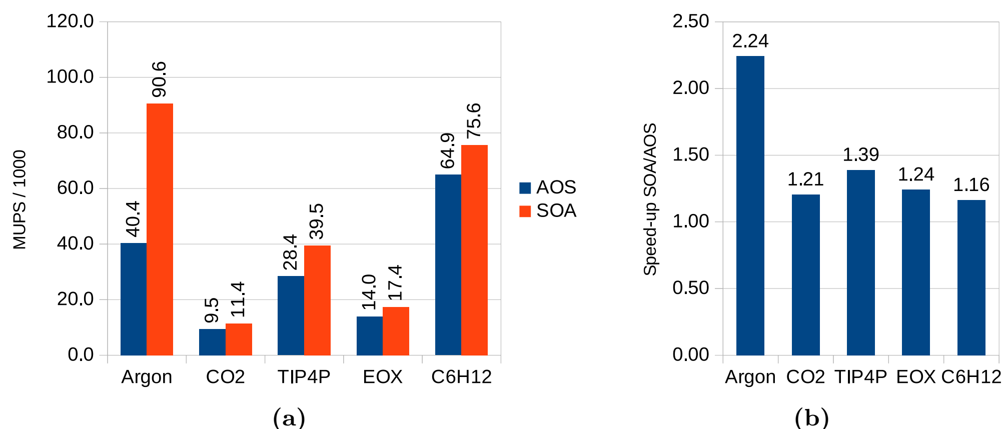


Figure 4.3: Performance comparison of the legacy AoS-based force calculation and the new default SoA-based one without vectorization. (a) Simulation rate in MUP/sec. (b) Speed-ups of the MUP/sec simulation rate.

As other MD packages also offer “mixed” precision, it was decided to incorporate that as well in the intrinsics wrappers.

4.2.1 Precision models

On CPUs, scientific codes are frequently developed using double precision. On early generations of GPUs, however, the single precision floating-point performance was much higher, than the double precision one [43]. For this reason, possibilities for using single precision in MD were explored in order to leverage the performance of GPUs. In [43], three floating-point precision modes were distinguished:

- **SPSP**
  Both calculations (of forces, torques, etc.) and accumulations (summing up force contributions, torque contributions, etc.) are performed and stored in single precision.
- **SPDP**
  Calculations are performed in single precision. Accumulations are performed in double precision. Variables used for the calculation (e.g. molecule position) are stored in single precision, while variables used for the accumulation are stored in double precision. This mode is referred to as “mixed” precision.
- **DPDP**
  Both calculations and accumulations are performed and stored in double precision.

The idea behind the SPDP model is that the forces can vary significantly in sign and magnitude, cf. Figure 1.2. Due to these variations, a loss of significance can occur, e.g. if forces of a similar magnitude, but opposite direction, are accumulated. Hence, the SPDP model was developed, in which forces are computed in single precision, then converted to double precision before being summed up. The investigations of [43] showed that the SPSP model was insufficiently precise for applications in the life sciences, while the SPDP one provided satisfactory precision, while delivering a performance gain.

The following gains from using single precision can be expected on CPUs:

- the simulation data fits better in memory and cache,

- some operations (such as division and square root) take less time to compute in single precision,
- the vectorization width is increased two-fold.

As all force kernels need to compute either $\frac{1}{r}$ or $\frac{1}{r^2}$, the division and square root time reductions could also be visible, especially since these operations are on the critical path of the dependency graph (see e.g. [10] for a detailed analysis of the point-charge kernel). In the case of mixed precision, the upcasting from single to double precision could also cause an overhead, however. Experiments to test the first two effects are presented in this section, while the third effect belongs to vector-level optimizations, and will be revisited in Chapter 5.

4.2.2 Implementation

In this section we describe how changing the precision mode for our intrinsics written kernels was realised in a way to avoid another three-fold code duplication. Listing 4.2 illustrates our approach for computing the forces on a cell pair. The two classes `RealCalcVec` and `RealAccumVec` were introduced. Depending on the precision mode, they are mapped to the classes `RealVecFloat` and `RealVecDouble` and `RealAccumVecSPDP` via conditional compilation and `typedef`-s, see Figure 4.4.

Listing 4.2: Extending the Force Kernels to Different Precision Modes

```
1  for(auto i in Cell1Molecules) {
2    RealCalcVec r_i_x, r_i_y, r_i_z; // position r_i of molecule i
3    RealAccumVec f_i_x, f_i_y, f_i_z; // force f_i of molecule i
4    ... // initialize r_i, f_i
5
6    for(auto j in Cell2Molecules) {
7      RealCalcVec r_j_x, r_j_y, r_j_z; // position r_j of molecule j
8      ... // initialize r_j
9
10     if (insideCutoff) {
11       RealCalcVec f_ij_x, f_ij_y, f_ij_z;
12       ... // compute LJ, Charge, ... force kernels
13
14       RealAccumVec a_f_ij_x = RealAccumVec::convertCalcToAccum(f_ij_x);
15       RealAccumVec a_f_ij_y = RealAccumVec::convertCalcToAccum(f_ij_y);
16       RealAccumVec a_f_ij_z = RealAccumVec::convertCalcToAccum(f_ij_z);
17
18       f_i_x += a_f_ij_x;
19       f_i_y += a_f_ij_y;
20       f_i_z += a_f_ij_z;
21       ...
22     }
23   }
24 }
```

In the SPSP and DPDP cases, both `RealCalcVec` and `RealAccumVec` are set to the same class — `RealVecFloat` or `RealVecDouble`, respectively — via `typedef` statements. In those modes, the upcasting method `RealAccumVec::convertCalcToAccum()` is empty and performs no work. In the SPDP case, two cases are distinguished — whether vectorization is used or not. If no vectorization is used, then `RealAccumVec` stores one instance of `RealVecDouble` and the conversions are simple implicit C-style casts. If vectorization is used, however, the fact, that the `RealVecFloat` class has two times more vector elements than the `RealVecDouble` class needs to be properly handled. For instance, in case of AVX compilation, `RealVecFloat` wraps

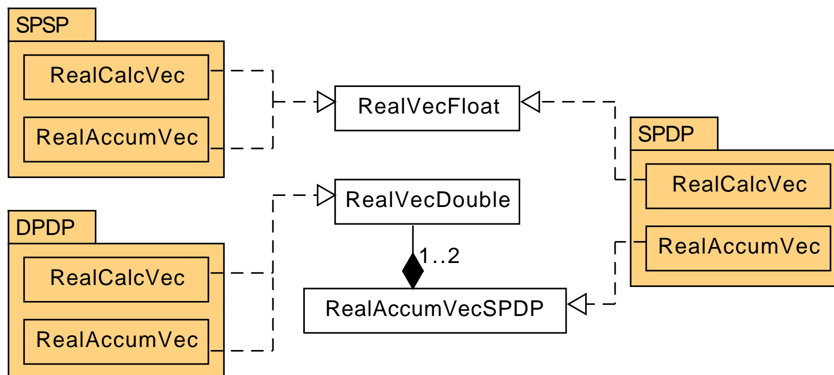


Figure 4.4: Diagram illustrating implementation of different precision modes. In the SPSP and DPDP cases, both the `RealCalcVec` and `RealAccumVec` classes are mapped to either `RealVecFloat` or `RealVecDouble`, respectively, via `typedef` statements. In the SPDP case, the `RealAccumVec` class is mapped to the `RealAccumVecSPDP` class, which works on either one or two instances of the `RealVecDouble` class, in order to be able to work on vectors of the same length as the `RealVecFloat` class.

scenario	model	N	cutoff $\frac{r_c}{\sigma}$	density $\rho\sigma^3$	Temperature
EOX	3CLJD	65 536	6.00	0.45	3.73
C_6H_{12}	6CLJ	16 000	3.00	0.10	4.74

Table 4.2: Simulation parameters for results in Figures 4.5 and 4.6.

eight single precision elements in one `__m256`, while `RealVecDouble` wraps four double precision elements in one `__m256d`. The `RealAccumVec::convertCalcToAccum()` method takes care of that and upcasts the precision in the process.

4.2.3 Results

Validation As changing the precision affects the simulation results, the first question is whether SPSP and SPDP produce results of sufficient physical accuracy. Figure 4.5 shows a comparison between the three modes. Simulation parameters are summarized in Table 4.2. The potential energy averaged over 1000 iterations is plotted versus the iteration number for two molecular systems. The first one is a homogeneous system, containing 65536 Ethylene Oxide molecules, modelled via a three LJ centers and one dipole (3CLJD). The second one is an inhomogeneous system for the study of vapour-liquid equilibrium properties (see e.g. [115]), containing a slab of liquid, surrounded by two slabs of vapour. It contains 16000 Cyclohexane molecules, modelled via six LJ centers (6CLJ). The Long-range correction method of [115] was switched on.

In the Ethylene Oxide system, the results show an excellent agreement between all three modes, not only between SPDP and DPDP. In the Cyclohexane system, the values fluctuate somewhat more, but are still in good agreement with each other. Similar results were also observed for pressure, as well as for systems of Argon (1CLJ) (including a more advanced simulation of coalescence of two droplets) and CO2 (2CLJQ). This is partially in contrast to the findings of [43], where it was found that SPSP was insufficiently accurate for the considered

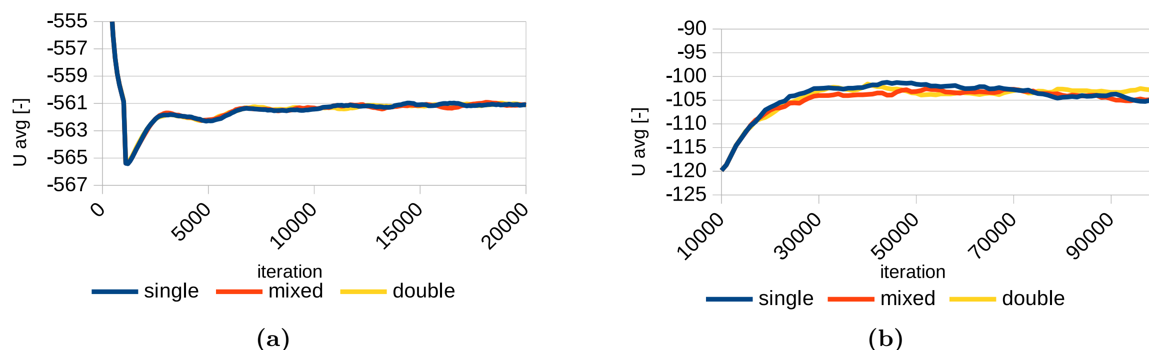


Figure 4.5: Potential energy over iteration number, averaged over the last 1000 iterations. (a) System containing 65536 Ethylene Oxide molecules. (b) System containing 16000 Cyclohexane molecules.

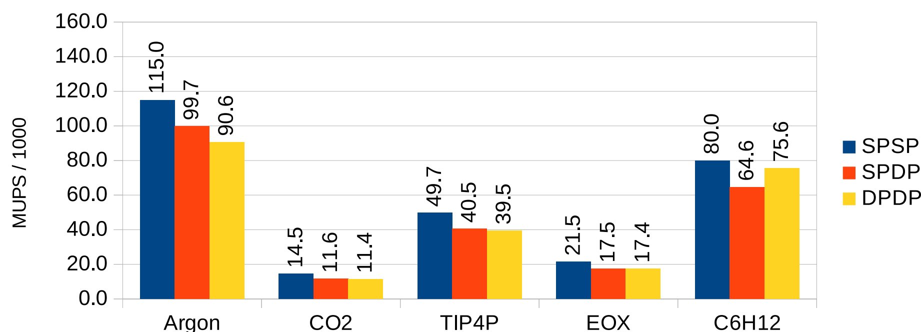


Figure 4.6: Performance of the three different modes for the systems from Table 4.1.

applications from life sciences. These findings suggest that perhaps at least some applications from the application area of `ls1 mardyn` could run in single precision without compromising the physical results. A further study of the effect of precision on other thermodynamic properties and for more systems would, thus, be of interest. It should, however, also include error analyses of the observed quantities, as the trajectories of the different simulations diverge quickly. Hence, these analyses should be performed by users from the application side.

Performance Figure 4.6 plots the performance of all three versions, when compiled without intrinsics, i.e. with vectorization width one. The simulations are carried out on a node of the Hazel Hen supercomputer running an Intel®Xeon®CPU E5-2680 v3. The parameters of the molecular systems are presented in Table 4.1.

Differences between the speed-ups of the force calculation and the total simulation (not shown), are very small, as the force calculation — expectedly — dominates runtime in all scenarios. The speed-ups of SPSP over DPDP are around 1.24× to 1.27×, except for C6H12 where it is only 1.06×. For SPDP, the speed-ups over DPDP are even more modest: 1.01× to 1.10× and drop down to 0.85× for the C6H12 case. This indicates that the kernel is not bandwidth bound, as we would have otherwise expected values closer to 2×.

In an attempt to gain further insight, the Argon and C6H12 scenarios were profiled via the Intel®Amplifier[3] and Intel®Advisor[4] tools on an Intel®Core™i7-4770 CPU @ 3.40GHz

[3] `https://software.intel.com/en-us/vtune`

[4] `https://software.intel.com/en-us/advisor`

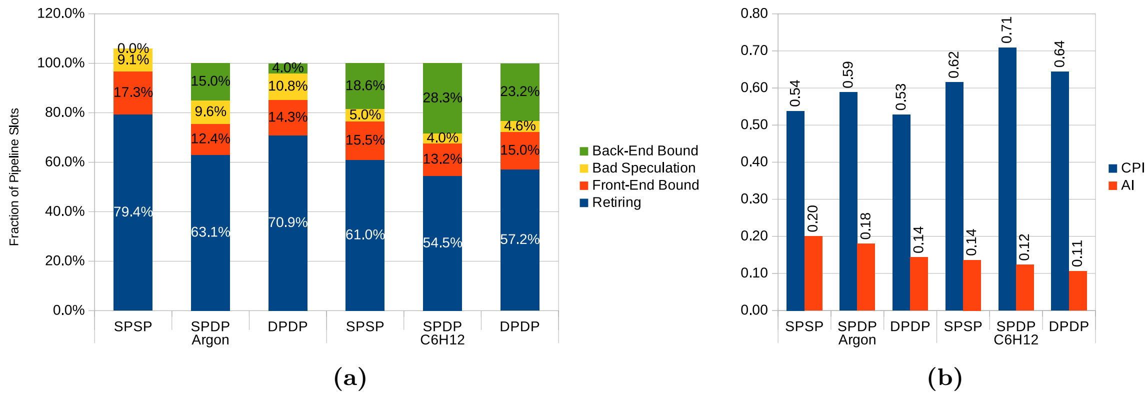


Figure 4.7: Results of Intel Amplifier and Advisor analyses of the Argon and C6H12 cases. (a) Distribution of pipeline slots. (b) Cycles Per Instruction and Arithmetic Intensity.

desktop CPU. The results are shown in Figure 4.7. "Retiring" instructions denote instructions which complete without bottlenecks. "Front-end bound" denotes pipeline stalls, which could be related to fetching operations to execute. "Bad speculation" refers to branch misprediction such as in the cutoff-condition evaluation in the force calculation. "Back-end bound" refers to pipeline stalls related to data-misses, long-latency operations (e.g. division in the force calculation), dependencies in instruction or data flow (e.g. the chained multiplications of the LJ kernel) and others.

Overall, the fraction of "retiring" instructions is quite high in Figure 4.7(a). This means that most of the time the different CPU pipelines are not stalling, but doing productive work and are not severely bounded by e.g. memory or compute throughput capabilities. Looking at the results for different precisions, the SPSP mode has a lower back-end bound fraction. This is — at least in part — due to the fact that the data fits better into cache and the floating-point division takes less time. The SPDP cases feature a higher back-end bound portion than either SPSP or DPDP. A reason could be the additional conversion instructions from `float` to `double`, which are not present in SPSP or DPDP. Comparing Argon and C6H12, the back-end bound fraction increases for C6H12. This could be because much more variables are loaded per molecule.

Looking at the cycles-per-instruction (CPI) and arithmetic intensity (AI, Flops-per-Byte) values in Figure 4.7(b), the AI values expectedly increase when going from DPDP to SPDP or SPSP. The CPI values are roughly equal for SPSP and DPDP, but considerably higher for SPDP and higher for C6H12 than for Argon. This indicates that the conversion operations present in the SPDP mode should not be neglected. Further metrics were also available and collected, both from Advisor and from Amplifier, but they complicate the analysis further and did not reveal significant insights to the author.

It is not immediately clear why the SPDP variant performs poorer than the DPDP variant only for the C6H12 case, although the higher CPI rate hints towards the conversion instructions. Another factor could be the special vapour-liquid configuration of the simulation. In the following sections, however, more cases where DPDP outperforms SPDP are observed. This serves to reinforce the statement that the analysis of the performance for various simulation parameters (including varying degrees of homogeneity) is quite difficult and the need for automatic selection of the best algorithm, see Section 9.5.

scenario	model	N	cutoff $\frac{r_c}{\sigma}$	density $\rho\sigma^3$	Temperature
low density	1CLJ	6 096 960	2.5, 3.5, 5.0	0.39	0.00063336
high density	1CLJ	12 189 632	2.5, 3.5, 5.0	0.78	0.00063336

Table 4.3: Simulation parameters for results in Figure 4.8.

4.3 Reduced memory mode

As part of our work in [108], a specific "Reduced Memory Mode" (`RMM`) was developed for `ls1 mardyn`. This represents a follow-up to the work of [27], in which such a reduced memory mode was developed in a specialized branch in order to maximize the number of molecules, which can be simulated. This time, however, in order to ensure maintainability of the code and to benefit from the other introduced improvements to the main code base, it was decided to integrate this functionality in the production trunk of `ls1 mardyn`.

The core idea is to reduce the storage per molecule objects close to the minimal possible. It is reduced to 32 Bytes: three `float` variables for the position, another three for the velocity and an 8 Byte `unsigned long` unique identifier.

This was achieved through C++ polymorphism and conditional compilation, the latter being kept to a minimum. Different classes were introduced for the needed functionality changes. The class `MoleculeRMM` manages reduced 1CLJ molecular storage, dropping all unnecessary variables and providing an interface to the existing user-code base. Similarly to the class `ConcatenatedAlignedArray`, the class `ConcatenatedAlignedArrayRMM` stores all data of molecules within a cell in a contiguous memory array. This consists of position, velocity and an 8-Byte `unsigned long` unique identifier, in this order. The force calculation is implemented in the `VCP1CLJRMM` vectorized cell processor. Data storage for the `ParticleCell` class was also reduced and resulted in the `ParticleCellRMM` class. It should be pointed out that the `ParticleCellRMM` class does not store an `std::vector<Molecule>`, but instead accesses data directly in the `CellDataSoARMM` structure. Since the storage for forces is dropped, the velocity update of the Leapfrog scheme is merged with the force calculation accumulation. The modified time integration is implemented in the `LeapFrogRMM` class.

Results

In this section we show comparisons between the `RMM` mode and the default one (denoted as `Normal` mode), for the SoA variant, in order to isolate structural changes from vectorization. Figure 4.8 shows the results for the two systems, whose parameters are given in Table 4.3. The simulations are carried out on a node of the Hazel Hen supercomputer running an Intel®Xeon®CPU E5-2680 v3.

Overall, the `Normal` mode performs only about 7% more FLOPs than the `RMM` mode, so the total number of operations performed is not a determining factor of performance differences. The differences lie primarily in the layout of data and specialization of functionality to the 1CLJ case. It can be observed, that the highest gains of `RMM` over `Normal` are for smaller cutoff radii and smaller density. Analysis through Intel VTune (not shown) indicates that in those cases the "Front-End Bound" fraction is high — as high as 38% in the $r_c = 2.5$, $\rho = 0.39$ case — and drops down to 6% in the $r_c = 5.0$, $\rho = 0.78$ case for SPSP. This can be thought of as the overhead incurred from the fragmentation of the data into too many and too small cells. This means that the limiting factor is related to fetching, decoding and issuing instructions, rather than floating-point arithmetic or memory accesses. As the `RMM` mode specializes to the 1CLJ

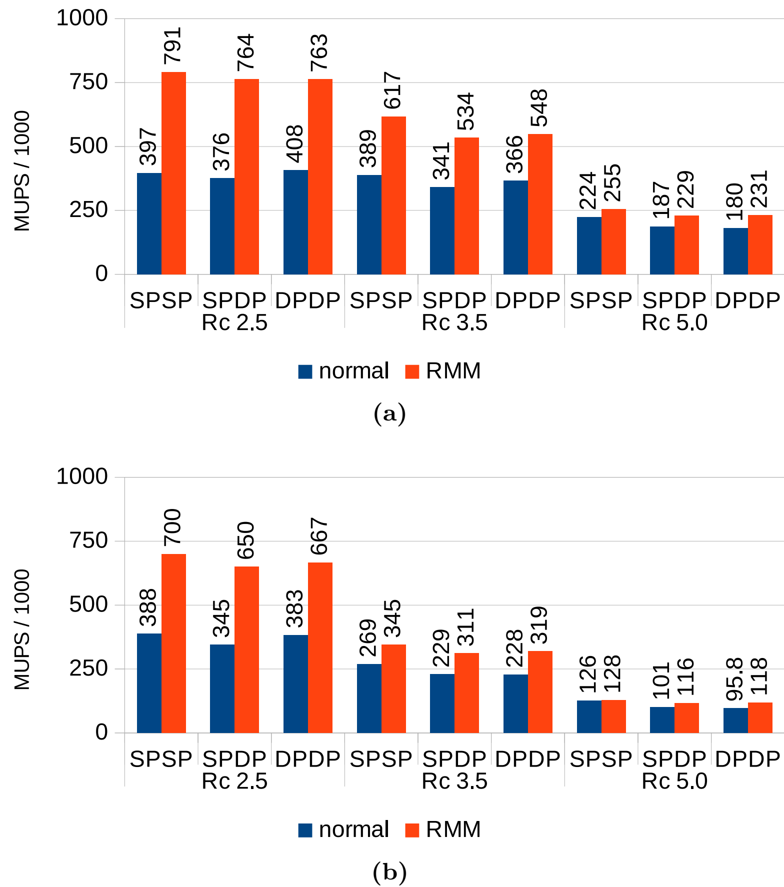


Figure 4.8: Comparison of the modes `Normal` and `RMM` modes. (a) Molecular system of density 0.39. (b) Molecular system of density 0.78.

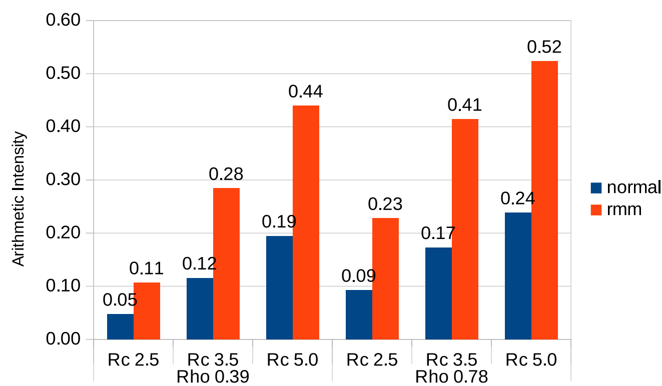


Figure 4.9: Arithmetic intensity as reported by Intel Advisor for the SPSP mode.

model and reduces the memory footprint of all data structures (cells, SoAs, etc.), fewer overall instructions are necessary, explaining the higher gains of `RMM` over `Normal` in the less compute intensive cases.

The observed dependencies on the cutoff radius and the density can also be attributed to this bottleneck. Namely, it explains why the MUP/sec rate barely changes for the $r_c = 2.5$ case between the $\rho = 0.39$ and $\rho = 0.78$ cases. If floating-point arithmetic were the limiting factor, the MUP/sec rate should have decreased by a factor of four with doubling the density, due to the locally $\mathcal{O}(N^2)$ behaviour. In the observation, however, the rate barely drops, meaning that the calculation becomes almost two times more efficient. When comparing the MUP/sec rates between the $\rho = 0.39$ and $\rho = 0.78$ cases for $r_c = 3.5$ and $r_c = 5.0$, the rates drop by factors closer to a factor of two. Furthermore, the weaker dependencies of the MUP/sec rate on the cutoff radius in the $\rho = 0.39$ case are explained. Increasing the cutoff radius from 2.5 to 3.5 means that every molecule has $(\frac{3.5}{2.5})^3 = 2.74$ times more neighbours to interact with, yet the MUP/sec rates drop only by 28% to 39%. The rate drops from 3.5 to 5.0 are around 2.4×, which is closer to the $(\frac{5.0}{3.5})^3 = 2.92\times$. As expected, these drops are higher in the $\rho = 0.78$ case.

Looking at the other extreme case, $\rho = 0.78$, $r_c = 5.0$, when the calculation is very FLOP intensive, `RMM` is only 1.5% faster than `Normal`. Analysis through Intel Advisor of the SPSP case shows that the inner loops of the force calculation run at the "Scalar ADD" peak performance for both the `RMM` and `Normal` modes. These loops run at an arithmetic intensity value of around 0.64 FLOPs-per-Byte, at which value the "Scalar ADD" roof of the roofline model is attained. This means that the calculation proceeds near the maximal speed that the kernel allows and no considerable further speed-ups are expected. The question is how much time is spent in that loop, as a fraction of the total execution time. Figure 4.9 shows the total Flops-per-Byte Arithmetic Intensity, as reported by Intel Advisor for the SPSP case. As the computational effort increases with increasing density and increasing the cutoff radius, more and more time is spent in that loop, bringing the total value higher closer to 0.64. The "Scalar ADD" peak value can be attained from a value of around 0.23 onwards, which is in agreement with the observation, that as the `Normal` mode approaches a value of 0.23, the `Normal` mode catches up to the `RMM` mode.

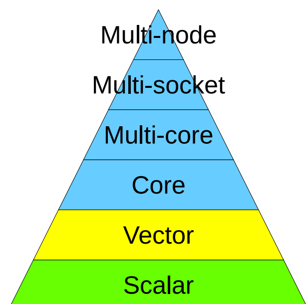
Finally, we comment on the SPDP and DPDP performance. When the simulation is not very compute intensive, the differences in performance are quite small — about 3.5% — which is smaller than the differences observed in Figure 4.6. In those cases, as issuing instructions and the traversal of cells are limiting factors, varying the precision does not affect the performance much. As the calculation becomes more floating-point intensive, the values increase somewhat — up to 10%. This value of 10% is still lower than in Figure 4.6, however. In that case, since scalar addition and multiplication operations take the same number of cycles whether working on data in single or double precision, the performance is again not affected strongly. It is notable that the SPDP calculation again runs slower than the DPDP one in more cases, which is now more easily attributable to the additional conversion operations. The differences are somewhat more pronounced for the `Normal` mode than for the `RMM` one.

4.4 Summary

In this chapter we investigated the scalar-level performance of `ls1 mardyn`. First, the changes to the data structures were presented, which lay the foundation for SIMD vectorization and parallelization. The performance of the legacy AoS version was compared to the SoA performance. Gains of up to 2.24× were observed for the SoA version, illustrating the importance of the memory layout. Throughout the remainder of this work we will consider almost exclusively

the SoA version. The AoS version is maintained primarily for backwards compatibility and development purposes as it is more compact and easy to read than the SoA version.

Next, the precision and `RMM` modes were introduced and evaluated prior to vectorization. Gains of up to 1.27× were demonstrated for the SPSP mode, while the `RMM` mode delivered gains of up to 2×. As the performance of these features depends heavily on SIMD vectorization, their final evaluation is presented in the following chapter.



5 Vector-Level Performance

In this chapter we present the different aspects of the SIMD vectorization of `ls1 mardyn` and our contribution to it. When vectorizing the Linked Cells-based force calculation, multiple possibilities arise.

Consider the computation of forces contained in two neighbouring cells, as illustrated in Figure 5.1(a). Every molecule in cell A needs to interact with all molecules in cell B, which lie within its cutoff sphere. Listing 5.1 shows a prototypical pseudocode implementation, prior to vectorization.

The first question is how to vectorize the nested `for`-loops in lines 1 and 4 of Listing 5.1. For example, one could vectorize the inner loop, the outer loop or a combination of the two. Section 5.1 discusses the possibilities. The next question is how to handle the arising SIMD divergence from the `if`-statement in line 6. That is, since one works on multiple molecules together in one SIMD register, it is possible that only some of the molecules are within the cutoff radius. The handling of these cases is discussed in Section 5.2. After that, the fact that both molecules have a variable number of interaction sites needs to be taken into account by the `compute_multicentered_forces()` function in line 7. The treatment of multicentered molecules is discussed in Section 5.3. Finally, the question how to realize all of this in a portable fashion for the multiple available intrinsic instruction sets is discussed in the context of the introduced wrappers in Section 5.4.

Most of the methodology for vectorization in `ls1 mardyn`, laid out in Sections 5.1 to 5.3, was developed prior to the start of this project in [26–28,53,55]. As part of this project, the changes to the data structures, the precision modes, the intrinsics wrappers and the `RMM` mode were introduced to the production trunk. This meant that the vectorization was almost completely rewritten and, hence, we partially reevaluate it in Section 5.5.

Listing 5.1: Calculation of forces between molecules in two neighbouring cells

```
1  for(int i = 0; i < num_mols_A ; ++i) {
2      mol_A = load_molecules(A,i);
3      force_buffers_a_i = 0.0;
4      for(int j = 0; j < num_mols_B ; ++j) {
5          mol_B = load_molecules(B,i);
6          if(distance(mol_A,mol_B) <= cutoff_Radius) {
7              F = compute_multicentered_forces(mol_A, mol_B);
8              accumulate_forces(force_buffers_a_i,F);
9              accumulate_forces(mol_B,-F);
10         }
11     }
12     accumulate_forces(mol_A,force_buffers_a_i);
13 }
```

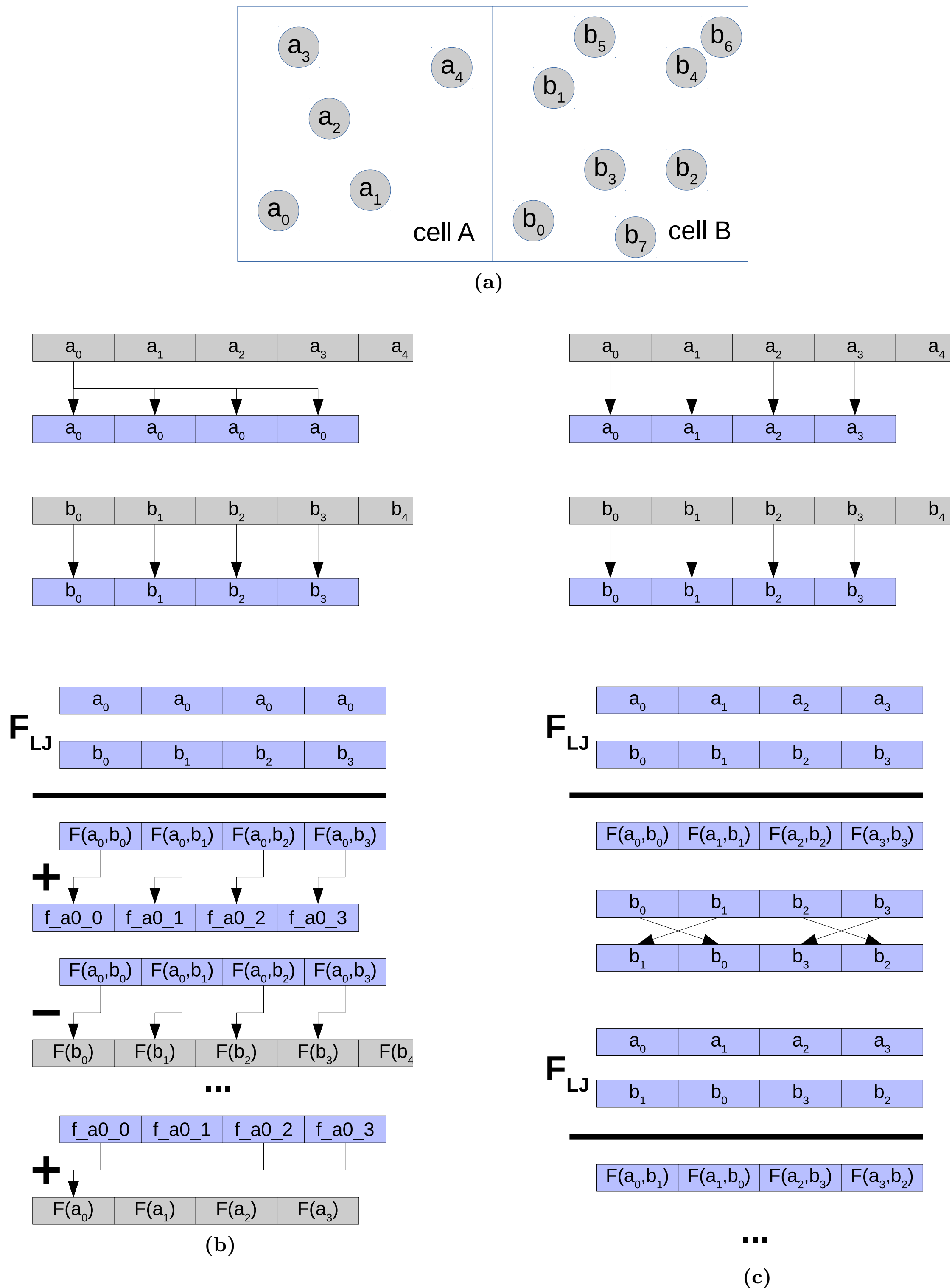


Figure 5.1: (a) Calculation of forces between two neighbouring cells A and B. (b) "Broadcast-reduce" approach for vectorizing the force calculation between cells A and B for vector length four. (c) "Permutation" approach for vectorizing the force calculation between cells A and B for vector length four.

5.1 Vectorization of nested loops

In the following descriptions, by molecule "data" we mean all attributes of a molecule, needed for computing the force on the molecules (in SoA format), as described in Chapter 4. This includes x-,y- and z components of its position, of its force and variables of its potential type, such as the Lennard-Jones ϵ and σ parameters or the Coulomb charge, dipole and quadrupole moments, etc. All variables are loaded in the same fashion, as indicated in the following paragraphs. In order to simplify the discussion, we will assume that the molecules have only one interaction site, for now.

Broadcast-reduce variant In Listing 5.1, it is possible to vectorize the inner loop (over molecules in the cell B, line 4), the outer loop (over molecules in the cell A, line 1) or both. The approach taken by `ls1 mardyn` is the first one [26]. It is illustrated in Figure 5.1(b). One molecule from cell A is loaded and its data is "broadcast" to all SIMD lanes (four lanes in Figure 5.1(b)). Let this molecule be a_0 and the broadcast SIMD register be $\begin{array}{|c|c|c|c|}\hline a_0 & a_0 & a_0 & a_0 \\ \hline\end{array}$. The loop over molecules in cell B is vectorized and a chunk of molecules' data corresponding to the SIMD lane size is loaded from memory — $\begin{array}{|c|c|c|c|}\hline b_0 & b_1 & b_2 & b_3 \\ \hline\end{array}$ in the example. The distances and forces between a_0 and the four molecules from B are computed together using vector operations, resulting in the register $\begin{array}{|c|c|c|c|}\hline F_{a_0,b_0} & F_{a_0,b_1} & F_{a_0,b_2} & F_{a_0,b_3} \\ \hline\end{array}$. A SIMD divergence arises if some, but not all, of the four molecules are inside the cutoff radius. Its handling is be clarified in Section 5.2. The SIMD vector of forces is added to a buffer register, reserved for storing the force contributions to a_0. We denote this buffer as $\begin{array}{|c|c|c|c|}\hline f_{a_0}_0 & f_{a_0}_1 & f_{a_0}_2 & f_{a_0}_3 \\ \hline\end{array}$. If *Newton3* is used, the forces to the molecules from B are accumulated in the respective memory locations (with negative sign). After this is done, the next chunk of molecules from B is loaded and their distances and forces to a_0 are computed. After molecule a_0 has been considered with all molecules from B, the force contributions to a_0 in $\begin{array}{|c|c|c|c|}\hline f_{a_0}_0 & f_{a_0}_1 & f_{a_0}_2 & f_{a_0}_3 \\ \hline\end{array}$ are "horizontally" added or "reduced" and added to the respective memory location. Then the next molecule from cell A is considered. We refer to this approach as "broadcast-reduce".

Permutation variant Figure 5.1(c) shows another alternative, which appears in other high performing codes, such as GROMACS [85]. In this variant, both loops are, in effect, vectorized. A chunk of molecules is loaded from cell A, $\begin{array}{|c|c|c|c|}\hline a_0 & a_1 & a_2 & a_3 \\ \hline\end{array}$ and cell B is traversed. For each chunk of molecules in B, however, e.g. $\begin{array}{|c|c|c|c|}\hline b_0 & b_1 & b_2 & b_3 \\ \hline\end{array}$, several interaction combinations are computed. After the calculation of the first pairs of forces, the forces $\begin{array}{|c|c|c|c|}\hline F_{a_0,b_0} & F_{a_1,b_1} & F_{a_2,b_2} & F_{a_3,b_3} \\ \hline\end{array}$ have been computed. Then the data inside the SIMD vector for the molecules from B is permuted. One such permutation can be performed, for example, via the `_mm256_permute_pd()` intrinsic function to achieve the ordering $\begin{array}{|c|c|c|c|}\hline b_1 & b_0 & b_3 & b_2 \\ \hline\end{array}$. Leaving the data from cell A unpermuted, the force contributions $\begin{array}{|c|c|c|c|}\hline F_{a_0,b_1} & F_{a_1,b_0} & F_{a_2,b_3} & F_{a_3,b_2} \\ \hline\end{array}$ can now be computed. In this fashion, n permutations are performed, where n is the number of SIMD lanes.

Compared to the broadcast-reduce approach, the permute approach has the advantage that it requires fewer load and store operations. However, it needs multiple operations for permuting all of the involved data, which limits the attained speed-ups. Moreover, wrapping these permutations in intrinsics code for differing vector lengths is not straightforward. The primary reason, however, why the broadcast-reduce approach was selected for `ls1 mardyn`, is that the handling of multicentered molecules, becomes very complex in the permute approach, see also Section 5.3. This is because molecules can begin or end across vector boundaries in either the first or the second SIMD register, which makes bookkeeping very tedious. Nevertheless, this is

a viable approach and will be considered for the autotuning library, described in Section 9.5.

5.2 SIMD divergence handling

As mentioned earlier, it frequently occurs that not all molecule pairs within a SIMD chunk should be computed. In the following we assume a broadcast-reduce approach was taken for the vectorization of the nested loops.

Masking The masking approach to handling the divergence proceeds as follows. Consider, for example, the computation of the forces between the SIMD chunks $\boxed{a_4 \mid a_4 \mid a_4 \mid a_4}$ and $\boxed{b_0 \mid b_1 \mid b_2 \mid b_3}$. Assume that only molecules b_1 and b_3 lie within the cutoff sphere, centered at a_4. In the masking approach, from the SIMD register, storing the distances between the molecules $\boxed{r_{a_4b_0} \mid r_{a_4b_1} \mid r_{a_4b_2} \mid r_{a_4b_3}}$, a mask is generated, by performing entrywise $\leq$ comparisons to a register holding the cutoff radius $\boxed{r_c \mid r_c \mid r_c \mid r_c}$. This results in $\boxed{0 \mid 1 \mid 0 \mid 1}$ in our example. After that, if at least one entry in the mask is non-zero, the whole chunk is computed, $\boxed{F_{a_4b_0} \mid F_{a_4b_1} \mid F_{a_4b_2} \mid F_{a_4b_3}}$ and multiplied by the mask $\boxed{0 \mid 1 \mid 0 \mid 1}$. As the resulting forces are to be added or subtracted to the respective memory buffers, no unwanted force contributions are, thus, accumulated.

Gather-scatter The gather-scatter approach proceeds in two phases. First, the distances to all molecules and the respective masks are computed, as in the masking approach. Assume that for the molecule a_4 they result in the mask $\boxed{0 \mid 1 \mid 0 \mid 1}$ for the chunk of molecules $\boxed{b_0 \mid b_1 \mid b_2 \mid b_3}$. Then, a list of indices is created, with the entries $\boxed{0 \mid 1 \mid 2 \mid 3}$ and the entries in which the mask has a value of 1 are "compressed" in a new SIMD vector, populated with zeroes at the end. The result of this is $\boxed{1 \mid 3 \mid 0 \mid 0}$ and gets stored to a memory buffer. Then, the distances to the next chunk of molecules is computed. Assuming that it results in the mask $\boxed{0 \mid 1 \mid 1 \mid 0}$ for the molecules $\boxed{b_4 \mid b_5 \mid b_6 \mid b_7}$, the indices $\boxed{4 \mid 5 \mid 6 \mid 7}$ get compressed into $\boxed{5 \mid 6 \mid 0 \mid 0}$. The index vector is, then, appended to the memory buffer, resulting in $\boxed{1 \mid 3 \mid 5 \mid 6}$. After all indices of interaction partners have been computed, a second `for`-loop is carried out, with the molecular data being collected via `gather` operations with the index vector $\boxed{1 \mid 3 \mid 5 \mid 6}$. A `mask_gather` instruction is used to handle potentially unpopulated entries at the end of the index vector. This results in compact SIMD register evaluations of the force kernels, resulting directly in $\boxed{F_{a_4b_1} \mid F_{a_4b_3} \mid F_{a_4b_5} \mid F_{a_4b_6}}$. If *Newton3* is used, the force contributions also need to be sent back to the force buffers of the molecules from B. This is done via `scatter` and `mask_scatter` instructions with the same index vector. In this way, only the kernel evaluations, which are really necessary, are computed, saving most of the unnecessary contributions, which, otherwise, would have to be masked out.

While this approach potentially spares many operations, the scatter option is required if *Newton3* is to be used, which is a goal of our work. The scatter option is only available since AVX512 and KNC. Hence, the gather-scatter approach has not been widely adopted by the user community of `ls1 mardyn`, since the users run mostly on AVX2 clusters as of writing this document. Moreover, alternative CPU vendors still do not support AVX512, even in very recent architectures, such as AMD's Rome CPU, implementing the Zen 2 michroarchitecture[1]. For this reason, while the gather-scatter variant is included in the production trunk of `ls1 mardyn`, it has been fully optimized only in a branch, specifically targeting Intel Xeon Phi [109]. Excerpts of the gather-scatter results from [109] are presented in Chapter 6.

[1] `https://en.wikichip.org/wiki/amd/cores/rome`

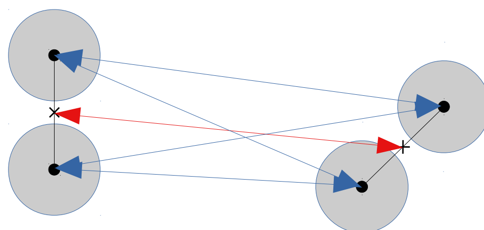


Figure 5.2: Force calculation between multicentered molecules. The cutoff-condition is evaluated on a center-of-mass basis (red arrow), but force contributions need to be evaluated on a site-basis (blue arrows).

5.3 Vectorization for multicentered molecules

In `ls1 mardyn`, the cutoff condition is evaluated on a center-of-mass basis, see Figure 5.2. This is done to prevent molecules from interacting only with some, but not all sites of another molecule. To see why this can cause problems, consider a molecule, which features several charge sites, but is overall electroneutral. In that case, if another molecule interacts only with some of the charges, it will be subjected to much stronger forces than if it would interact with the whole molecule, as the contributions from the whole — electroneutral — molecule would mostly cancel each other out. For this reason, the cutoff condition is evaluated between entire molecules.

In the following we discuss three possible implementations of this calculation and their vectorization potential. For the following discussion, let N_A and S_A be the total number of molecules and total number of sites, respectively, in cell A ($N_A \leq S_A$). Similarly, let N_B and S_B be the number of molecules and sites, respectively, for cell B. Moreover, note that whenever we discuss the innermost loops between sites from the first molecule and sites from the second cell, multiple such loops are necessary for the different kernels. In other words, there is one such loop for LJ interactions, one such loop for charge-charge interactions, charge-dipole interactions and so on. These loops, however, can be considered as replications of the illustrated loop.

Center-of-mass to center-of-mass approach This is the most straightforward approach, considering the center-of-mass handling of the cutoff condition. In `ls1 mardyn`, it is used in the legacy AoS implementation. Listing 5.2 illustrates this approach.

Listing 5.2: CoM-CoM Variant

```
1  for mol_A in molecules_cell_A {
2      for mol_B in molecules_cell_B {
3          if(distance(mol_A,mol_B) <= cutoff_Radius) {
4              for site_a in mol_A
5                  for site_b in mol_B
6                      compute_Force(site_a, site_b);
7                      ...
8          }
9      }
10 }
```

In this variant, the distances between the centers of mass of the two molecules are calculated, resulting in only $N_A \cdot N_B$ cutoff checks. Distances between molecular sites are computed only if the molecule falls within the cutoff sphere. The complex control logic arising from this approach is, however, a hurdle for vectorization, as two loops over the number of sites of a molecule are

needed (lines 4 and 5 in Listing 5.2). The trip count of those loops can vary from molecule to molecule and is usually low — between one and four most of the time, but reaching as high as 12 in some use cases of `ls1 mardyn`. Such short, nested loops are bad candidates for vectorization. Unrolling the loops would be preferable, but the fact that they can vary in length from one molecule to the next renders unrolling also difficult. For this reason, other approaches were needed for `ls1 mardyn`.

Site to site approach In [55] another approach was developed, in which one works directly on the molecule centers in SoA format. Listing 5.3 illustrates it. In this approach, one iterates directly over the molecular sites in the cells so that the loop in line 2 can be vectorized. The cutoff condition is still evaluated on a center-of-mass basis by replicating the position of the centers of mass for each molecular site. Thus, for molecules a and b, the same center-of-mass to center-of-mass check is performed (redundantly) $M_a \cdot M_b$ times instead of only once, M_a and M_b denoting the number of sites in a and b. This approach results in $S_A \cdot S_B$ cutoff condition checks and the benefits of using vectorization were, in practice, outweighed by the extra calculations [55]. Yet another approach was needed.

Listing 5.3: Site-Site Variant

```
1  for site_a in sites_cell_A {
2      for site_b in sites_cell_B {
3          if(distance(mol_A,mol_B) <= cutoff_Radius) {
4              compute_Force(site_a, site_b);
5              ...
6          }
7      }
8  }
```

Center-of-mass to site approach As an intermediate solution, a center-of-mass to site implementation was developed [28]. This is the current implementation in all vectorized branches of `ls1 mardyn` and its production trunk. Listing 5.4 illustrates this version.

Listing 5.4: CoM-Site Variant

```
1  for mol_A in molecules_cell_A {
2      for site_b in sites_cell_B {
3          force_mask[site_b] = compute_force_mask(mol_A, site_b);
4      }
5      for site_a in mol_A {
6          for site_b in sites_cell_B {
7              if(force_mask[site_b]) {
8                  compute_Force(site_a, site_b);
9                  ...
10             }
11         }
12     }
13 }
```

The cutoff condition is evaluated between molecules from the first cell and sites from the second cell. Again, it is evaluated on a center-of-mass basis by replicating the position of the centers of mass for each molecular site. This time, however, for molecules a and b, the same center-of-mass to center-of-mass check is performed (redundantly) $1 \cdot M_b$ times, M_b denoting the number of sites in b. This happens in a separate `for`-loop, carried out prior to the calculation of force contributions, in a vectorized fashion (line 2 in Listing 5.4). The force-masks resulting from this loop are stored for subsequent use in line 7 and reused for all sites of the current

molecule. Overall, $N_A \cdot S_B$ checks of the cutoff condition are performed in this approach which is a considerable reduction of the computational overhead. As we shall see in Section 5.5 this renders the vectorization profitable.

This is the current implementation in `ls1 mardyn` underlying all (SoA-based) vectorization modes. The difference between the $N_A \cdot S_B$ evaluations of this variant and the $N_A \cdot N_B$ evaluations of the AoS variant is one of the sources of the different number of operations mentioned in Section 4.1.5.

Outlook: vectorized center-of-mass to center-of-mass approach Before we proceed, we discuss another possibility, for computing the force masks. Ideally, in line 2 in Listing 5.4, one would want to compute the force mask on a center-of-mass to center-of-mass basis. Then, the force mask would be extended by the respective — variable — number of sites of each molecule. For example, if the force mask on a CoM basis is [0|1|0|1] and the number of sites of the corresponding molecules are [2|3|1|4], then the resulting site force mask would be [0|0|1|1|1|0|1|1|1|1]. Listing 5.5 illustrates such an approach. The challenge in this approach is to perform the `extend_mask_from_CoM_to_sites(...)` function efficiently. If unvectorized, the presence of very short `for`-loops of variable length is possible to outweigh the performance gains from a reduced number of cutoff-comparison operations. It is not obvious how this can be done in a vectorized fashion. One possibility would be a `gather` operation. Gather operations are, however, intended for “truly sparse” data accesses [64]. Since this is not the case here, this may lead to unsatisfactory performance. The `permutevar` intrinsic function up to AVX2, or the `permutexvar` from AVX512 onwards, could present a performant alternative.

In this implementation, the number of cutoff-evaluations is reduced down to $N_A \cdot N_B$. This change, however, would also decrease the trip count of the loop in line 2, which may again affect overall performance negatively.

Listing 5.5: Vectorized CoM-CoM Variant

```
1  for mol_A in molecules_cell_A {
2      for mol_B in molecules_cell_B {
3          force_mask_mol[mol_B] = compute_force_mask(mol_A, mol_B);
4      }
5      force_mask = extend_mask_from_CoM_to_sites(force_mask_mol);
6      for site_a in mol_A {
7          for site_b in sites_cell_B {
8              if(force_mask[site_b]) {
9                  compute_Force(site_a, site_b);
10                 ...
11             }
12         }
13     }
14 }
```

5.4 Intrinsics wrappers

When programming with intrinsic functions for different vector lengths and different data types (e.g. `float` or `double`), different functions need to be invoked. For example `_mm_mul_ps()` needs to be called for multiplication of four `float` elements, while `_mm512_mul_pd()` is needed for multiplication of eight `double` elements. This means that porting scalar code to 128-bit arithmetic or to 256-bit arithmetic requires writing separate code. Moreover, changing the precision from e.g. `float` to e.g. `double` can no longer be done in the traditional way of using a

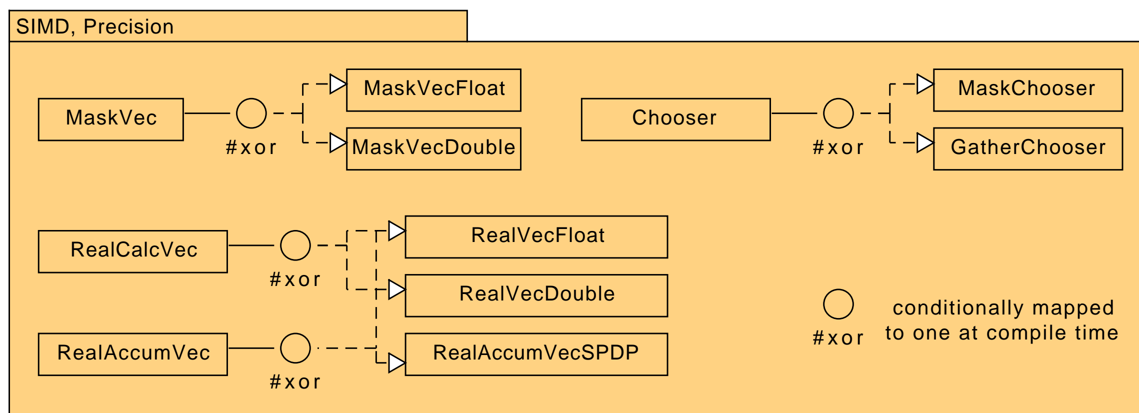


Figure 5.3: Class diagram of the classes, which are part of the intrinsics wrappers.

`typedef`. Thus, if the developer wants to have support for multiple instruction sets or wants to be able to change the vectorization type or length, he or she would need to maintain multiple versions of the code with small changes between them. For this reason people either avoid intrinsics altogether by using alternative means of vectorization or develop intrinsics "wrappers".

A significant effort had been invested in developing intrinsics versions of the force kernels for `ls1 mardyn`, cf. [26–28,53,55]. These were already well performing and adopted by the user community of `ls1 mardyn`, prior to the start of this project. The need for higher portability and maintainability, however, demanded the introduction of wrappers, which was done as part of this work. While other libraries for wrapping SIMD intrinsics exist (e.g. [36,66]), it was found to be easiest to wrap the already existing code in wrappers arising from our particular use cases and demands.

Figure 5.3 shows the major classes, which constitute the SIMD wrappers. The code for floating-point operations resides in a class `RealVec<T>` with specializations `RealVecFloat` and `RealVecDouble`. Which one is used for the calculation depends on the choice of precision, as discussed in Section 4.2. Listing 5.6 shows an excerpt from the class `RealVecDouble`. Depending on the compilation mode, the class has a data field of one of the types `double`, `__m128d`, `__m256d` or `__m512d`. The vector functionality is provided through member functions and class operators, to enhance code readability. Care is taken that all functions are always inlined. Implementing the code for different vector widths in this way — one file with conditional compilation at every function inside — keeps related code as close as possible. The other option — separating the code into further files, as done in e.g. [36] — has the disadvantage that multiple files need to be compared to detect bugs or performance loss between the different vector lengths.

The classes `RealVecFloat`, `MaskVecFloat` and `MaskVecDouble` are implemented in a similar fashion. The `MaskVec` classes wrap around either masks represented by integer data types (prior to AVX512) or the dedicated types introduced in AVX512, such as `__mmask8` and `__mmask16`. The choice of the handling of the SIMD divergence via either masking or gather-scatter is implemented through the `Chooser` class, which maps either to the `MaskChooser` or `GatherChooser` classes. In the masking case, a mask is loaded from memory for the current chunk of data to mask any elements, which should not be computed. In the gather-scatter case, the indices of molecules in the interaction region are loaded. The resulting code is relatively easy to read and maintain and is nearly free of code duplication. Appendix A shows the wrapped Lennard-Jones kernel as a more comprehensive example.

Listing 5.6: RealVecDouble class

```
template<>
class RealVec<double> {
protected:
        #if     VCP_VEC_WIDTH == VCP_VEC_W__64
                typedef double real_vec;
        #elif VCP_VEC_WIDTH == VCP_VEC_W_128
                typedef __m128d real_vec;
        #elif VCP_VEC_WIDTH == VCP_VEC_W_256
                typedef __m256d real_vec;
        #elif VCP_VEC_WIDTH == VCP_VEC_W_512
                typedef __m512d real_vec;
        #endif

protected:
        real_vec _d;
        ...

public:
        inline __attribute__((always_inline))
        RealVec operator * (const RealVec& rhs) const {
        #if     VCP_VEC_WIDTH == VCP_VEC_W__64
                return _d * rhs;
        #elif VCP_VEC_WIDTH == VCP_VEC_W_128
                return _mm_mul_pd(_d, rhs);
        #elif VCP_VEC_WIDTH == VCP_VEC_W_256
                return _mm256_mul_pd(_d, rhs);
        #elif VCP_VEC_WIDTH == VCP_VEC_W_512
                return _mm512_mul_pd(_d, rhs);
        #endif
        }
        ...
};
```

Special functionality available from AVX2 onwards Before proceeding, we point out that our AVX2 and AVX512 version wrappers make use of two additional features of these intrinsic instruction sets. The first one is the use of the lower-precision functions `_mm256_rcp_ps` or `_mm256_rsqrt_ps` for the calculation of, respectively, $\frac{1}{x}$ and $\frac{1}{\sqrt{x}}$. The use of these low-precision variants mandates two Newton-Raphson iterations, but it was found that if fused-multiply-add instructions are available, this can sometimes be beneficial [78]. The second feature is the use of `gather` instructions available from AVX2 onwards. These instructions are used for setting up look-up tables for the ε_{ij} and σ_{ij} parameters, to be used with the Lorentz-Berthelot mixing rules (cf. Section 1.4). The use of two gather instructions there spares up to 18 other instructions for loading the appropriate variables for vector lengths of eight. In some cases it was observed that this could lead to a drop in performance, however. Hence, a variant, which makes use of the aforementioned `permutevar` instruction might be considered in the future. Again, this is another point, which could be tuned automatically if deemed necessary, see Section 9.5.

5.5 Results

5.5.1 Kernel Studies

In this section we present studies of the kernels and vectorization modes, carried out in [32] as part of this project. The studies were performed on a Intel SandyBridge-EP Xeon E5-2670

machine in double precision using AVX intrinsics. All measurements are performed in the `Normal` mode with the masking approach. Though partially outdated, these measurements still present interesting insights.

The goal of these measurements is to obtain information on the performance of the kernels as a function of the number of molecules in a cell. In order to reduce the otherwise large parameter space, a simplified setup was created. Two cells were initialized with molecules. The cells are constructed in such a way, that the cutoff condition is evaluated as would be evaluated in production scenarios, but it is always fulfilled. In other words, the distance between any two points in the cells is smaller than the cutoff radius. This removes the inherent variability in the performance, due to the strong dependence of the hit-rate on the cell configuration, cf. Section 1.6.2. The number of molecules in the cells is then varied, while the performance of the force calculation was measured. In this section, the internal FLOP counters of `ls1 mardyn` are used, as described in Section 2.4.2.

The results are shown in Figure 5.4. Figures 5.4(a) and 5.4(b) show the performance of the different vectorization modes for 1CLJ molecules. It can be observed, that the performance of the AoS and SoA variants is quickly saturated. The AoS performance is saturated at around 0.54 GFLOP/sec, while the SoA one — at 1.42 GFLOP/sec.

The SSE and AVX curves exhibit a more interesting behaviour. A saw-like behaviour is observed in the SSE performance, as odd numbers exhibit a lower performance than even numbers. This can be easily attributed to the vector length of two: for odd numbers, the loop has a remainder, which is executed less efficiently. Similarly, the AVX curve has maxima at multiples of four and performance drops slightly in between. The SSE curve saturates at a performance of about 3.33 GFLOP/sec. Since the `for`-loops in the AVX implementation have a two times smaller trip count than in the SSE one, they take longer to saturate. Ultimately, they saturate at around 4.54 GFLOP/sec in Figure 5.4(b). Overall, the AVX performance is at about 21% of the theoretical peak performance of the SandyBridge core, which is in agreement with previous analyses [56]. To see why a value of 21 % is appropriate, consider the following. As the kernel performs more multiplication operations, only one of the two SandyBridge units is stressed, which lowers the attainable theoretical performance to 50%. The presence of a costly division operation in the LJ kernel further lowers that value to around 20–25% [56].

In the idealized measurements of Figure 5.4(a), we observe speed-ups of around 2.2× of the SSE implementation over the SoA one at 32 molecules per cell. The value is slightly higher than the theoretical value of 2×. This could be attributed to the fact, that — apart from vectorizing with length two — the SSE version uses the higher instruction set SSE3 (enabling SSE3 automatically switches intrinsics vectorization on, so running the SoA mode with the SSE3 instruction set is not possible). For the AVX variant, the value is 2.8× at 32 molecules per cell and is reasonable to rise to 3.15× at around 256 molecules per cell. An explanation why a perfect value of 4× is not attained could be that the floating-point division of four double precision elements (`_mm256_div_pd`) has an almost two times higher latency than the division of two elements (`_mm_div_pd`) [19]. Other reasons could be the limitations on vectorization speed-ups discussed in Section 2.2.2.

Figure 5.4(c) plots the performance of the AVX variant for one-, two- and four Lennard Jones sites per molecule. As the innermost loop is over the number of sites, not molecules, it has larger trip counts for multicentered molecules, which leads to a quicker saturation of the performance. The “roof”, however, remains fairly constant, as it depends on the particular mix of instructions in the kernel. It can be observed that the saw behaviour changes as well: for 2CLJ, the period becomes two, down from four for 1CLJ. This is again because the innermost loop, whose remainder determines this period, is over the number of sites, not molecules. For two- and four-centered molecules, this number becomes a multiple of two and four, respectively,

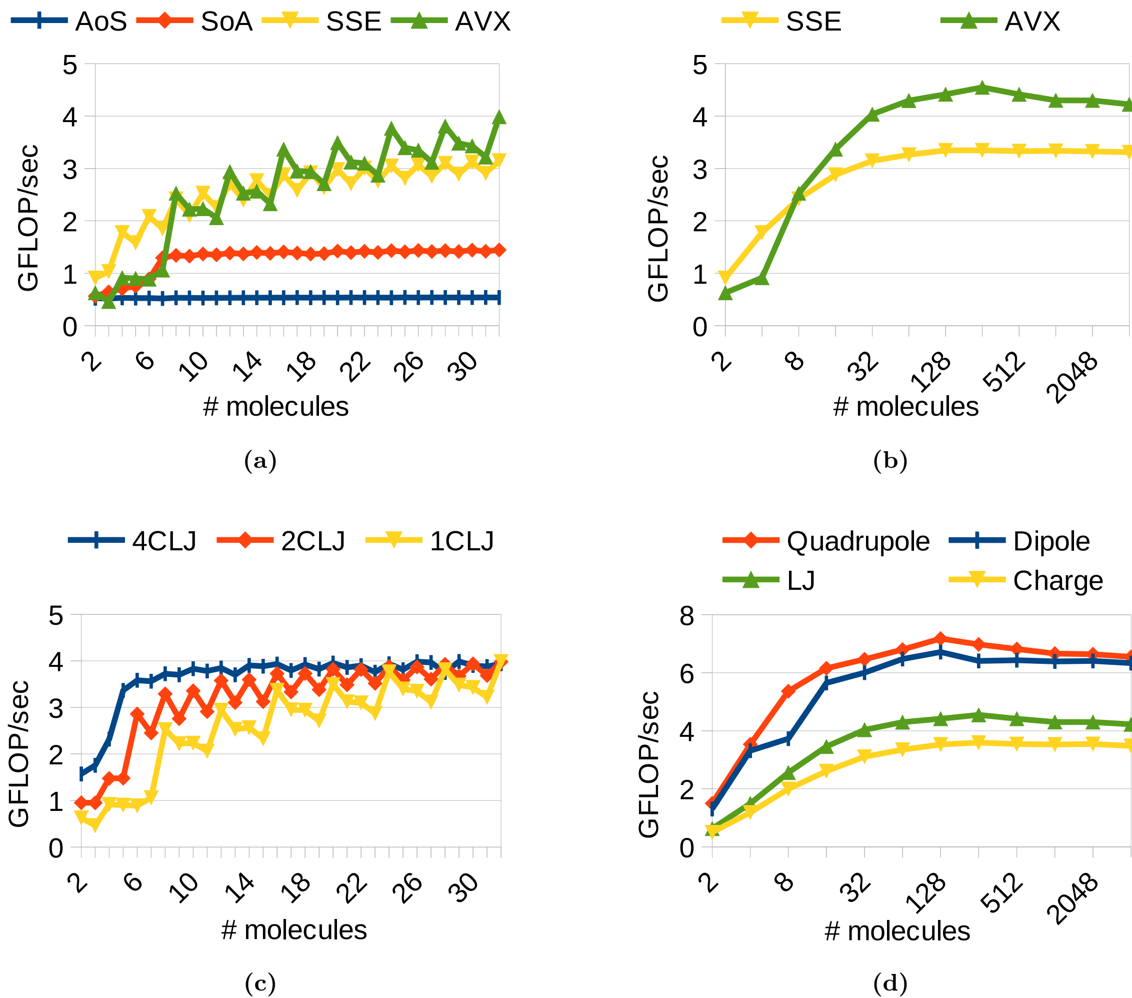


Figure 5.4: Performance study of cell-pair force calculations as a function of the number of molecules in a cell. The cutoff condition always evaluates to true. (a) Vectorization modes for 1CLJ molecules. (b) Vectorization modes for 1CLJ molecules, logarithmic scale. (c) AVX performance for different numbers of LJ sites per molecule. (d) Different potentials' AVX performance, logarithmic scale.

thus, explaining the exhibited behaviour.

In Figure 5.4(d), the performance of the different potentials is plotted. The charge potential has a slightly lower GFLOP/sec performance: it performs fewer FLOP operations for a similar number of data elements to be loaded, leading to a lower computational intensity. It is not multiplication dominated, but features a square root operation, apart from the division operation, which is also a high-latency operation.

The dipole and quadrupole potentials reach considerably higher values in their performance — up to 6.4 and 7.2 GFLOP/sec, respectively. This is because much more FLOP operations are performed, which are also balanced better in terms of the multiplications and additions. While still featuring division and square root operations, these potentials attain up to 34% of the theoretical peak performance of the core.

Some light peaks of performance are observed at 256 molecules for the Lennard-Jones and charge potentials and at 128 for the dipole and quadrupole potentials. This is likely due to data no longer fitting in the L1 data cache of the CPU. This fact also explains why the peak is reached earlier for the dipole and quadrupole potentials: they require more variables to be loaded. After that peak, however, the performance drops by less than 9%, which we consider tolerable.

5.5.2 Single Center Lennard Jones molecules

In this section we continue the study of 1CLJ molecules from Section 4.3. The simulations are again performed on a node of the Hazel Hen supercomputer running an Intel®Xeon®CPU E5-2680 v3. As we saw in the analysis of Figure 4.8, the extreme cases are $r_c = 2.5$, $\rho = 0.39$ and $r_c = 5.0$, $\rho = 0.78$. For this reason, we will analyze only those two cases, as all other cases are somewhere in between. Figure 5.5 shows the obtained results for the different vectorization, precision and memory modes for those two cases.

In Figure 5.5(a) it can be observed that the vectorization modes have little to no influence on performance. Indeed — as already discussed in Section 4.3 — the limiting factor is not the time spent within a cell, but rather the time spent traversing the cells. As the vectorization decreases the time spent within a cell, but does not improve the traversal speed, we do not see notable increases in overall performance.

In Figure 5.5(b), on the other hand, we see a much greater influence of the vectorization modes. The speed-ups are more pronounced for the `RMM` mode than for the `Normal` one. Recall the analysis of Section 4.3 — as the system parameters made the calculation more compute intensive (high values of ρ and r_c), the `Normal` mode caught up in performance to the `RMM` mode, as they both reached the scalar compute-bound roof. However, the `RMM` calculation was proceeding at a much higher arithmetic intensity than the `Normal` one, cf. Figure 4.9. When moving from scalar to vectorized code, the roof becomes higher, meaning that the arithmetic intensity value, at which the roof is attained, also becomes higher. For this reason, the vectorization speed-ups are more pronounced for the `RMM` mode.

Still, the speed-up values appear rather low, compared to the theoretical expectations. In the `Normal` mode, the single precision speed-up is 1.97× (out of 8×), while the double precision one — 1.84× (out of 4×). For the `RMM` mode, the values are 4.27× (out of 8×) and 2.52×, respectively (out of 4×). Investigation through Intel Advisor, however, indicates that this is an Amdahl's law-type limitation. In our setup, comparing SSE or AVX to SoA can be considered as parallelization of the innermost loops of the force calculation. If, through this parallelization, high or even perfect speed-ups are attained, the "sequential" part (the entire code outside of those loops) poses a limit on the total speed-up, which can be observed.

In an attempt to investigate further, the SoA and AVX2 runs for $\rho = 0.78$, $r_c = 5$ in

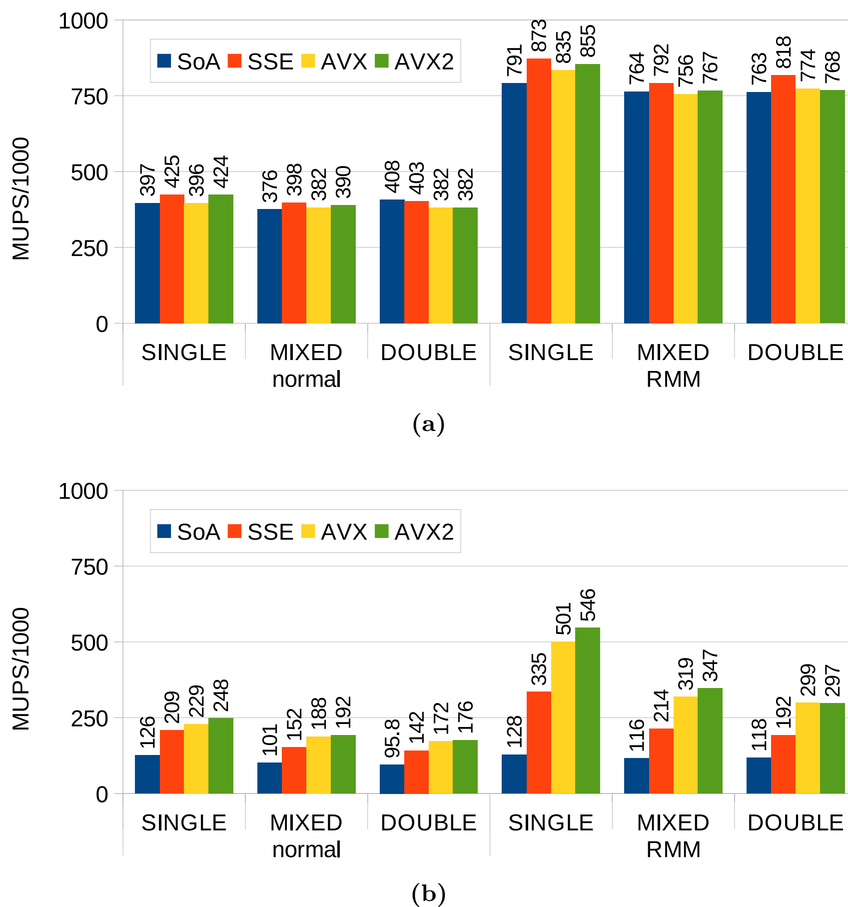


Figure 5.5: Vectorization performance for 1CLJ systems, in the different precision and memory modes. System parameters given in Table 4.3. (a) $\rho = 0.39$, $r_c = 2.5$. (b) $\rho = 0.78$, $r_c = 5.0$.

`Normal` and `RMM` mode were analyzed in single precision. The analyses were performed on a Intel Core i7-4770 CPU @ 3.40GHz desktop Haswell machine. The measurements are, thus, not one-to-one comparable for several reasons. First, the CPU is a different model with different frequency. Second, the program needs to be compiled with debugging information, in order for Advisor to be able to analyze it. Finally, the system size was reduced, in order to speed-up the (long) profiling time and reduce the amount of sampling data collected. Nevertheless, we obtain representative insights on the observed behaviour.

Considering the most time-consuming loop of the force calculation, Advisor reports a value of up to 2.39 GFLOP/sec for SoA and 12.66 GFLOP/sec for AVX2. The total time, however, is 89.81 seconds and 55.02 seconds, respectively. This gives a 5.30× increase in performance in that loop, much higher than the overall $\frac{89.81}{55.02} = 1.63\times$ observed speed-up. Similarly, in the `RMM` mode, the SoA variant reports 3.67 GFLOP/sec, while the AVX2 variant reports 26.34 GFLOP/sec and runtimes of 88.12 seconds and 24.15 seconds total runtime, respectively. This gives a 7.18× increase in performance of the vectorized loop, out of theoretical possible 8×. Overall, however, the speed-up value drops to $\frac{88.12}{24.15} = 3.65\times$. Looking in more detail, Advisor reports 7.67 seconds of scalar code and 16.48 seconds in the vectorized loops. Plugging in 7.67 seconds of scalar time, 88.12 seconds of total time and vector length of eight in Amdahl's law gives a theoretical limit of 4.97×. Considering this limit, we regard the achieved 3.65× in the Advisor measurements and the 4.26× in the production measurements as excellent results.

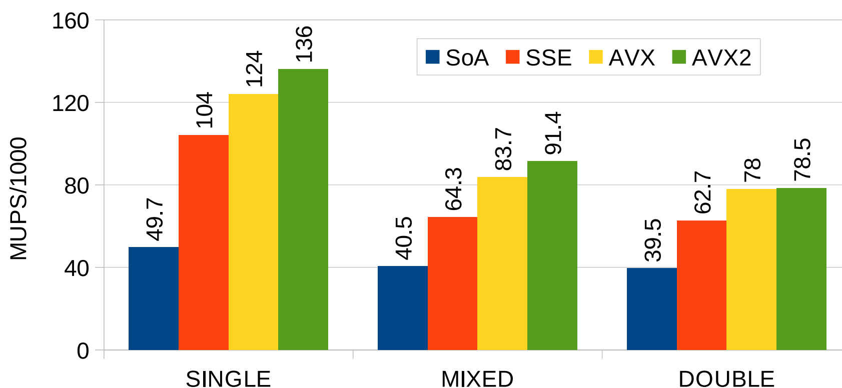


Figure 5.6: Performance comparison of the different vectorization modes for TIP4P system from Table 4.1

Comparing the double precision AVX speed-ups to the values of Section 5.5.1, the values are, of course, smaller. Here we observe speed-ups up to 1.80× in the **`Normal`** mode and 2.53× in the **`RMM`** mode, whereas we observed up to 3.15× in Figure 5.4(b). Considering that the measurements in Figure 5.4(b) are isolated for the force calculation and the cutoff condition is always fulfilled, we consider the values observed here to be appropriate.

When comparing the different precision modes, the following observations can be made. For the **`RMM`** mode with vectorization, the SPSP mode is 1.68×–1.84× faster than the DPDP mode, which is close to the theoretical value of 2×. For the SPDP mode the values are, expectedly, lower: 1.07×–1.17×, corroborating that the other overhead of the mixed precision implementation is indeed significant. In the **`Normal`** mode, the speed-ups are lower, for the same argument as for the vectorization speed-ups. The values are 1.33×–1.47× for SPSP and 1.07×–1.09× for SPDP.

Finally, comparing **`RMM`** and **`Normal`** mode, the **`RMM`** mode is about 2× faster for the low density scenario, as this was already the value of the baseline SoA performance. For the high density scenario with AVX2, **`RMM`** is now 1.7×–2.2× faster due to more pronounced vectorization gains.

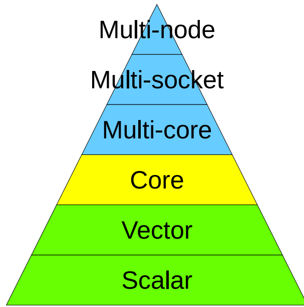
5.5.3 Multicentered molecules

Figure 5.6 shows the results for the TIP4P system from Table 4.1. The simulation was performed again on a node of the Hazel Hen supercomputer running an Intel®Xeon®CPU E5-2680 v3. The masking approach is being used with AVX2. The results are for the **`Normal`** mode, as an **`RMM`** mode is not available for multicentered molecules.

The observed speed-ups due to vectorization are 2.74× for SPSP, 2.26× for SPDP and 1.99× for DPDP. Compared to the theoretical 8× for SPSP and 4× for DPDP, these values seem quite modest. However, as we shall see in Section 9.2, the overall attained performance is similar to that of established, high-performing codes like **LAMMPS** and **GROMACS**. Similarly to the 1CLJ results in the previous section, it is reasonable to expect that the speed-ups can further improve for larger cutoff radii or density of the simulated system. Similar speed-ups were also observed for the CO2 or EOX systems from Table 4.1 (not shown). Comparing the precision modes, SPSP is 1.73× faster than DPDP and SPDP is 1.16× faster than DPDP. These improvements are higher than the **`Normal`** mode in the 1CLJ cases and reach the **`RMM`** values.

5.6 Conclusion

In conclusion, the gains through the SIMD vectorization depend strongly on the simulation parameters. While for low cutoff radii or low densities the values can be disappointing, total speed-ups of up to 4.27× of the full simulation are observed, which is a considerable improvement. These values are also in agreement with the previous results of [27], as will be shown in Section 9.3. For low densities and cutoff radii, perhaps other methods, such as Verlet Lists, are more appropriate than Linked Cells, as will also be discussed in Section 9.2. Considering precision, SPSP was demonstrated to perform up to 1.84× faster than DPDP, out of theoretically possible 2×. This is a great improvement and should be used whenever the application requirements allow it. The SPDP variant outperforms the DPDP variant only by 1.17×. Unless it can be further improved, it is questionable whether it is worth using it over DPDP. Regarding the `RMM` mode, it was demonstrated that, after vectorization, it always runs about 2× faster than the `Normal` mode and, hence, should also be used whenever possible.



6 Core-Level Performance

As introduced in Section 2.2.3, on the core-level of performance analysis we concern ourselves almost exclusively with hyperthreading. Hyperthreading exposes two (or more) logical cores to the operating system for each physical core. Although some hardware units are doubled, almost all of them are shared and used competitively by the threads. The potential benefits of hyperthreading are, thus, mostly due to a better utilisation of the existing units.

Figure 6.1 shows a schematic of the different execution ports of the Intel Haswell microarchitecture. Eight ports are available, which are responsible for the execution of various operations [37]. Floating-point operations, for example, are executed on ports 0 and 1, while load and store operations are executed on ports 2, 3 and 7. Integer operations can be performed on ports 0, 1, 5 and 6. Hyperthreading can, thus, help improve parallelism on the level of instructions. The second thread can co-schedule instructions on idling ports or overlap pipelined operations, increasing overall throughput, subject to certain restrictions. Up to four instructions can be retired per cycle, setting also upper bounds on the gains. Examples for better utilisation of the resources include one thread executing floating-point arithmetic, while the other one executes integer arithmetic or is waiting on data to be fetched from memory, disk storage or the network.

How much an application gains from utilising the hyperthreading functionality of a core depends strongly on the program at hand. There are many cases in which hyperthreading can lead to performance degradation and for this reason many sources advise caution when using it and even to consider disabling it [37]. For this reason, in this section we perform tests to determine if and how much our program can gain from this hardware feature.

Regarding `ls1 mardyn` in particular, there are several sources of gains from hyperthreading. Mitigating latency due to fetching data from memory or even from cache is one possibility. Other possibilities become clear after examining the particular mix of instructions. As mentioned earlier, the Lennard-Jones kernel itself features many, chained multiplications. The

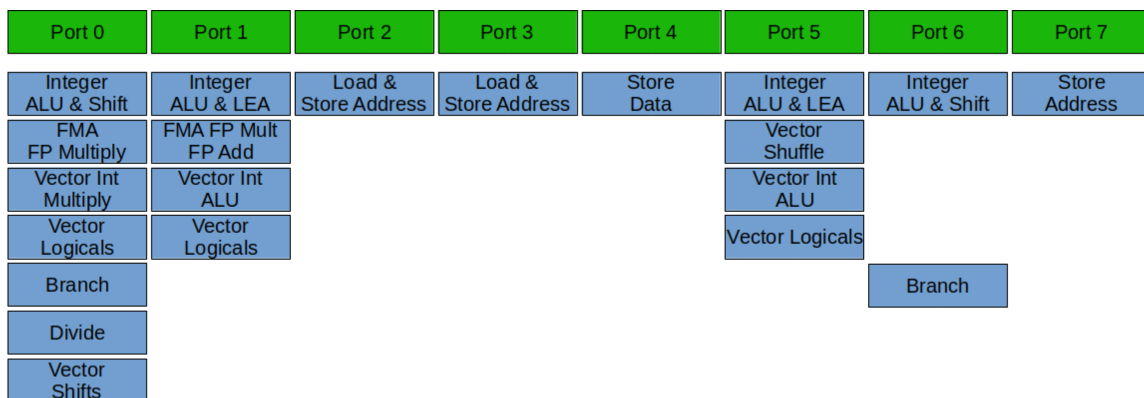


Figure 6.1: Diagram of execution ports on the Intel Haswell microarchitecture.

second hyperthread could, hence, coschedule additions for force accumulation or even independent multiplications, which help to fill the multiplication pipeline. Last, but not least, the cutoff condition leads to branching, which can also cause stalls due to misprediction. Thus, through hyperthreading we expect gains from mitigated memory access stall-time, co-scheduling of operations on different ports, improved pipeline utilisation or reduced stalling due to branch mispredictions.

In this section we present experiments on the gains through hyperthreading, by pinning two OpenMP threads or MPI processes on one Xeon core and pinning four threads on one Xeon Phi core. This allows us to set expectations for gains due to hyperthreading in later chapters. The results were collected with the developed OpenMP schemes, present in `ls1 mardyn`.

This is a good means to evaluate the gains due to hyperthreading, because the parallel overhead of both the parallelization schemes and the execution environment is low. The parallelization schemes themselves will be discussed at great length in Chapter 7 when we talk about multi-core scalability. As we shall see in that chapter, the schemes used here scale up to hundreds of threads with over 90% parallel efficiency, hence their parallel overhead when running on two or four threads, is very small. In the current execution environment, i.e. when running multiple threads on a single core, most of the parallel overhead is negligible. To see that, consider the following:

- synchronization among threads on a single core is cheap: it can be realized in the L1 cache,
- data transfer is also very cheap, as both hyperthreads have common access to all data storage locations, hence, they can just write to a common location, e.g. in the cache,
- if there are any load imbalances, their negative impact on runtime will be mitigated by the fact that the threads essentially compete for the hardware resources.

To elaborate on the last point, threads with a lower workload would finish computing earlier, leaving more hardware resources for the threads with a higher workload. This, in turn, speeds-up the rate at which the threads with a higher load perform their calculation. For example, assume that when using one thread, the calculation proceeds at 1.0 MMUP/sec and when using two threads, it proceeds at 1.2 MMUP/sec. In the latter case, each thread is essentially running at 0.6 MMUP/sec, as long as they are computing simultaneously. If one thread finishes earlier, the other thread speeds-up from 0.6 MMUP/sec to 1.0 MMUP/sec and completes its workload faster, reducing the negative impact of the load imbalance.

6.1 Force calculation study on KNC

In [109] we performed in-depth studies of core performance. This was done in order to leverage the performance of the Knights Corner (KNC) architecture, which features hardware support for up to four threads per core. KNC is an in-order architecture, which means it is even more dependent on hyperthreading for improving port utilisation. Moreover, no back-filling was available, which meant that a single thread could issue instructions only every second cycle [63]. Thus, in order to leverage its full performance, one needed to run with at least two threads per core. For these reasons, hyperthreading played a crucial role on KNC. In [109], also a more advanced version of the gather-scatter implementation was developed. We now present it briefly, as it was used in the collection of the results.

Listing 6.1: Gather-scatter kernel schematic

```
1  Indexvector iV;  // indices of interaction-partner sites
2  for m1 in soa1 {
3      iV.clear();
4      // evaluate CoM cutoff-condition
5      for m2 in soa2 {
6          ...
7          mask_packstore(iV, m2.sites...)
8      }
9      // compute LJ-Kernel forces
10     if (m1 is single-centered) {
11         for i in iV {...}
12     } else {
13         for i in iV {
14             r2 = gather_load(i[0:7], soa2.R, ...) // get partner pos.
15             for site in m1 {
16                 r1 = bcast_load(m1.r);
17                 f_LJ = LJ(r2-r1);
18                 ...
19                 f1 = reduce_add(f_LJ);
20                 m1.F += f1;
21             }
22             scatter_store(i[0:7], soa2.F, -f_LJ...)
23         }
24     }
25 }
```

Gather-scatter optimization Listing 6.1 illustrates the optimized gather-scatter implementation. The optimization consists of a loop interchange between the loops in line 13 and line 15. In this way, for multicentered molecules, the innermost loop in line 15 is not over sites from the second cell, but over the sites of the current molecule. This results in a trade-off between the relatively cheap broadcast and reduce operations in lines 16 and 19 and the expensive gather and scatter operations in lines 14 and 22. This reduces the overall count of the latter operations by a factor of the number of sites in a molecule, resulting in an increase of performance, especially for larger numbers of sites. For single centered molecules, the loop in line 15 has a trip count of 1, rendering it very inefficient. Hence, it has been peeled in line 10. This optimization is, unfortunately, not supported in the production trunk of `ls1 mardyn` yet, as it is not obvious how to integrate it in the wrappers (see Section 5.4) without code-duplication.

The simulations were performed on a dual-socket eight-core Intel IvyBridge host processor Xeon R E5-2650@2.6 GHz (IVB) with enabled hyperthreading and two Intel Xeon Phi 5110p coprocessors with 60 cores @ 1.1GHz. The simulation parameters are given in Table 6.1. The simulations were performed with the dedicated Xeon Phi branch of `ls1 mardyn`, in double precision. The OpenMP scheme `c08` is used, presented in Section 7.3.

Figure 6.2 shows the obtained results on the speed-up due to hyperthreading. As can be observed from Figure 6.2(a), when going from 1 to 4 threads, speed-ups of 2× to 2.5× were

scenario	model	N	cutoff $\frac{r_c}{\sigma}$	density $\rho\sigma^3$	Temperature
LJ fluid	1CLJ	159014	3.00, 5.00	0.73, 0.76	0.00095005
Acetone	4CLJ	159014	3.50, 4.61	0.44, 0.45	2.8714

Table 6.1: Simulation parameters.

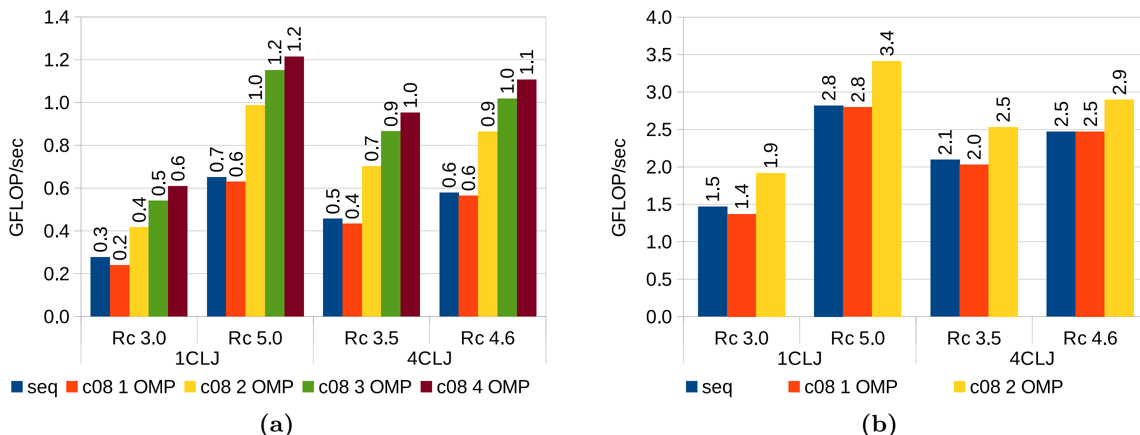


Figure 6.2: Performance for pinning hyperthreads on a single core. “seq” denotes the sequential variant, compiled without OpenMP. (a) Pinning up to four hyperthreads on a KNC core. (b) Pinning up to two hyperthreads on a IVB core.

measured. Adding the second thread gives the largest boost in performance, as two threads instructions can be issued in every CPU cycle. The third and fourth threads contribute less, but nevertheless keep increasing performance. The difference between the sequential version and running the OpenMP version with one thread is very small, testifying that the parallel overhead of the OpenMP version is very low. Experiments with oversubscription with more than four threads (not shown) lead to performance degradation. Hence, considerable gains are obtained from running with the full number of hyperthreads and no more.

On IVB the measured gains are more moderate, but, nevertheless, significant. They reach up to 1.4× for the 1CLJ, $r_c = 3$ scenario and between 1.17× and 1.24× for the other ones. Hence, for the remaining of the current work, we will aim for scalability up to the full number of hyperthreads supported by hardware.

Before proceeding, we comment on the gather-scatter optimization and the attained core performance on the KNC architecture. Figure 6.3 shows the obtained results for running similar scenarios with an increasing number of sites per molecule. It is observed that the optimized version has become “scalable” in the number of interaction sites per molecule. More sites per molecule lead to a better amortization of the costly gather and scatter operations. The basic implementation’s performance stagnates, on the other hand, likely at a limit depending on the gather and scatter operations’ performance. For both the optimized and the basic implementation, the performance is slightly higher at four and eight centers. Considering that a cache line can store eight double precision elements, this means that in those cases a maximum of one or two cachelines need to be accessed. Overall, the optimized version performs 1.1×–1.7× faster than the normal one over the range of two to ten sites per molecule. The attained absolute performance is in the range of 4%–8% of the theoretical peak of the core. This is around two times lower than the values observed on SNB or IVB. This is explained by the fact that most of the time the kernel is doing either additions or multiplications, but not fused multiply-adds. Since the peak performance of KNC can only be attained through fused multiply-adds, but not separate additions or multiplications, the attained fraction of peak performance is halved.

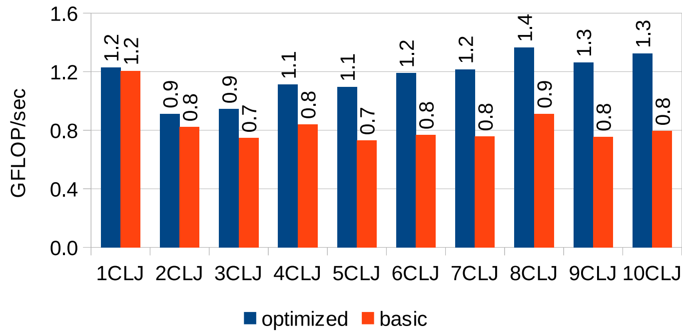


Figure 6.3: Performance of optimized and basic gather-scatter version with four threads pinned on the same KNC core.

6.2 Entire simulation

We now turn our attention to the systems, investigated in the previous sections. The results in this section are for the 1CLJ systems from Table 4.3 and the TIP4P system from Table 4.1. The simulations were carried out again on one Intel Xeon CPU E5-2680 v3 Haswell (HSW) node of Hazel Hen with Turbomode frequency scaling switched off. We have restricted the configurations to AVX2 and single and double precision. The `sli` scheme, presented in Section 8.1 was used.

Figure 6.4 presents the results. As can be observed, a performance increase of around 20% was observed in nearly all configurations and scenarios. The only exception is the `RMM` mode for the scenario $\rho = 0.39$, $r_c = 2.5$, in which case 32% were observed.

In order to investigate these gains further, the `RMM` configurations in single precision were profiled with Intel VTune Amplifier on a Intel Core i7-4770 @ 3.40GHz Haswell workstation. In the profiling setup, however, the gains due to hyperthreading for both cases became equal at 20%, in contrast to the observed 32% the low density scenario. Figure 6.5 plots some of the obtained metrics in a “Microarchitecture Exploration” analysis. The full reported metrics are provided in Appendix B for reference.

In Figures 6.5(a) and 6.5(b) it is observed that the fraction of retiring pipeline slots is increased by roughly 20% when going from one to two threads. The portion of back-end bound pipeline slot stalls decreases most notably. Taking into account that the absolute number of pipeline slots used by the program decreases by about 20%, the pipeline stalls due to back-end issues are practically halved. Front-end bound and bad speculation stalls also decrease, when going from one to two threads, except for front-end in the high density scenario. It is not immediately clear why they increase in that case (also in absolute numbers), even when considering all reported metrics in Appendix B.

Looking at the port utilisation in Figures 6.5(c) and 6.5(d), the utilisation of all ports is increased by about 20%. Comparing the two simulated scenarios, the scenario $\rho = 0.78$, $r_c = 5.0$ has a pronouncedly higher utilisation of ports 0 and 1. This is to be expected, since this is where floating-point operations execute, and the $\rho = 0.78$, $r_c = 5.0$ setup performs many more such operations than the $\rho = 0.39$, $r_c = 2.5$ one. Scenario $\rho = 0.39$, $r_c = 2.5$ has a higher utilisation of ports 5 and 6, suggesting high fractions of integer operations, which is also explained by the fact that relatively more time is spent traversing the cells.

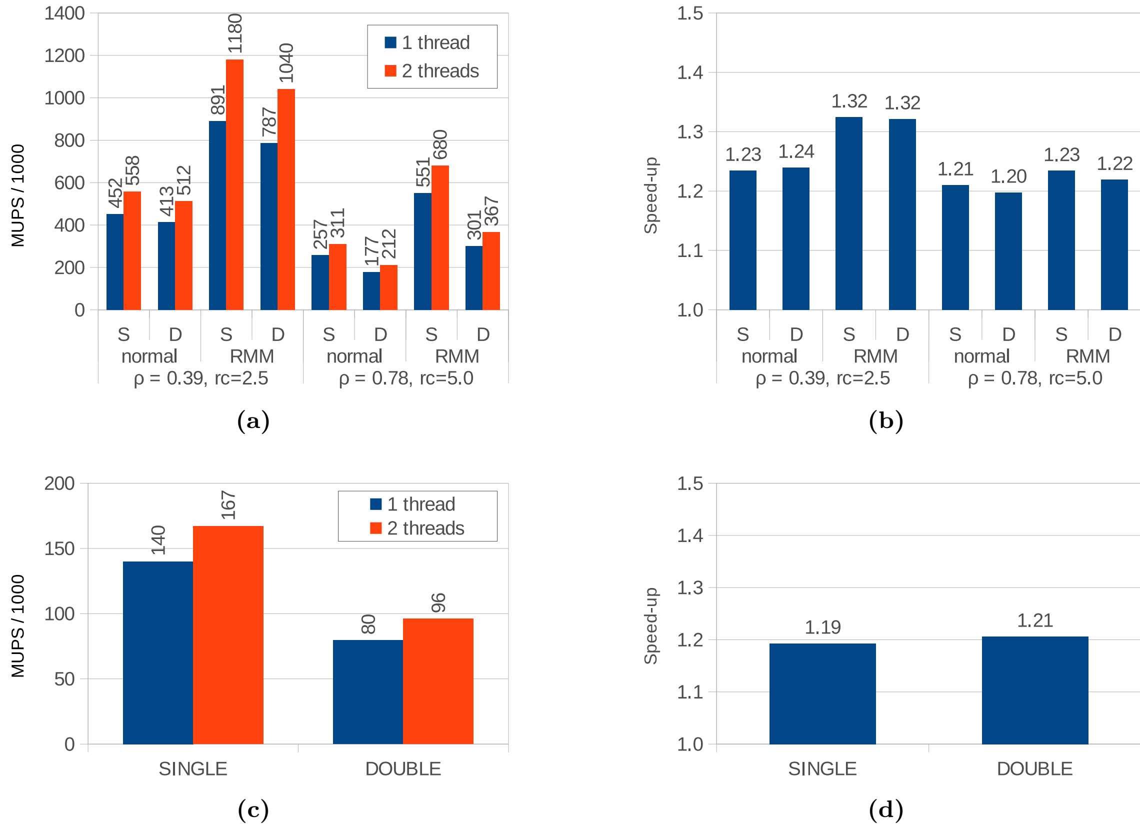


Figure 6.4: Performance and parallel speed-up for running one and two threads pinned on the same HSW core. (a) Performance for 1CLJ systems. (b) Speed-up for 1CLJ systems. (c) Performance for TIP4P system. (d) Speed-up for TIP4P system.

6.3 Conclusion

In conclusion, hyperthreading is a must-have on KNC, delivering around 2× higher performance. Similarly high gains are reported in [10] for running charge kernels in the context of FMM on another in-order CPU, PowerPC A2 in a BlueGene/Q supercomputer. Hence, one can conclude that for in-order architectures large gains can be expected.

On Xeon architectures, we can expect around 20% performance increase, although exceptions with up to 30% or 40% are possible. In [27], a value of around 12% is reported with an earlier version of `ls1 mardyn` also featuring a (different) reduced memory mode. A special OpenMP scheme was used for hyperthreading there, different from our (subsequently developed) ones. In that scheme, the threads work on spatially close data and synchronize often (once per three cells processed in the force calculation), which could be a reason for the low gains. The schemes presented here work on spatially disjoint data and synchronize only once (in **`sli`**) or eight times (in `c08`) for the entire force calculation.

In summary, the results of this section imply that, ideally, we always want to run with the full number of threads supported by the hardware. Thus, we have a requirement for the OpenMP schemes to scale up to 256 threads for Xeon Phi architectures.

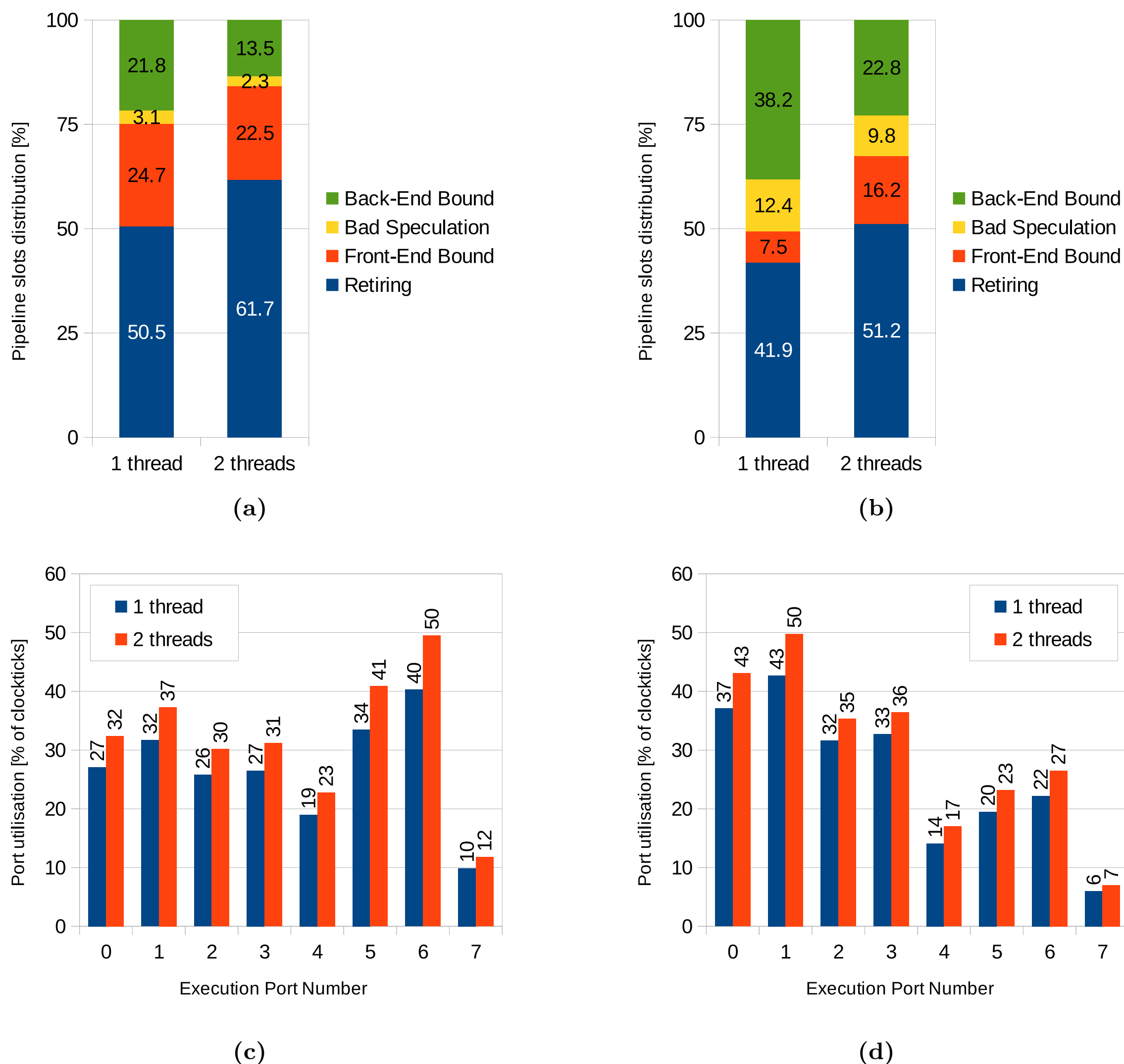
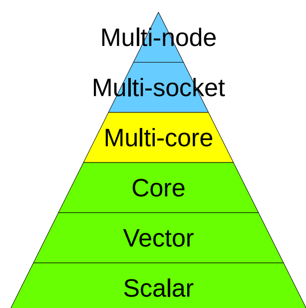


Figure 6.5: Intel VTune Amplifier analysis of `RMM` in single precision on a HSW desktop workstation. (a) Pipeline slots distribution for $\rho = 0.39$, $r_c = 2.5$. (b) Pipeline slots distribution for $\rho = 0.78$, $r_c = 5.0$. (c) Port utilisation for $\rho = 0.39$, $r_c = 2.5$. (d) Port utilisation for $\rho = 0.78$, $r_c = 5.0$.



Multi-Core-Level Performance

In this chapter we present the OpenMP parallelization of `ls1 mardyn`. We focus on multi- and many-core execution, which means that we will examine all aspects of OpenMP parallelization, except NUMA aspects. The NUMA aspects of modern hardware and their implications for HPC software will be discussed separately in Chapter 8.

Motivation The introduction of the Intel Knights Corner architecture in 2013 posed a challenge to HPC developers. On the one hand, featuring 60 cores with four-way hyperthreading meant massive parallelism on the node level. On the other hand, being an accelerator architecture, it requires a sufficiently high workload, in order to leverage its performance. However, the limited amount of RAM (8 or 16 GBytes) means that one cannot run arbitrarily large simulations on the card alone (in "native" mode).

As part of the Intel Parallel Computing Center[1] in Munich[2], it was decided to optimize `ls1 mardyn` for KNC. From the start it was decided to optimize `ls1 mardyn` for native mode execution, in order to prepare for the then-upcoming Knights Landing architecture. Since `ls1 mardyn` was pure MPI at the time, this meant running 60 to 240 MPI ranks per card in order to leverage the full performance of the architecture. Early experiments in [57] showed that fragmenting the domain into 240 MPI ranks exceeded the available RAM on the card, due to excessive storage for replicated cells in halo layers. For this reason, OpenMP schemes needed to be devised, in order to avoid this duplicated storage. Already in [57] some experimentation with OpenMP was done, but, unfortunately, did not deliver satisfactory performance. For this reason, considerable effort was invested in `ls1 mardyn` into memory-efficient OpenMP schemes, which is the primary contribution of this thesis.

The goal in this chapter is to establish efficient schemes, which can scale well up to hundreds of threads for execution on Intel Xeon Phi and x86 architectures in general. In Section 7.1 we discuss OpenMP parallelization of MD from a top-level perspective. In Section 7.2 related work is described, both in other MD codes and within `ls1 mardyn`. Section 7.3 presents the first schemes for the force calculation, which were demonstrated to scale sufficiently well. Results for the full simulation are included as well. Section 7.4 discusses further schemes for the force calculation, aimed at small systems. Section 7.5 evaluates alternative approaches, based on tasking.

Throughout this chapter we frequently present results on the KNC architecture. As of writing this document, this hardware architecture has been discontinued. Nevertheless, it provided an excellent testbed for developing OpenMP schemes. The slow sequential performance, lack of turbo-boost and large number of threads actually proved valuable in exposing any and all bottlenecks of the developed shared-memory implementations.

[1] `https://software.intel.com/en-us/ipcc`
[2] `https://www.lrz.de/services/compute/labs/astrolab/ipcc/`

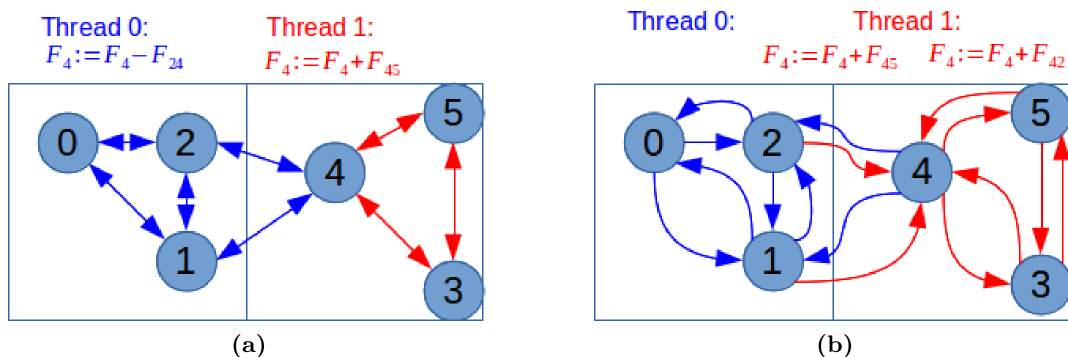


Figure 7.1: Illustration of thread-parallel force calculation. (a) Race conditions may arise due to Newton's third law, when threads work on adjacent cells. Thread 0 needs to read and write to the force buffer of molecule 4, in order to accumulate the force contribution $F_{42} = -F_{24}$. In the same way, thread 1 needs to read and write to the force buffer of molecule 4, in order to accumulate the force contribution F_{45}. (b) Without Newton's third law optimization, thread 1 computes both F_{42} and F_{45}.

7.1 OpenMP parallelization of MD simulations

Parts of the simulation to parallelize Molecular dynamics simulations are inherently very parallel. Most of the compute-intensive operations are performed on all molecules in the system. Thus, these operations easily lend themselves to parallelization over the number of molecules, which is usually much larger than the number of threads. Per MPI rank, molecules typically range from thousands to millions, while threads range from one up to 288 on Xeon Phi systems. Thus, it is almost always suitable to parallelize operations over the number of molecules.

The force calculation features complex dependencies and a high fraction of total runtime, hence, most of our effort was dedicated to it. While not time intensive, re-sorting molecules into cells features similar dependencies and, hence, we will discuss it briefly below. Apart from these two operations, most of the other operations are either embarrassingly parallel or straightforward to parallelize via prefix-sum-like schemes. For example, the velocity and position update of a molecule depend only on that molecule's data and no other molecules. For this reason, it is embarrassingly parallel and performed through a general-purpose parallelization via the `ParticleIterators` discussed in Section 4.1.3. As shall be seen in Section 7.3, the scalability of the full simulation highly mimics the scalability of the force calculation alone, as the other operations do not pose additional bottlenecks in a pure OpenMP setting. Thus, throughout this chapter and Chapter 8, we concern ourselves primarily with the force calculation.

Parallelization of the force calculation The main difficulty when parallelizing molecular dynamics simulations with OpenMP is how to resolve the race conditions, which arise when employing Newton's third law optimization. Figure 7.1 illustrates the problem. Both threads 0 and 1 need to read, update and write the force buffer of molecule 4, which may lead to race conditions and, respectively, wrong results. Even if, by chance, the results turn out correct, performance will suffer due to "false" sharing of the cache lines containing the memory buffers. For this reason, we want to develop OpenMP schemes in which threads are guaranteed to either work on disjoint cells or to write to disjoint memory locations.

Re-sorting molecules into cells In the Linked Cells algorithm (and also in the construction of the neighbour lists in the Verlet Lists algorithm), it is needed to sort molecules into cells. Without further assumptions, essentially a full-scale parallel sorting algorithm would be needed. In `ls1 mardyn`, the additional assumption is made that the cells are sufficiently large and the molecules move sufficiently slow so that over one timestep molecules may only travel from one cell to an adjacent one. As the cells are fairly large ($\geq r_c$), this is not a hard restriction and always holds in practice.

After every iteration, the molecules may have moved from one cell to another and need to be re-sorted. Under the aforementioned assumption, they may only move to a neighbouring cell. This involves removing a molecule from one cell's vector and inserting it in another cell's vector. If two threads re-sort molecules in neighbouring cells, race conditions may again arise. These race conditions are of the same nature as in the force calculation — read-write dependencies between neighbouring cells. For this reason, every OpenMP scheme for the force calculation we present here can be adapted to become a scheme for re-sorting molecules into cells. Throughout this chapter — whenever included in the measurements — the following OpenMP scheme is used:

1. All cells are traversed in parallel. Each cell is filtered for "leaving" molecules and they are removed from its primary `std::vector <Molecule>` into a second, auxiliary `std::vector <Molecule>` which stores only the molecules leaving the current cell.
2. All threads synchronize at a `#pragma omp barrier`.
3. The cells are traversed in parallel and for every cell, all $3^D - 1$ neighbouring cells' vectors of leaving molecules are traversed. If a molecule is found in a neighbouring cell, which travels to the current cell, it is appended to the primary vector of the current cell.
4. All threads synchronize at a `#pragma omp barrier`.
5. All cells are traversed in parallel and their vector of leaving molecules is cleared.

The core idea of this approach is to remove the write-dependencies to neighbouring cells. Thus, it can be considered a variant of the `no3` scheme, presented in Section 7.3.

7.2 Related work

Two standard approaches for resolving the race conditions of the force calculation are present in the literature:

- allocate additional force buffers for each molecule and for each thread,
- drop Newton's third law optimization altogether.

These approaches can be found in the source-codes of other high-performing MD codes, such as LAMMPS and GROMACS.

Force buffers per thread The first approach is typical for usage with moderate thread counts on multi-core architectures. Each thread allocates a separate copy of the force buffer of each molecule. For T threads, this means T times extra force-storage.

In the example from Figure 7.1, in which race conditions arise on the force-buffer of molecule 4, that would mean that threads do not write to its force storage F_4 directly, but to individual

copies F_4^0 and F_4^1, denoting the copy of thread 0 and thread 1, respectively. Then, once the calculation is complete, all of the results are reduced to the original copy:

$$F_4 = \sum_{i=0}^{T} F_4^i. \tag{7.1}$$

This approach is a competitive one, as we shall see in Section 9.2. Nevertheless, the memory overhead is unfeasible for going to hundreds of threads on the Xeon Phi. For brevity we will refer to this scheme as thread-molecule copies (**`tmc`**).

Dropping Newton's third law optimization The second approach, which appears commonly in high-performing MD codes is to drop *Newton3* altogether, as in Figure 7.1(b). This is also a common approach for GPUs, where the extremely high number of threads and limited amount of RAM render the force buffer approach infeasible. This leads, however, to doubling the computational complexity within cells from $\mathcal{O}\left(\frac{N(N-1)}{2}\right)$ to $\mathcal{O}\left(N(N-1)\right)$. For some Verlet Lists-based codes, this turns out to still be competitive (see Section 9.2), but as demonstrated in Section 7.3, this is not the case for Linked Cells-based codes. We refer to this approach as **`no3`** and present it in detail in Section 7.3.

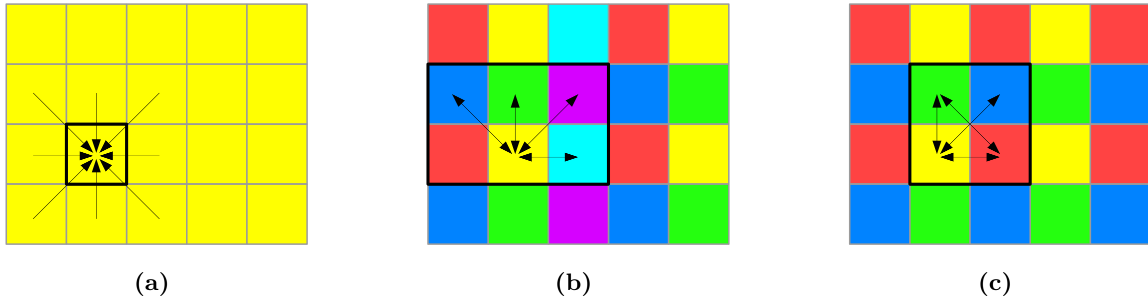
Early `ls1 mardyn` approaches There were considerable early attempts at introducing shared-memory parallelism in `ls1 mardyn` almost ten years ago, see [15]. Different variants were tried, including atomic operations, colouring, locking of cells, force buffers per thread, etc. Some of those schemes were tried with either OpenMP or Intel TBB. Unfortunately, none of the schemes was found to perform sufficiently well and they were outperformed by the pure MPI version by significant margins. For this reason, they were not integrated in the production trunk at that time and `ls1 mardyn` remained MPI-only.

Possible explanations why the investigated schemes did not perform as well as the ones presented in this work, could be the old software design or further data structures, which were not fully cleansed of false sharing or other bottlenecks. It was not possible to revitalize those solutions, as they were developed in old branches of `ls1 mardyn`, which had diverged too early and too far from the production trunk. For this reason, new OpenMP schemes had to be developed and reimplemented from scratch.

7.3 First scalable schemes: `no3`, `c08`, `c18`

In [109] we introduced the first scalable OpenMP solutions in `ls1 mardyn` which retain *Newton3*. This was done in the specialized Xeon Phi branch of `ls1 mardyn`, building on top of the implementation developed in [57]. The Xeon Phi version used the gather-scatter optimization described in Section 6.1. In this work, the **`no3`** version used in [57] was greatly sped-up through some of the software design changes described in Section 4.1. A total of three OpenMP schemes were evaluated: **`no3`**, **`c18`** and **`c08`**.

`no3` scheme Figure 7.2(a) illustrates the **`no3`** scheme. This scheme sacrifices *Newton3* and recomputes forces between all cell-pairs. This removes write-dependencies to neighbouring cells, thus, allowing all cells to be processed in parallel by the threads. No synchronization steps are needed, apart from at the end of the calculation. Forces within a cell are still computed with the *Newton3* optimization.



(a) (b) (c)

Figure 7.2: OpenMP schemes. Arrowheads denote read-and-write dependencies, arrow tails denote read-only dependencies. (a) `no3` scheme. Write-dependencies to neighbouring cells are removed by not using Newton's third law and recomputing all contributions. (b) `c18` scheme. Colouring scheme with $2 \times 3 \times 3 = 18$ colours in three dimensions to resolve read-write dependencies to neighbouring cells. (c) `c08` scheme. Colouring scheme with $2 \times 2 \times 2 = 8$ colours in three dimensions to resolve read-write dependencies to neighbouring cells.

`c18` scheme The `c18` scheme is illustrated in Figure 7.2(b). It is the direct result of applying colouring to the dependencies, introduced when *Newton3* is applied. When *Newton3* is used, one only needs to traverse the neighbouring cells, whose index (in a one-dimensional linearised storage) is larger than the current index. In order to parallelize this traversal via colouring, one needs to apply a stride of two in the first dimension and a stride of three in subsequent dimensions. In 2D, this results in $2 \times 3 = 6$ colours, while in 3D it results in $2 \times 3 \times 3 = 18$ colours, hence, the name we have assigned to it. This scheme requires 18 synchronization steps per force calculation, which is a fairly high amount. This scheme appears already in [15].

`c08` scheme Figure 7.2(c) illustrates this scheme. The scheme is a modification of the `c18` scheme, which reduces the number of colours and, respectively, synchronization steps down to eight. The crux of the `c08` scheme lies in the order of the calculation of the cell-pairs. In Figure 7.2(c), while working on the yellow cell, instead of computing the interaction between the yellow cell and its blue upper-left neighbour, the same diagonal interaction is computed between its red, right neighbour and its green, upper neighbour. This idea was inspired by [80], where a similar reordering of the operations appears in the context of Lattice Boltzmann simulations. This allows a more compact access of the cells being worked on, thus, also resulting in a better data reuse. More importantly, it decreases the striding requirements down to 2 in each dimension, resulting in a total of $2^3 = 8$ colours in 3D. This is more than a two-fold reduction of the number of synchronization steps. Apart from less synchronization, this also implies more work per colour, which means that more threads can work efficiently in parallel.

We will refer to this way of traversing the cells as the "compact" traversal for the remainder of this work. A package of eight cells (in a $2 \times 2 \times 2$ configuration), accessed during the processing of one cell via the compact traversal will be referred to as a "8-pack" of cells.

Both this scheme and the `c18` one were implemented with a `schedule(dynamic,1)` OpenMP scheduling, meaning that 8-packs are assigned to threads dynamically on a first come, first serve basis. The use of this scheduling is intended to mitigate potential load-imbalances to some extent.

7.3.1 Results: force calculation

In this section, we summarize our results from [109], where investigations were made for the force calculation only. The simulation parameters are given in Table 7.1. The measurements were performed on dual-socket Intel Xeon E5-2650v2 IvyBridge (IVB) processors and Intel Xeon Phi 5110p (KNC) coprocessors used in native mode. We ran the simulations with `no3`, `c18` and `c08` on KNC. On the host IVB machines, instead of the OpenMP `no3` scheme, the MPI version available at the time was run for a comparison. That version used the *Newton3* optimization and the legacy sliding window implementation. In the measurements, since only the time within the force calculation is measured, the `mpi` measurements are at an advantage, as they appear as embarrassingly parallel in this setting. They perform some extra calculations, however, due to the full shell MPI scheme (cf. Section 2.2.6).

scenario	N	cutoff $\frac{r_c}{\sigma}$	density $\rho\sigma^3$	Temperature
1CLJ	159014, 1317006	3.00, 5.00	0.73, 0.76	0.00095005
4CLJ	159014, 1317006	3.50, 4.61	0.44, 0.45	2.8714

Table 7.1: Simulation parameters.

Figure 7.3 shows strong scaling results going up to the maximal number of supported threads or processes. First we comment on the effect of the *Newton3* optimization. It is most cleanly seen by comparing `c08` and `c18` to `no3` on KNC, when running with 1 thread. In those cases, `c18` and `c08` perform 1.33×–1.57× faster, giving a considerable boost and, thus, justifying the development of the new schemes.

Next, we comment on the scalability of the schemes. Overall, all schemes scale quite well. Expectedly, the larger scenarios scale better, because they contain more cells. When increasing the number of threads, the performance increases for all measurements in the large systems. For the smaller systems, the scalability sometimes breaks down, particularly for the larger cutoff radii. This is, again, explained by the number of cells — the systems have an equal volume, so the larger cutoff radii subdivide the domain into fewer cells.

Comparing the `c18` and `c08` schemes for the lowest attained runtime overall, `c08` is between 2% and 26% faster for all, except two cases. In the two cases IVB 0.16M 1CLJ rc3 and KNC 1.32M 4CLJ rc3.5, `c18` is 7% and 3% faster, respectively. It is not completely clear why `c18` is faster in those cases. Perhaps a subdivision of the cells, which matches better the stride-3 patterns than the stride-2 patterns could be a reason. Still, `c08` is considerably faster in most of the cases.

Looking at the parallel efficiencies at the maximal number of cores, the values are mostly between 80% and 88% for `no3` (first- and third quartiles of all values), between 78% and 91% for `c18` and between 84% and 95% for `c08`, as compared to their values at one thread. Thus, it is observed that `no3`, apart from having a lower sequential performance, also sometimes features a lower scalability in some cases. In this way, in the KNC 4CLJ rc4.6 1.32M case at 240 threads, `c08` outperforms `no3` by a factor of 1.72×. This is a surprising observation, suggesting that perhaps the `no3` scheme could be optimized further. It was not investigated in greater detail, however, as the two coloured schemes are clearly more promising.

For the `c08` case, we can use the hyperthreading experiments from Section 6.1 to obtain a normalized value of parallel efficiency at 240 threads. According to that section, when going from 60 to 240 threads, we can expect speed-up values of 2.5×, 1.9×, 2.2× and 2.0× for the 1CLJ rc3, 1CLJ rc5, 4CLJ rc3.5 and 4CLJ rc4.6 cases, respectively. Thus, for the 1.32M cases, we arrive at parallel efficiencies of 77%, 83%, 77% and 84%. Taking into account that

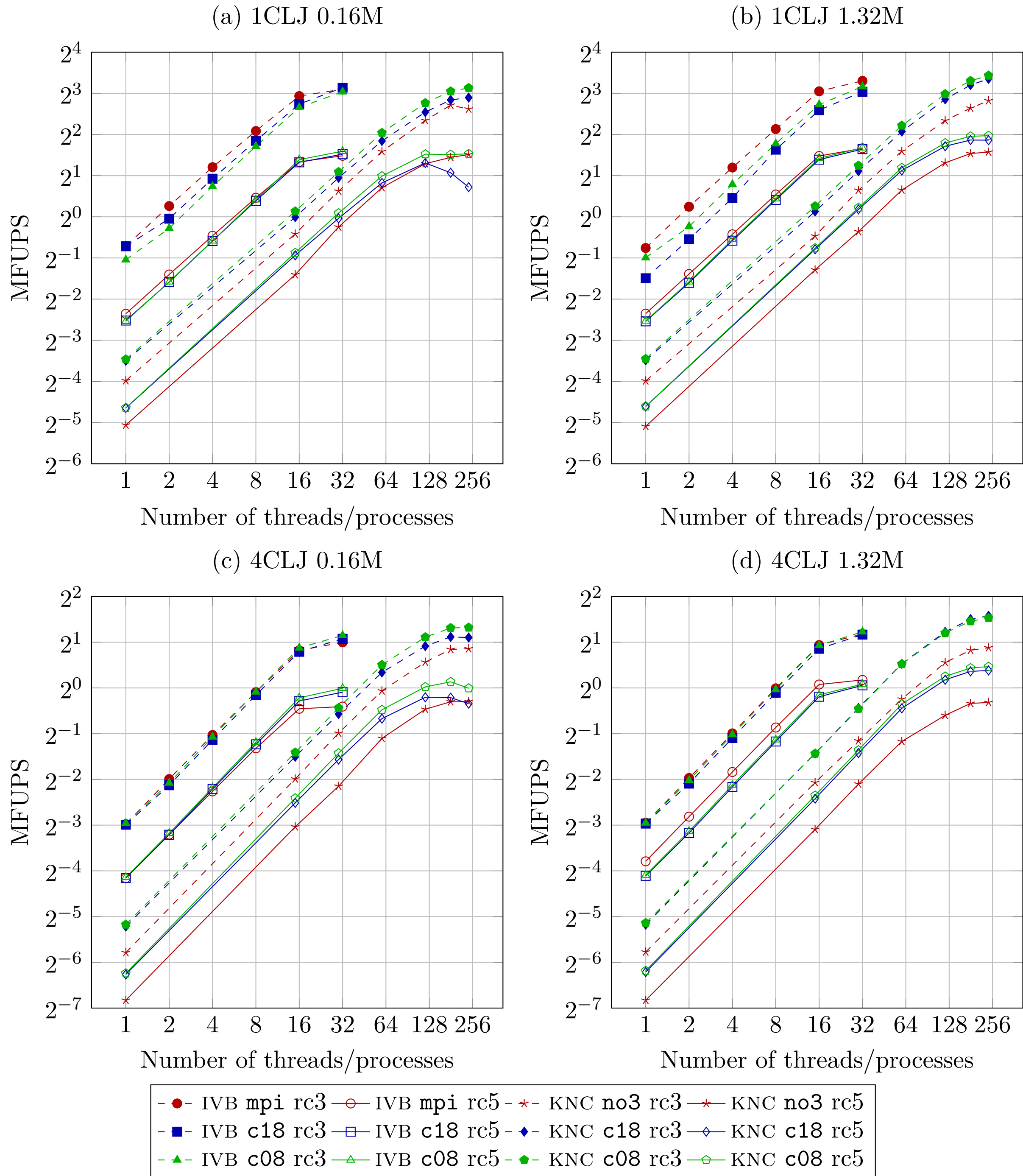
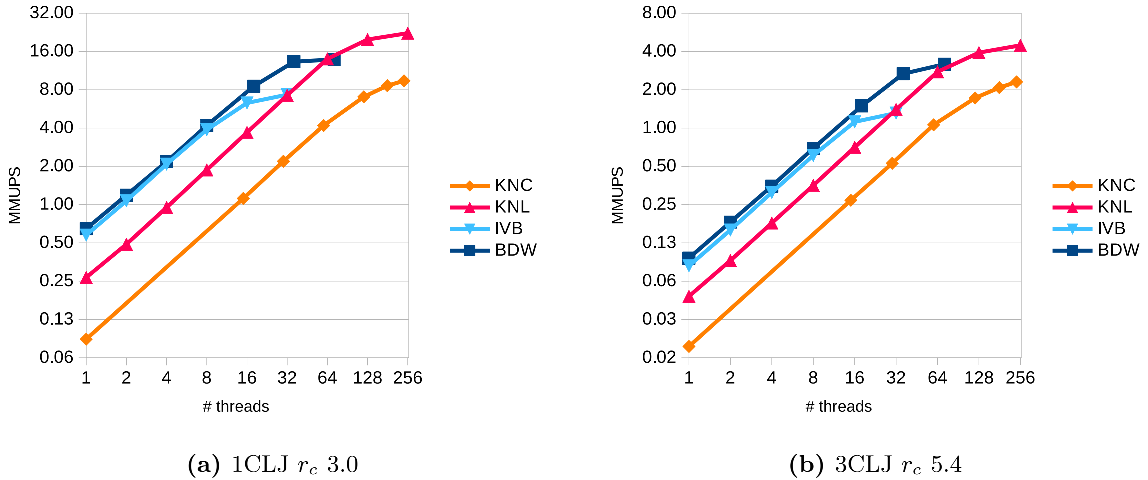


Figure 7.3: Performance in million force-updates per second for strong scaling experiments on KNC and IVB. rc3 denotes $r_c = 3.0$ for the 1CLJ scenarios and $r_c = 3.5$ for the 4CLJ scenarios. rc5 denotes $r_c = 5.0$ for the 1CLJ scenarios and $r_c = 4.6$ for the 4CLJ scenarios.

scenario	N	cutoff $\frac{r_c}{\sigma}$	density $\rho\sigma^3$	Temperature
1CLJ	1317006	3.00	0.73	0.00095005
3CLJ	1317006	5.35	0.29	3.4605

Table 7.2: Simulation parameters.



(a) 1CLJ r_c 3.0

(b) 3CLJ r_c 5.4

Figure 7.4: Strong scaling results for system with parameters given in Table 7.2.

this is a strong-scaling measurement going from 1 to 240 threads, this represents an excellent achievement.

Comparing the **mpi** version to **c08** on IVB, the values range between 0.91× (meaning that **mpi** outperforms **c08**) and 1.32× (meaning that **c08** outperformed **mpi**). **mpi** performs better than **c08** when the extra calculations due to the halo regions matter less: for the larger systems, for the smaller cutoff radii and for the lighter molecular model. Thus, the extreme value of 0.91× is attained for the 1.32M 1CLJ rc3 case. Conversely, the more the extra calculations matter, the more **c08** outperforms **mpi**, giving the extreme value of 1.32× for the 0.16M 4CLJ rc4.6 case.

Comparing KNC to IVB, KNC is observed to perform the force calculation between 0.96× and 1.27× faster than IVB.

7.3.2 Results: entire simulation

In this section, we present results for the entire simulation. The measurements were carried out with the c08 OpenMP scheme for the force calculation, again with the optimized Xeon Phi branch of ls1 mardyn. The remaining parts of the program were parallelized with OpenMP as indicated in Section 7.1. Apart from introducing OpenMP parallelization to the whole program, further optimizations were performed in the direction of changing the data structures towards the ones from Section 4.1.3.

The measurements were performed on the same IvyBridge and Knights Corner processors as in the previous section, together with an Intel Xeon E5 2697v4 Broadwell (BDW) system and an Intel Xeon Phi 7210 Knights Landing (KNL) system. Turbo-boost was enabled for Broadwell and KNL. Table 7.2 gives the simulation parameters.

Figures 7.4 and 7.5 show the simulation results. Excellent scalability is attained for all ar-

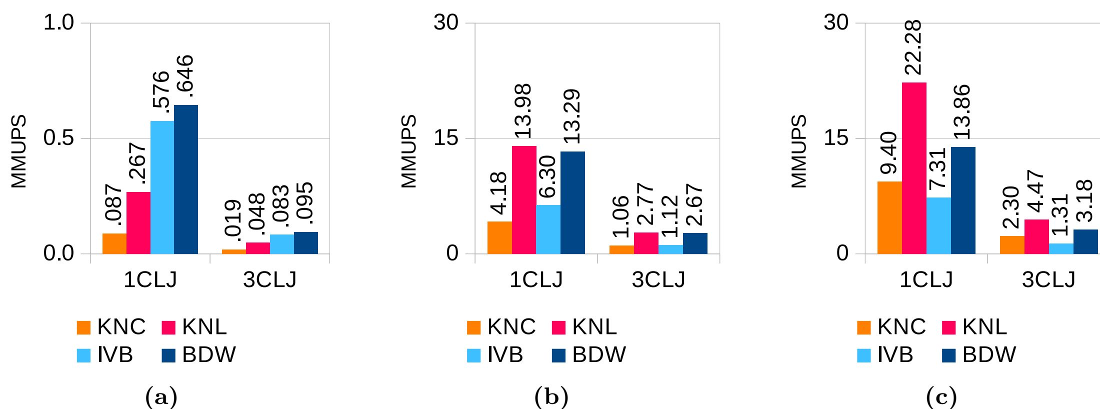


Figure 7.5: (a) Simulation performance at one thread. (b) Performance at maximal number of cores. (c) Performance at maximal number of threads.

	Speed-up T_{max}		Efficiency C_{max}	
	1CLJ	3CLJ	1CLJ	3CLJ
KNC	107.4	120.1	80%	92%
KNL	83.4	94.1	82%	91%
IVB	12.7	15.9	68%	85%
BDW	21.5	33.5	57%	78%

Table 7.3: Parallelization quality for system from Figure 7.4. Speed-up is given at the maximal number of threads, while parallel efficiency is given at the maximal number of cores.

chitectures, demonstrating that the developed solution is platform independent and suitable for future architectures with increasing core-counts. Table 7.3 summarizes the obtained speed-ups and parallel efficiencies. Speed-ups of 16×–100× are observed, highlighting both the potential and the importance of node-level performance. The values are better for the 3CLJ case, as the calculation proceeds, expectedly, at lower rates, due to the more compute intensive molecular model. The more work to do, the better all parallel overhead is amortized. The BDW speed-up values are somewhat lower, due to the fact that turbo-boost was used — the CPU frequency boost is higher when only one core is working than when multiple cores are working. Comparing the KNC 1CLJ speed-up of 107.4× to the one from Section 7.3.1 (118.0×) we observe that the rest of the simulation is not a bottleneck to the scalability of the program.

Hyperthreading again plays an important role, cf. Figures 7.5(b) and 7.5(c). On Xeon it gives a boost of about 1.2×, except for the BDW 1CLJ case, where it is only 1.04×. On Xeon Phi the gains are 2.2× for KNC and 1.6× for KNL. The value is lower for KNL, as the cores have a much higher single-thread performance, cf. Figure 7.5(a). This is because the KNL cores support out-of-order execution, can issue instructions in every cycle, feature a higher base-frequency and support turbo-boost [64]. KNL is 3.1× and 2.5× faster than KNC on 1 thread for the 1CLJ and 3CLJ cases, respectively, which fits well with Intel projections of 3× [64]. KNC, however, gains more in the hyperthreading range, so at the maximal number of threads, the speed-ups of KNL over KNC go down to 2.4× and 1.9× for the 1CLJ and 3CLJ cases, respectively.

Comparing BDW and IVB, sequential performance improved by only 1.1× and 1.2× for the 1CLJ and 3CLJ cases. Thus, the overall gains at the maximal number of threads are mostly

due to the higher number of cores. The final values are 1.9× and 2.4× in favour of BDW.

One can also draw a comparison between KNL and BDW. At one thread, KNL is at only 0.4× and 0.5× of the BDW performance. This is mostly due to the lower CPU frequency: KNL has a base frequency of 1.3 GHz which can go up to 1.5 GHz with turbo-boost, while BDW — 2.3GHz up to 3.6 GHz. At the maximal number of threads, however, the massive parallelism of KNL manifests itself and it outperforms BDW by 1.6× and 1.4× in the simulated scenarios. Similarly, KNC starts off at 0.2× of IVB's performance, but manages to outperform it significantly at 1.3× and 1.8×.

7.3.3 Summary and conclusion

In Section 7.3 we have reevaluated the `no3` and `c18` schemes and shown that they can scale excellently. Furthermore, we have introduced `c08`. Thus, we have managed to recover the *Newton3* optimization in a memory-efficient fashion at great benefits. We demonstrated that the `c08` scheme, is very promising and outperforms `c18` by up to 1.26× and `no3` by up to 1.72×. For this reason, `c18` and `no3` will not be considered further in this work, though they could be incorporated in the autotuning library of Section 9.5. Furthermore, it was shown that the new schemes perform on par with the MPI force calculation (even without MPI messaging overhead), which we consider a significant achievement.

In the second part of this section, it was demonstrated that the excellent OpenMP scalability extends from the force calculation to the entire simulation. This showed that a pure-OpenMP approach is viable for small molecular systems and sets good expectations for hybrid MPI-OpenMP execution.

Moreover, the developed solution was shown to be platform-independent as the same scheme ran excellently on essentially four architectures. This confirms that going for the native mode on Knights Corner was a good decision, as the Knights Landing results were obtained without prior tuning, apart from adapting the intrinsic instructions from KNC to AVX512. This allowed to draw extensive comparisons of the different hardware platforms and demonstrate that there are use cases where Xeon Phi can outperform Xeon by significant margins.

Some drawbacks of `c08` are the eight barriers and the fact that the data is streamed through the CPU eight times. Moreover, once a pack of cells is processed, a new pack of cells needs to be fetched from memory, which is disjoint from the old one. This reduces data reuse. A possible extension of `c08` was discovered in [44], which can address the latter drawback. It will be presented in Section 7.5.1, where the results of [44] are presented.

As another drawback of `c08`, it can be said that it is a relatively coarse-grain approach. Indeed, performance began to break down already for systems with 160 000 molecules in Section 7.3.1. Moreover, it has a clear limitation of requiring at least eight cells per thread, which — in the limits of MPI strong scaling — can prove to be a bottleneck. For this reason, further possibilities for the force calculation will be explored in the next section, aiming at smaller molecular systems.

7.4 Parallelization of small systems: `cfb`, `tfb`

The schemes discussed in Section 7.3 have a clear granularity requirement of at least eight cells per thread for `c08` and eighteen for `c18` (in 3D). For small systems, or in the limit of strong scaling with MPI, this poses a strong limitation. For this reason, further, more fine-granular systems are also of interest. In this section, we investigate schemes that allocate extra buffers for forces on a cell-basis so that threads write to disjoint memory locations.

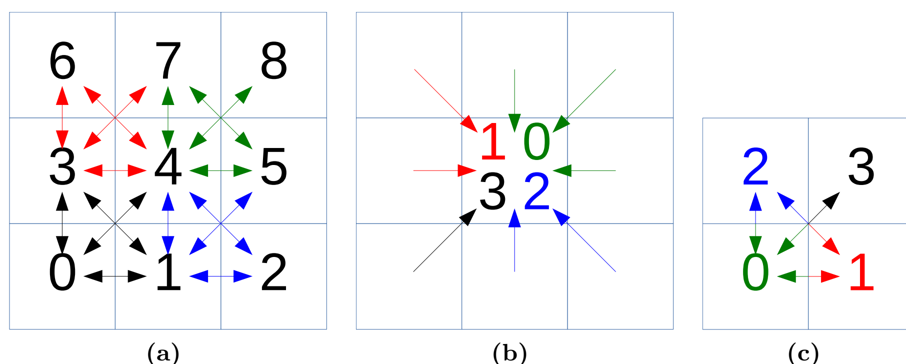


Figure 7.6: Illustration of the `cfb` scheme. (a) Cell number 4 can be accessed only from within four packs of cells (eight in 3D) — the ones based at cells number 0, 1, 3 and 4. Thus, at most four threads can access the molecular data simultaneously. (b) Buffers with IDs 0 to 3 are allocated for each cell. Buffer with ID 0 is for writing the contributions from the right cell, the upper cell and the upper-right cell and so on, as indicated by the colours of the arrows and IDs. (c) When computing the contributions between cells, they are written to buffers of different IDs in the different cells. E.g. when processing the lower two cells, the forces are written in buffer 0 for the left cell and in buffer 1 for the right cell.

7.4.1 Parallelization schemes

Cell Force Buffers `cfb`: The first scheme can be considered to be a buffered variant of `c08`. It is based on the observation that in the compact traversal a cell is accessed only from within eight 8-packs, see Figure 7.6(a). This means that at most eight threads can access a cell simultaneously. Thus, only eight copies of the force storage per cell suffice, instead of the T copies in the thread-molecule copies `tmc` scheme from Section 7.2. Then, depending on where a cell is located within an 8-pack, each thread writes in disjoint memory locations so that no race conditions arise, see Figures 7.6(b) and 7.6(c). In the example in Figure 7.6(a), the interactions are grouped in the following fashion. The interactions of cell 4:

- with itself and with cells 5, 7 and 8 are written in buffer 0 of cell 4,
- with cells 3 and 6 are written in buffer 1 of cell 4,
- with cells 1 and 2 are written in buffer 2 of cell 4,
- with cell 0 are written in buffer 3 of cell 4.

Note that e.g. buffer 0 could, technically, also be the primary force storage, thereby reducing the requirements for buffers a little more. After all forces have been computed, a second traversal of the cells is necessary to accumulate all contributions of the buffers together. In this fashion, all packs can be processed in parallel and in any order. In the presented implementation, the distribution of work to the threads was done via a `schedule(guided)` scheduling.

We now compare the memory requirement of this scheme and `tmc`. `cfb` requires $N_c \cdot c_f \cdot 8$ storage units, where N_c is the number of cells in the system and c_f is the force storage of one cell. The requirements of `tmc` are then $N_c \cdot c_f \cdot T$, where T is the number of threads. If the number of threads used is less than eight, the requirements of this scheme exceed those of `tmc`, but, for T in the range of hundreds, this can mean considerable savings.

scenario	model	N	cutoff $\frac{r_c}{\sigma}$	density $\rho\sigma^3$	Temperature
ethane	2CLJ	1000–64000	4.0	0.07746	1.6228

Table 7.4: Simulation parameters.

Thread Force Buffers `tfb`: This is a scheme, which aims at allowing an even finer granularity than `cfb`. In this scheme, each thread allocates two memory buffers, each of which is large enough to store the forces of the molecules in a cell. During the calculation, all force contributions are computed and accumulated in those buffers, thus race-conditions are excluded. As soon as the calculation of forces between a pair of cells is completed — and before proceeding with the next pair of cells — the results are reduced to the primary storage of the cells. Since race-conditions may now arise during the — potentially simultaneous — reduction of results to the primary storage, this is done under either OpenMP atomic operations or locks. In the presented version, locks were used, as atomic updates could not be performed on intrinsic types such as `__m512d`. The distribution of work to the threads was also done with a `schedule(guided)` scheduling.

The granularity of this scheme is now down to pairs of cells. In a scenario with N_c cells, the number of pairs of cells to be computed is about $N_c \cdot \frac{3^D+1}{2} = N_c \cdot 14$ (including computation within the cell). We can then start a simulation, theoretically, with up to $N_c \cdot 14$ threads and have no idle threads. In practice this cannot be expected to perform perfectly, however, since having many threads working on every cell will inevitably lead to multiple threads waiting to accumulate their contributions to the primary storage.

Two variants of this scheme were tested — `tfb` and `tfb-3lck`. In the first one, `tfb`, one OpenMP lock is allocated per cell. During the reduction, the thread sets the lock of the cell, accumulates all force contributions and then releases the lock. In the second variant, `tfb-3lck`, three locks are allocated per cell — one for each dimension of the force contributions. The idea of the second variant is that while one thread accumulates in the x-direction force buffer, a second thread could accumulate in the y-direction force buffer.

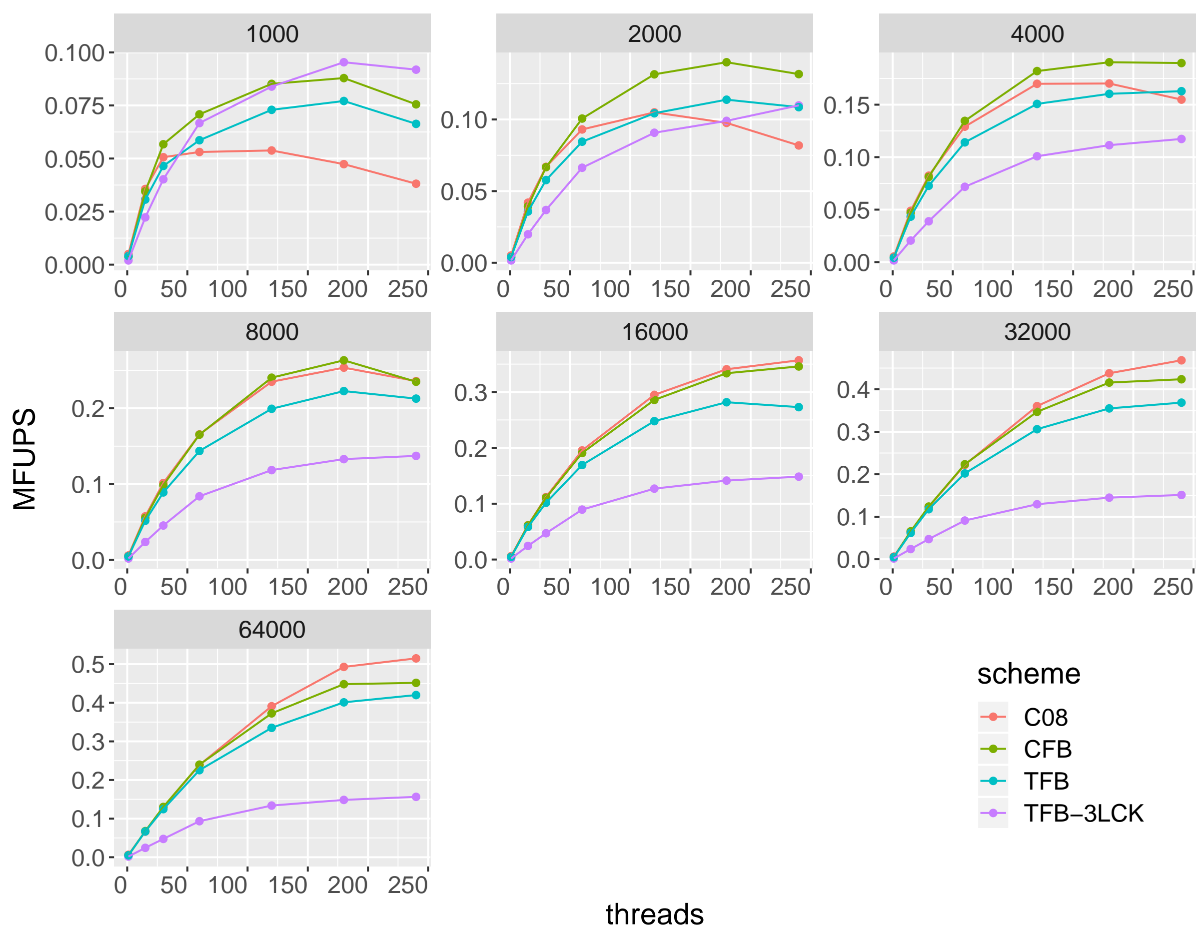
In [15], a variant with thread-local copies was also discussed. However, it appears that a variant of `tmc` was implemented, not of `tfb` as presented here.

Again comparing to `tmc`, the memory requirements of this scheme are $c_f \cdot T \cdot 2$, where T is the number of threads and c_f is the force storage of one cell. Clearly, if $N_c > 2$, `tfb` is more memory-efficient than `tmc`. Comparing `cfb` and `tfb`, the question is whether $N_c \cdot c_f \cdot 8$ is larger or smaller than $c_f \cdot T \cdot 2$. If $N_c \cdot 4 < T$, then `cfb` is more memory efficient, otherwise `tfb` is.

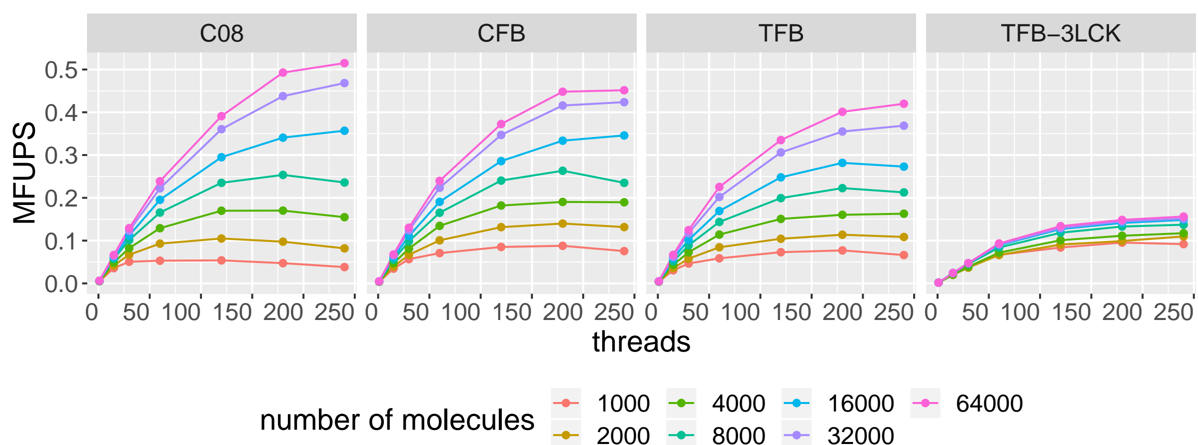
7.4.2 Results

The `cfb` and `tfb` schemes were tested and compared to `c08` for systems of ethane molecules containing between 1000 and 64000 molecules. The simulation parameters of the systems are given in Table 7.4. The measurements were performed on the Intel Xeon Phi 5110p coprocessors in native mode (KNC). The optimized Xeon Phi branch of `ls1 mardyn` was used. Only the time for the force calculation was measured for clean comparisons to `c08`.

Figure 7.7 presents the results of the measurements. Since these schemes are targeted at small systems, we will focus on the question of which scheme can advance the simulation fastest, not necessarily on parallel efficiency and speed-up. It is clear, that increasing the system size improves the performance, hence, larger systems can provide better values of efficiency and speed-up. It should also not be forgotten that accelerator architectures, such as the KNC used here, are generally aimed at large workloads, not small ones. It is, nevertheless, instructive to



(a)



(b)

Figure 7.7: Strong scaling performance on KNC for `cfb`, `tfb` and `tfb-3lck` for systems of variable size. Performance in million force-updates per second. (a) Results grouped by number of molecules in system. (b) Results grouped by OpenMP scheme.

investigate where the limits lie.

For the larger systems, the `c08` scheme was found to outperform the memory-buffer ones. In Figure 7.7(a), from about 16000 molecules onwards, `c08` is found to be the best performing scheme in these settings, despite the fact that it traverses the data eight times, while the other schemes traverse it only once or twice.

The border of 8000 molecules, when the performance of `c08` breaks down, agrees well with the number of 8-packs per colour. That scenario features $12 \times 12 \times 12$ packs to be processed ($11 \times 11 \times 11$ inner cells). Per colour, this gives $6^3 = 216$ packs and indeed we observe rising performance until 180 threads and a drop at 240 threads. The performance of `cfb` and `tfb` in that case also drop, however, between 180 and 240 threads. The drop for `cfb` and `tfb` is by a larger factor than in the 4000 or 2000 cases, so there might be further reasons, why this configuration is particularly unfavourable for execution at 240 threads.

The scheme `cfb` performs best for the intermediate range of systems sizes between 2000 and 8000 molecules. In the 8000 case, even though the $12 \times 12 \times 12 = 1728$ packs to be processed give ample work per thread, the performance also drops from 180 to 240 threads. In fact, between 1000 and 8000 molecules, the best performance for `cfb` is always attained at 180 threads, not 240, even though from 2000 molecules onwards, the number of packs to be processed exceeds 240. This illustrates that at least several cells per thread are required in practice to make the scheme perform well. One reason for this could be that when the number of cells per thread is small ($1728 / 240 = 7.2$), load imbalances of ± 1 cell start to become significant.

Overall, `tfb` exhibits very similar scaling behaviour to `cfb`, but was everywhere marginally slower, cf. Figure 7.7(b). Again, similarly to `cfb`, for systems smaller than 4000 molecules, `tfb` also surpasses `c08`. The similarity between `cfb` and `tfb` performance is surprising, considering, that they work quite differently — `cfb` uses eight times the force storage, while `tfb` uses one lock per cell.

The `tfb-3lck` variant exhibits a more distinct behaviour. While it represents only a small modification of the `tfb` variant, it shows drastically different performance from it, see Figure 7.7(b). One reason could be that the performance becomes dominated by the tests to obtain the locks for reducing the thread-buffers into the primary storage. The performance for all systems is low and relatively constant. Interestingly, however, this variant delivers the highest performance for the 1000 molecule case. The performance increases up to 180 threads, which gives just 5.5 molecules per thread, or 16.6 molecules per KNC core, which is also impressive.

7.4.3 Conclusion

The buffered schemes `cfb` and `tfb` present interesting alternatives to `c08` for small systems. It was demonstrated that `cfb`, `tfb` and `tfb-3lck` can outperform `c08` with respect to time to solution.

Since `tfb` theoretically offers much finer granularity than `cfb`, perhaps further optimizations of the `tfb` scheme and the `tfb-3lck` schemes are possible. On the other hand, the `tfb` schemes involve setting and unsetting OpenMP locks multiple times per cell, which might be the reason for the lower performance they exhibit.

The results in this section emphasize (again) that determining which scheme will perform fastest is not an easy matter. The choice is difficult both for the developer and for the application user and should best be left to runtime autotuning as described in Section 9.5. The use of runtime autotuning would further allow the exploration of different OpenMP schedules and chunk sizes.

Due to technical issues with the software design, the buffered schemes were not added to the production trunk of `ls1 mardyn`. Nevertheless, they are considered for the library of Section 9.5.

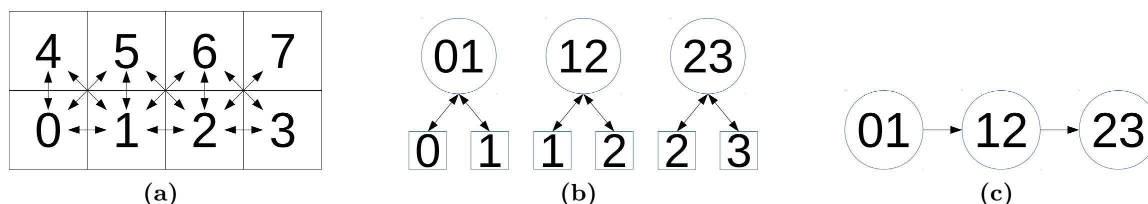


Figure 7.8: Graph serialization arising from attempting to use OpenMP tasks with dependencies for the force calculation. (a) Group of cells for the force calculation. (b) In order to compute forces between cells n and m without race conditions, `inout` dependencies must be specified between task nm and cells n and m. (c) In the OpenMP dependency model, if task nk is spawned after task nm and they both feature `inout` dependencies on cell n, then task nk *must* be executed after task nm.

7.5 Tasks with dependencies: Quicksched, OpenMP

Up to this point, all investigated shared-memory parallelization schemes were loop-based. In this section we explore alternative, task-based approaches and compare their performance to the established `c08` method. We summarize the results of [44], which was carried out as part of this work.

Task-based parallelism offers an alternative which is worth exploring, as it can potentially allow the removal of nearly all synchronization points in a multithreaded program. If the programming model supports dependencies between the tasks, one can specify for example that once the force-calculation tasks of a cell are completed, a task with the calculation of velocities of the molecules in that cell can be started. Thus, one thread can already work on the velocity update, while other threads work on force updates in far-away cells. This allows to potentially remove synchronization points not only within the force calculation (such as the eight barriers in `c08`), but also among different subroutines of the program.

Multiple libraries are available for task-based parallelism. Tasks were introduced in OpenMP with the 3.0 standard and dependencies among them were introduced with the 4.0 standard [106]. Unfortunately, the introduced functionality for dependencies proved to be too rigid for the requirements of N-body codes. Namely, the dependencies between the tasks impose a strict order on their execution. This strict order is too rigid for the force calculation, where simple reduction operations are needed. Consider the example of cells in Figure 7.8(a). Forces need to be computed between the pairs of cells 01, 12 and 23. In order to ensure that the calculation of 01 and 12 does not happen concurrently, `inout`-dependencies need to be specified between task 01 and cells 0 and 1, see Figure 7.8(b). In this tasking model, since task 12 is spawned after 01, it has to be executed after it, because it also has an `inout` dependency on cell 1. In the same way, task 23 has to be executed after 12. This results in the execution graph depicted in Figure 7.8(c), meaning that task 23 must be processed after task 01, when in fact they could have been executed simultaneously. In this fashion, complicated dependency graphs are generated (even more so, when the other dimensions are taken into account), which contain high degrees of artificially created serialization. This results in severe limitations to scalability. As a possible remedy to this issue, it was attempted to spawn the tasks through a `c08` traversal. Spawning the tasks colour by colour should imply that at least tasks of the same colour can be processed in parallel. As will be seen below, however, this did not result in a good scalability either.

In order to overcome these restrictions, the `Quicksched` library was developed in the context

of the code SWIFT[3] [42]. Stemming from another N-body code, Quicksched implements all the necessary features, which would be needed by ls1 mardyn. In particular, apart from dependencies between tasks, the library also supports "conflicts" between resources worked on by tasks. This means that two tasks accessing the same resources (e.g. the force storage of a cell) can be executed in any order, as long as they are not executed simultaneously. This resolves the issue with OpenMP's tasking model described in the previous paragraph.

Moreover, Quicksched supports further features, which are useful for N-body simulations. One such feature is assigning weights to the tasks, which can be useful to distinguish between cells with few molecules and cells with many molecules. Another useful feature is that the graph of dependencies is constructed explicitly and in advance of the calculation. This allows additional scheduling optimizations, such as calculation of the critical path of the graph and taking memory locality into consideration when distributing the tasks among the threads. Furthermore, if threads happen to have an unequal load towards the end of the calculation, work-stealing is used to rebalance the work. Finally, hierarchical dependencies among the tasks are supported, which, while not needed for the short-range force calculation, will be needed for the multithreaded parallelization of FMM in Section 12.2. Both OpenMP and POSIX threads are supported.

7.5.1 Parallelization schemes

In [44] several variants with Quicksched were developed for the force calculation: **qui_pair**, **qui_111**, **qui_222**, **qui_333**, **qui_444**[4]. They aim at achieving different levels of granularity of the tasks, by grouping different numbers of cell pairs. The calculation was implemented by specifying conflicts between tasks working on common cells. The number of FLOPs performed was used as a measure of the task weight, although further measures are possible [99]. The OpenMP variant of Quicksched was used for better interoperability with the rest of the code, which is OpenMP parallelized.

qui_pair In this variant, one Quicksched task is spawned per cell pair combination. There are, thus, around $N_c \cdot 14$ tasks in total, where N_c denotes the number of cells. Every 27 cells share a resource, however, which means that about $\frac{1}{27}$ of the tasks can be run in parallel. This gives around $\frac{N_c \cdot 14}{27} \approx \frac{N_c}{2}$ tasks that can run in parallel.

qui_111 This variant is a more coarse-grained variant of **qui_pair**: it generates one task for every 8-pack of cells. Thus, **qui_111** exploits the compactness of the traversal to improve the ratio of cell-pair calculations to resources locked. A total of 14 calculations are performed on eight locked resources, which is a much higher ratio than in **qui_pair**, where one calculation is performed on two resources. The ratio for **qui_111** is $\frac{14}{8} = 1.75$ is 3.5 times higher than the ratio $\frac{1}{2} = 0.5$ of **qui_pair**. This ratio is a measure of the connectivity and, respectively, complexity of the arising dependency graph. For N_c cells, there are approximately N_c tasks generated. Overall, about $\frac{N_c}{8}$ tasks can run in parallel, analogously to the eight colours sufficing in **c08**.

qui_nmk In an attempt to further coarsen granularity, new possibilities were discovered, which are essentially extensions of the **c08** colouring, see Figures 7.9(a) to 7.9(c). Even further colouring extensions are possible (Figures 7.9(d) and 7.9(e)); their potential advantages are discussed

[3] http://icc.dur.ac.uk/swift/

[4] Note that we have changed the names of the schemes for consistency: **qui_pair**, **qui_111**, **qui_222**, **qui_333** and **qui_444** were named, respectively, **qui1**, **qui8**, **qui_adj_3**, **qui_adj_4**, **qui_adj_5**.

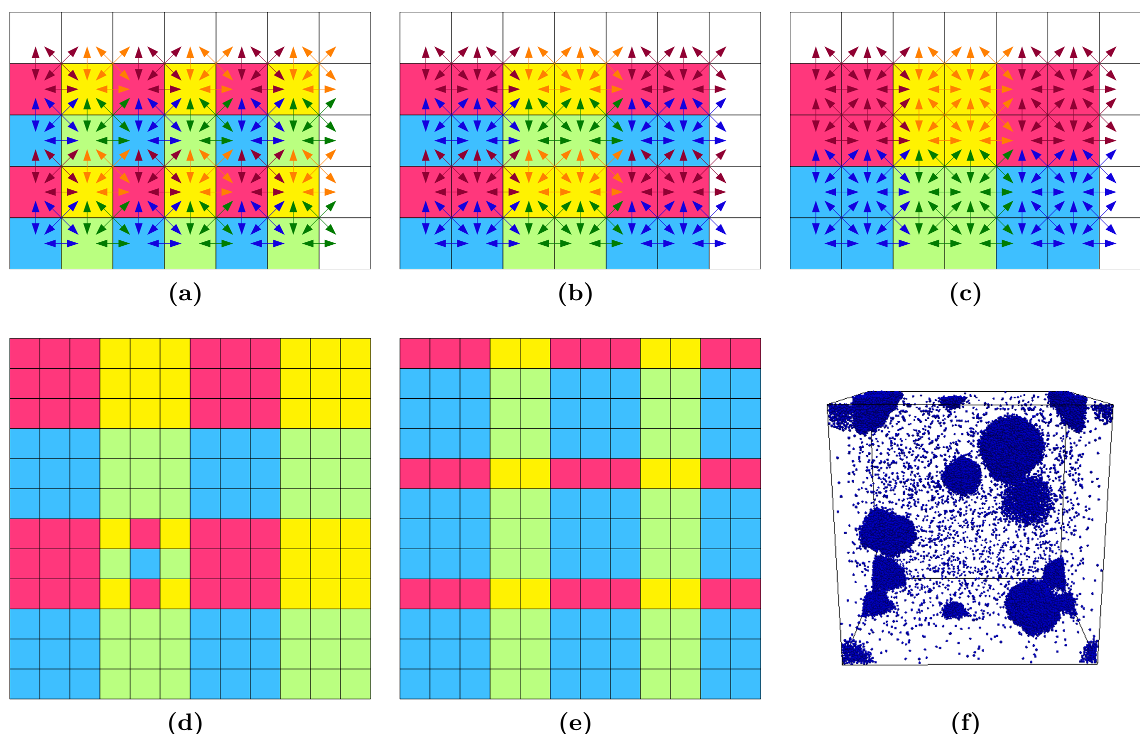


Figure 7.9: Coloured cells in the `c08` scheme can be agglomerated into larger patches of the same colour, without breaking the pattern. No race conditions arise, as long as each patch is processed by an individual thread. (a) Classic `c08` scheme. (b) Agglomerating 2×1 packs of cells together. (c) Agglomerating 2×2 packs of cells together. (d) Patches of sufficiently large size can be adaptively subdivided. (e) Patches of different colour may be of different sizes. (f) Visualization of molecule distribution in inhomogeneous scenario used in Section 7.5.2.

in Section 7.5.3. In the `qui_222` and `qui_333` variants, 8-packs of cells are processed together in agglomerated "patches". One color patch has to be processed by a single thread. `qui_222` agglomerates $2\times$ $2\times$ 2 packs together, while `qui_333` agglomerates $3\times$ $3\times$ 3 packs. This further improves the ratio of calculations performed to resources locked, due to surface-to-volume arguments. For `qui_222` the ratio is $\frac{2^3 \cdot 14}{3^3} \approx 4.15$, while for `qui_333` it is $\frac{3^3 \cdot 14}{4^3} \approx 5.91$. This alleviates the issue of `c08` mentioned in Section 7.3.3 that packs of cells are discarded immediately after use and, thus, improves memory locality. This benefit, however, comes at the cost of a reduced total number of tasks and, respectively, reduced amount of parallelism in the system. The total numbers of tasks for `qui_222` and `qui_333` are $\frac{N_c}{8}$ and $\frac{N_c}{27}$. The fraction of tasks, which can run in parallel, remains constant, however, at about about $\frac{1}{8}$ of the total number of tasks. Thus, about $\frac{N_c}{64}$ and $\frac{N_c}{216}$ tasks can run in parallel for `qui_222` and `qui_333`.

`task_c08` For a comparison, measurements with an OpenMP-based variant, `task_c08`, were also collected. This variant spawns the tasks colour by colour, as mentioned earlier. Overall, execution should, theoretically, proceed very much like in the `c08` traversal, but without the barriers between the colours.

scenario	N	cutoff $\frac{r_c}{\sigma}$	density $\rho\sigma^3$	Temperature
1CLJ	40000	5.00	0.044	0.70
4CLJ	1317006	3.50	0.45	2.87

Table 7.5: Simulation parameters.

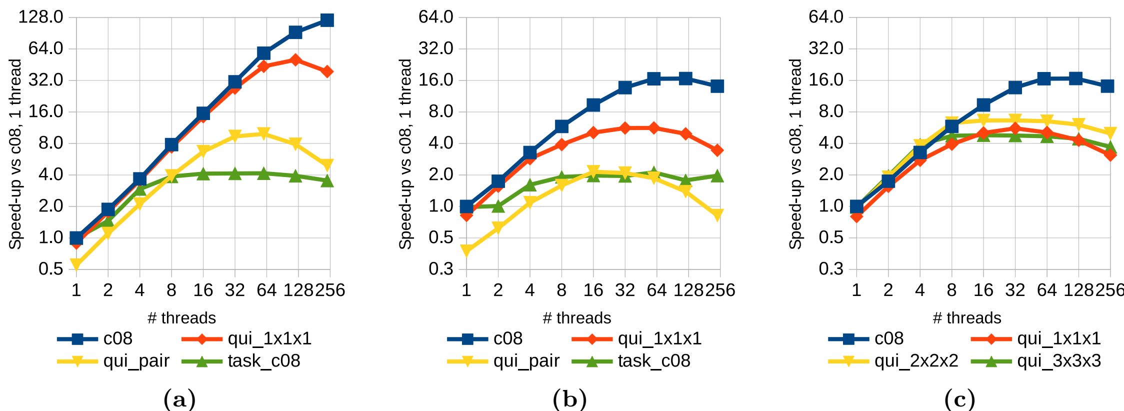


Figure 7.10: Performance of different schemes on KNC, relative to one thread of `c08`. (a) Large, homogeneous 4CLJ system. (b), (c) Small, inhomogeneous 1CLJ system.

7.5.2 Results

We present results for a large homogeneous system and for a smaller, inhomogeneous system, which features clusters of molecules, surrounded by gas. The simulation parameters are given in Table 7.5. The first system was chosen to see how `Quicksched` behaves for a large and "easy" to parallelize system, to see its overall scalability potential. The second system was chosen as a scenario, which should be more favourable for `Quicksched`. The lack of barriers, weighting of tasks and computation of the critical path of the task graph should give it an edge over `c08` in a small load-imbalanced system. A visualization of the molecule distribution for the second scenario is given in Figure 7.9(f). Simulations were executed on Intel Xeon Phi 5110p coprocessors in native mode (KNC).

The results are shown in Figure 7.10. We first comment on the `c08` performance. The performance of `c08` in the large, homogeneous scenario in Figure 7.10(a) was analyzed in Section 7.3.1. Excellent scalability is observed with a total speed-up of 120.39×.

We have not examined the performance of `c08` for inhomogeneous scenarios yet, such as in Figures 7.10(b) and 7.10(c). For this scenario `c08` attains a value of 16.6× at 60 threads and then performance begins to drop. The scenario contains a total of 20×20×20 8-packs of cells, which means $10 \times 10 \times 10 = 1000$ cells per colour or about four cells per thread at 240 threads. While four cells per thread is not a high value, it should still suffice to see a performance rise beyond 60 threads and also higher speed-ups for fewer than 60 threads. The poorer performance should, thus, be attributed to the load imbalance in the system. Even though the use of `dynamic,1` scheduling mitigates load imbalance to an extent, `c08` clearly leaves something to be desired. A possible explanation is that the 8-packs of a colour are processed in an order, stemming from their spatial positioning in the three-dimensional grid of cells. If some, but not all, threads encounter 8-packs with a high workload shortly before the "end" of the domain, the remaining threads will have to wait and cannot proceed on to packs of the next colour, due to the OpenMP

barriers between the colours. Thus, the penalty due to the load-imbalance manifests itself up to eight times because of the eight barriers of the `c08` scheme, explaining the loss in performance.

Next, we comment on **`task_c08`**. For the homogeneous system in Figure 7.10(a), it actually shows moderate scaling up to eight threads: a speed-up of 2.93× is attained at four threads, but it levels off at about 4× from eight threads onwards. For the inhomogeneous system in Figure 7.10(b), it shows no speed-up at two threads and stagnates at only 2× for more threads. Clearly, even after (at least theoretically) resolving the serialization issue, performance is far from competitive to either `c08` or the `Quicksched` variants.

Considering **`qui_pair`** in Figures 7.10(a) and 7.10(b), the curves scale somewhat, but begin at a low performance on one thread. Apparently, too many and too small tasks are spawned, which leads to a considerable sequential overhead.

The scheme **`qui_111`** shows a very good scaling up until 60 threads in Figure 7.10(a). The curve begins at 0.89× of the `c08` performance, but reaches 43.64× at 60 threads. In the hyperthreading range, however, performance deviates from `c08` considerably. Ultimately, **`qui_111`** attains 50.42× compared to `c08` at one thread. Compared to the 120.39× attained by `c08`, it can be said that `c08` becomes 2.39× faster, which is a considerable margin.

In the inhomogeneous scenario in Figure 7.10(b), `qui_111` begins at 0.82× and reaches 5.63× at 60 threads. Up until around four threads, scalability is moderate. At four threads, it is at 2.86× of the `c08` performance at one thread.

In Figure 7.10(c), the more coarse schemes `qui_222` and `qui_333` are compared. `qui_222` scales visibly better than `qui_111`. It begins at 0.96× on one thread and even slightly outperforms `c08` for low thread counts, giving 6.26× at eight threads, where `c08` is at 5.82×. It is, thus about 15% faster on four threads and about 7% faster on eight threads. After that, performance levels off from 16 threads onwards. `qui_333` begins at 0.98× and is actually faster than both `qui_222` and `c08` until four threads. At four threads, `qui_333` is 19% faster than `c08`. After that, performance levels off from eight threads onwards at about 4.8×. The trend continues with `qui_444` (not plotted) — it begins higher at one thread, exhibits highest performance until two threads and then levels off earlier — at four threads. The values are 1.00× at one thread, 1.99× at two threads (14% faster than `c08`) and a levelling off at 3.2×. Overall, the best speed-ups are attained with `qui_222`— up to 6.64×. Compared to the attained 16.69× by `c08`, `c08` is again about 2.51× faster.

7.5.3 Conclusion and outlook

Several task-based schemes were investigated and compared to the loop-based `c08` variant in this section. While `c08` is overall fastest, multiple interesting conclusions can be drawn. Although `task_c08` showed disappointing performance, extensions provided by later OpenMP standards seem promising and the scheme can be revisited. Since OpenMP 4.5, specifying priorities for the tasks is also possible. This can be used in a fashion similar to the weighting of `Quicksched` so that heavier tasks are processed earlier and lighter tasks can be used to balance the workload better. But most importantly, with the introduction of reduction operations to tasks with OpenMP 5.0, great performance improvements can be expected.

The experiments with `Quicksched` provided multiple insights, especially as it sometimes outperformed `c08` for low numbers of threads. This illustrates that the approach has potential. The first observation is that there is a sequential overhead, related to the total number of tasks — spawning too many or too small tasks is observed to be undesirable (such as in `qui_pair`). This is somewhat in contrast to loop-based parallelism, where more iterations usually lead to better performance. Increasing the task size too much, however, reduces parallelism. Further analysis in [44] on the scheduling of the tasks over time showed clearly that the levelling off of

performance for e.g. `qui_222` and `qui_333` is because there are only a few tasks left in the queue, which have conflicts among them and can not be executed simultaneously. The investigations also revealed a bottleneck in the work stealing, which was reported to the developers of the library.

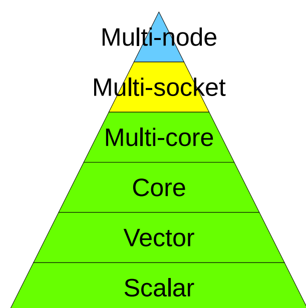
A balance between task size and count must, therefore, be achieved. We devised schemes, with which one can achieve arbitrary granularity of the tasks (e.g. **`qui_nmk`**), which greatly helps in this direction. The autotuning approaches described in Section 9.5 could be used to automatically select the best task size. When the right balance between task granularity and thread count is struck, `Quicksched` has the potential for parallel efficiencies of over 90% even in load-imbalanced scenarios. In those cases, the additional features such as task weighing and critical path analysis allowed it to outperform **`c08`**.

We now point out that further experimentation with the grouping of work into tasks could be of interest. The illustrated extensions of the colouring in Figures 7.9(d) and 7.9(e) provide some ideas for optimization of the **`Quicksched`** variants. For example, for load-imbalanced scenarios, one can begin from a coarse blocking of e.g. 3×3×3 cells in a patch (as in **`qui_333`**) and subdivide patches, which contain a high computational load, as illustrated in Figure 7.9(d). This can be used to equalize the load among the tasks and, respectively, threads, in order to achieve a better balancing. Moreover, patches with multiple cells, which contain few molecules, are kept together, which improves the memory reuse precisely for those cells, for which it matters most. Furthermore, this should also reduce the cases where only a few, large patches remain in the queue, which block each other's execution due to resource conflicts. On the other hand, the colouring illustrated in Figure 7.9(e) can be used to create tasks of unequal load from the start. In this case, the larger tasks will be scheduled first and it is guaranteed that there will be plenty of smaller tasks to balance out the workload at the end. This could, potentially, speed-up execution for homogeneous scenarios. Finally, we point out that one could take the agglomeration of colour patches all the way to consume one or even two dimensions. This would reduce the dependencies in the graph considerably.

It should also not be forgotten that `Quicksched` supports greater generality than needed in the use case presented here. First, the code outside the force calculation can be parallelized with tasks, which provides more work to be distributed among the threads and, potentially, removes further OpenMP barriers. Moreover, the library features support for hierarchical locking of resources, which is intended for tree-based methods such as Barnes-Hut or FMM. We will make use of this latter feature to provide a thread-level parallelization for our FMM implementation Section 12.2. The library is, hence, of interest and has been integrated in the production trunk of `ls1 mardyn`.

While **`c08`** provided the best performance overall, it leaves room for optimization for load-imbalanced scenarios. First, the block sizes can be varied in the same fashion as for `Quicksched`, as illustrated in Figures 7.10(b) and 7.10(c). This can reduce the scheduling overhead for homogeneous scenarios and potentially further improve performance. Next, the subdivision approach of Figure 7.9(d) can be applied similarly to `Quicksched` for the same reasons.

Drawing further inspiration from `Quicksched`, one can also implement a prioritization of the patches according to their weights for **`c08`** and **`cfb`**. Like in `Quicksched`, the patch weights can be computed and the tasks can be sorted by decreasing workload. A list of indices is then created and parallelized over with the `schedule(dynamic,1)` scheduling. In this fashion, the big tasks are processed first and the small ones are used to even out the workload imbalances. As molecules move slowly, the list of indices and, respectively, workloads can be rebuilt only once per e.g. 100 or 1000 iterations. Thus, improvements of **`c08`** can be hoped for for both homogeneous and inhomogeneous scenarios.



8 Multi-Socket-Level Performance

In this chapter, we investigate performance on multi-socket systems in more detail. We present the shared-memory optimizations that were carried out in our work [108]. The goal is to obtain maximal memory efficiency, in order to determine the maximal number of molecules, which can be simulated on entire supercomputers such as SuperMUC or Hazel Hen. In order to reduce the memory overhead for internal MPI buffers and halo cells, the number of MPI ranks should be minimized in the hybrid MPI×OpenMP execution. We are, hence, aiming at efficient execution with one rank per dual-socket node. Thus, NUMA considerations become important as we want to access the memory available from both sockets from a single process. As introduced in Section 2.2.5, threads experience penalties for accessing the RAM from foreign sockets, which the OpenMP schemes should address as best as possible.

Designing NUMA-aware OpenMP schemes has several implications. The data that each thread works on, should be placed as close as possible to the CPU that the thread is pinned on. For this reason, care must be taken during the initialization, which should now be done in parallel. Following the first-touch policy, threads should allocate and initialize the data they will be working on. Moreover, the patterns of access of each thread should be predictable and each thread should aim to always access the same data. “Data exchange” between threads should be minimized, i.e. the intersection of the memory regions accessed by each two threads should be small, not only during the force calculation, but for the entire program. Last, but not least, synchronization between threads should be minimized, as synchronizing threads lying on different sockets is more expensive.

While `c08` was already observed to perform well in dual-socket configurations (e.g. on Ivy Bridge and Broadwell in Section 7.3.2), in [108] two new schemes were introduced, which further improve performance for NUMA execution. In this chapter, we present these two schemes.

8.1 Low-synchronization schemes: `sli` and `c04`

In [108] the schemes `sli` and `c04` were introduced for the force calculation, which have in common that they both reduce the synchronization costs. The `sli` scheme aims for maximal NUMA efficiency by taking an approach similar to the domain decomposition one. The `c04` scheme is another colouring variant, which brings the number of colours and, respectively, synchronization steps, in three dimensions down to four. This is a twofold reduction from the eight steps of `c08`.

The `sli` scheme When running with T threads, this scheme cuts the domain into T slices of approximately equal size. It can be considered as a one-dimensional domain decomposition, see Figure 8.1(a). In order to prevent race conditions at the boundaries between the slices, OpenMP locks are used. At the beginning of the force calculation, thread t sets a lock, which serves to protect the first layer of cells from being written on by thread $t-1$. As soon as thread

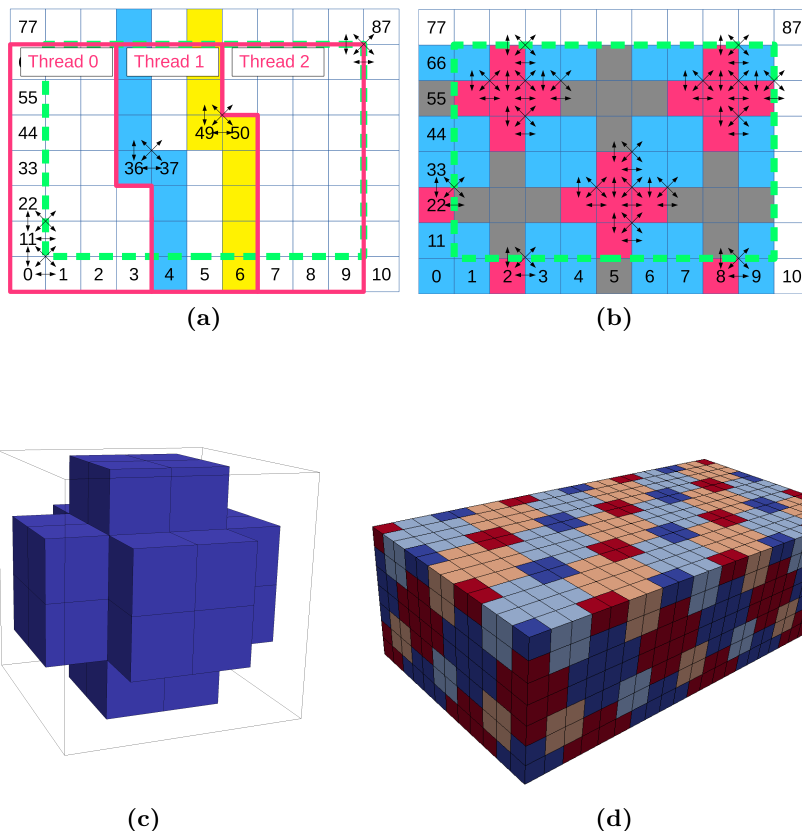


Figure 8.1: (a) The **sli** scheme with three threads. Thread 1 works on the cell packs between cells 36 and 50. At the start of the calculation, it sets a lock to protect its cells from being accessed by thread 0. As soon as thread 1 processes all cells in the blue region (after cell 37), it releases the lock so that thread 0 can complete its calculation. When thread 1 reaches the yellow region (cell 49), it needs to access cells from thread 2's slice for the first time and, hence, waits to obtain the lock from thread 2. Assuming a homogeneous distribution of the workload and more than two layers of cells per thread, thread 2 should have released the lock already. Thread 0 has a slightly lower workload than the other ones, because its region contains a lot of halo-cells (outside the green dashed square), forces between which are not computed. (b) The two-dimensional **c03** scheme. By going for larger patches, this scheme reduces the number of colours needed in two dimensions from four to three. (c) One patch (building block) from the three-dimensional **c04** scheme. It contains 32 cells within a 4^3 cube of cells, arranged in a +-like fashion. (d) The **c04** scheme. It can be regarded as two interwoven Cartesian grids of building blocks. Each of the grids is coloured in the familiar "red-black" fashion known from parallelization of the Gauss-Seidel algorithm.

t is done processing the first layer of cells, it releases the lock. As soon as thread $t-1$ reaches a region, at which it will touch cells in the region of thread t, it waits in a blocking fashion to obtain the lock from thread t.

Since every thread has a well-defined region to work on, the `sli` scheme is NUMA-friendly. In every iteration, every thread always accesses the same cells in the force calculation. Furthermore, the `ParticleIterator` and initialization of particles were modified so that each thread works in the same region as in the force calculation, to increase NUMA-awareness even further. Some operations cannot be thread-parallelized without cross-access, however. For example, applying boundary conditions in parallel on the lower and upper boundaries in Figure 8.1(a) is not possible without cross-access.

The `sli` scheme is clearly very coarse-grained. It requires at least two layers of cells along the dimension along which the decomposition is done. For a small scenario and a large number of threads, this may be prohibitive. For large scenarios, however, or small numbers of threads, this is not a problem and ample work is provided, which can hide the synchronization costs well. Moreover, this scheme is cache-friendly and NUMA-aware, which makes it an excellent candidate for the (extra large) scenarios computed in [108].

This scheme is generally not load-imbalance tolerant. However, if the domain is very large, the boundaries between threads can be shifted somewhat, which — depending on the scenario being computed — may improve load-balancing. The "data exchange" of the scheme is constant at two layers of cells per thread. Depending on the length of the domain, this may be either a very small or a very large fraction.

The `sli` scheme has the advantage of being the least synchronization-intensive of all schemes discussed in this work. In `sli`, each thread must only wait to obtain one single lock from another thread, which is cheaper than an OpenMP barrier. As all other schemes require at least one barrier or multiple locks, we regard this scheme as optimal in terms of synchronization.

Another advantage of `sli` is that the molecular data passes through the CPU only once. In the colouring variants, on the other hand, it passes through the CPU as many times as the number of colours in the scheme. These multiple traversals reduce the overall arithmetic intensity of the program, as the same number of floating-point operations are performed in multiple passes, which decreases vectorization gains.

The `c04` scheme In [108] we also introduced another family of colour-based parallelization schemes. While the `c08` family uses 2^D colours in D dimensions, this family reduces the number of colours down to $D+1$. It was inspired by the extensions to the `c08` colourings presented in Figure 7.9 and the additional observation that increasing the size of color patches (or building blocks) may be used to disconnect patches, thereby reducing the number of colours.

First, the `c03` variant was constructed in two dimensions, see Figure 8.1(b). It is an extension of the known red-black colouring of the Gauss-Seidel algorithm with the third colour used to break diagonal links. The reduced number of colours comes at the cost of the scheme being more coarse-grained, since each patch is 4–5 times larger than in the two-dimensional version of `c08`. The larger patches, however, improve memory reuse as well. A feature, which makes the scheme somewhat more difficult to implement, are the differing forms of the patches and multiple possibilities for intersection with the bounding box of the domain.

The `c04` scheme is an extension of the `c03` variant to three dimensions. The building blocks of the four colours are uniform and shown in Figure 8.1(c). Each block consists of 32 cells within a cube of $4^3 = 64$ cells, excluding all cells, which touch the edges of the cube. Again, this improves data reuse, at the cost of making the scheme more coarse-grained. The building blocks form two interwoven Cartesian grids, each coloured in an alternating red-black fashion. The centers of the building blocks are essentially arranged on a body-centered cubic lattice. Similarly

scenario	model	N	cutoff $\frac{r_c}{\sigma}$	density $\rho\sigma^3$	Temperature
homogeneous	1CLJ	46787312	3.5	0.78	0.000317
vapour-liquid	1CLJ	351805	3.0	0.6223 & 0.06482	0.95

Table 8.1: Simulation parameters.

to `c03`, a difficulty arises from describing the iteration and handling all possible intersections with domain boundaries. For `c04` the problem is more pronounced because the number of edge- and corner cases increases with the dimension.

As in `c08`, we implemented `c04` with a `schedule(dynamic, 1)` scheduling. This makes it more load-imbalance tolerant at the cost of predictable access patterns and NUMA-awareness.

8.2 Results

Experiments were conducted to compare the performance of the **`sli`** and **`c04`** schemes to the **`c08`** scheme. The simulations were carried out on SuperMUC Phase 1 and Hazel Hen supercomputers. The hardware used was Intel SandyBridge-EP Xeon E5-2680 (SNB) and Intel Haswell Xeon E5-2680 v3 (HSW). Up to 32 (hyper-) threads are available on SNB and up to 48 on HSW. Turbomode on SuperMUC Phase 1 is disabled by default. On Hazel Hen it was switched off for the measurements in this section, in order to get clean values of the parallel performance. The **`RMM`** mode of the production trunk of `ls1 mardyn` was used[1]. Single precision was used, together with AVX/AVX2 on SNB/HSW. Results for the entire simulation are shown.

Two molecular systems were simulated. The first system is a large, homogeneous one. It is only slightly larger than the minimal (cubic) system required to run 48 threads in the **`sli`** mode — it contains 112×112×112 packs, which gives 2.33 layers of cells per thread. The second system is a smaller, inhomogeneous one in a setup frequently used by `ls1 mardyn` — vapour-liquid equilibrium simulations (see e.g. [24]). This system consists of a liquid region and a vapour region in equal volumes, see Figure 8.3(a). The system contains 28×55×28 linked cells. The density in the liquid region is 0.62, while in the vapour region — 0.065. Table 8.1 provides the rest of the simulation parameters.

Figure 8.2 shows results of a strong-scaling measurement for the homogeneous scenario. In the used **`RMM`** mode, the scenario takes less than 2 Gigabytes of memory, which fits in the RAM of one socket. Default thread placement was used, which means that threads get placed first on one socket. Threads get placed on the second socket only when running more than 16 threads on SNB and more than 24 on HSW.

Expectedly, the **`sli`** scheme gives near-perfect performance for the homogeneous scenario: the parallel efficiency at 16/24 threads on SNB/HSW is 97/94%. For this reason, it was used in hyperthreading experiments with two threads pinned on the same core (HT curve). On both SNB and HSW, gains of about 23% are observed due to hyperthreading. These values were used to normalize ideal expectations in the hyperthreading range. Overall, the parallel efficiency of **`sli`** is 99%/97%/96% and 97%/94%/93% on SNB and HSW, respectively, on the maximal number of cores on one socket, on two sockets and on two sockets with hyperthreading. These values are a considerable achievement for any strong scaling experiment. Between 8 and 16 threads and between 12 and 24 threads, where NUMA accesses emerge, the performance drop is one of the smallest drops observed, compared to the other schemes. This is because of the

[1] A slightly modified version of the production trunk, available for download here: `http://www5.in.tum.de/mardyn/WR_2017-Release.tar.gz`

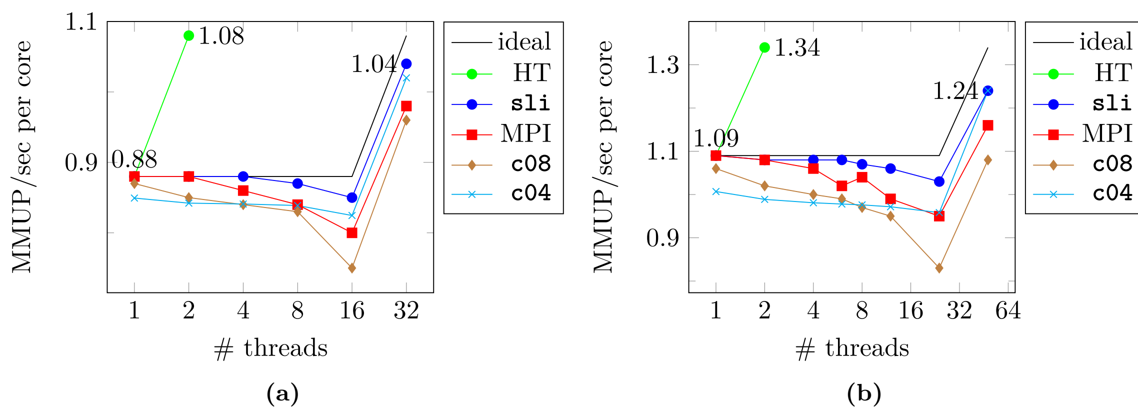


Figure 8.2: Strong scaling experiment for homogeneous scenario. Performance is given in MMUP/sec per compute core. One thread per core is used, except for the HT set and the measurements with 32/48 threads on SNB/HSW, where two threads per core are used. Measurements with two threads pinned on the same core (HT) were used to extrapolate ideal performance behaviour in the hyperthreading regime (at 32/48 threads). (a) Results on SNB. (b) Results on HSW.

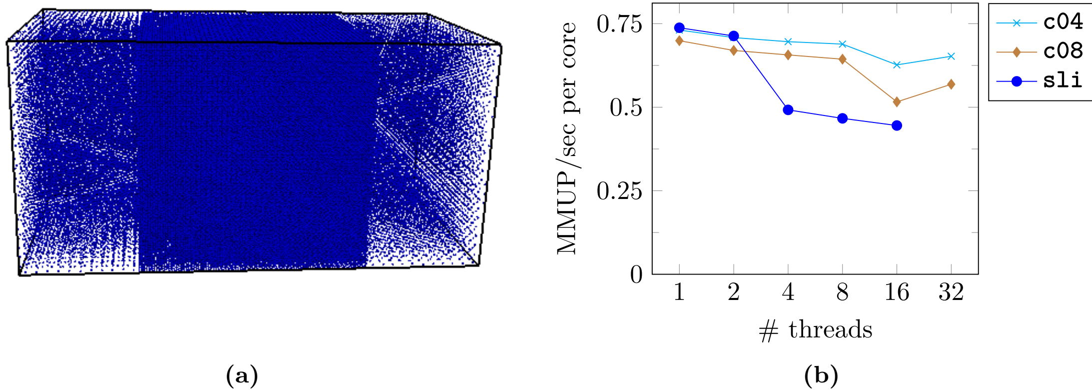
NUMA-awareness of the scheme. Finally, it is notable that `sli` manages to outperform the pure MPI version for all numbers of threads, which is also a significant achievement.

`c08` performs slightly poorer. Compared to `sli`, there is a sequential overhead of 1%/3% on SNB/HSW, due to the better memory locality of `sli`. Parallel efficiency is at 94%/85%/89% and 87%/76%/81%. The performance drop at 16/24 threads is notable — this is where NUMA accesses emerge and this scheme is not NUMA-friendly. At 32/48 threads, performance recovers by up to 5% because hyperthreading helps to mitigate memory accesses.

`c04` has a notable sequential overhead at 3%/8% on SNB/HSW. This is likely due to the complex description of the `c04` traversal and building blocks. The multiple possible intersections of the complex building blocks with domain boundaries require a considerable amount of `if`-statements, which could be a cause for the performance drop. However, the scheme features the best scalability, as evident from the flatness of the `c04` curves. The scalability is even retained when cross-socket accesses emerge at 16 and 24 threads, where `c08` experienced a large drop in efficiency. This is probably due to the larger building blocks, which need to fetch new data much less frequently than `c08`. `c04` is, thus, also a good choice for NUMA-execution. The excellent scalability of `c04` ultimately gives the second-best performance after `sli` on 32/48 threads on both SNB and HSW.

Figure 8.3(b) shows the performance results for the inhomogeneous scenario. `sli` gives the best performance for one and two threads, as cutting the domain along the longest dimension gives equal load in both halves of the domain. There is a big drop in performance between two and four threads, as then two threads work in the liquid domain and two threads work in the less computationally intense vapour domain. From four to 16 threads scalability (versus four threads) is again good, as the workload in the liquid domain gets subdivided equally. The scheme could not be applied for 32 threads, because that leaves less than two layers of cells per thread. Cutting along the x- or z-direction would lead to perfect load balance, but would further decrease the number of threads that can be used.

`c04` and `c08` experience a small sequential overhead. The overhead is about 1% for `c04` and about 5% for `c08`. The penalty is larger for `c08`, because the large fraction of vapour volume is not computationally intense and, hence, influenced considerably by memory-locality, which is



(a) **(b)**

Figure 8.3: (a) Visualization of inhomogeneous, vapour-liquid scenario. (b) Strong scaling experiment for inhomogeneous scenario on SNB. Performance is given in MMUP/sec per compute core. One thread per core is used, except for the measurements with 32 threads, where two hyperthreads per core are used.

poor for `c08`. The larger building blocks in `c04` improve data reuse and, hence, the penalty is small, despite also the overhead of the complex traversal.

Between one and eight threads, `c04` and `c08` scale quite well, as indicated, again, by the flatness of the curves. The parallel efficiency at eight threads is 93% and 87% for `c04` and `c08` versus `sli` at one thread. This is much higher than the observed 73% in the load-imbalanced scenario of Section 7.5.2. The reason for this is that only vapour is present towards the "end" of the traversal. The longest dimension is y, and the dimensions are (always) traversed in the order x, then y, then z, with x being the innermost loop. Under this ordering, apparently no load-imbalances arise towards the "end" of the traversal and, hence, the `schedule(dynamic,1)` scheduling allows the threads to finish at roughly the same time, resulting in good speed-up and efficiency values. This is also the reason why the difference between `c04` and `c08` remains nearly constant in this range at about 6%. The difference in performance should be attributed only to the better data reuse, but not the smaller number of colours. Since the load-imbalances are hidden, the reduction of barriers from eight to four does not affect performance so much in this range.

At 16 threads, NUMA-accesses emerge. `sli`, being NUMA-aware, experiences no significant drop between 8 and 16 threads. `c04` performance drops by 8.4%, while the `c08` performance drops two times more, by 17.3%. This should be attributed to both the better data-reuse and the two times smaller number of colours. In the hyperthreading range at 32 threads, `c04` gains 4% additional performance (vs its performance at 16 threads), while `c08` gains 10%. As in Figure 8.2, hyperthreading mitigates the NUMA-accesses to an extent, which is more beneficial for `c08`, due to its poor data-reuse.

Overall, `c04`, `c08` and `sli` deliver maximal values of 10.44, 9.10 and 7.12 MMUP/sec, respectively. Thus, `c04` outperforms `c08` by 15% and `sli` by 47%, which represents considerable gains.

8.3 Conclusion

In this chapter, two further schemes were presented, which improve performance for multi-socket execution. The **sli** scheme is NUMA-friendly and gives near-optimal performance for homogeneous scenarios. It outperformed the hitherto most convenient and performant **c08** scheme by 15%–24% in the homogeneous scenario. It features the lowest synchronization costs of all schemes investigated in this work. This comes at the cost of being the most coarse-grained scheme, requiring at least two layers of cells per thread along the longest dimension of the domain. The current implementation aborts for less than two layers of cells per thread, although a change of the implementation to leave idle threads is also possible and may be beneficial in some cases.

c04 also showed excellent scalability and overall performance in NUMA execution environments. It requires a low number of colours and features a higher reuse of cell data, which proved beneficial when the domain contains low-density regions and when cross-socket memory accesses emerge. It was observed to deliver up to 15% more performance than **c08** in both simulated scenarios. As a possible extension, a memory-buffer variant of **c04** was discussed in [108]. Analogous to the relation between **c08** and **cfb**, such a scheme would be able to work with at most four memory buffers per cell.

8.4 Outlook

As we have seen in Section 8.2, both the **sli** and the **c04** schemes show excellent scalability. They both have drawbacks, however, which restrict their applicability or impose overhead. In this section we outline extensions to the schemes, which should remedy their drawbacks. As already **sli** and **c04** showed excellent performance, the extensions are very promising.

Slicing along multiple dimensions As discussed in the previous section, the main drawback of the **sli** scheme is the very coarse granularity, resulting from the decomposition only along one dimension. In Figure 8.4(a) we propose an extension, which can circumvent this restriction. The scheme, which we dub **sli_blk**, makes use of two OpenMP locks per thread: a “horizontal” and a “vertical” one. Analogically to **sli**, they are exchanged only to the “previous” and “next” neighbours in, respectively, the horizontal and vertical directions. The own locks are exchanged to the “previous” neighbours (locks `my_H` and `my_V`), while the neighbours’ locks (locks `nb_H` and `nb_V`) are obtained from the “next” neighbours. The threads exchange locks in order to protect regions of the domain, they currently work on, from being accessed by neighbouring threads. The four combinations of locks each thread holds during an iteration — (`my_H`, `my_V`), (`my_V`, `nb_H`), (`nb_H`, `nb_V`) and (`nb_V`, `my_H`), in this order — serve to protect the blue, green, yellow and red quadrants, respectively, from being accessed by other threads. Locks are released as soon as possible and obtained as late as possible. In other words, locks are released as soon as the boundary to the lock recipient has been processed and obtained only when no other work is left, but the boundary to the next lock’s former owner. The order of processing of the coloured blocks is inspired by the Hilbert space-filling curve [58], which minimizes the number of times locks change hands. Moreover, the use of the space-filling curve ordering gives an easy way to extend to higher dimensions. A Hilbert ordering could also be used inside of a thread’s domain, within and across the coloured quadrants, which gives well defined and spaced-out stages for setting and unsetting the locks.

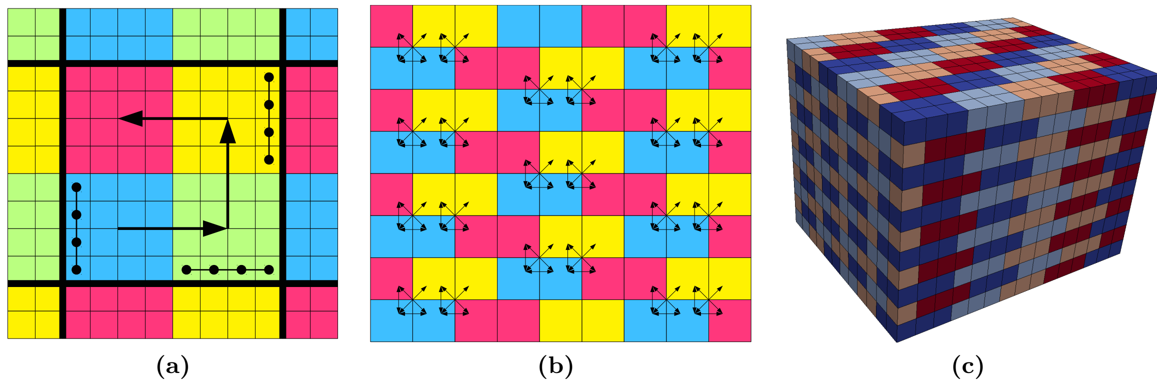
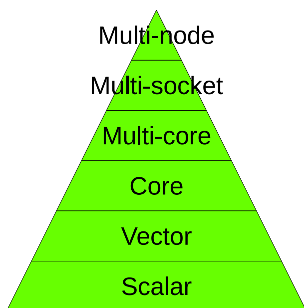


Figure 8.4: (a) The **sli_blk** scheme. Thick black lines mark the boundaries of a thread's domain. Threads begin at the blue block and set locks my_H and my_V. After the cells marked with circles are processed, my_H is unset and threads proceed with the rest of the blue region. After the blue region is finished, threads wait to obtain the lock nb_H from the right neighbour. The process continues with the green, yellow and red blocks, in this order. (b) Two-dimensional **c03_hcp** colouring scheme. (c) Three-dimensional **c04_hcp** colouring scheme.

A practical family of colourings with $D+1$ colours While also quite promising, the main drawbacks of **c04** are the complex traversal and shape of the building blocks. In Figures 8.4(b) and 8.4(c) we outline another family of colourings with $D+1$ colours in D dimensions. The grid formed from the centers of the blocks resembles a hexagonal close packing, hence, the names of the schemes. The patches are $2{\times}1$ in two dimensions and $3{\times}2{\times}1$ in three dimensions. They are coloured cyclically along the first dimension. Along the second dimension, the pattern from the first dimension is taken and a shifted, so that blocks of the same colour do not touch (shift by $D+1$ cells). In the same fashion, along the third dimension, the two-dimensional pattern is taken and a shifted so that blocks of the same colour do not touch (shift by $(D+1,-1)$ cells). The scheme should extend to D dimensions by taking blocks of size $D \times (D-1) \times \ldots \times 1$ and periodic tilings with periods of $D+1, D, \ldots, 2$. Finally, we point out that these are the minimal block sizes. If desired, the data-reuse can be increased by enlarging the block sizes, similarly as for **c08**.



9 Putting it all Together

Our optimizations throughout Part III have led to a significant increase of the node-level performance of `ls1 mardyn` and Linked Cells-based codes in general. In this chapter we present further results from [108] in order to quantify the performance increase and also determine how `ls1 mardyn` fares in comparison to other codes.

Our work of [108] can be seen as a follow-up to the work of [27]. In the latter paper, `ls1 mardyn` was used to set a world record for the largest simulation of a molecular system in 2013. In that work, strong and weak scaling experiments were performed up to the full SuperMUC Phase 1 machine. At the limit of the weak scaling, $4 \cdot 10^{12}$ molecules were simulated at up to 591 TFLOP/sec, setting a world record in the process. While the performance was excellent, several bottlenecks were identified. Among them was the use of 128-bit intrinsics, AVX being still a nascent technology at the time. Another limitation was that running too many MPI ranks (16 per node) caused a visible memory overhead, thereby, reducing the number of simulated molecules by about 25%. Having addressed the former bottleneck through SIMD wrappers and the latter one through the different schemes for OpenMP parallelism, the question of how many molecules can be simulated efficiently was readily revisited. The runs for the largest simulation were planned for the Hazel Hen machine, which features more than three times more RAM than SuperMUC. In order to isolate code improvements from hardware improvements, however, extensive benchmarks on SuperMUC were also conducted. As in this chapter we draw frequent comparisons to the code and simulations of [27], we will abbreviate the work as WR13.

In Section 9.1, we present the newly attained hybrid MPI×OpenMP performance. Next, a comparison to other state-of-the-art molecular dynamics software is provided in Section 9.2 as validation of the performance of `ls1 mardyn`. After that, we illustrate the achievements of this work by comparing to a benchmark on eight nodes from WR13 in Section 9.3. Finally, we present the scalability on the full supercomputers SuperMUC Phase 1 and Hazel Hen and draw comparisons to WR13 in Section 9.4. In Section 9.5 we outline how our work laid the foundation for a promising, new library for N-body kernels. Section 9.6 summarizes this chapter.

scenario	model	N	cutoff $\frac{r_c}{\sigma}$	density $\rho\sigma^3$	Temperature
Lennard-Jones fluid	1CLJ	up to $2 \cdot 10^{13}$	3.5	0.78	0.000317
LAMMPS benchmark	1CLJ	512 000, 524 288	2.5	0.84	1.4
TIP4P	1CLJ3C	512 000	3.2	1.01	4.00
Benzene	6CLJ6Q	$64 \cdot 10^6$, $9 \cdot 10^9$	6.0	0.25	2.563

Table 9.1: Simulation parameters.

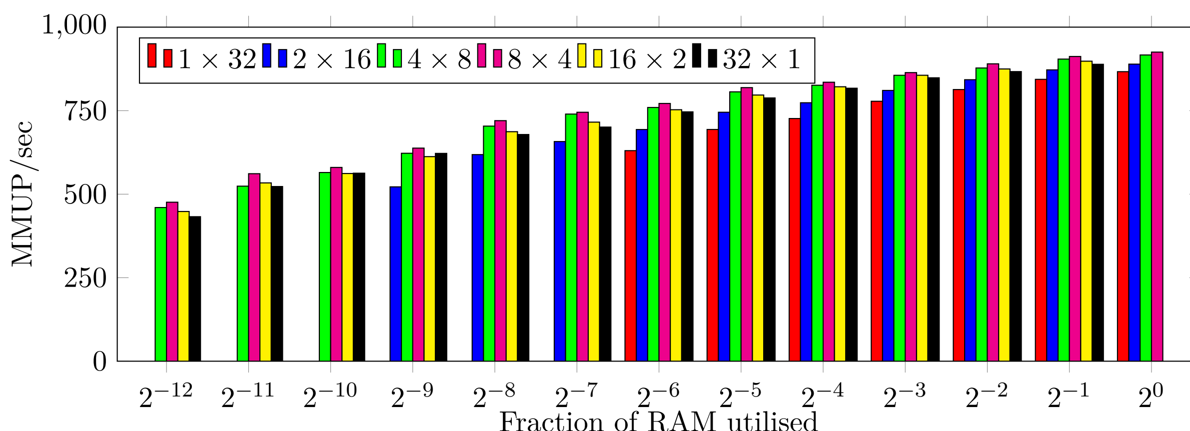


Figure 9.1: Hybrid MPI-OpenMP analysis on 64 nodes of SNB with **sli** for decreasing system sizes. 2^0 denotes 100% RAM utilization, 2^{-1} denotes 50% RAM utilization, etc. $M \times O$ denotes M MPI $\times$ O OMP. Missing values for 1×32 MPI×OMP and 2×16 are due to the **sli** scheme not being applicable for such small systems. Missing values for 16×2 and 32×1 at full RAM utilisation are due to MPI buffers and halo-cells causing out-of-memory exceptions.

Setup Results on four molecular systems will be presented. Table 9.1 summarizes the simulation parameters. The first system is chosen to be directly comparable to the system used for the world record simulations from WR13. It is a dense system of single-site Lennard-Jones molecules at a low temperature. The second system is a known benchmark from the LAMMPS suite[1]. It is used for a comparison to this Verlet Lists-based code. The third system is a system of TIP4P water molecules at the boiling point, used for a comparison to LAMMPS and GROMACS for multi-centered molecules. The fourth system is used to showcase the attainable performance on the whole SuperMUC for complex molecular models such as benzene, featuring six Lennard-Jones and six quadrupole sites.

For the first two systems, single precision and the `RMM` mode are used. For the third and fourth systems, double precision and the `Normal` mode is used. AVX is used on SuperMUC Phase 1, which features Intel SandyBridge Xeon E5-2680. AVX2 is used on Hazel Hen, consisting of Intel Haswell Xeon E5-2680 v3 nodes. Results for the entire simulation are shown with a slightly modified version of the production trunk of `ls1 mardyn`[2].

9.1 Hybrid MPI-OpenMP performance

Figure 9.1 shows the performance of a hybrid MPI×OpenMP study for differing system sizes. System parameters from the first system detailed in Table 9.1 were used. The study was performed on $4^3 = 64$ nodes of SuperMUC. The high number of nodes was selected so that full communication patterns to distinct MPI neighbours arise, also over the periodic boundaries of the simulation box. The number of molecules was varied between $9{\cdot}10^6$ and $36{\cdot}10^9$. The maximal number of molecules represents utilisation of nearly 100% of the available RAM memory. The lower values are representative of the per-node system sizes, that will arise in the strong scaling setting when going from 8 to 9216 nodes. The **sli** scheme was used.

Overall the hybrid performance is very good, as indicated by the flatness of the bars per

[1]LJ benchmark from `https://lammps.sandia.gov/bench.html#lj`

[2]This modified version is available from `http://www5.in.tum.de/mardyn/WR_2017-Release.tar.gz`.

fixed system size. When going from 1×32, the performance rises until 8×4 and then drops down when going to 32×1. The variances are quite small: for the 2^{-1} case (50% RAM utilisation) the performance ranges between 844 and 912 MMUP/sec, which gives a difference of only 8%. Nevertheless, we comment on the shape of the curves.

Considering the excellent OpenMP performance observed in Chapter 8, one would expect the peak of the curve to be more to the left. However, while the **`sli`** scheme is optimized for NUMA execution, not having cross-socket accesses at all is always going to be beneficial. Moreover, when running in pure OpenMP mode, the periodic boundary conditions can be handled in a practically embarrassingly parallel fashion. While the routines necessary for MPI communication — such as packing and unpacking molecules to and from MPI buffers — are generally straightforward for OpenMP parallelization, they intrinsically do not scale so well. Packing molecules into send-buffers requires prefix-sum-like operations, scale like $\mathcal{O}(\log T)$, instead of $\mathcal{O}(T)$, where T is the number of threads. Unpacking molecules from receive-buffers and inserting them into the Linked Cells structure also has a higher overhead, because it requires either OpenMP locks for each cell or additional preprocessing passes. For this reason, as soon as MPI communication is present, OpenMP efficiency inevitably drops somewhat. On the other hand, the packing and unpacking operations scale with MPI: increasing the MPI count for a fixed system size shortens the per-process length of the boundaries. This is why the peak of the performance is shifted more to the right. Despite these effects, however, the maximal performance is attained at 8×4 and it drops down to 32×1, which is also a good result.

Next, we comment on the effects of decreasing the system size. For the smallest system sizes, the system uses about 4 MB of memory per node, which is extremely small — it is less than the cache of one socket (20 MB). Performance is, thus, sustained very well for varying system size: over a reduction of system size by a factor of 4096, performance drops only by a factor of two. This is thanks to the new data structures and design. Further measurements (not shown) suggest that the performance of the force calculation remains fairly constant. This means that the reduction in performance is primarily due to other operations, which do not scale so well. As mentioned, MPI communication is an example, also because network latency plays an ever increasing role with decreasing system and message sizes.

Similar effects were observed on Hazel Hen, both in terms of per-size hybrid behaviour, as well as sustained performance for small sizes [108]. The best performance was attained for the 6×8 configuration, running 6 MPI ranks with 8 OpenMP threads each. The differences at 50% RAM utilisation are at 6%, thus, even better than on SuperMUC. When decreasing system size by a factor of 4096, the performance drop is also smaller: 1.66×. Thus, hybrid performance on Hazel Hen is even better than on SuperMUC, despite the higher number of cores per socket (12 vs 8).

Considering the small differences between 8×4 and 32×1 on SuperMUC (and also 6×8 and 48×1 on Hazel Hen), one might question whether going for OpenMP was worth it. Indeed, given an excellent MPI performance, a hybrid MPI×OpenMP scheme can only achieve moderate speed-ups on scenarios and architectures, on which the MPI scheme already scales well. Going for more OpenMP threads, however, also has other advantages apart from total performance. Memory efficiency is higher, which was one of our main motivations for developing the OpenMP schemes (cf. Chapter 7). Moreover, given a load-imbalance-tolerant scheme such as `c08` and `c04`, one can shift work away from MPI load-balancing, which is generally more difficult to achieve. Finally, the reduced MPI count simplifies MPI communication patterns and stress on the network architecture of the cluster.

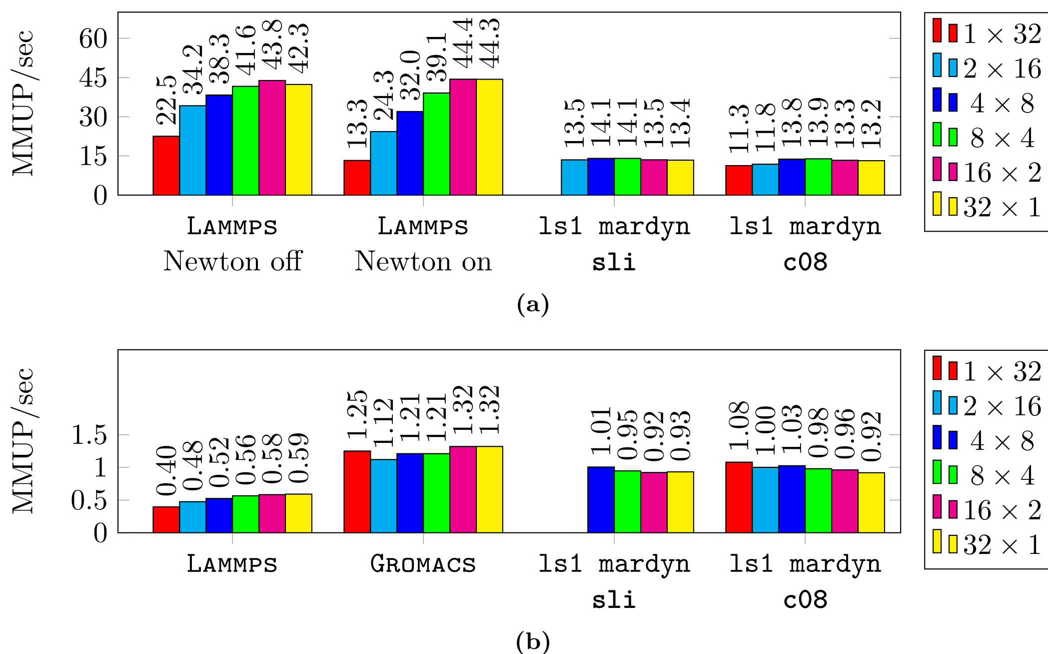


Figure 9.2: (a) Performance comparison between `ls1 mardyn` and LAMMPS USER-INTEL. Hybrid MPI×OMP study on the LAMMPS benchmark. Newton "off" and "on" indicates whether Newton's third law optimization was turned on for LAMMPS. (b) Performance comparison between `ls1 mardyn`, LAMMPS USER-OMP and GROMACS 2016.3d for TIP4P system.

9.2 Comparison to LAMMPS and GROMACS

In this section we present comparisons to LAMMPS and GROMACS, as representatives of well-optimized Verlet Lists-based codes. Figure 9.2(a) shows performance for the second system from Table 9.1. It is a known benchmark for LAMMPS, for which reference performance values are available on different architectures at `https://lammps.sandia.gov/bench.html#accelerator`. The LAMMPS runs were performed with the USER-INTEL optimization package. The stable version from 11 August 2017 was used. The measurement was done on one node of SuperMUC.

In this scenario, LAMMPS clearly outperforms `ls1 mardyn`, as seen from Figure 9.2(a). The difference between the best performance attained by each code is almost a factor of three. The fact that the **`sli`** scheme could not be applied for `ls1 mardyn`, however, indicates that this is a rather small scenario for the current version of `ls1 mardyn`.

We now look at the hybrid MPI×OpenMP performance. The bars are very flat for `ls1 mardyn`, as one would expect from the results in the previous section. The best performance is given for the 4×8 configuration, which is marginally better than 8×4. For LAMMPS, however, performance strongly increases to the right, indicating that the OpenMP performance leaves something to be desired. This is true even for the "Newton off" case, in which the force calculation should be embarrassingly parallel. The difference between "Newton on" and "Newton off" case for LAMMPS is large for many OpenMP threads (1.7×), but almost vanishes for the 16×2 and 32×1 configurations. This suggests that retaining Newton's third law with OpenMP parallelization is also a non-trivial topic for Verlet Lists-based codes.

Further exploration of parameters for the comparison between LAMMPS and `ls1 mardyn` was performed in [108]. The system size, cutoff radius and Verlet Lists-rebuild settings were varied. This investigation found that for cutoff radius 3.5 (which was used for the large runs of WR13) `ls1 mardyn` could fit about 50 times more molecules in the RAM, at the cost of only about

30% performance. Due to the higher memory consumption, LAMMPS would, thus, not be a good candidate for large-scale runs, such as the ones in WR13.

The primary advantage of Verlet Lists-based codes over Linked Cells-based codes is the better hit rate, as discussed in Section 1.6.2. For this configuration of parameters, the hit-rate is 72% for LAMMPS, while only 16% for `ls1 mardyn`. For cheap molecular models, such as the used 1CLJ, evaluating the cutoff condition represents a significant portion of the total number of floating-point operations performed. The primary application area of `ls1 mardyn`, however, is multicentered molecular models, such as the TIP4P water model, containing one Lennard-Jones center and three charges per molecule. For such models the evaluation of the cutoff condition represents a much smaller fraction of the total number of floating-point operations performed. Hence, it can be expected that `ls1 mardyn` performs better for multicentered molecules.

In order to test this hypothesis, a comparison was performed between `ls1 mardyn`, LAMMPS and GROMACS for the TIP4P system from Table 9.1. The comparison was performed in double precision. For LAMMPS, the USER-OMP package was used, which features a specialized kernel for TIP4P molecules. For GROMACS, the 2016.3d release was used, which also features a specialized kernel. For `ls1 mardyn`, the **`Normal`** mode was used. The general-purpose multicentered force calculation was used for `ls1 mardyn`, as the code base does not feature kernel specializations, apart from the **`RMM`** mode for 1CLJ molecules. Care was taken to create comparable simulation scenarios [108]. Features, which are handled in a different fashion by the different codes (e.g. long-range contributions), were excluded from the measurements.

The results are shown in Figure 9.2(b). For this scenario, `ls1 mardyn` now outperforms LAMMPS by a good margin: a factor of 1.83×. A reason is the vectorization developed with multicentered molecules in mind, as outlined in Section 5.3. In particular, the evaluation of the cutoff-condition on a center-of-mass-to-site basis gives an advantage to `ls1 mardyn` in this case. The other Verlet Lists-based code, GROMACS, still outperforms `ls1 mardyn`, but by a small margin: only 22%. One reason for this is the specialization of the force kernels for TIP4P in GROMACS, together with the advanced intrinsics vectorization and clustering of the Verlet Lists, as described in [2]. Clearly, the OpenMP performance of GROMACS is also excellent, as indicated by the small variation of performance among the different MPI×OMP combinations.

For `ls1 mardyn` it is interesting to note that **`c08`** consistently outperforms **`sli`** by a small margin. It is not immediately clear why, as the scenario is homogeneous. A reason could be the small load-imbalance in the halo-layer of the first thread, cf. Figure 8.1.

Clearly, extending the comparison among the three codes for other molecular systems and simulation parameters would be of interest. In particular, this holds for multi-node performance and performance for inhomogeneous systems. However, this would be a considerable undertaking and is beyond the scope of the current work.

9.3 Comparison to WR13 on eight nodes

Before going to comparisons to WR13 on the full SuperMUC Phase 1, we look at a smaller configuration on eight nodes. The low number of nodes and, respectively, MPI ranks, allows to isolate MPI performance to an extent, as network communication for such a low number of nodes should not be a determining factor of performance, as opposed to comparing on thousands of nodes. The new version of `ls1 mardyn` was run in the fastest, 8×4 configuration and the **`sli`** scheme. Thanks to the intrinsics wrappers introduced in Section 5.4 we can easily compare the performance of both the AVX and SSE version of the code. For WR13 available performance data from [27] was used for the first system from Table 9.1. In both versions, FLOP operations were counted with `ls1 mardyn`'s internal counters. System size varies between 4.5 and 575

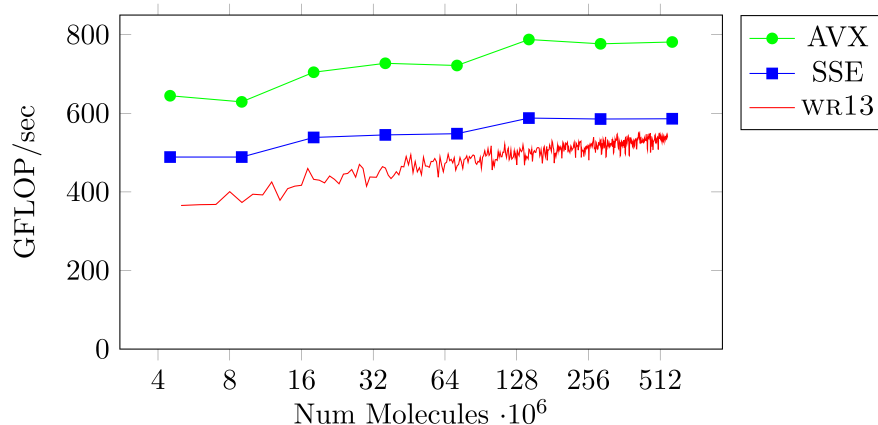


Figure 9.3: Comparison to WR13 nodes on eight nodes of SuperMUC. For SSE and AVX, the system size was varied geometrically between $4.5 \cdot 10^6$ and $575 \cdot 10^6$ molecules. Data for WR13 varies linearly between $5 \cdot 10^6$ and $550 \cdot 10^6$ molecules.

million molecules.

Figure 9.3 shows the results. Our performance with the 128-bit SSE version matches the performance of the 128-bit AVX intrinsics of WR13 well. The difference is only about 7% at large systems. For small systems, however, the difference rises to up to 33%. This means that the new version of the code sustains performance for low particle counts much better than the old one. This should be attributed to the changes in the core data structures, as outlined in Section 4.1. Moreover, it is reasonable to assume that the difference will become even more pronounced for even smaller systems. This is significant, because systems of about ten times fewer molecules per node will come into play in the limit of strong scaling in Section 9.4.

Comparing the AVX performance to the SSE one, AVX consistently gives about 32% more performance than SSE. Thus, comparing AVX to WR13, we observe 43% more performance at high particle counts and up to 74% more at low particle counts as a result of our work.

9.4 Petaflop molecular dynamics: a new world record

In this section we present multi-node scalability results up to the full SuperMUC and Hazel Hen supercomputers. The purpose of weak scaling, generally, is to simulate as large problems as possible. For this reason, one MPI rank per node was used, as it gives the best memory efficiency. The idea of strong scaling, on the other hand, is to simulate problems in the lowest runtime possible, so we selected the hybrid configuration, which minimizes runtime. Thus, we used 8×4 on SuperMUC and 6×8 on Hazel Hen. The first system from Table 9.1 was used with single precision and the `sli` scheme.

On SuperMUC scalability runs were collected up to the full 9216 nodes. On Hazel Hen the runs were performed up to 7168 nodes, which is about 93% of the total 7712 nodes of the machine. Repetitions could not be performed for the weak scaling runs on the full machine, due to the long duration of the runs (more than 30 minutes) and limited amount of time the full machines could be reserved for our measurements. The variations in performance were small, however, so this was not deemed necessary[3]. For strong scaling, as the runtime on the full machine was smaller, they were repeated about 10–11 times and the best runtime was selected. Some fluctuations were, nevertheless, present, despite the repetitions. When running on more

[3]One outlier seems to be present in the weak scaling at 2.7 GHz on SuperMUC with 32 nodes.

than 4096 nodes on SuperMUC, issues with the power supply of the machine emerged. In order to overcome them, the CPU frequency had to be lowered from 2.7 GHz down to 2.3 GHz.

On Hazel Hen, some hardware issues were encountered at large node counts, particularly for the weak scaling simulations. Drops in performance by up to 50% and failing nodes were observed. The cause of these performance decreases was investigated, but could not be pinpointed, due to the difficulty and long waiting times for repeating the measurements on the full supercomputer. These problems illustrate the need for fault-tolerant execution at such scales. For this reason, the measurements on Hazel Hen were repeated several times and the performance of the best five iterations was used.

Weak scaling runs on SuperMUC Figure 9.4(a) shows our strong and weak scaling results in the basic configuration for 2.3 and 2.7 GHz, together with data from WR13. In the weak scaling scenario, the sixteen-fold reduction in MPI ranks allowed us to simulate 27% more particles than before ($5.2 \cdot 10^{12}$). Despite the lower frequency, the simulation proceeded at 0.72 PFLOP/sec, which is 22% higher than the 0.59 PFLOP/sec of WR13. Comparing the weak scaling values for 2.7 GHz and 2.3 GHz at 4096 nodes, we see that the difference is 16.4%, which matches the ratio of CPU frequency of $\frac{2.7}{2.3} \approx 17.4\%$ excellently. At 4096 nodes our 2.7 GHz performance is 44% higher than that of WR13, which matches the expectations from the comparison on eight nodes in Section 9.3 of 43% almost perfectly. Thus, the increase in weak scaling performance should be attributed to the use of 256-bit AVX intrinsics.

Lastly, we compare the observed weak scaling efficiency. When going from 1 to 9216 nodes, efficiency is 87% for the 2.3 GHz curve of the new code. This is slightly lower than the observed 91% of WR13. Looking at the curve of the new code, however, one observes that the drop in performance is mostly between 1 and 64 nodes. This is because at 64 nodes communication to distinct MPI ranks emerges along all boundaries, including the periodic ones. For less than 8 nodes, some boundaries are handled with OpenMP, which gives greater efficiency. On one node, the performance difference is, thus, higher: the 2.7 GHz run is 48% faster than the WR13 one. Next, between 2 and 32 nodes some MPI neighbours appear on more than one boundary, due to the periodic boundary conditions. The new version of `ls1 mardyn` optimizes for this case by merging distinct MPI messages to the same recipient into a single message. This explains why efficiency drops by 10% from 1 to 64 nodes and then only by further 3% from 64 to 9216 nodes. For WR13 no such drop is experienced until 64 nodes, because already on one node 16 MPI ranks are running and the aforementioned optimizations were not yet implemented.

Strong scaling runs on SuperMUC Looking at the strong scaling performance, we were able to obtain 0.55 PFLOP/sec at 2.3 GHz. This is more than double the 0.26 PFLOP/sec obtained by WR13, despite the lower frequency. Considering that good strong scaling performance is more difficult to achieve than good weak scaling performance, and that these values are for a whole supercomputer, this represents one of the strongest achievements of our work.

The increase is primarily due to better performance than WR13 for small system sizes, as discussed in Section 9.3. In Section 9.3 differences in performance by up to 1.7× were observed on eight nodes. Here, the number of molecules per node is about ten times smaller, so even larger margins are expected. As discussed in [108], some MPI improvements were also performed, which also manifest themselves most strongly at the limits of the strong scaling. However, at least 1.7× of the 2.1× are due to the optimized data structures. Comparing the 2.7 GHz and 2.3 GHz scenarios at 4096 nodes, additional 16.5% are obtained from the frequency scaling, which again matches the value of $\frac{2.7}{2.3} \approx 17.4\%$ well. If we, thus, upscale the value of 0.55 PFLOP/sec, we obtain an estimate of 0.64 PFLOP/sec for a full run at a frequency of 2.7

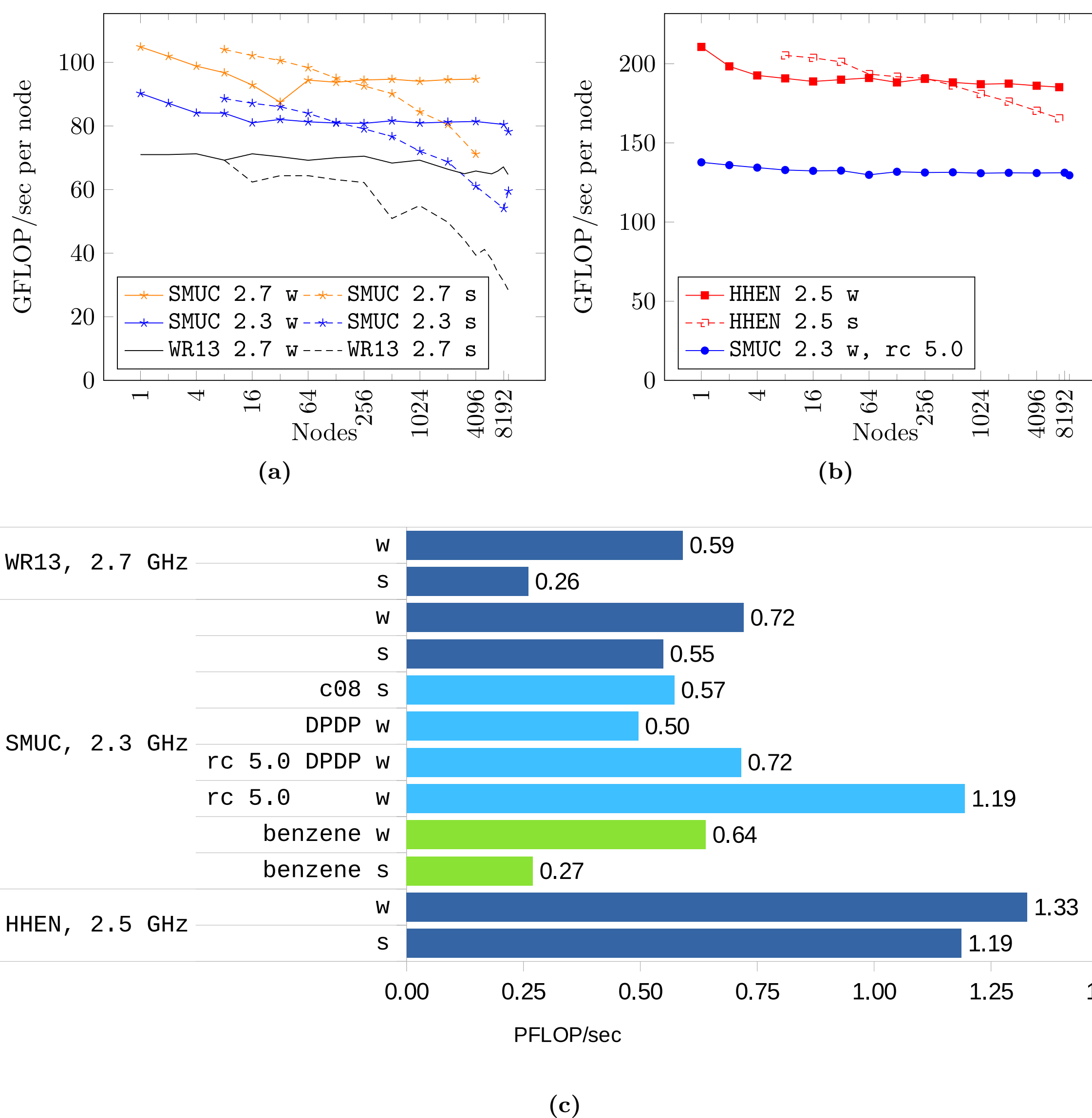


Figure 9.4: Scalability experiments on up to 9216 nodes on SuperMUC Phase 1 (`SMUC`) and 7168 nodes on Hazel Hen (`HHEN`). "w" denotes weak scaling setup, "s" denotes strong scaling setup. (a),(b) Performance per node. The numbers after the cluster name denote the CPU frequency. For example, `SMUC 2.7` denotes the run on SuperMUC at 2.7 GHz. (c) Performance on maximal number of nodes. Bars in dark blue indicate simulations for single-center Lennard-Jones molecules, in **`RMM`** mode, single precision mode SPSP, for a cutoff radius $r_c = 3.0\sigma$ with the **`sli`** OpenMP scheme. Bars in light blue explore how performance for the same base configuration is affected when the scheme is changed to **`c08`**, when double precision DPDP is used and when cutoff radius $r_c = 5.0\sigma$ is used. Bars in green show performance for the benzene system, in **`Normal`** mode, double precision.

GHz, which is even 2.5× higher than the WR13 value of 0.26 PFLOP/sec. Finally, the strong scaling efficiency when going from 8 to 9216 nodes is 67%, while it was 41% for WR13.

Further runs on SuperMUC Next, we discuss further measurements performed on SuperMUC, which were allowed by the greater performance and flexibility of the code attained through our work. A full weak scaling simulation was performed for a cutoff radius of 5.0σ, see Figure 9.4(b). With this run, a performance of 1.19 PFLOP/sec was obtained at a parallel efficiency of 94%. This amounts to 21% of the theoretical peak performance of the machine, after accounting for the lower frequency. Figure 9.4(c) shows the performance of some more simulation configurations on 9216 nodes. It was observed that `c08` outperformed `sli` mildly in the strong scaling scenarios starting from 4096 cores. The reason is not immediately clear: at 4096 nodes, the domain still has 17^3 cells, which should give ample work for four threads in the `sli` scheme. Nevertheless, `c08` performed between 4 and 10% better than `sli` between 4096 and 9216 nodes. Next, the `RMM` mode now can also be combined with the double precision DPDP format. Two (weak scaling) runs were performed for cutoff radii 3.5 and 5.0. We were able to simulate $3.2 \cdot 10^{12}$ molecules in double precision at up to 0.72 PFLOP/sec, which represents 26% of the double precision peak performance, after accounting for the frequency. This would also be a record for number of molecules simulated in double precision.

Before looking at the Hazel Hen results, we comment on the performed measurements for the benzene system. The weak and strong scaling runs proceeded, respectively, at 637 TFLOP/sec and 270 TFLOP/sec. This amounts to 23% and 9%, respectively, of the double precision peak performance, after accounting for the lower frequency. One could consider the performance of the strong scaling run to be relatively low. This run was performed at the limits of the scheme, however, where each thread has about one 8-pack to process per colour (the `c08` scheme was used in a 2×16 MPI×OMP configuration), which explains the low value. The simulations proceeded at 706 and 248 MMUP/sec, respectively. This means that the strong scaling run proceeded at almost four iterations per second, which enters the realm of what can be computed in practice. Thus, these runs demonstrate that we can now utilise a high fraction of the supercomputer's peak performance not only for single-center molecules in the specialized `RMM` mode, but also for realistic simulations with complex molecules, such as the ones used for the production simulations of `ls1 mardyn`. Keeping in mind that for multicentered molecules `ls1 mardyn` fares well also compared to some of the fastest MD codes (cf. Section 9.2), it can be said that these are some of the fastest, large-scale MD simulations of complex molecules.

Results on Hazel Hen Finally, we comment on the performance observed on Hazel Hen. For the weak scaling 1CLJ setup, a performance of 1.33 PFLOP/sec was obtained. This represents about 9% of the single precision peak performance of the machine. The value is lower than on SuperMUC, because while AVX2 doubles theoretical peak performance of the CPU, it increases `ls1 mardyn`'s performance only by a small margin, cf. Chapter 5. Parallel efficiency was at 88% and, similarly to results on SuperMUC, the drop in efficiency is mostly until 64 nodes, when a value of 91% is attained. The molecular systems contained up to $2.1 \cdot 10^{13}$ molecules, which is a fivefold increase in terms of total number of molecules simulated to date. The simulation proceeded at a rate of up to $1.89 \cdot 10^{11}$ MUP/sec.

The strong scaling experiment was started from 8 nodes, as for SuperMUC and WR13. It contained $2.4 \cdot 10^{10}$ particles. Performance of up to 1.19 PFLOP/sec was obtained for up to 81% parallel efficiency. This is, again, an excellent parallel efficiency for strong scaling experiments on the full cluster. The simulation proceeded at a rate of $1.78 \cdot 10^{11}$ MUP/sec. This means that simulations containing billions of particles can be executed efficiently for longer periods of time.

The simulations were performed in dimensionless units. If one dimensionalizes the largest system with $2.1 \cdot 10^{13}$ for xenon atoms with $\sigma = 3.9450$ Å, the simulated cubic box becomes of side-length 11.8 μm and diagonal of $11.8 \cdot \sqrt{3} = 20.4$ μm. This enters the range of the diameter of human hair 17–181 μm[4]. While this falls outside of the range of smallest objects visible to the naked eye[5] (55–75 μm), it is only a factor of two away. Thus, MD simulations for objects large enough to be seen should soon become a reality.

9.5 Foundations for a new, autotuning library

Throughout Part III, we saw on multiple occasions that there is no "silver bullet" OpenMP scheme or configuration, which would give optimal performance for all simulation parameters. For example, memory-buffer schemes perform best for small scenarios, `c08` performs best for intermediate ones, `sli`— for large scenarios and perhaps a `Quicksched`-based one for inhomogeneous ones. Moreover, Verlet Lists might be the more appropriate for single-centered molecules, while Linked Cells— for multicentered ones. Considering further also vectorization modes, precision and the fact that many of the configurations have tunable parameters, the total number of possible configurations becomes quite large, see Figure 9.5. Lastly, which configuration performs best may change during the course of a simulation as the distribution of particles changes, for example in nucleation scenarios [82].

A serious question, then, arises: should a code choose one scheme, which performs reasonably well across all scenarios, or should it support many schemes? The majority of codes take the former approach, as the latter one requires a considerable effort in software design and maintenance. We took the latter approach in order to maximize performance for a wider range of applications. If the code supports many configurations, however, a second question arises: how to select the best configuration among all available ones? As we saw in Part III, the answer to this second question is not always obvious even to the developers of the code, so some experimentation and testing must be performed.

What alleviates the problem, is that the distribution of particles changes slowly during a simulation. This means that once the best configuration has been determined, it will continue to perform well over the course of several hundred or several thousand iterations. This is the idea underlying load-balancing schemes in MD: balance the load, run for example for 1000 iterations, balance the load again, run for another 1000 iterations and so on [15,98]. This idea, together with the flexible software design, which resulted from the incorporation of multiple OpenMP schemes and the `Normal` and `RMM` modes, laid the foundations for and inspired the creation of a new library, called `AutoPas` [46][6].

`AutoPas` aims to deliver optimal node-level performance for short-range pairwise kernels not only in MD, but potentially also for other N-body problems. It is a templated, C++ library, intended to be used as a container for the particles (similarly to STL containers, such as `std::vector<T>`). Through autotuning the library automatically tries out multiple configurations, selects the one delivering the lowest time per iteration and uses this configuration for a fixed number of iterations. Hardware-specific solutions for different models of CPUs or GPUs are also envisaged, giving portability of the performance. In this fashion, the users do not need to concern themselves with the choice of the algorithm, which makes it user-friendly. Through iterator classes, efficient, OpenMP-parallel access to the particles from outside is also available.

Another advantage of `AutoPas` is sustainability of the software. The algorithms, performing

[4] https://hypertextbook.com/facts/1999/BrianLey.shtml
[5] https://en.wikipedia.org/wiki/Naked_eye#Small_objects_and_maps
[6] https://github.com/AutoPas/AutoPas

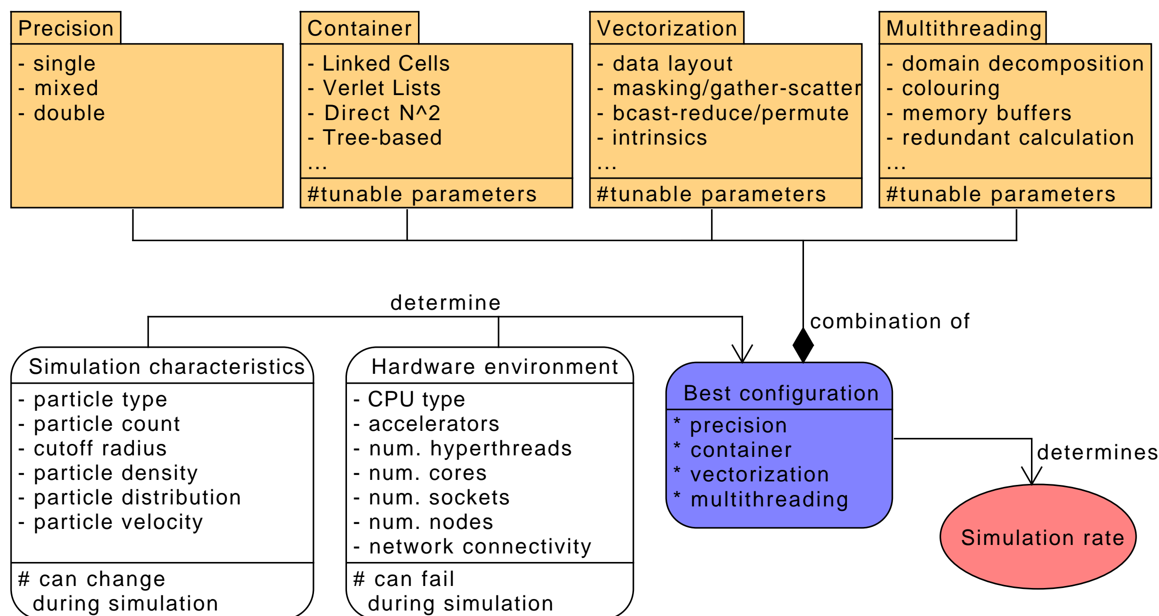


Figure 9.5: The choice of precision, particle container, vectorization mode and multithreading algorithm depend on the simulation parameters and execution hardware. Many of the options further have tunable parameters such as cell size for Linked Cells, skin radius for Verlet Lists, patch size for `c08`, etc. All of these variables influence which configuration will perform best and, hence, the rate at which the simulation proceeds.

best on today's hardware may not be the same as the ones performing best on tomorrow's hardware. It can, thus, happen that an algorithm gets developed, but discarded for some reason, only to be reimplemented several years later. This was the case with `c18` for example, which was implemented in [15], only to be lost and reimplemented again in [109]. In theory, a good software design of application codes can allow to keep multiple solutions in the code base for later reevaluation on new hardware or new setups. In practice, however, it is questionable how much effort the developers would be willing to invest in maintaining features, which are currently not used by the users of the code. A library, heavily oriented at interchangeability and automatic selection of the best algorithm, is, thus, better suited for such a purpose. In this fashion, `AutoPas` is very promising, as it can potentially provide optimality, portability, maintainability and user-friendliness for a wide range of applications and users. Early results in [46] are already providing evidence to the viability of the approach.

9.6 Summary

In this chapter we have seen that our optimization work in Part III has enabled a significant increase in the performance and flexibility of the code. The novel OpenMP schemes allowed efficient hybrid MPI×OpenMP execution, as seen in Section 9.1. The best performance was delivered by 8×4 or 6×8 configurations, while the 1×32 and 1×48 configurations were only within a few percent of them, which is evidence of the good OpenMP performance.

Important comparisons to LAMMPS and GROMACS were performed in this chapter, which serve to relate the performance of `ls1 mardyn` to that of other high-performing and well-established codes. While for some scenarios these Verlet Lists-based codes outperformed our Linked Cells-

based one by considerable margins, for others our code was on par and even better in some cases. Memory efficiency was clearly in favour of `ls1 mardyn`, however, enabling the large-scale runs in Section 9.4.

The comparison to WR13 on eight nodes in Section 9.3 demonstrated that the redesign of the data structures and intrinsics wrappers improved performance considerably. Gains of up to 1.74× were observed for small systems and up to 1.43× for large systems.

Next, big improvements in performance were observed when going to the large-scale runs in Section 9.4. The high OpenMP thread count in the weak scaling scenarios allowed an increase in 27% in the number of simulated molecules on the same hardware as WR13. On the whole supercomputer, 1.43× more performance was attained in the weak scaling scenarios and 2.1–2.5× in the strong scaling scenarios. Performance of over 1 PFLOP/sec at up to 94% parallel efficiency on SuperMUC was also demonstrated. The introduction of different precision modes allowed also the simulation of $3.2 \cdot 10^{12}$ molecules in double precision for the first time, at up to 26% of the theoretical peak performance of the supercomputer. Furthermore, it was demonstrated that utilisations of 10-20% of the supercomputer's peak performance can also be achieved for complex molecules in double precision with the full functionality of the code, at speeds comparable to those required for production execution.

On Hazel Hen runs with up to $2.1 \cdot 10^{13}$ molecules were performed at 80% parallel efficiency in both the strong and weak scaling scenarios. This is a fivefold increase in the number of molecules that have been simulated to date and the simulated molecular system is almost visible with the naked eye.

Last, but not least, this work laid the foundations for a new, powerful library with potential gains for both users and developers of N-body simulation codes and with the promise for considerable savings of computational resources.

PART IV

FAST MULTIPOLE METHOD

In this part we present our work on algorithmic improvements of `ls1 mardyn` by introducing an FMM implementation into the code. As explained in Chapter 3, a new implementation was developed. For this reason, in Chapter 10 we, first, introduce the basic implementation. This chapter then verifies the convergence of the implemented algorithm and its runtime complexity. Next, Chapter 11 presents our work sequential on accelerations of the algorithm. The choice of basis functions is discussed and our acceleration of the M2L phase is presented, together with comparisons to the established `ExaFMM` code. Finally, in Chapter 12 we present our work on the parallelization of the algorithm. Experiments on up to 256 threads are presented for the shared-memory implementation. A novel optimization is presented in the distributed-memory parallelization and scalability on up to 32768 MPI processes is shown. Comparisons to `ExaFMM` are also presented as a means of relating our attained performance to that of known HPC implementations.

10

Basic implementation

As discussed in Section 3.10, FMM is a complex algorithm and reusing externally developed implementations is not always possible. For `ls1 mardyn` in particular, the code and its use cases rely strongly on multicentered molecules and complex electrostatic potentials, such as the dipole and quadrupole ones. These need to be treated with care by an FMM implementation, in order to provide correct results. Moreover, the code is often used to simulate inhomogeneous particle distributions, which require a careful MPI load-balancing [98,99]. Combining the `ls1 mardyn` MPI load-balancing with an external FMM library's MPI implementation would not be easy, because the distribution of particles to MPI ranks has implications about which rank holds which multipole cells, which is information needed to construct the higher levels of the tree. Lastly and most importantly, `ls1 mardyn` aims for maximal implementational efficiency, which is also best achieved through an internal FMM implementation. For these reasons, a new implementation of FMM for `ls1 mardyn` was begun.

As the implementation is developed from scratch, in Section 10.1 we outline the core algorithm, used as a starting point for optimizations and integration in `ls1 mardyn`. The first question to be addressed is the correctness of the computed forces and potentials, as well as the correctness of the complexity of the algorithm. These are shown in Section 10.2 and Section 10.3, respectively. In doing this, we also illustrate and comment on some of the basic FMM relationships, which have implications for the design and execution of the algorithm.

10.1 Implementation

Expansion storage Our implementation is based on the one outlined in [93]. It uses the form of the multipole expansions introduced in Section 3.3. The storage of real and imaginary parts (recall $\mathbf{M}_{l,m} = \mathbf{M}^c_{l,m} + i\mathbf{M}^s_{l,m}$) is done in an SoA fashion, with the ^{c} entries consecutively in memory and the ^{s} entries as well (ccc…sss…). Other authors suggest also interleaving the two is possible (cscscs…) [10]. Following [93], only the $m > 0$ entries are stored. The entries for $m < 0$ are computed on the fly via the symmetry relations of Equations (3.11) and (3.12), when needed. This is again in contrast to [10], where also the $m < 0$ entries are stored.

Translation operators The shifting expansions for the translation operators M2M, M2L and L2L are created up to order p, i.e. the same order as the multipole and local expansions. This is also how it is done in [69,93,121]. Some authors create the expansions for shifting up to order $2p$, however [65]. This increases the accuracy of the M2L translation, but would also considerably increase the computational cost. The four loops in the M2L translation of Equation (3.9) are nested in the `l-m-l'-m'` order, with `l` being the index of the outermost loop and `m'` being the index of the innermost loop. This means that the outer two loops iterate over the destination expansion, while the inner two loops iterate over the expansion, which is being translated and

the translation expansion. Since the involved operations are associative (additions), however, the loops may also be reordered [10].

Tree structure The FMM implementation is an LBT one. This means that a uniform octree of the desired depth is created. The cells of the different levels lie one after the other in a contiguous array. A basic adaptivity for inhomogeneous particle distributions is supported by skipping M2L operations, for which either of the underlying cells contain no particles, following the original implementation of [93]. It is not a very potent approach, however, as the cells usually need to contain multiple particles for efficient execution. The intended (future) extension is to skip M2L operations for cells, which contain *few* particles and to handle them either directly or via M2P and P2L operations as mentioned in Section 3.9.

Acceleration and parallelization Different approaches were tried out for sequential acceleration of the M2L phase. They will be detailed in Chapter 11. The shared-memory parallelization is discussed in Section 12.2 and the distributed memory parallelization is discussed in Section 12.3. In this chapter we look at the unoptimized $\mathcal{O}(p^4)$ M2L calculation and the sequential algorithm.

Outside code We point out that our implementation was developed first outside of `ls1 mardyn` and was integrated afterwards. This was done to reduce the complexity of developing the new functionality and integrating it in the existing software design at the same time. As part of this work, [38,62,88] were conducted in the outside code, while [39,45,83] were conducted in `ls1 mardyn`. In this chapter we present work done in the outside code.

Since after integration in `ls1 mardyn`, the P2P phase was able to utilise the extensive intrinsics wrappers and SoA structures, no effort was spent in optimizing the P2P phase of the outside code. The numbers, which depend on P2P performance, are, hence, intended to illustrate the intrinsic FMM relationships and should not be taken as absolute performance indicators (e.g. in Section 10.3.1).

10.2 Convergence

In this section we verify the correctness of the implementation. Section 10.2.1 verifies the correctness by comparing the forces computed via FMM to the forces computed via a direct N^2 calculation. In Section 10.2.2 we calculate the potential for a known lattice of charges, for which reference estimates are available.

10.2.1 Comparison to direct N^2

We now present an analysis of the convergence of the forces in dependence of the truncation order p of the multipole and local expansions. Multiple measures for evaluating the error on the forces in a system of N particles can be used [113]. Similarly to [25,103], we consider the relative root mean square (RMS) error of the forces. The relative RMS error δF_{rms} is defined as follows:

$$\delta F_{rms} = \frac{\Delta F_{rms}}{F_{rms}} = \sqrt{\frac{\sum_{i=1}^{N} |\vec{F}_i^{FMM} - \vec{F}_i^{direct}|^2}{\sum_{i=1}^{N} |\vec{F}_i^{direct}|^2}}. \tag{10.1}$$

In Equation (10.1), $|\vec{F}_i|^2$ denotes the squared norm of the force vector $|\vec{F}_i|^2 = F_x^2 + F_y^2 + F_z^2$ on particle i. $\vec{F}^{FMM}$ denotes the FMM obtained value and $\vec{F}^{direct}$ the value obtained from a

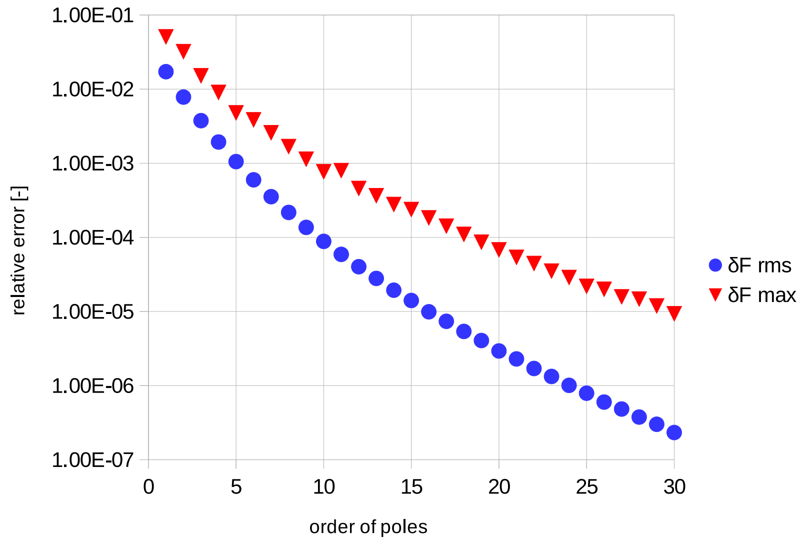


Figure 10.1: Dependence of force RMS error on truncation order p of the expansions. System contains 12288 particles in a tree of depth 4 ($8^4 = 4096$ cells in the finest level; about three particles per cell). Particles are uniformly distributed in a cube. Charges are randomly assigned to values of ± 1 with an equal probability. The curves for the force begin at order $p = 1$, because the order $p = 0$ approximation gives a constant function approximation within the local cells, whose derivative is identically zero.

direct N^2 calculation. Of interest is also the maximal error in the system δF_{max}. It is, again, defined in relation to the RMS of the force [103]:

$$\delta F_{max} = \frac{\Delta F_{max}}{F_{rms}} = \sqrt{\frac{\max_{i=1,\ldots,N} |\vec{F}_i^{FMM} - \vec{F}_i^{direct}|^2}{\sum_{i=1}^{N} |\vec{F}_i^{direct}|^2}}. \tag{10.2}$$

Figure 10.1 shows the results for a system of 12288 uniformly distributed particles with randomly assigned charges of value ± 1. As expected, the error decays quickly with increasing order of the multipoles. For applications in MD, typical requirements are that the RMS error of the force is below 10^{-4} or 10^{-5} [25,91]. In our implementation, this falls in the range of order $p \in [10, 16]$. This is in agreement with [70], where typical truncation orders for MD are said to range between 8 and 30.

The ratio between δF_{max} and δF_{rms} can be considered to be quite large. It begins at a factor of 3 for $p = 1$ and reaches a factor of 40 at $p = 30$. This is in agreement with other results [103]. Indeed, the topic of error analysis in FMM is quite complex. The errors depend on multiple factors, including the distribution of particles within the cells [65] and the depth of the FMM tree [103]. Further analysis of the errors falls outside of the scope of the current work.

10.2.2 Computation of the Madelung constant

A well-known test case for the correctness of long-range electrostatic schemes is the calculation of the Madelung constant [25,65]. It is named after E. Madelung, who investigated it in [76]. The problem consists of evaluating the potential energy on a positive charge in an infinite lattice of alternating ± 1 charges. Figure 10.2(a) illustrates the two-dimensional lattice. In [65] the following values of the Madelung constant are summarized:

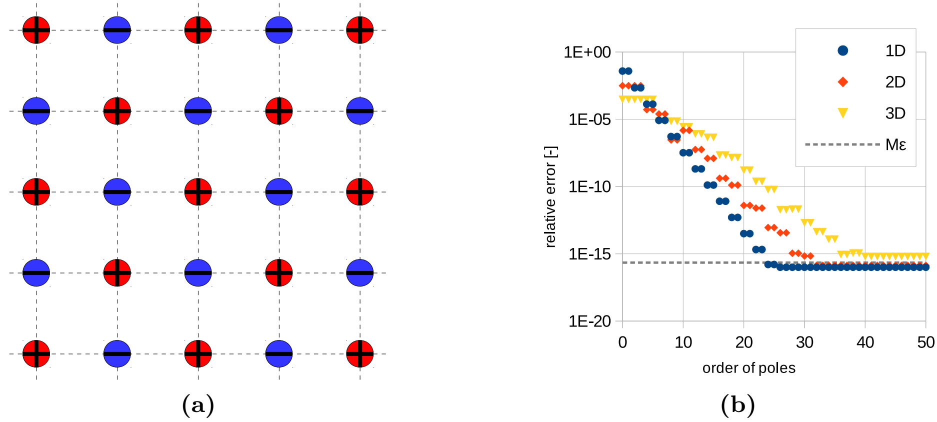


Figure 10.2: (a) Infinite lattice of alternating charges for computation of the Madelung constant in two dimensions. (b) Relative error in the potential for computing the Madelung constant with FMM of increasing order of the poles. $M\varepsilon$ denotes the machine precision $M\varepsilon = 2^{-52} \approx 2.22 \cdot 10^{-16}$. A value of $N_{iter} = 50$ was used.

- 1D: -1.3862943611198906...,
- 2D: -1.6155426267128247...,
- 3D: -1.7475645946331821....

Similarly to [65], the addition of periodic boundary conditions was done via the method of [67]. It works in the following fashion. Up to level 1 of the tree, which contains four cells in two dimensions, the multipole expansions are simply copied periodically. This creates a halo layer in the standard fashion, with the exception that the halo layer has a thickness of two cells, instead of one. If the simulation volume is $[0, L]^3$, then in this fashion the contribution of the volume $[-L, 2L]^3$ can be accumulated.

For treating level 0 of the tree (one cell) and above, a special periodic operator is computed, similar to an M2L translation. This operator is computed once at the initialization of the program, for a user-specified number of iterations N_{iter}. Every iteration has the effect of roughly tripling the simulation box along each dimension, resulting in an exponential increase in the number of periodic images. This approach also allows to switch periodicity on or off along individual dimensions, which is a desirable feature for MD simulations.

Figure 10.2(b) shows the obtained results for the potential in dependence of the order of the expansions p. The relative error in the potential is plotted:

$$\left| \frac{U_{Ma} - U_{FMM}}{U_{Ma}} \right|,$$

where U_{Ma} denotes the analytic value and U_{FMM} denotes the FMM approximation. The method converges well, reaching machine precision easily. It can be observed that our convergence rate is slightly lower than that of [65]. There, the one-dimensional system reaches machine precision at order 17, while we reach it at order 22. For the three-dimensional system, [65] attains it at order 23, while we attain it at order 36. A possible explanation for this is that they perform the M2L operation with a translation expansion up to order $2p$, while we perform it with a translation expansion up to order p. Nevertheless, we also reach machine precision and consider that the differences are not a cause for concern.

Another observation is that the curves exhibit a somewhat “stepwise” nature. For example, in the 1D data points, it can be observed that the odd orders (corresponding to dipole, octupole, ... contributions) do not decrease the error. This effect is also present in [65]. The reason for this is probably the high symmetry of the systems. To see this, consider in one dimension a local expansion, centered at the positive charge at $x = 0$ and capturing the influence of all other charges. Due to the symmetry to the left and right, the potential function, approximated by this local expansion must be an even function satisfying $f(x) = f(-x)$. The expansion terms of odd order, however, all represent odd functions, satisfying $f(x) = -f(-x)$. Hence, their coefficients in the local expansion are zero and they do not improve the approximation. For two- and three dimensions even more contributions are (nearly) zero, due to even stronger symmetry constraints. This leads to wider “plateaus” which, respectively, lower the convergence rate, explaining also the slower convergence for two and three dimensions.

10.3 Complexity

While the theoretical complexity of FMM is known to be $\mathcal{O}(N)$, achieving the linear scaling depends on a careful choice of the parameters. As already mentioned in Section 3.7, minimizing runtime depends on a proper balance between the P2P and M2L phases. In Section 10.3.1 we illustrate the need for this balance and motivate the accelerations of the M2L phase in Chapter 11. In Section 10.3.2 we demonstrate the linear complexity of the implementation.

10.3.1 Balancing the P2P and M2L phases

The first question, which the user faces when using FMM, is how to select the depth of the tree. As there are no characteristic length scales in the calculation of this type of electrostatic interactions, the depth of the tree is free to vary and choose. The choice of the depth of the tree affects runtime drastically, however. Figure 10.3 illustrates the dependence of FMM runtime on tree depth. If the tree is too shallow, the P2P phase dominates runtime. If the tree is too deep, however, the M2L phase dominates. One must, therefore, find the trade-off between the two phases for the setup at hand.

The question of finding the balance comes down to whether it is faster to interact two groups of particles directly via P2P or indirectly through pseudoparticles via M2L. Interacting them through P2P is a N^2 operation and, hence, depends primarily on the number of particles in the group (or cell). Interacting them through M2L, on the other hand, depends primarily on the truncation order p, due to its high complexity in p. As discussed in Section 3.2, M2L has a complexity between $\mathcal{O}(p^2 \log^2 p)$ and $\mathcal{O}(p^6)$. In the measurements in Figure 10.3 it is $\mathcal{O}(p^4)$. Since the order is selected a priori by the precision requirements of the application, the question is then at what number of particles per group (cell) the balance between P2P and M2L is achieved. In Figure 10.3, the balance is attained at around 12 particles per cell (tree depth 4) for $p = 4$ and at about 98 particles per cell (tree depth 3) for $p = 8$. Clearly, in general, the balance strongly depends on the optimization degree of both the P2P phase and the M2L phase. For this reason, it is difficult to compare FMM implementations and cross-overs versus direct N^2 summation or alternative methods such as Ewald summation.

From this dependence, it follows that optimizations of either the P2P phase or the M2L phase can shift the runtime balance of the entire algorithm, leading to large reductions in runtime. For this reason, great effort has been invested in the optimization of the M2L phase [10,30,39,50,69,119,121]. As part of this work, within `ls1 mardyn` the P2P phase reuses the developed intrinsics wrappers for the short-range kernels. For the M2L phase, we use FFT accelerations, which will be detailed in Chapter 11.

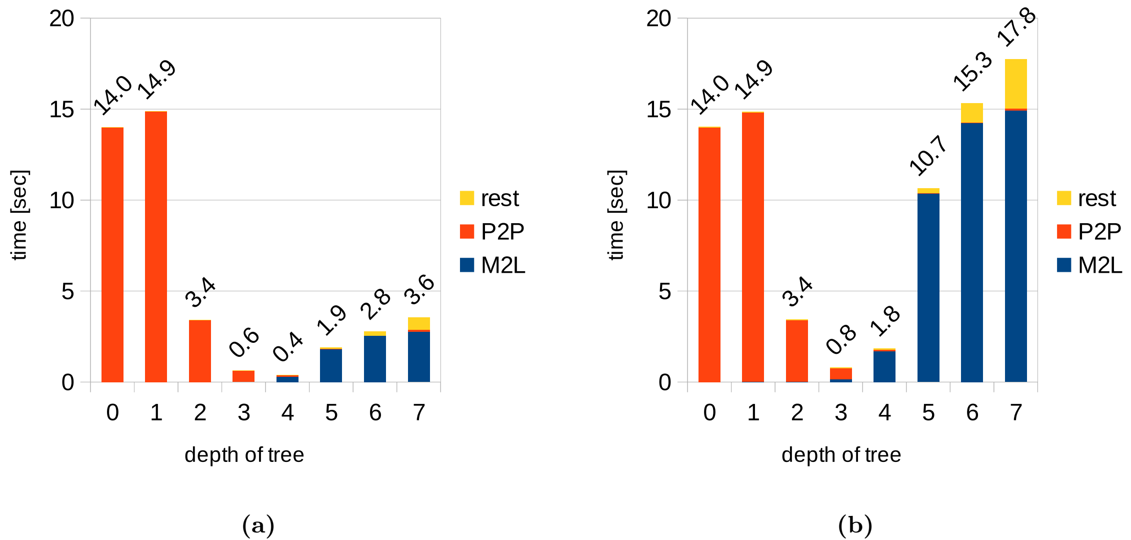


Figure 10.3: Runtime of FMM in dependence of tree depth. Setup contains 50000 uniformly distributed charges in a cube. Charges are randomly assigned values of ± 1 with equal probability. Simulations performed on an Intel Core i7-4770 CPU @ 3.40GHz desktop machine. (a) Truncation order $p = 4$. (b) Truncation order $p = 8$.

10.3.2 Computational complexity

Figure 10.4 shows the time per iteration for up to 67 million particles. In Figure 10.4(a) the time is plotted for trees of depth 2–4. Each of the individual curves exhibits a quadratic behaviour. This is because along each curve the number of cells remains constant, while the number of particles in the cells increases. As the number of particles in the cells increases, the time spent in the P2P phase rises quadratically, due to its N^2 nature. The time spent in M2L, on the other hand, is dependent only on the number of cells, which remains constant. Hence, the curves — per tree depth — exhibit a quadratic scaling behaviour.

Figures 10.4(b) and 10.4(c) show how the overall linear scaling is attained. Trees of increasing depth give the minimal runtime overall. Each depth gives the best runtime between 16 and 64 particles per cell for the chosen value of $p = 4$. For every increase of the number of particles by a factor of 8, it becomes more efficient to add a new level to the tree. Thus, the overall $\mathcal{O}(N)$ scaling is attained from piecewise quadratic curves, with each "piece" spanning a range of particle numbers eight times larger than the previous one.

Depths 0 and 1 contain, respectively, 1 and 8 cells. For these depths there are no well-separated cells and, thus, the calculation is identical to a direct N^2 calculation. The crossover between the direct N^2 calculation and the FMM algorithm can, thus, be considered as the first switch to the "next" depth, i.e. to depth 2. If one includes trees of depth 0 and 1 in the formal definition of FMM, it can, thus, be said that FMM is always faster or as fast as the N^2 direct calculation.

Before concluding this chapter, we point out that it is possible to improve runtime by reducing the length of the "pieces". The "fractional tiers" method [118] can do this for the list-based traversal, while the alternative, dual-tree traversal variant of FMM [121] should allow an even flatter curve, due to its more flexible control of the number of particles per cell.

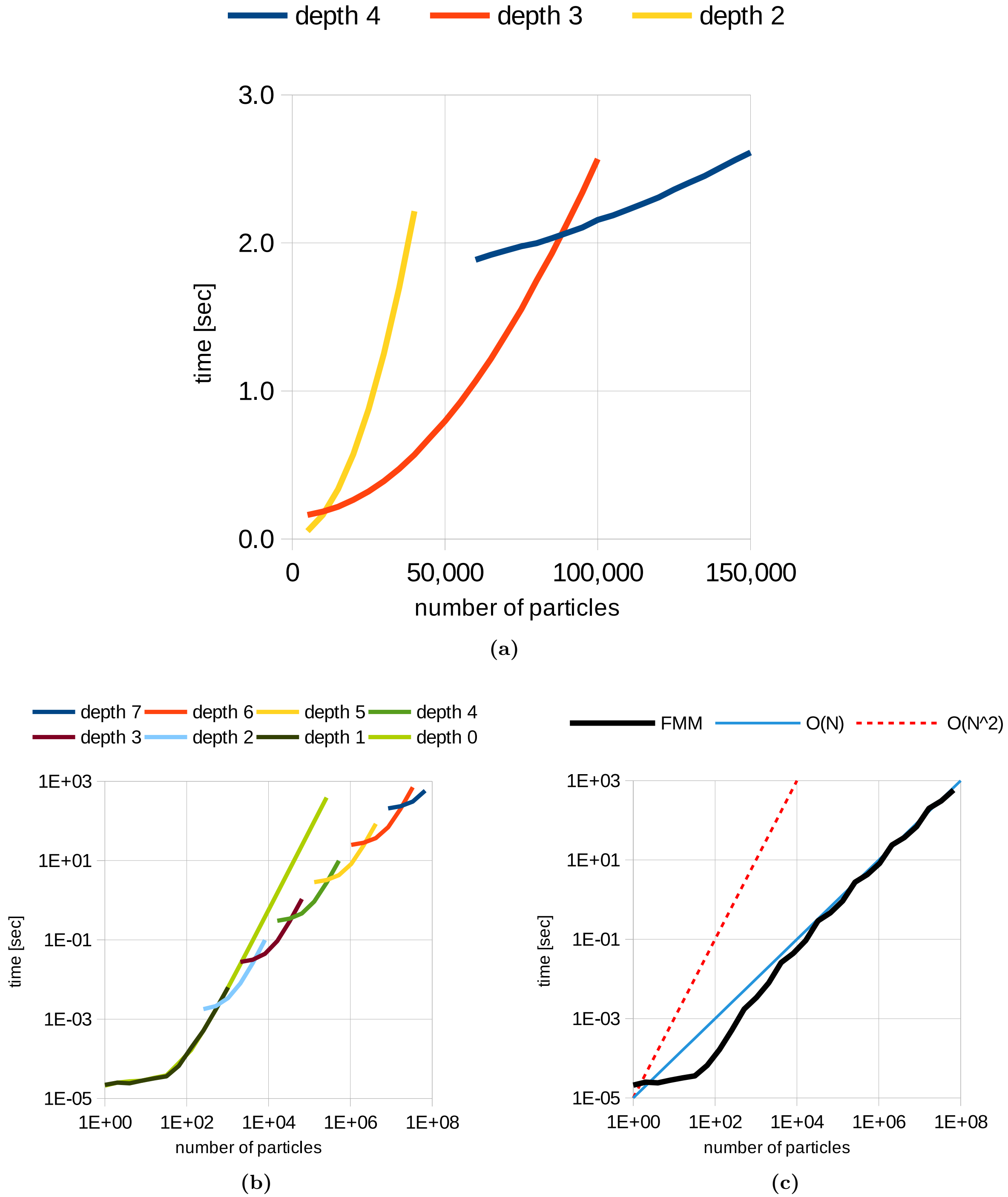


Figure 10.4: Runtime of the basic FMM implementation on an Intel Core i7-4770 CPU @ 3.40GHz desktop. The particles are uniformly distributed in a cube and with random charges of ±1. (a) FMM runtime for trees of depth 2–4. Number of particles increases from 5000 to 150000. Truncation order $p = 8$. (b) FMM runtime for trees of depth 0–7. Number of particles increases from 2^0 to 2^{26} (≈ 67 million). Truncation order $p = 4$. Axes feature log-log scaling. Tree of depth 0 is identical to the direct N^2 calculation. (c) Minimal FMM runtime across all trees. Setup same as for (b).

11 Acceleration of the M2L Phase

Having shown the correctness of our implementation in Chapter 10, we now turn our attention to runtime optimizations of our FMM implementation. As mentioned in Chapter 10, the most important functions to optimize are P2P and M2L, but P2P is easily vectorized via the intrinsics wrappers developed in Section 5.4. Hence, in this chapter we focus on the M2L phase.

In Section 11.1 we look at some of the different mathematical formulations for the multipole expansion, as there is generally no consensus in the literature about which expansions are the fastest. In Section 11.2 we give an overview of some of the possible optimization techniques for the M2L phase and describe the ones that were implemented. Finally, in Section 11.3 we present the results on the selection of mathematical formulation and acceleration of the M2L phase. A comparison to the established `ExaFMM` code is also included.

The results presented in this chapter are based on [38,39,62], which were carried out as part of this work.

11.1 Choice of basis functions

Different options exist for the mathematical formulation for the multipole and local expansions in FMM. A summary of the available formulations can be found in [121]. There seems to be no consensus about which formulation is fastest, however. For example, [117] argues for solid harmonics instead of spherical harmonics, but spherical harmonics still exist in modern codes, such as `ExaFMM`. Moreover, some authors advocate the use of Cartesian Taylor expansions instead of harmonics [51,111,121]. One of the first questions is, then, which formulation(s) will be used, as this potentially affects the software design of the algorithm. Suppose, for example, that for some parameters one formulation is faster, but for other parameters another formulation is faster. Then an application, which aims for the best performance across the whole range of parameters, would need to support switching between the formulations in some way. Hence, as part of this work, solid harmonics, spherical harmonics and Cartesian Taylor expansions were considered. Table 11.1 provides the storage and runtime complexity in terms of the truncation order p.

expansions	storage	runtime
spherical harmonics	$\mathcal{O}(p^2)$	$\mathcal{O}(p^4)$
solid harmonics	$\mathcal{O}(p^2)$	$\mathcal{O}(p^4)$
Cartesian Taylor	$\mathcal{O}(p^3)$	$\mathcal{O}(p^6)$

Table 11.1: Storage and runtime complexity of FMM in terms of truncation order p for expansions in different basis functions.

Solid harmonics Up to this point the algorithm has been presented in solid harmonics. The expansions in solid harmonics are closely related to the traditional spherical harmonics up to multiplicative factors [113]. These different factors make them more computationally favourable, however [87,117]. The implementation in solid harmonics boils down to evaluating Legendre polynomials in the spatial coordinates x, y, z (e.g. $\frac{z(x^2-y^2)}{r^3}$ or $\frac{z(2xy)}{r^3}$). As indicated by Table 11.1 and Fig. 3.3(a), truncating up to order p results in a triangular array of complex numbers. Per multipole, $\frac{(p+1)(p+2)}{2}$ complex numbers are stored, which gives the $\mathcal{O}(p^2)$ storage. For the M2L translation (Equation (3.9)), each of the $\mathcal{O}(p^2)$ entries is the result of a sum of $\mathcal{O}(p^2)$ elements, hence, the complexity is $\mathcal{O}(p^4)$.

Spherical harmonics FMM was originally developed using spherical harmonics [48]. Since then, many authors have switched to solid harmonics [70,87,93,119] due to more favourable runtime. A part of the difference to solid harmonics is that spherical harmonics evaluate the Legendre polynomials in spherical coordinates, which means that position and distance vectors need to be converted to spherical form. This, in turn, implies additional expensive square-root and trigonometric operations. Nevertheless, spherical harmonics still appear in some modern libraries today, such as `ExaFMM`. Like solid harmonics, the spherical harmonics feature $\mathcal{O}(p^2)$ storage and $\mathcal{O}(p^4)$ runtime.

Cartesian Taylor Some authors advocate the use of expansions in Cartesian Taylor basis functions [51,111,121,125]. Unlike solid or spherical harmonics, in this case the basis functions are simple monomials of the spatial coordinates, e.g. $1, x, y, z, x^2, xy, y^2, yz \ldots$. Overall, when truncating up to order p, the monomials of the type $x^{p_1}y^{p_2}z^{p_3}$ are used, where the powers sum up to p: $p_1 + p_2 + p_3 = p$. Per multipole $\frac{(p+1)(p+2)(p+3)}{6}$ elements are stored, leading to "pyramidal" storage with $\mathcal{O}(p^3)$ elements [62]. In this fashion, the M2L operation has complexity $\mathcal{O}(p^3) \cdot \mathcal{O}(p^3) \in \mathcal{O}(p^6)$, which is considerably higher than for spherical or solid harmonics. Advocates of Cartesian Taylor expansions, however, report that their simplicity can lead to a better runtime when p is low, although they do report that the different methods are not optimized to the same degree [121].

11.2 Acceleration techniques

Different acceleration techniques exist for the M2L operators. Some of these are implementational, such as unrolling and low-level optimizations, while others are algorithmic, such as FFT, Wigner rotations and plane wave accelerations. In Section 11.3 we will only present results for unrolling and FFT. For the sake of completeness, however, we will now briefly introduce all mentioned techniques. The exposition assumes expansions in spherical harmonics, however, the discussion applies to the other basis functions as well.

Unrolling and low-level optimizations Consider the implementation of Equation (3.9) for low values of p, for example for $p = 5$. In such cases, the four nested loops with trip counts of ≤ 5 lead to a very complex control flow and the overhead of the loops is very big. Moreover, computing the entries $\mathbf{M}_{l,-m}$ on the fly from the entries $\mathbf{M}_{l,m}$ and the use of complex numbers make the issue even more pronounced. For this reason, a viable optimization strategy is to reduce the control statements. One would, ideally, hope that the compiler would be able to unroll the loops either by itself in the optimization stages or via `#pragma` statements inserted by the programmer. This turns out to not be the case, unfortunately, as the code is too complex. Further intervention of the programmer is, thus, necessary. In [121] and [10], the code

was unrolled through the use of template metaprogramming. As part of this work, this was achieved through a code generator in [62]. The code for the M2L translation was taken and the arithmetic statements were simply replaced to print the unrolled code to a file for different values of p. Since we used a code generator, it was also easy to take into account non-trivial optimizations, such as removing entries, which are identically zero (for example $\mathbf{M}^s_{l,0}$) and taking the $\mathbf{M}_{l,\pm m}$ symmetries into account.

This approach is expected to work well for low values of p. For high values of p, the nested loops are amortized better, thus, reducing the gains due to unrolling. Moreover, due to the $\mathcal{O}(p^4)$ complexity, the generated code through unrolling grows quartically, which leads to long compilation time, large binaries and less cache storage available for program data. Hence, for larger values of p, other accelerations are preferable. Overall, this approach retains the $\mathcal{O}(p^4)$ complexity, but improves on the asymptotic constants.

FFT acceleration One of the first algorithmic accelerations of the M2L phase was to perform it in Fourier space [49]. The M2L operation is essentially a convolution operation between the two involved expansions and can, thus, be performed as a simple entrywise multiplication in Fourier space, due to the convolution theorem [38].

The conversions to and from Fourier space carry, of course, some overhead. The calculation of a two-dimensional FFT for the expansions has a $\mathcal{O}(p^2 \log p)$ complexity. It should be kept in mind that zero-padding is also needed to avoid wrap-around effects due to the FFT, so the expansions become larger in Fourier space. Once in Fourier space, however, an M2L operation becomes simply an entrywise multiplication and, hence, $\mathcal{O}(p^2)$. The Fourier coefficients of a multipole expansion can then be reused for M2L translations to different local cells. Due to the linearity of the Fourier transform, the contributions of different multipole cells to a certain local cell can also be added in Fourier space. The idea is, thus, to compute the Fourier transforms of all multipole expansions prior to the start of the M2L phase, perform all M2L operations and, finally, convert the resulting local expansions from back from Fourier space.

Since the M2L operation in Fourier space is an entrywise multiplication, it is a memory-bound operation. For this reason, it is worthwhile to consider performing it in single precision, as this halves the memory requirements, thereby giving a theoretical speed-up of a factor of $2\times$.

Overall, this is a mathematical optimization of the complexity of the M2L operation. Hence, it is expected to have larger benefits for larger values of p, while for lower values the overhead of the conversions to and from Fourier space is expected to be visible.

This technique, unfortunately, quickly runs into numerical instabilities with increasing p, which might be one of the reasons why it is not universally adopted. As a remedy, in [30] the method was extended to work on sub-blocks of the expansions, at the cost of some computational efficiency. As part of this work, both the basic FFT acceleration and the sub-block variants were studied extensively in the works [38,39]. In Section 11.3 we present only a small subset of the results therein.

Wigner rotation acceleration Another popular acceleration technique for the M2L phase is to use Wigner rotation matrices [21,65,119]. The core idea is that if one translates a multipole expansion along the z-axis (i.e. for translation vector $\vec{r} = (0, 0, z)$), then the expansion becomes "sparse" and only $p + 1$ real entries of the $\frac{(p+1)(p+2)}{2}$ complex entries are non-zero. If one takes the sparsity into account in Equation (3.9), one of the loops can be collapsed, leading to an $\mathcal{O}(p^3)$ complexity. In general, however, only about three out of the $6^3 - 3^3 = 189$ translations per cell are along the z-axis. In order to accelerate the remaining ones, the multipole and local pseudoparticles are rotated appropriately before and after each M2L translation so that

the translation vector connecting them is along the z-axis. These rotations are performed via Wigner matrices, hence, the name of the scheme. As part of this work, an implementation of the Wigner rotation acceleration was developed, but abandoned. Although not fully optimized, it was found to be considerably less performant than the FFT implementation. As also its theoretical complexity is higher than that of FFT, it was, hence, not pursued further.

Plane wave acceleration Another acceleration technique is the use of plane wave expansions [50]. An advantage of the plane wave acceleration over FFT or Wigner rotations is that it can work for arbitrary positions of the pseudoparticles. The FFT acceleration and (especially) the Wigner acceleration rely heavily on the fact that the pseudoparticles are arranged on a grid, so that certain translations and rotations can be precomputed. For the plane wave acceleration, however, the translation vector needs to be rotated only so that the z-direction is dominant, but not necessarily along the z direction. For this reason, it can work for arbitrary displacement vectors, which allows its application to the more adaptive DTT-based codes, such as `ExaFMM`. Hence, while this acceleration was not implemented in this work, it may be considered for future extensions.

11.3 Results

In this section we present results on the choice of basis functions for the multipole expansions, as well as on acceleration techniques. The simulations were performed with the basic code outside of `ls1 mardyn`. In order to determine which basis functions perform best, the Cartesian Taylor expansions were implemented in [62] as part of this work. A comparison to the version of `ExaFMM` used for [121] was also performed[1]. For a clean comparison, both our solid harmonics and the Cartesian Taylor expansions were optimized to the same degree as in [121] via unrolling [62]. The time per M2L call was calculated by measuring the time for the whole M2L phase and dividing by the number of M2L calls.

Figure 11.1 shows the results of the measurements. As seen from Figure 11.1(a), our basic solid harmonics version performs faster than the (basic) spherical harmonics version of `ExaFMM`. At $p = 0$, the solid harmonics version is $1.6\times$ faster and decreases down to $1.2\times$ at $p = 12$. The differences decrease with increasing order p, which is what we expect, since both implementations are $\mathcal{O}(p^4)$ and only differ in their asymptotic constants. Thus, as p increases, the p^4 term comes to dominate the calculation and diminishes the effect of this difference.

In Figure 11.1(b) we show comparisons between solid harmonics and Cartesian Taylor (CT) expansions. Comparing our CT expansions to those of `ExaFMM`, we observe that the implementations perform very similarly. For higher orders, the values match very well. For low orders, some differences are visible. The differences at low orders are not significant, however, as our code is LBT-based, while `ExaFMM` is DTT-based. Hence, the overhead of the tree traversals is different and most visible at the low orders where M2L calls are cheap. Overall, we can draw the conclusion that our CT expansions have been optimized well and are, hence, representative of CT in general.

Having established that our Cartesian Taylor version is performant, we can now draw significant comparisons to our solid harmonics version. For orders $p \leq 3$, their performance is basically equal and from $p \geq 4$, the solid harmonics version begins to outperform the Cartesian

[1] A version of the code from the repository at `https://bitbucket.org/exafmm/` has been used. Since then the code has migrated to `https://github.com/exafmm/exafmm` and changed considerably. In the present version of `ExaFMM` available on `https://github.com/exafmm/exafmm` (as of 24.05.2019), it appears that only the spherical harmonics version is present for the Laplace equation, without accelerations.

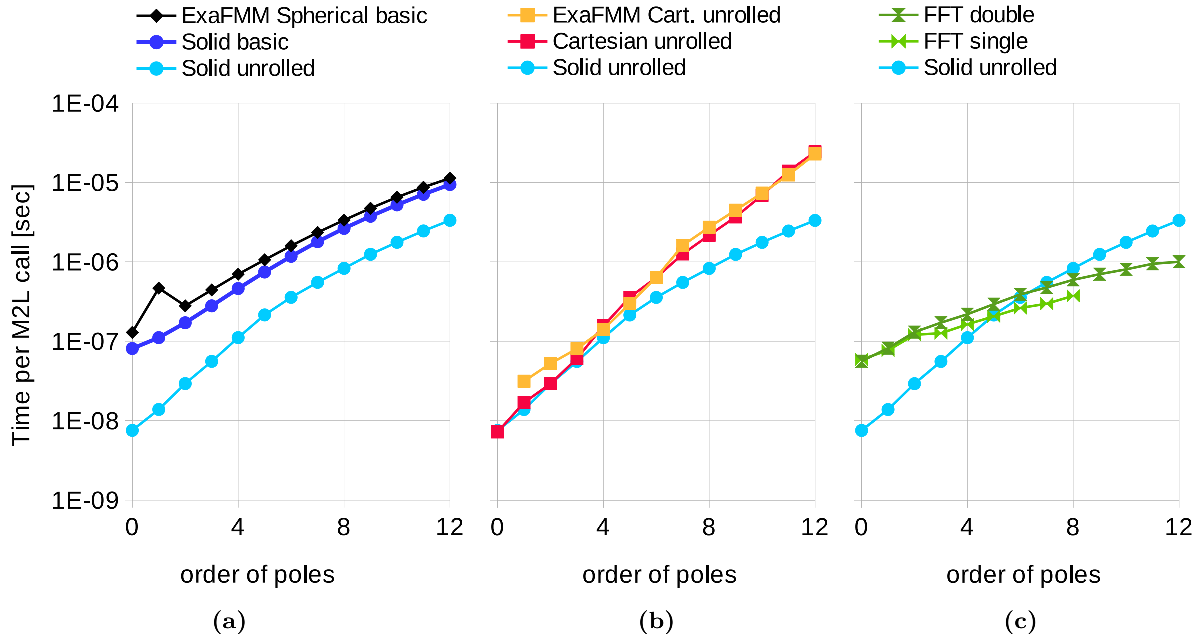


Figure 11.1: Time per M2L call for different mathematical formulations and different optimizations on an Intel Core i7-4770 CPU @ 3.40GHz desktop. Time for whole M2L traversal taken and divided by number of performed M2L calls. System contains 512000 particles in a tree of depth 6. For `ExaFMM`, $\theta = 0.4$ was used for the multipole acceptance criterion and $ncrit = 16$ for the subdivision of cells (`ExaFMM` is DTT-based [121]). "Basic" refers to basic implementation prior to optimizations. "Unrolled" refers to optimization of the M2L phase by fully unrolling the four nested loops of Equation (3.9) via code generation [62]. "FFT" refers to optimization of the M2L phase by performing the M2L convolution in Fourier space. "FFT single" uses single precision, which becomes unstable at $p = 9$. "FFT double" uses double precision, which is stable in this range of p. (a) Comparison between spherical harmonics and solid harmonics. (b) Comparison between solid harmonics and Cartesian Taylor expansions. (c) Comparison between unrolling optimization and FFT optimization of the solid harmonics code.

Taylor one considerably. This means that we can restrict our FMM implementation to only solid harmonics expansions.

The accuracy of the potentials and forces calculated by the Cartesian Taylor and solid harmonics implementations was also compared (not shown). They were found to give identical results, which is to be expected, as whether one uses one or the other is only a matter of a change of basis of the involved vector space.

We now comment on the accelerations of the M2L operations. In Figure 11.1(a) we can compare the basic solid harmonics implementation and the unrolled one. The unrolled one is clearly much faster: at $p = 0$ it is 10.7× faster and for $p = 12$ it is 2.8× faster. Because p increases, the trip counts of the nested loops increase and improve the performance of the basic version. Nevertheless, the difference is dramatic.

In Figure 11.1(c), we can compare the unrolled version to the FFT one. The FFT performance is shown for both single and double precision. The single precision starts at about 0.95× the double precision performance and grows to up to 1.6× faster at the last stable order $p = 8$. As the theoretical speed-up is 2×, we consider this a good value. At $p = 0$, the unrolled version is 7.5–7.9× faster than the FFT versions. This is due to the overhead that they carry for conversion to and from Fourier space, as well as the additional (and larger) memory storage in the frequency domain.

The unrolled version gets outperformed by FFT in single precision at $p = 5$ and by FFT in double precision at $p = 6$. Single becomes 2.2× faster than unrolled at $p = 8$, while double becomes 3.3× faster at $p = 12$. For larger values of p, even larger speed-ups due to the FFT version were observed in [38,39], as it has a lower computational complexity.

Overall, comparing the basic solid harmonics version and the minimal time delivered by the accelerated versions, speed-ups between 3.6× and 10.7× were attained. In this section, we also showed that our performance compares well to the established and high-performing `ExaFMM` code. For larger values of p, our code even outperforms both `ExaFMM` versions, which were available in the investigated version of `ExaFMM`. From this, we can conclude that our implementation has a high performance also relative to other implementations. As mentioned in Chapter 10, maximal performance was one of our main motivations for developing our own FMM code for `ls1 mardyn`. The primary (sequential) ingredient for achieving this is, hence, attained.

12

Parallelizing the Fast Multipole Method

In this chapter we present the parallelization of our FMM implementation. For the shared-memory parallelization we present results from [45] and for the distributed-memory parallelization we present results from [83]. Both were carried out as part of this work. The implementations were performed within `ls1 mardyn`, on top of the FFT-accelerated version implemented in [39].

The works [45] and [83] are both extensive. Here we only summarize the most important results and refer to the original works for further details. As the parallelization of FMM is not trivial, in Section 12.1 we discuss the sources of parallelism and the dependencies between the different phases of FMM. Then, in Section 12.2 we present the shared-memory parallelization and, finally, in Section 12.3 we present the distributed-memory parallelization.

12.1 Sources of parallelism in FMM

Figure 12.1 shows (again) the different passes of the algorithm. In serial execution, the passes are simply performed in the order: upwards, horizontal and, finally, downwards. In parallel execution, however, things are more complex, because the upwards and downwards passes do not exhibit a lot of parallelism and have many dependencies. While these passes are not very costly, care must be taken so that they do not become a bottleneck to the scalability of the whole program. The horizontal pass, on the other hand, exhibits a lot of parallelism, but is subject to many dependencies on the upwards and downwards passes.

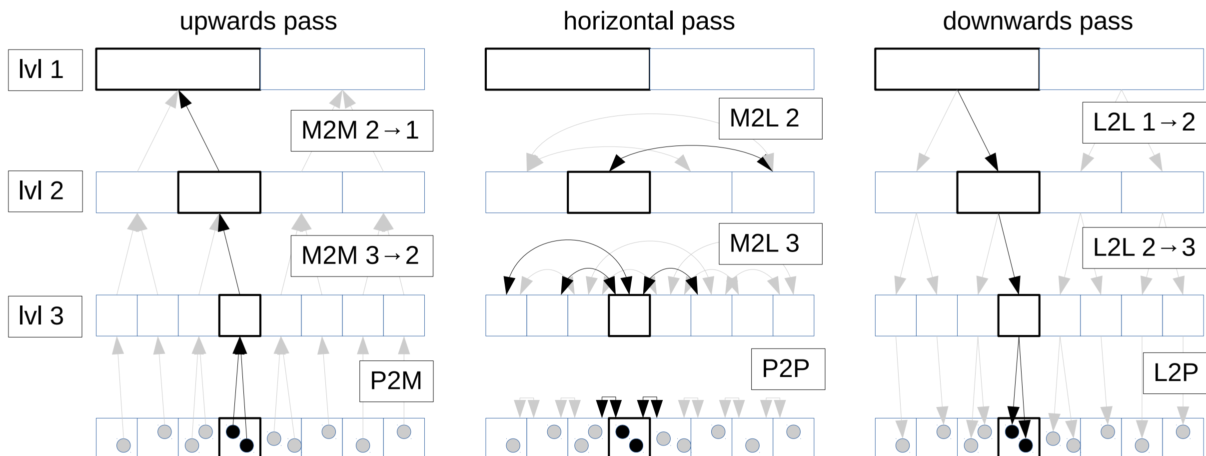


Figure 12.1: Illustration of the different FMM passes. Figure identical to Figure 3.7.

Upwards pass During the construction of multipoles, information flows from the leaves of the tree and propagates "upwards" towards the root, cf. Figure 12.1. The P2M phase is performed first. It has read-only dependencies on the particles' positions, so it can be performed simultaneously with any other operations on the particles. In the M2M phase, 2^D cells from level k combine their multipoles in one cell from level $k-1$, where D denotes the dimension of the simulation. Thus, in order to form the multipole expansion of a cell on level $k-1$, the multipoles of all child cells at level k must have already been computed.

Usually a single thread performs the 2^D M2M operations from the children to their parent cell, because of the write-dependencies to the parent's multipole coefficients. This means, however, that if level k contains $(2^D)^k$ cells, then only $(2^D)^{k-1}$ threads can work in parallel on M2M operations from level k to level $k-1$. In this fashion, parallelism decreases towards the root of the tree, leading to a serialization of the algorithm. Moreover, as processing level $k-1$ requires work on level k to be completed, some form of synchronization between the levels is needed. While resolving these synchronizations via global barriers is easy, it is, unfortunately, expensive and restrictive. If one is able to resolve the synchronizations requirements in a local fashion, it is possible to continue work upwards in the tree in one region, while a distant region is still being worked on in the lower levels.

Horizontal pass As mentioned in previous chapters, most of the work in FMM is in the horizontal pass, as this is where the P2P and M2L phases are. This pass also exhibits the highest degree of parallelism. This is due to the fact that work on different levels is independent, cf. Figure 12.1. Thus, a thread working on a pair of cells needs to lock the resources of (at most) two cells on the same level.

Considering the P2P phase, the involved operations are practically the same as the short-range operations discussed in Part III, except that no cutoff-radius is applied. Instead, whole cells interact with whole cells. A thread working on two cells, thus, needs to protect its write-access to both of them, due to the *Newton3* optimization. The P2P work can be started immediately and parallelized by any of the methods outlined in Chapters 7 and 8.

For the M2L phase, a thread working on M2L between two cells typically needs to lock only the local expansion of the target local cell. The multipole expansion is accessed in read-only fashion. Similarly to *Newton3*, however, it is sometimes beneficial to perform M2L also for the other pair of cells. In other words, when computing M2L between the multipole expansion of cell A and the local expansion of cell B, it may be beneficial to compute also the M2L operation between the multipole expansion of cell B and the local expansion of cell A [39,45,107]. This type of M2L interaction is referred to as "symmetric," "mutual" or "two-way". If the calculation is performed in symmetric fashion, then the thread working on the pair of cells needs to lock the local expansions of both cells.

Regarding the execution order of the operations, the M2L work on level k can start as soon as the M2M work from level $k+1$ to k has been completed. Parts of the work can begin earlier, if a means is a available for identifying which of the cells have already been processed.

Downwards pass Similarly to the upwards pass, the downwards pass does not represent a large portion of FMM runtime, but care is needed to prevent it from becoming a sequential bottleneck. The work begins from the top of the tree and propagates downwards, cf. Figure 12.1. The L2L work from level k to level $k+1$ can start only after the M2L work on level k has been completed. The L2L operation from a cell on level k to the respective cells on level $k+1$ can be executed in parallel, as the level k cell is accessed in read-only fashion. As the M2L and L2L operations are "additions", the M2L operations to a cell on level k and the L2L operation

to the same cell can be executed in any order, but not simultaneously. Analogously, the same goes for P2P and L2P: they can be executed in any order, but not simultaneously.

Thus, the downwards pass exhibits the same bottlenecks as the upwards pass: a high degree of serialization and multiple synchronization steps. Overall, these bottlenecks are due to a serialization, which is inherent to the FMM algorithm itself. Thus, they cannot be avoided.

Implications of FFT acceleration for the parallelization of FMM Adding the FFT acceleration to the parallel algorithm is generally not difficult, because the FFT coefficients are stored separately from the solid harmonics multipole and local coefficients. The calculation of the Fourier coefficients from the solid harmonics coefficients of multipole expansions must happen just before any M2L operations. The calculation back of the solid harmonics coefficients of local expansions from their Fourier analogues must happen right after all M2L operations have been completed.

12.2 Shared-memory parallelization

12.2.1 Implementation

The shared-memory parallelization of FMM was done in `ls1 mardyn`. Due to the multiple, complex dependencies between the different stages, the FMM algorithm is clearly more suited to task-based parallelization than to loop-based parallelization. A comparison of the two parallelization strategies in [107] demonstrated that this is indeed the case and, hence, also the choice of other implementations [75]. The ability of task parallelism to handle dependencies and conflicts between resources allows the execution of all three passes together in one parallel region. In this fashion, the negative effect of the serialization around the root of the FMM tree is mitigated. For this reason, our approach to the parallelization of FMM has been to use `Quicksched`. As introduced in Section 7.5, `Quicksched` is an open-source library with support for tasks with dependencies among them and resource conflicts. Unlike in Section 7.5, however, this time we are employing the library's full functionality, also making use of the support for hierarchical locks on resources, which are needed for the dependencies in the upwards and downwards passes.

As we saw in Section 7.5, the right balance of the granularity of the tasks needs to be found. Spawning too many tasks can lead to considerable overhead, while spawning too few can cause starvation of the threads. In the context of FMM, this has also been observed in [75], where spawning one task per M2L operation was observed to lead to the generation of too many tasks. It is, hence, necessary to consider appropriate groupings of the operations together into tasks.

In our implementation, the P2P tasks are grouped in the same fashion as in the scheme **`qui_111`**, detailed in Section 7.5. This time, however, tasks for building the SoAs and transferring the data from SoAs to the primary AoS storage, are needed as well. They are referred to as P2P "preprocessing" and P2P "postprocessing" tasks in the figures in Section 12.2.2. The M2M tasks are grouped so that the eight M2M operations sharing the same parent cell are executed together. The L2L tasks are also grouped in the same way — the L2L operations from one cell to all of its eight children are grouped together. Two groupings for the M2L tasks were implemented: `pair2way` and `CompleteCell`. In the `pair2way` variant, one task is spawned for every M2L pair of cells. The tasks perform the M2L operation symmetrically and, hence, block both cells. In the `CompleteCell` approach, all M2L operations, which share a local expansion, are grouped together. Thus, a single task performs a total of 189 M2L operations in the `CompleteCell` variant. While the two-way optimization is not applied in this case, a high degree of parallelism is present because all such tasks (one per cell) can run in parallel.

A feature of Quicksched to automatically determine the weight of a task was also employed, to account for the fact that M2L tasks have different runtimes from e.g. M2M and L2L tasks. The library computes these automatic weights by measuring the time spent inside tasks via the high-precision __rdtsc function. These weights are computed during the first force calculation. They are then used to optimize the scheduling of the tasks to the threads, as discussed in Section 7.5. Since the first iteration is performed during the initialization, it is not included in the measurements shown here and, hence, has no negative effect on runtime.

12.2.2 Results

In this section we present results of the Quicksched parallel implementation. The measurements were performed on an Intel Xeon Phi 7210-F KNL node of the CoolMUC3 cluster[1]. The simulation system contains 85805 1CLJ2C molecules, each with one Lennard-Jones center and two charges with a total charge of zero. As we are only measuring the parallel performance in this chapter, the physical parameters of the simulation have not been tuned to real molecular models.

The simulation was chosen to be moderately inhomogeneous. It consists of a small, spherical droplet, surrounded by a gaseous phase. The droplet has a radius of $\frac{1}{8}$ of the domain length and a density, which is $3.8\times$ higher than the surrounding gaseous phase. The system contains 16^3 cells on the finest FMM level, which gives about 21 molecules (42 point-charges) per cell, on average.

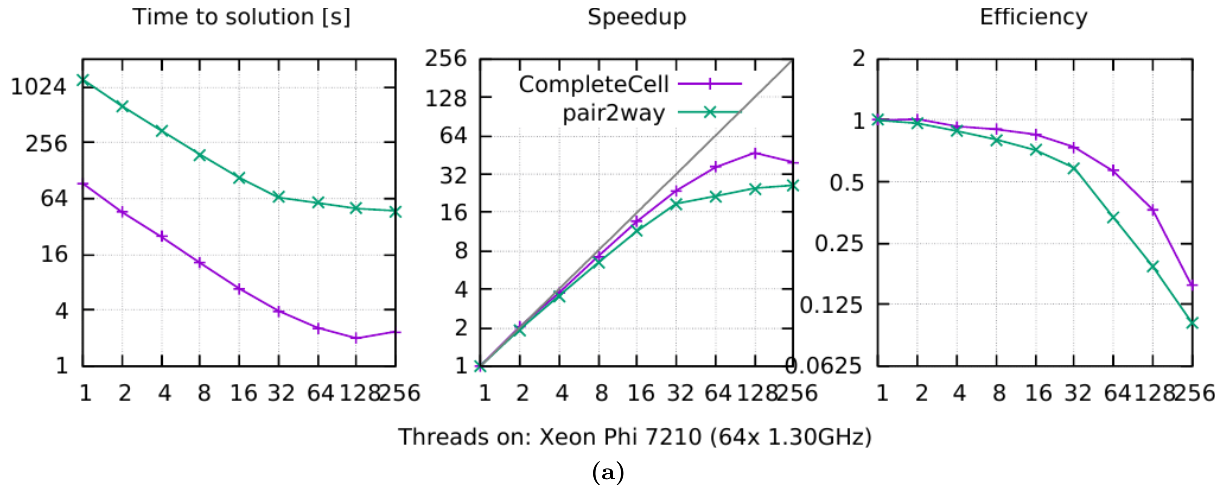
The measurements were performed in ls1 mardyn with the FFT acceleration and SIMD vectorization of the P2P phase. In ls1 mardyn the size of the FMM cells is related to the size of the Linked Cells by construction, in order to optimize the communication of the data for the P2P phase with MPI, as will be detailed in the next section. In order to allow control of the depth of the FMM tree, a “subdivision factor” was implemented, which subdivides the Linked Cells uniformly, thereby increasing the depth.

Figure 12.2 presents the results. Figure 12.2(a) shows a comparison between CompleteCell and pair2way for the horizontal phase only. As seen from the results, the CompleteCell strategy is vastly superior. Similarly to the observation in Section 7.5, the Quicksched implementation comes with some overhead, which depends on the total number of tasks in the queue. The pair2way variant has approximately $\frac{189}{2} = 94.5$ times more M2L tasks, which manifests itself in about $16\times$ sequential overhead. While the speed-ups themselves are similar, the speed-up of CompleteCell is also better, which can probably be attributed to the fact that tasks lock only one, instead of two cells. For these reasons, the pair2way variant has not been considered further.

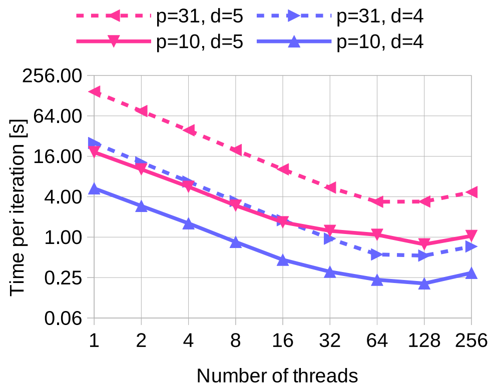
In Figures 12.2(b) and 12.2(c) strong scaling experiments of the entire simulation are shown for two different FMM parameters. Considering the abundant dependencies, conflicts and even load imbalance to be resolved, the results are excellent. Time to solution decreases to up to 64 or 128 threads at high parallel efficiencies.

Looking at Figure 12.2(c), the effects of increasing the order and of increasing the tree depth become clear. The $p = 31$ curves feature a considerably better scalability, meaning that the Quicksched library is better at handling the larger, more computationally intense tasks in this setting. Increasing the tree depth d raises total time considerably, due to more tasks and a less favourable balance between the P2P and M2L stages (cf. Section 10.3.1). The higher tree depth also comes together with a marginal decrease in parallel efficiency, as another level in the tree comes together with even more dependencies to be resolved than before.

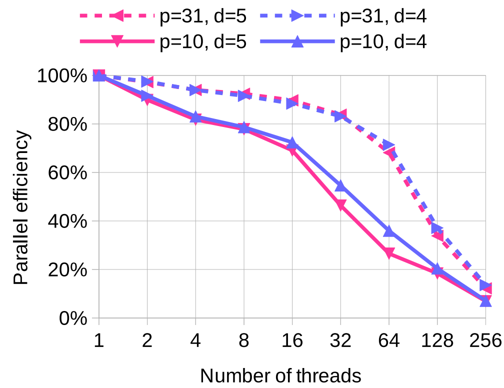
[1] https://www.lrz.de/services/compute/linux-cluster/coolmuc3/overview/



(a)



(b)



(c)

Figure 12.2: Strong scaling measurements for the shared-memory parallelization on the KNL platform. (a) Comparison of the `pair2way` and `CompleteCell` strategies for 16^3 cells in the last FMM level. Figure adapted from [45]. Time only for P2P and M2L included. (b), (c) Time per iteration and parallel efficiency for full simulation with `CompleteCell` for different orders of the poles p and different tree depths d.

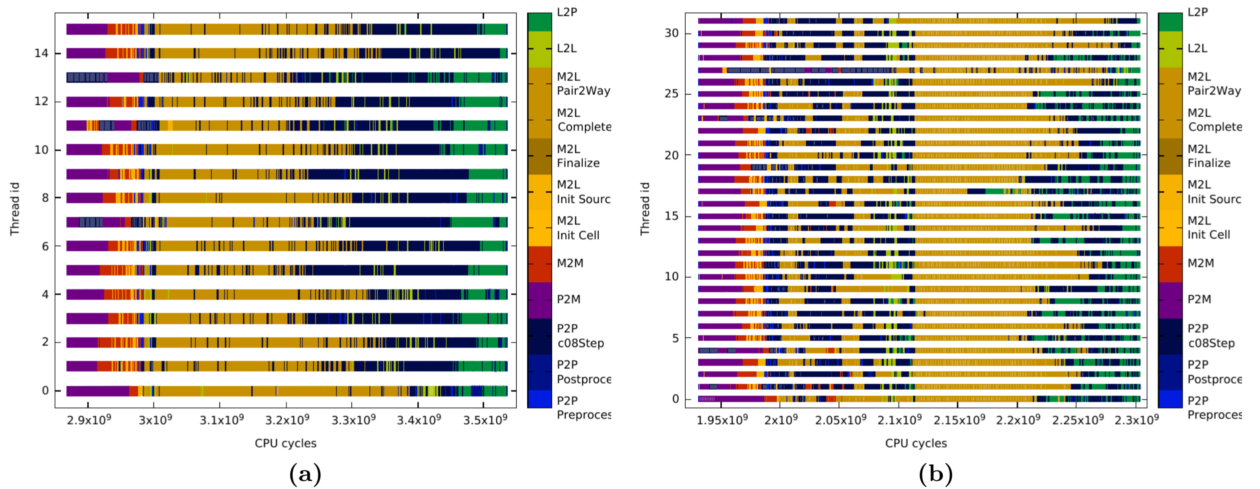


Figure 12.3: Scheduling of the various FMM tasks over time to threads for $p = 10, d = 4$. Figure adapted from [45]. (a) 16 threads. (b) 32 threads.

In an attempt to gain further insight into the drop-off of efficiency between 16 and 32 threads for $p = 10, d = 4$, the scheduling of tasks to threads was plotted in Figure 12.3. While no particular idle time is observed (which would have been indicated by white gaps, cf. also [45]), it can be observed that the scheduling is markedly different. Namely, in Figure 12.3(a) each thread works first mostly in M2L and then mostly in P2P. In Figure 12.3(b), however, the P2P and M2L workloads are more mixed for every thread. This could lead to a poorer data-locality of the execution. While the P2P phase is mostly compute bound and, hence, not affected strongly by data locality, the M2L operations in Fourier space are memory-bound, and, hence, affected more strongly from the mixing of the two phases. This explanation is also supported by the visibly longer duration of the M2L phase at 64 threads, available in [45].

Further tests were conducted in [45], also with increased total system size. They confirmed the observations made here that increasing order is beneficial to parallel efficiency, but not increasing the system size or tree depth. Thus, the granularity of the tasks can affect parallel efficiency considerably. This is also in agreement with [75], where a more flexible granularity of the tasks is investigated in the context of DTT-based FMM.

Performance tends to drop in the hyperthreading range in all examined cases (here and in [45]). As this was also observed in Section 7.5, it may be a drawback of the `Quicksched` library or even of the tasking approach in general.

Note that Figure 12.3 also highlights the benefits of using task-based parallelism for FMM instead of loop-based parallelism. Multiple different routines are executed in the same parallel region with only one barrier at the end of the region, which would not have been possible with loop-based parallelism. Moreover, it can be seen from the figure that while some threads are still working on P2M and M2M, other threads have already begun working on M2L and P2P, thereby practically resolving the serialization bottlenecks of the algorithm.

12.3 Distributed-memory parallelization

The complex dependencies between FMM operations and the inherent serialization towards the root of the tree make the distributed-memory parallelization of FMM also challenging. As every particle interacts with every other particle in the system, some form of global communication

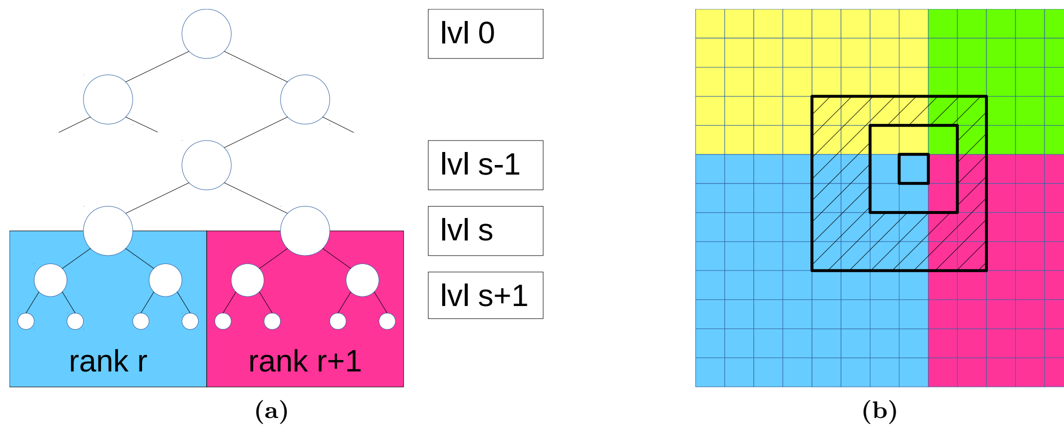


Figure 12.4: (a) Splitting of FMM tree to "local" and "global" for the distributed-memory parallelization of FMM. Levels coarser than level s are referred to as global part of the tree (including level s), while levels finer than level s are referred to as local part of the tree. (b) Parallelization of the local part of the tree for M2L. Regions in different colour belong to different processors. The hatched region marks the M2L interaction list of the corner cell of the blue processor. The halo layer must, hence, be two cells wide.

is necessary between every two processes.

12.3.1 Parallelization approach

The basic approach for parallelizing FMM was inspired by the idea of "locally essential tree" (LET), borrowed over from Barnes-Hut simulations [114]. The idea is that the amount of data required from distant processes decreases geometrically when going upwards in the tree. The FMM tree is, then, usually split into a "local" and a "global" part [61,123]. The notions are illustrated in Figure 12.4(a).

Local part of tree The local part of the tree is relatively straightforward to handle. One simply introduces halo-layers in the same fashion, typical for domain decomposition methods. Care must be taken only that the halo-layers are two-cells thick, so that all local M2L operations can be performed, see Figure 12.4(b). The calculation, then, proceeds as usual: compute own multipole coefficients, send and receive multipole coefficients to neighbours to populate halo layers and proceed with the M2L phase. Assuming that the global part of the tree can be handled to receive local coefficients at the local root of the tree, the downwards pass then proceeds within the domain.

Global part of tree Different possibilities exist for the handling of the global tree. The first question is how deep the global tree is. For a problem of a fixed size, the depth of the global tree is given by $\log_8 P$, where P is the number of processors. If the number of processors is small, the whole global tree can simply be stored on all processes redundantly. The upwards pass can be completed with the help of global collective operations. The MPI functions `MPI_Alltoall`, `MPI_Allgather` or `MPI_Allreduce` can be used for that purpose. The horizontal and downwards passes can be computed only for cells, which will have an influence on the local domain, or even redundantly for all cells.

If the global part of the tree is large, however, a different implementation may be required.

This happens especially in strong-scaling scenarios, where a fixed problem is solved on as many processors as possible. In such cases, the global tree can easily be larger than the local tree, which may render redundant storage or computation inefficient. More advanced strategies are needed in such cases, such as outlined in [61,123].

12.3.2 Implementation

The implementation, which was developed as part of this work in [83], follows ideas outlined in [123]. Some original ideas were also included, which — to the best of the author's knowledge — represent new optimizations. The MPI parallelization was done in `ls1 mardyn`.

First implementation The first implementation, which served as a baseline for improvement, consisted of handling the global tree via an `MPI_Allreduce`. This was realized in the following fashion. Each processor computes the upwards pass until the root of its local tree as in serial mode. The global tree is constructed on all processors and initialized with zeros. The multipole value of the local root is then entered at the appropriate place. The M2M contributions from the local root are computed upwards all the way up to the global root. An `MPI_Allreduce` call is then performed on the multipole coefficients of the entire global tree with a simple `MPI_SUM` operator. Since the individual multipole contributions are simply summed up, this implementation gives correct results. After that call is executed, the correct multipole coefficients in the entire global tree are available on every process. In this fashion, the upwards pass is completed. The horizontal and downwards passes are, then, performed only for cells, which are parents of the local root cell. This means that some work in the global tree is performed redundantly: processors, which share common parents will both compute the parents' local coefficients. However, as only one cell out of every level is computed, this is not a large portion of work.

Through the use of non-blocking MPI communication, a considerable portion of the MPI waiting times can be overlapped with computation. For instance, after the upwards pass in the local tree is completed, the `MPI_Allreduce` of the global tree can be initiated in a non-blocking fashion via `MPI_Iallreduce`. Work on the horizontal pass in the local tree can, then, be performed, while the MPI call is being processed in the background. Finally, when there is no more work to be done, but to begin with the downwards pass, the `MPI_Wait` command on the `MPI_Iallreduce` request is called. The first implementation already includes overlapping of communication and computation.

Optimizations Following [123], multiple improvements to the baseline implementation were performed in [83]. The first optimization approach is to avoid the collective `MPI_Allreduce` communication, because it becomes inefficient for large processor counts. It becomes inefficient, because the number of (internal) synchronization stages keeps rising ($\mathcal{O}(\log P)$ stages for P processors) and the volume of data communicated increases. Moreover, the fraction of summed-up entries, which are zero, increases (every processor has only one non-zero coefficient per tree level) and, finally, the whole multipole tree becomes available to all processors, which becomes much more than what each processor needs. For these reasons, alternatives are of great interest when running with many MPI ranks. The way to avoid the collective `MPI_Allreduce` is to simply perform the minimal necessary communication via point-to-point communication calls in $\mathcal{O}(\log P)$ stages [83,123].

This scheme, however, also requires communication in the horizontal and downwards passes. For small processor counts this may lead to a slow down, but for larger ones it is expected to

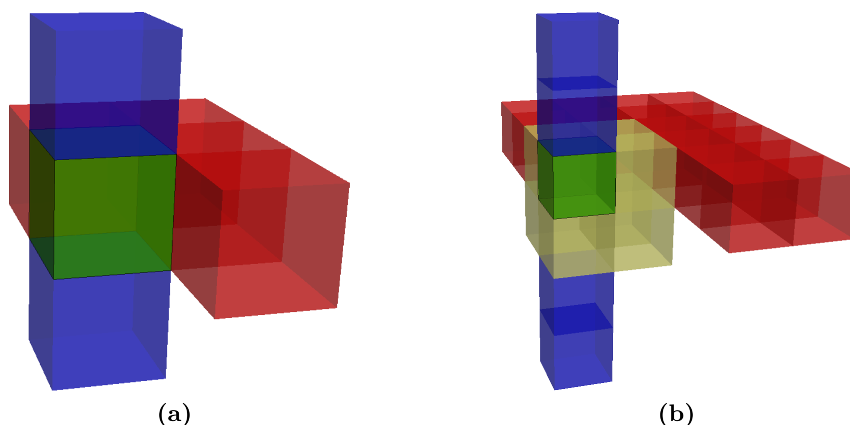


Figure 12.5: (a) Neutral territory method illustrated for a cubic domain, whose side-length is equal to one cutoff-radius. The processor, which owns the domain in green, computes all interactions between the blue "tower" and the red "plate" region. All cells interact with the cell in green. Force contributions to the tower and plate regions are communicated back to their respective processors. Note that communication to the processors in the "other half" of the red plate (symmetrically about the green cell) is also necessary, in order to receive their force contributions to the green cell. (b) Application of neutral territory idea to M2L calculations in global part of tree. A yellow region is introduced, which consists of the green cell's siblings in the tree. The processor owning the cell in green computes all interactions between the cells in the blue tower and the cells in the red plate region. Interactions of all cells with the cells in yellow are also computed. Computed local contributions to blue, red and yellow cells are communicated back to the processors, which own those cells. Interactions for cells, which are not well-separated, are not computed.

perform faster. For this reason, a version was implemented, which automatically switches between the Allreduce-version and the other one at a certain level of the tree.

Other optimizations were also performed, including groupings of processors, which are siblings in the tree. This allows for reduction of the number of communication partners. Also different MPI sending modes, such as `MPI_Rsend` were tested [83].

Neutral territory optimization A novel optimization idea was to apply the neutral territory method ([100]) to the M2L communication. The neutral territory method is part of a class of methods, called Zonal Methods [13]. The aim of these methods is to reduce range-determined communication volume in short-range MD simulations in the limit of strong-scaling. In other words, to reduce the halo region when the per-processor domain length approaches — or even falls below — the cutoff-radius of the short-range force calculation. Figure 12.5(a) illustrates the neutral territory method when the side-length of the domain is equal to the cutoff-radius. In this fashion, by changing the halo-cells so that they are no longer read-only, the total communicated volume — and number of communication partners — is decreased. In Figure 12.5(a), if the calculation was to be computed with the full shell method (cf. Section 2.2.6), 26 communication partners would be needed and a communication volume 26 times larger than the domain of the processor. Through the neutral territory method the number of communication partners is reduced to 10 and the communication volume to 12 times that of the processor domain.

In Figure 12.5(b) we illustrate how this idea was applied to reduce the number of communication partners for the horizontal pass in the global part of the tree in [83]. As in the neutral

territory method, cells of the blue tower interact with cells of the red plate and all cells interact with the green cell, which is the cell of the current processor. Of course, interactions are not computed for cells, which are not well-separated. Due to the fact that the M2L stencil is slightly different for each of the eight children of the same parent cell, the method needs to be modified in order to be applied to FMM. The modification is the introduction of the cells in yellow, which are siblings of the cell in green and the new requirement is that all cells also interact with the cells in yellow. Figure 12.5(b) illustrates the resulting scheme. After the calculations are done, the contributions to all foreign cells are sent back to their owning processor.

Overall, this represents a decomposition of the M2L operations between the different processors such that no M2L operation is computed redundantly. The number of communication partners is reduced from 189 down to 43, which is a fourfold reduction.

In [123], an alternative optimization for reducing the number of communication partners was presented. It reduces the number of partners from 189 down to 26, albeit introducing more stages in the communication. The neutral territory idea could also be applied on top of it, however, bringing the number further down from 26 to 10. The details can be found in [83].

Hybrid MPI-OpenMP parallelization As of writing this document, our `Quicksched` implementation has not yet been extended to the MPI communication routines. This means that hybrid MPI-OpenMP execution is not yet supported by our code. Considering the multiple dependencies between the MPI communication stages and the possibilities for overlapping communication with computation, however, the MPI communication would fit very well in the tasking model of `Quicksched`. One could, thus, expect that the approaches would fit well with each other, leading to a good hybrid MPI-OpenMP performance.

12.3.3 Results

In this section we investigate the MPI performance of our FMM implementation. In [83] measurements were performed only for the FMM part, excluding the rest of the algorithm. This was done to isolate the scalability of the FMM implementation from the rest of the algorithm, which contains other collective communication calls. Measuring in this way also meant that data for the P2P communication is not included because it is reused from the short-range force calculation communication. The communication of data for the P2P part, however, is known to scale excellently [27].

Except in Figure 12.8, hyperthreading is not considered in the measurements, i.e. at most one MPI process was started per core.

Scalability on SuperMUC and Shaheen Figure 12.6 presents a strong scaling study on up to 32768 MPI processes on the supercomputers SuperMUC Phase 1 and Shaheen II[2], presented also in [84]. Shown is the time per iteration of the FMM part of `ls1 mardyn` for different system sizes. The system sizes — S, M, L and XL — are chosen to come close to the minimal systems that the code can support ($2\times2\times2$ cells per processor). Due to limitations of the cluster, the strong scaling curves were broken down into three (S, M, L), instead of running from 1 to 32768 processors. The measurements are collected with the final, best version of the MPI parallelization. Breakdowns of the speed-ups due to the individual optimizations can be found in [83].

The first thing to notice is that the simulations run about $2\times$ faster on one processor of Shaheen than on one processor of SuperMUC. This is due to the different hardware: SuperMUC

[2] `https://www.hpc.kaust.edu.sa/content/shaheen-ii`

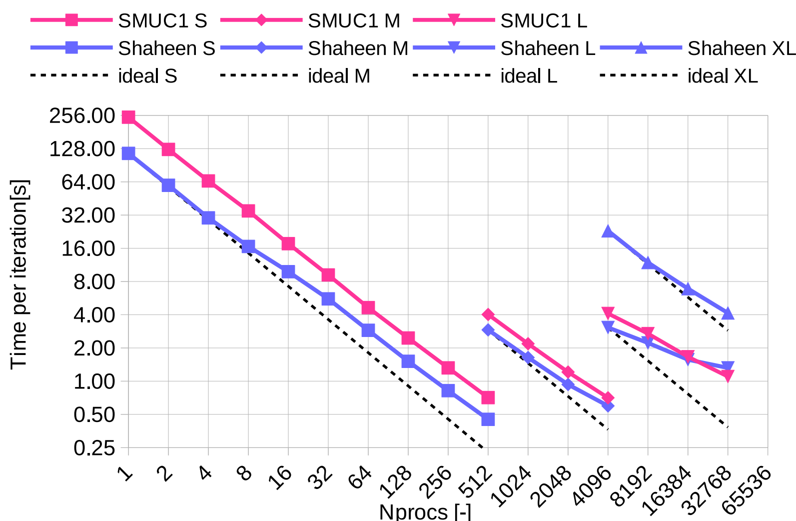


Figure 12.6: Strong scaling of the FMM part of `ls1 mardyn`. Order of poles $p = 10$ with FFT acceleration. The letters "S", "M", "L" and "XL" denote the system sizes 0.7, 5.4, 43.3 and 343 million particles, respectively. Each particle consists of one Lennard-Jones site and two Coulomb sites with a net charge of zero. Simulations were carried out on the SuperMUC Phase 1 and Shaheen II clusters. Parameters are chosen such that in the S, M and L configurations the system contains only two local levels in the tree at the respective maximal number of processes. E.g. the L scenario contains only $4^3 = 64$ cells per processor in the local tree at 32768 processes. The XL scenario was performed only on Shaheen and illustrates that much better scaling is achieved for larger systems.

Phase 1 runs Intel SandyBridge Xeon E5-2680 with AVX, while Shaheen runs Intel Haswell Xeon E5-2698v3 with AVX2. As these runs are with FFT, the kernel profits more from AVX2. For the M and L setups the differences are smaller because sequential differences matter less on larger numbers of processors.

Next, we comment on the observed speed-ups. The values for SuperMUC are 347.4×, 5.7× and 3.7× for the S, M and L scenarios, respectively, computed against 1, 512 and 4096 processors. For Shaheen the values are 257.1× 4.9× 2.3× and 5.6× for the S, M, L and XL scenarios, respectively, computed in the same fashion. Considering that the scenarios are quite small and that every process communicates with every other process in every iteration, we consider these to be excellent results. For the same number of processors, increasing the system size (i.e. comparing L and XL scenarios on Shaheen) improves scalability greatly: speed-up improves more than twofold from 2.3× to 5.6×. This emphasizes the importance of the size of the local tree relative to the size of the global tree and implies that the low observed speed-ups are primarily due to the small system size.

Finally, we comment on the fact that the speed-ups for SuperMUC are better than for Shaheen. On the one hand, this is expected, since the calculation runs slower on SuperMUC. For the L scenario, however, SuperMUC even surpasses Shaheen, which is not covered by this explanation. This observation should, hence, be attributed to another explanation, such as the different network topology of the machines. SuperMUC Phase 1 has a fat tree topology, which is a more natural fit to the FMM algorithm than Shaheen's dragonfly topology. This explanation should manifest itself for large processor counts, which is what we observe in the L scenario. Hence, the network topology should be the reason for the better scalability on SuperMUC.

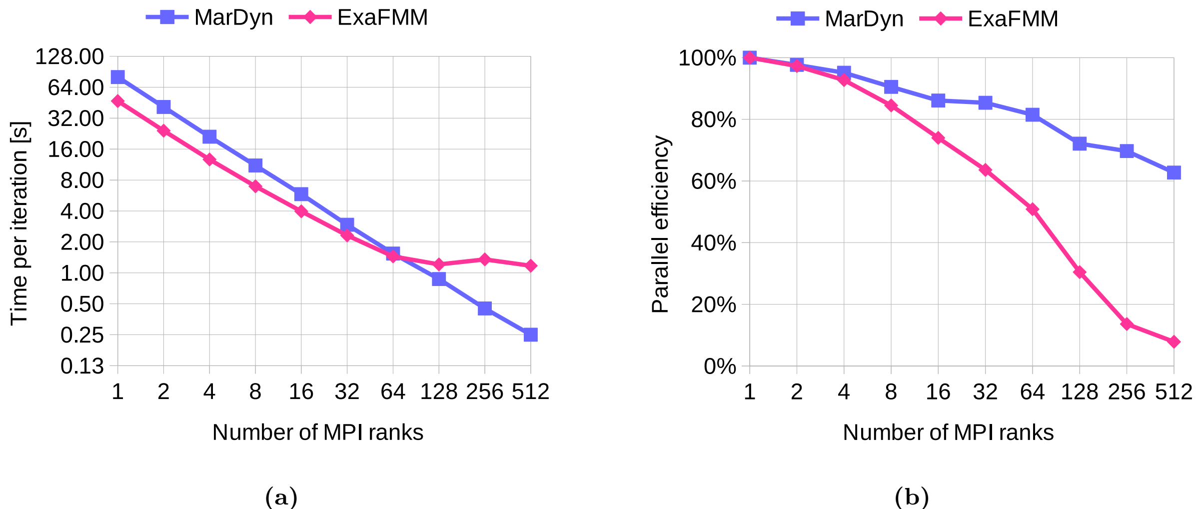


Figure 12.7: Comparison between `ls1 mardyn` and `ExaFMM` for 0.7 million particles. (a) time per iteration. (b) parallel efficiency versus one process of each code.

Comparison to `ExaFMM` In order to relate our parallelization to established codes, a comparison was set up between `ls1 mardyn` and `ExaFMM` in [83]. For `ExaFMM`, a version from 6.5.2016, which was available on `https://github.com/exafmm` was used. The comparison was performed on up to 512 MPI processes on SuperMUC Phase 1.

Since we want to compare the parallel implementations, effort was made to make the serial execution as comparable as possible. Keeping in mind the remarks about comparing different FMM implementations outlined in Section 3.10, this comparison is also to be interpreted with caution for multiple reasons. First, `ls1 mardyn` works with solid harmonics, optimized with FFT, while the used version of `ExaFMM` featured Cartesian Taylor expansions optimized by unrolling the loops via template metaprogramming. This means that `ls1 mardyn` is optimized for higher values of the truncation order p than `ExaFMM`. In order to account for this, a value of $p = 5$ was used, which is the value, which gives the closest time per M2L call, as per the comparisons in Figure 11.1. Next, `ls1 mardyn` is LBT-based with a (hardcoded) well-separatedness value of 1. `ExaFMM` is DTT-based, on the other hand, so a value of $\theta = 0.53$ was selected, which should give comparable separation to $ws = 1$. Only one periodic image (on all sides) was included in the measurements, as the codes treat larger numbers of images differently. For `ls1 mardyn`, a tree depth of 5 was selected, which should correspond to splitting criterion $ncrit = 21$, i.e. the number of particles per cell in the leaf level of the tree is ≤ 21. The runs were performed with 0.7 million uniformly distributed particles in `ls1 mardyn`, again having two charges with a net charge of zero. Since `ExaFMM` does not support multicentered particles, the number of particles was doubled and they are assigned charges of ± 1. Only MPI was used in this comparison, no shared-memory parallelization. Symmetric M2L calculations were disabled in both codes.

Looking at the results in Figure 12.7(a), it can be observed that `ExaFMM` has almost two times lower runtime on one process, despite the attempts to make the sequential execution similar. This should be attributed primarily to the dual-tree traversal, which can be more flexible than the list-based one in balancing the M2L and P2P phases [121]. However, the scalability of `ls1 mardyn` is better, which allows it to eventually outperform `ExaFMM`. Runtime stagnates for `ExaFMM` from about 64 processes onwards, while it keeps decreasing for `ls1 mardyn` all the way

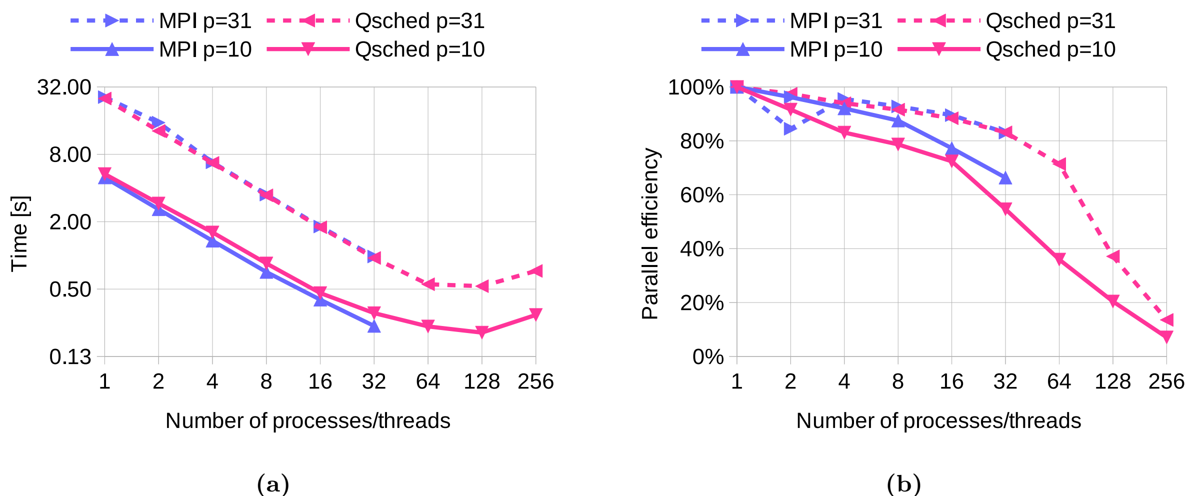


Figure 12.8: Comparison between shared-memory parallelization and distributed-memory parallelization of the FMM implementation in `ls1 mardyn` for different truncation orders with FFT acceleration. Due to technical issues on the cluster, the MPI version could not be executed for more than 32 processes. (a) Time per iteration. (b) Parallel efficiency versus one process of each version.

up to 512. Overall, the speed-up for `ls1 mardyn` from 1 to 512 processors is 321.2×, while only 40.1× for `ExaFMM`. The value of 321.2× is slightly lower than observed in Figure 12.6 because these measurements are for order $p = 5$ and not $p = 10$. This makes the sequential calculation faster, which, expectedly, lowers the parallel speed-up.

One reason for the lower parallel performance of `ExaFMM` could be that `ls1 mardyn`'s implementation communicates fewer multipole coefficients in total due to the lower storage requirements of the solid harmonics than of Cartesian Taylor ones. At order $p = 5$, however, the difference is small: `ls1 mardyn` communicates 42 double precision values, while `ExaFMM` communicates 56. While P2P communication time is not included in the `ls1 mardyn` values, this should also not cause such a difference in this range of particles per process.

The primary reason for the stagnating performance of `ExaFMM` should be that the version of `ExaFMM`, which was available at the time, featured a `MPI_Alltoall` collective call in the communication of the global tree. This can cause performance to decrease, as explained in Section 12.3.2. This explanation is also supported by the fact that `ExaFMM` observes lower parallel efficiency also in the range 1–64 processors in Figure 12.7(b). Due to the performance stagnation of `ExaFMM`, it was decided not to continue the comparison for higher processor counts.

Comparison between `Quicksched` and MPI parallelization Finally, we present a comparison between our shared-memory and our distributed-memory parallelization of FMM in `ls1 mardyn`, carried out in [45]. Measurements were performed on an Intel Xeon Phi 7210-F (KNL) node for a system with 85805 molecules for orders $p = 10$ and $p = 31$. Due to configuration issues, no more than 32 MPI processes could be started on a node.

Figure 12.8 shows the results. It can be observed that both version scale excellently for the computationally intense $p = 31$ case. Parallel efficiency is above 83% for both codes. `Quicksched` marginally outperforms the MPI version, by about 3%. For the $p = 10$ case, the MPI version outperforms the `Quicksched` version. Sequentially it is about 7% faster and it scales better overall, leading to up to 30% difference in time per iteration at 32 processes/threads.

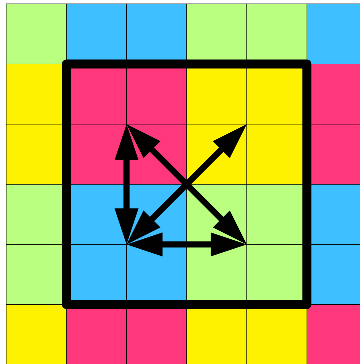


Figure 12.9: Extending the `c08` and `qui_111` schemes to the M2L phase of FMM. The cells on a level are coloured based on a `c08` colouring of their parents in the tree: all sibling cells are coloured with the same colour. The respective pairs of coloured cells within the "pack" of cells are traversed and the well-separated ones are interacted through (symmetric) M2L.

Exploration of further parameters in [45] also showed that sometimes one is faster, sometimes the other, but both perform overall excellently. This suggests that there is, perhaps, a small room for improvement for both the `Quicksched` and the MPI version.

12.4 Conclusion

Shared-memory parallelization In Section 12.2 we have presented our shared-memory implementation of FMM in `ls1 mardyn` via `Quicksched`. Despite the complex dependencies of FMM, the implementation performed very well, even handling a modest load-imbalance. Excellent scalability was observed for the entire calculation with a decreasing runtime on up to 128 threads on a KNL node. It was observed that task granularity and groupings of operations to be performed into tasks has a considerable effect on parallel performance. The implementation handles big tasks well for a high truncation order p and leaves some room for improvement for lower orders. Similarly to the observations in Part III, this again points to the fact that the granularity of the tasks must be chosen carefully. As this was also the case in [75], who used the `MassiveThreads` library [79], this can probably even be said about task-based parallelism in general. For this reason, further possibilities for grouping the work into tasks are of interest.

In Figure 12.9 we outline how to construct further strategies for shared-memory parallelization. Observe that all cells, which should interact via M2L, are children of cells, which are nearest neighbours (in LBT-based FMM). Hence, if groups of siblings are considered, the dependencies between groups of siblings look very much like the dependencies in the short-range force calculation: every group has read-write dependencies to all nearest neighbour groups. Based on this observation, we can reduce the problem of parallelization of M2L to the problem of parallelizing the short-range force calculation. For example, Figure 12.9 shows how `c08` and, respectively, `qui_111` can be applied to the M2L parallelization. In this way, virtually every scheme discussed in Chapters 7 and 8 can be applied, giving a large number of new possibilities to explore.

Distributed-memory parallelization In Section 12.3 we presented our MPI parallelization of FMM. Excellent scalability was demonstrated on two supercomputers for up to 32768 processors. It was observed that the network topology of the computers plays an important role and that even better strong scaling efficiencies can be obtained for larger scenarios.

Relating performance to other FMM implementations In Section 12.3 we presented a comparison between our MPI-parallel FMM and that of ExaFMM for up to 512 processors. While ExaFMM provided superior sequential performance, ls1 mardyn was able to outperform it through a better parallel performance. Considering that ExaFMM has been shown to compare excellently to other FMM implementations and even outperform many of them [121], it can be concluded that our code also performs very well relative to other implementations.

A comparison was then made between our shared-memory and our distributed-memory parallelizations. Both performed almost equally well with sometimes one scaling better, sometimes the other, depending on FMM parameters. Thus, our Quicksched implementation is as good as the MPI one and, by inference, also compares well to other FMM codes. In this way, we consider that our goal of developing a high-performing FMM implementation for ls1 mardyn has been met.

PART V

SUMMARY

In this thesis, we presented our work on the optimization of the `ls1 mardyn` simulation program. In Part III, we discussed our implementational work on improving the node-level performance of the program. In Part IV, we summarized our work on introducing the FMM algorithm to `ls1 mardyn`.

The primary contributions of this project were presented in Part III. The exposition was separated into different layers, which gradually build on top of each other. In Chapter 4, our refactoring of the core data structures was introduced. These changes paved the way for changing the floating-point precision, efficient SIMD and OpenMP execution and enabled the introduction of the **`RMM`** mode. The changes were a key ingredient for the observed much greater performance for small systems in Section 9.3, which manifested itself in about 33% higher performance than before. Next, in Chapter 5, our work on SIMD intrinsics wrappers was presented. The changes allowed portability of the implementation and a flexible handling of the instruction sets. This allowed up to 40% higher performance in Section 9.3 and enabled runs in either single or double precision on the full supercomputers in Section 9.4. In Chapter 6, we demonstrated that hyperthreading is important for `ls1 mardyn`, giving up to 40% more performance on Xeon architectures and up to 150% more performance on Xeon Phi architectures. Most notably, in Chapters 7 and 8, we introduced multiple new schemes for shared-memory parallelization. We demonstrated that these schemes can scale excellently, giving up to 84% strong scaling efficiency, when going from one to 240 threads on Xeon Phi and up to 96% when going from one to 32 threads on Xeon architectures.

All of these optimizations resulted in considerable gains for simulations on the full SuperMUC Phase 1 and Hazel Hen supercomputers in Chapter 9. Up to 43% higher weak scaling performance at up to 27% better memory utilisation were observed. Most notably, strong scaling performance was increased by more than 150%. Moreover, the flexibility of the code also allowed many further simulations on the full number of nodes, illustrating that application simulations are also greatly enhanced. This paved the way for new scientific insight on the application side, such as the simulations conducted in [99]. Finally, $2.1 \cdot 10^{13}$ molecules were simulated for the first time at over 80% parallel efficiency and more than 1PFLOP/sec in both strong and weak scaling experiments.

On the algorithmic side, we extended the capabilities of `ls1 mardyn`, by introducing the FMM algorithm for long-range electrostatics to it. We validated the correctness of the implementation in Chapter 10 and presented an analysis of the runtime of the individual phases of the algorithm. In Chapter 11, we showed our sequential optimization of the M2L phase, which is a necessary prerequisite for efficient execution. In Chapter 12, we presented our shared- and distributed-memory parallelizations of the algorithm. Excellent scaling on up to 256 threads on KNL were observed, as well as on up to 32768 processes on the supercomputers SuperMUC and Shaheen.

We now discuss the most promising directions for further research. We consider that the **`c04_hcp`** and **`sli_blk`** shared-memory schemes presented in Section 8.4 have a very high potential. This is because they build on top of the **`c04`** and **`sli`** schemes, which already perform very well, but address their primary weaknesses. Moreover, these schemes should be applicable

also to other operations on Cartesian grids with read-write dependencies to nearest neighbour points, which makes them applicable outside of molecular dynamics. Next and most notably, the autotuning library outlined in Section 9.5 has the potential for considerable gains for N-body simulations and early results have already illustrated the promise of the approach. Finally, on the FMM side, the observation in Section 12.4 opens many new possibilities for improving the shared-memory parallelization of the algorithm.

Overall, we consider that our work has significantly improved the capabilities of the simulation program `ls1 mardyn`. The scientific insights, gained in the process, can be used to accelerate other molecular dynamics and N-body simulation packages as well. This can be done either directly by, for example, applying some of the new OpenMP schemes or through the autotuning library inspired by this work. The resulting performance increase of the simulation codes, in turn, enables new scientific insights on the application side, as scientists can perform more simulations or simulate larger and longer physical processes.

Lennard-Jones Kernel in Intrinsics Wrappers

Listing A.1: LJ kernel

```
template<bool calculateMacroscopic>
inline __attribute__((always_inline))
void VectorizedCellProcessor :: _loopBodyLJ(
    const RealCalcVec& m1_r_x,     // CoM of molecule from cell 1
    const RealCalcVec& m1_r_y,
    const RealCalcVec& m1_r_z,
    const RealCalcVec& r1_x,       // site of molecule from cell 1
    const RealCalcVec& r1_y,
    const RealCalcVec& r1_z,
    const RealCalcVec& m2_r_x,     // chunk of molecules' CoM from cell 2
    const RealCalcVec& m2_r_y,
    const RealCalcVec& m2_r_z,
    const RealCalcVec& r2_x,       // chunk of molecules' sites from cell 2
    const RealCalcVec& r2_y,
    const RealCalcVec& r2_z,
    RealCalcVec& f_x,              // force vector
    RealCalcVec& f_y,
    RealCalcVec& f_z,
    RealAccumVec& V_x,             // virial vector
    RealAccumVec& V_y,
    RealAccumVec& V_z,
    RealAccumVec& sum_upot6lj,     // sum of LJ potential (times 6)
    RealAccumVec& sum_virial,      // sum of virials
    const MaskCalcVec& forceMask,  // force mask
    const RealCalcVec& eps_24,     // epsilon values (times 24)
    const RealCalcVec& sig2,       // sigma values
    const RealCalcVec& shift6)     // shift values (times 6)
{
    // distances between molecules' sites
    const RealCalcVec c_dx = r1_x - r2_x;
    const RealCalcVec c_dy = r1_y - r2_y;
    const RealCalcVec c_dz = r1_z - r2_z;

    const RealCalcVec c_r2 =
        RealCalcVec::scal_prod(c_dx, c_dy, c_dz, c_dx, c_dy, c_dz);

    // the force mask is applied here:
    const RealCalcVec r2_inv =
        RealCalcVec::fastReciprocal_mask(c_r2, forceMask);

    // calculation of the 12-6 formula
    const RealCalcVec lj2 = sig2 * r2_inv;
    const RealCalcVec lj4 = lj2 * lj2;
    const RealCalcVec lj6 = lj4 * lj2;
    const RealCalcVec lj12 = lj6 * lj6;
```

```
46      const RealCalcVec lj12m6 = lj12 - lj6;
47      const RealCalcVec eps24r2inv = eps_24 * r2_inv;
48      const RealCalcVec lj12lj12m6 = lj12 + lj12m6;
49      const RealCalcVec scale = eps24r2inv * lj12lj12m6;
50
51      f_x = c_dx * scale;
52      f_y = c_dy * scale;
53      f_z = c_dz * scale;
54      const RealCalcVec m_dx = m1_r_x - m2_r_x;
55      const RealCalcVec m_dy = m1_r_y - m2_r_y;
56      const RealCalcVec m_dz = m1_r_z - m2_r_z;
57
58      // calculation of virial values
59      V_x = RealAccumVec::convertCalcToAccum(m_dx * f_x);
60      V_y = RealAccumVec::convertCalcToAccum(m_dy * f_y);
61      V_z = RealAccumVec::convertCalcToAccum(m_dz * f_z);
62
63      // (compile-time) check whether macroscopic values should be accumulated
64      if (calculateMacroscopic) {
65
66          const RealCalcVec upot_shifted =
67              RealCalcVec::fmadd(eps_24, lj12m6, shift6);
68
69          const RealCalcVec upot_masked =
70              RealCalcVec::apply_mask(upot_shifted, forceMask);
71
72          const RealAccumVec upot_accum =
73              RealAccumVec::convertCalcToAccum(upot_masked);
74
75          sum_upot6lj = sum_upot6lj + upot_accum;
76
77          sum_virial = sum_virial + V_x + V_y + V_z;
78      }
79  }
```

B

Full Metrics of VTune Analysis

In Table B.1 we present the full obtained metrics from Intel VTune Amplifier for 1CLJ `RMM` single precision scenarios from Figure 6.5. Scenario A in the following table refers to $\rho = 0.39$, $r_c = 2.5$, while scenario B refers to $\rho = 0.78$, $r_c = 5.0$.

metric	A 1 thr	A 2 thr	B 1 thr	B 2 thr
Elapsed Time:	1671.544s	1413.430s	4099.689s	3427.572s
Clockticks:	$2.75 \cdot 10^{11}$	$4.56 \cdot 10^{11}$	$6.95 \cdot 10^{11}$	$1.15 \cdot 10^{12}$
Instructions Retired:	$3.81 \cdot 10^{11}$	$3.88 \cdot 10^{11}$	$9.84 \cdot 10^{11}$	$9.96 \cdot 10^{11}$
CPI Rate:	0.72	1.178	0.706	1.157
Retiring:	50.50%	61.70%	41.90%	51.20%
General Retirement:	40.10%	48.70%	41.20%	50.10%
Microcode Sequencer:	10.40%	13.00%	0.70%	1.10%
Assists:	0.00%	0.00%	0.00%	0.00%
Front-End Bound:	24.70%	22.50%	7.50%	16.20%
Front-End Latency:	19.00%	14.20%	3.20%	12.50%
ICache Misses:	0.00%	0.00%	0.00%	0.00%
ITLB Overhead:	0.00%	0.00%	0.00%	0.00%
Branch Resteers:	1.20%	0.80%	4.70%	3.60%
DSB Switches:	2.70%	3.90%	0.40%	0.60%
Length Changing Prefixes:	0.00%	0.00%	0.00%	0.00%
MS Switches:	5.60%	11.70%	0.40%	1.00%
Front-End Bandwidth:	5.70%	8.20%	4.30%	3.70%
Front-End Bandwidth MITE:	8.00%	16.60%	1.40%	2.10%
Front-End Bandwidth DSB:	12.20%	12.60%	16.20%	16.60%
Front-End Bandwidth LSD:	0.00%	0.00%	0.20%	0.30%
(Info) DSB Coverage:	62.60%	50.30%	93.60%	91.40%
(Info) LSD Coverage:	0.20%	0.20%	2.10%	2.50%
Bad Speculation:	3.10%	2.30%	12.40%	9.80%
Branch Mispredict:	3.00%	2.20%	12.40%	9.80%
Machine Clears:	0.00%	0.00%	0.00%	0.00%
Back-End Bound:	21.80%	13.50%	38.20%	22.80%
Memory Bound:	9.30%	6.50%	17.60%	13.80%
L1 Bound:	6.50%	12.30%	11.50%	21.80%
DTLB Overhead:	1.20%	8.00%	0.20%	0.50%
Loads Blocked by Store Forwarding:	4.70%	0.30%	0.60%	0.00%
Lock Latency:	0.00%	0.10%	0.00%	0.00%
Split Loads:	0.00%	0.00%	0.10%	0.10%
4K Aliasing:	5.90%	1.20%	10.50%	1.90%

FB Full:	0.30%	0.40%	5.70%	3.20%
L2 Bound:	N/A	N/A	N/A	N/A
L3 Bound:	N/A	N/A	N/A	N/A
Contested Accesses:	0.00%	0.00%	0.00%	0.00%
Data Sharing:	0.00%	0.00%	0.00%	0.00%
L3 Latency:	0.70%	0.40%	0.70%	0.30%
SQ Full:	0.00%	0.30%	0.20%	0.50%
DRAM Bound:	N/A	N/A	N/A	N/A
Memory Bandwidth:	N/A	N/A	N/A	N/A
Memory Latency:	N/A	N/A	N/A	N/A
LLC Miss:	5.00%	8.20%	3.30%	3.30%
Store Bound:	2.30%	0.60%	0.40%	0.10%
Store Latency:	99.40%	78.30%	95.90%	72.90%
False Sharing:	0.00%	0.00%	0.00%	0.00%
Split Stores:	1.00%	1.20%	0.80%	0.90%
DTLB Store Overhead:	0.10%	0.50%	0.00%	0.10%
Core Bound:	12.40%	7.00%	20.60%	9.00%
Divider:	13.50%	16.30%	1.00%	1.20%
Port Utilization:	16.40%	20.20%	15.30%	15.40%
Cycles of 0 Ports Utilized:	42.00%	89.70%	41.80%	91.10%
Cycles of 1 Port Utilized:	11.20%	16.60%	11.30%	20.10%
Cycles of 2 Ports Utilized:	12.10%	23.60%	12.50%	27.10%
Cycles of 3+ Ports Utilized:	18.70%	49.50%	18.00%	43.90%
Port 0:	27.10%	32.40%	37.10%	43.10%
Port 1:	31.70%	37.30%	42.70%	49.80%
Port 2:	25.80%	30.20%	31.60%	35.30%
Port 3:	26.50%	31.20%	32.70%	36.40%
Port 4:	19.00%	22.80%	14.10%	17.00%
Port 5:	33.50%	40.90%	19.50%	23.20%
Port 6:	40.30%	49.50%	22.20%	26.50%
Port 7:	9.90%	11.80%	6.00%	7.00%
Total Thread Count:	20	40	20	40
Paused Time:	119.282s	119.218s	120.577s	121.534s

Table B.1

Notes:

- The values are given as reported by VTune. Some values need to be divided by two for a fair comparison, e.g. “Clockticks”. For further information, see `https://software.intel.com/en-us/vtune-amplifier-help`.
- Executed with option “allow multiple”, which was observed to rerun every simulation 20 times. This is why the total thread count is reported as 20, and 40, instead of 1 and 2.
- Gains due to hyperthreading in profiling configuration on both systems were about 1.2× in contrast to the observed 1.3× for configuration A in Figure 6.4.
- “N/A” is abbreviated for the reported “N/A with HT on”, i.e. metrics, which are not available when hyperthreading is switched on.

Bibliography

[1] Mustafa Abduljabbar, Mohammed Al Farhan, Noha Al-Harthi, Rui Chen, Rio Yokota, Hakan Bagci, and David Keyes. Extreme scale FMM-accelerated boundary integral equation solver for wave scattering. *SIAM Journal on Scientific Computing*, 41(3):C245—-C268, 2019.

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

[3] M Abramowitz and I A Stegun. *Handbook of mathematical functions*. Dover Publications, New York, 1968.

[4] Michael P Allen and Dominic J Tildesley. *Computer simulation of liquids*. Clarendon Press, 1989.

[5] Christopher R Anderson. An implementation of the fast multipole method without multipoles. *SIAM Journal on Scientific and Statistical Computing*, 13(4):923–947, 1992.

[6] Axel Arnold, Florian Fahrenberger, Christian Holm, Olaf Lenz, Matthias Bolten, Holger Dachsel, Rene Halver, Ivo Kabadshow, Franz Gähler, Frederik Heber, et al. Comparison of scalable fast methods for long-range interactions. *Physical Review E*, 88(6):63308, 2013.

[7] Axel Arnold, Olaf Lenz, Stefan Kesselheim, Rudolf Weeber, Florian Fahrenberger, Dominic Roehm, Peter Košovan, and Christian Holm. Espresso 3.1: Molecular dynamics software for coarse-grained models. In *Meshfree methods for partial differential equations VI*, pages 1–23. Springer, 2013.

[8] J A Barker and R O Watts. Monte Carlo studies of the dielectric properties of water-like models. *Molecular Physics*, 26(3):789–792, 1973.

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

[10] Andreas Beckmann and Ivo Kabadshow. Portable node-level performance optimization for the fast multipole method. In *Recent Trends in Computational Engineering-CE2014*, pages 29–46. Springer, 2015.

[11] Daniel Berthelot. Sur le mélange des gaz. *Compt. Rendus*, 126:1703–1706, 1898.

[12] Kevin J Bowers, David E Chow, Huafeng Xu, Ron O Dror, Michael P Eastwood, Brent A Gregersen, John L Klepeis, Istvan Kolossvary, Mark A Moraes, Federico D Sacerdoti, et al. Scalable algorithms for molecular dynamics simulations on commodity clusters. In *SC'06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, page 43. IEEE, 2006.

[13] Kevin J Bowers, Ron O Dror, and David E Shaw. Zonal methods for the parallel execution of range-limited N-body simulations. *Journal of Computational Physics*, 221(1):303–329, 2007.

[14] W Michael Brown, Peng Wang, Steven J Plimpton, and Arnold N Tharrington. Implementing molecular dynamics on hybrid high performance computers–short range forces. *Computer Physics Communications*, 182(4):898–911, 2011.

[15] Martin Buchholz. *Framework zur Parallelisierung von Molekulardynamiksimulationen in verfahrenstechnischen Anwendungen.* Dissertation, Institut für Informatik, Technische Universität München, München, aug 2010.

[16] Oral Büyüköztürk, Markus J Buehler, Denvid Lau, and Chakrapan Tuakta. Structural solution using molecular dynamics: Fundamentals and a case study of epoxy-silica interface. *International Journal of Solids and Structures*, 48(14-15):2131–2140, 2011.

[17] David A Case, Thomas E Cheatham III, Tom Darden, Holger Gohlke, Ray Luo, Kenneth M Merz Jr, Alexey Onufriev, Carlos Simmerling, Bing Wang, and Robert J Woods. The Amber biomolecular simulation programs. *Journal of computational chemistry*, 26(16):1668–1688, 2005.

[18] Philippe Chatelain, Alessandro Curioni, Michael Bergdorf, Diego Rossinelli, Wanda Andreoni, and Petros Koumoutsakos. Billion vortex particle direct numerical simulations of aircraft wakes. *Computer Methods in Applied Mechanics and Engineering*, 197(13-16):1296–1304, 2008.

[19] Intel Corporation. Intel 64 and IA-32 architectures optimization reference manual. `https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf`, 2016.

[20] Intel Corporation. Intel VTune Amplifier 2019 user guide. `https://software.intel.com/en-us/vtune-amplifier-help-cpi-rate`, 2019.

[21] Holger Dachsel. Fast and accurate determination of the Wigner rotation matrices in the fast multipole method. *The Journal of chemical physics*, 124(14):144115, 2006.

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

[23] J W Eastwood, R W Hockney, and D N Lawrence. P3M3DP-The three-dimensional periodic particle-particle/particle-mesh program. *Computer Physics Communications*, 35, 1984.

[24] S Eckelsbach and J Vrabec. Fluid phase interface properties of acetone, oxygen, nitrogen and their binary mixtures by molecular simulation. *Physical Chemistry Chemical Physics*, 17:27195–27203, 2015.

[25] Wolfgang Eckhardt. *Efficient HPC Implementations for Large-Scale Molecular Simulation in Process Engineering.* Dissertation, Institut für Informatik, Technische Universität München, München, 2014.

[26] Wolfgang Eckhardt and Alexander Heinecke. An efficient vectorization of linked-cell particle simulations. In *Proceedings of the 9th conference on Computing Frontiers*, pages 241–244. ACM, 2012.

[27] Wolfgang Eckhardt, Alexander Heinecke, Reinhold Bader, Matthias Brehm, Nicolay Hammer, Herbert Huber, Hans-Georg Kleinhenz, Jadran Vrabec, Hans Hasse, Martin Horsch, Martin Bernreuther, Colin Glass, Christoph Niethammer, Arndt Bode, and Hans-Joachim Bungartz. 591 TFLOPS multi-trillion particles simulation on SuperMUC. In *International Supercomputing Conference (ISC) Proceedings 2013*, volume 7905 of *Lecture Notes in Computer Science*, pages 1–12, Heidelberg, Germany, June 2013. Springer.

[28] Wolfgang Eckhardt, Alexander Heinecke, Wolfgang Hoelzl, and Hans-Joachim Bungartz. Vectorization of multi-center, highly-parallel rigid-body molecular dynamics simulations. In *Supercomputing 2013, The International Conference for High Performance Computing, Networking, Storage and Analysis.*, Denver, nov 2013. IEEE.

[29] Wolfgang Eckhardt and Tobias Neckel. Memory-efficient implementation of a rigid-body molecular dynamics simulation. In *Parallel and Distributed Computing (ISPDC), 2012 11th International Symposium on*, pages 103–110. IEEE, 2012.

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

[31] Paul P Ewald. Die Berechnung optischer und elektrostatischer Gitterpotentiale. *Annalen der physik*, 369(3):253–287, 1921.

[32] Marat Faizov. Evaluation of the performance of vectorized force calculations for molecular dynamics. Master's thesis, Institut für Informatik, Technische Universität München, sep 2015.

[33] Scott E Feller, Richard W Pastor, Atipat Rojnuckarin, Stephen Bogusz, and Bernard R Brooks. Effect of electrostatic force truncation on interfacial and transport properties of water. *The Journal of Physical Chemistry*, 100(42):17011–17020, 1996.

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

[35] Hanno Flohr. Coupling the molecular dynamics simulation ls1 Mardyn with the lattice boltzmann simulation Walberla using the Macro-Micro-Coupling tool. Master's thesis, Institut für Informatik, Technische Universität München, 2015.

[36] Agner Fog. C++ vector class library. `https://www.agner.org/optimize/`.

[37] Agner Fog. The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers. `https://www.agner.org/optimize/`.

[38] Jean-Matthieu Gallard. Fast multipole method in MarDyn: FFT acceleration of the M2L phase. Master's thesis, Institut für Informatik, Technische Universität München, 2015.

[39] Jean-Matthieu Gallard. Optimization, implementation and evaluation of the FFT-accelerated fast multipole method. Master's thesis, Institut für Informatik, Technische Universität München, apr 2016.

[40] Aravindhan Ganesan, Michelle L Coote, and Khaled Barakat. Molecular dynamics-driven drug discovery: leaping forward with confidence. *Drug discovery today*, 22(2):249–269, 2017.

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

[42] Pedro Gonnet, Aidan B G Chalk, and Matthieu Schaller. QuickSched: Task-based parallelism with dependencies and conflicts. *arXiv preprint arXiv:1601.05384*, 2016.

[43] Andreas W Götz, Mark J Williamson, Dong Xu, Duncan Poole, Scott Le Grand, and Ross 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.

[44] Fabio Gratl. Implementation and evaluation of task-based approaches for molecular dynamics simulations. Master's thesis, Institut für Informatik, apr 2017.

[45] Fabio Gratl. Task based parallelization of the fast multipole method implementation of ls1-mardyn via QuickSched. Master's thesis, Institut für Informatik 5, Technische Universität München, Garching, nov 2017.

[46] Fabio Gratl, Steffen Seckler, Nikola Tchipev, Hans-Joachim Bungartz, and Philipp Neumann. AutoPas: Auto-Tuning for Particle Simulations. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2019.

[47] Christopher G Gray and Keith E Gubbins. *Theory of Molecular Fluids: Volume 1: Fundamentals*. Clarendon Press, 1984.

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

[49] Leslie Greengard and Vladimir Rokhlin. On the efficient implementation of the fast multipole algorithm. Technical report, Yale University, 1988.

[50] Leslie Greengard and Vladimir Rokhlin. A new version of the fast multipole method for the Laplace equation in three dimensions. *Acta numerica*, 6:229–269, 1997.

[51] Michael Griebel, Stephan Knapek, and Gerhard Zumbusch. Numerical simulation in molecular dynamics. numerics, algorithms, parallelization, applications, 2007.

[52] A Hainoun and A Schaffrath. Simulation of subcooled flow instability for high flux research reactors using the extended code ATHLET. *Nuclear Engineering and Design*, 207(2):163–180, 2001.

[53] Robert Hajda. Effiziente Vektorisierung des Dipolpotentials in parallelen Molekulardynamik-Simulationen. Bachelor's thesis, Institut für Informatik, Technische Universität München, 2014.

[54] David J Hardy, Zhe Wu, James C Phillips, John E Stone, Robert D Skeel, and Klaus Schulten. Multilevel summation method for electrostatic force evaluation. *Journal of chemical theory and computation*, 11(2):766–779, 2015.

[55] Johannes Heckl. Effiziente Vektorisierung von Simulationen für starre mehrzentrige Molekülmodelle. Bachelor's thesis, Institut für Informatik, Technische Universität München, 2012.

[56] Alexander Heinecke. *Boosting Scientific Computing Applications through Leveraging Data Parallel Architectures*. Dissertation, Institut für Informatik, Technische Universität München, München, 2014.

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

[58] David Hilbert. Über die stetige Abbildung einer Linie auf ein Flächenstück. In *Dritter Band: Analysis· Grundlagen der Mathematik·Physik Verschiedenes*, pages 1–2. Springer, 1935.

[59] Wolfgang Hölzl. Development of a particle-based streaming-solver with long-range interactions based on the fast-multipole-method. Bachelor's thesis, Fakultät für Physik, Technische Universität München, sep 2016.

[60] Martin Horsch. *Molecular Dynamics Simulation of Heterogeneous Systems.* Habilitation thesis, Laboratory of Engineering Thermodynamics, University of Kaiserslautern, 2016.

[61] Huda Ibeid, Rio Yokota, and David Keyes. A performance model for the communication in fast multipole methods on high-performance computing platforms. *The International Journal of High Performance Computing Applications*, 30(4):423–437, 2016.

[62] Benedikt Jaeger. Implementation and optimization of expansions for the fast multipole method in the Cartesian coordinate system. Bachelor's thesis, Institut für Informatik, Technische Universität München, apr 2015.

[63] James Jeffers and James Reinders. *Intel Xeon Phi coprocessor high performance programming.* Morgan Kaufmann Publishers, 2013.

[64] James Jeffers, James Reinders, and Avinash Sodani. *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition.* Morgan Kaufmann Publishers, 2016.

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

[66] Matthias Kretz. *Extending C++ for explicit data-parallel programming via SIMD vector types.* PhD thesis, Johann Wolfgang Goethe-Universität, Frankfurt am Main, 2015.

[67] Konstantin N Kudin and Gustavo E Scuseria. Revisiting infinite lattice sums with the periodic fast multipole method. *The Journal of chemical physics*, 121(7):2886–2890, 2004.

[68] Jack B Kuipers et al. *Quaternions and rotation sequences*, volume 66. Princeton university press Princeton, 1999.

[69] Jakub Kurzak, Dragan Mirkovic, B Montgomery Pettitt, and S Lennart Johnsson. Automatic generation of FFT for translations of multipole expansions in spherical harmonics. *The international journal of high performance computing applications*, 22(2):219–230, 2008.

[70] Jakub Kurzak and Bernard M Pettitt. Fast multipole methods for particle dynamics. *Molecular simulation*, 32(10-11):775–790, 2006.

[71] Scott Le Grand, Andreas W Götz, and Ross C Walker. SPFP: Speed without compromise — A mixed precision model for GPU accelerated molecular dynamics simulations. *Computer Physics Communications*, 184(2):374–380, 2013.

[72] John Edward Lennard-Jones. On the determination of molecular fields. II. From the equation of state of gas. *Proc. Roy. Soc. A*, 106:463–477, 1924.

[73] M B Liu and G R Liu. Smoothed particle hydrodynamics (SPH): an overview and recent developments. *Archives of computational methods in engineering*, 17(1):25–76, 2010.

[74] H A Lorentz. Ueber die Anwendung des Satzes vom Virial in der kinetischen Theorie der Gase. *Annalen der physik*, 248(1):127–136, 1881.

[75] Hatem Ltaief and Rio Yokota. Data-driven execution of fast multipole methods. *Concurrency and Computation: Practice and Experience*, 26(11):1935–1946, 2014.

[76] Erwin Madelung. Das elektrische Feld in Systemen von regelmäßig angeordneten Punktladungen. *Phys. Z*, 19(524):32, 1918.

[77] MPI Forum. Message Passing Interface (MPI) Forum Home Page. `http://www.mpi-forum.org/`.

[78] Micha Müller. Neighborhood lists in molecular dynamics simulations: SIMD-vectorization. Bachelor's thesis, Institut für Informatik, Technische Universität München, aug 2017.

[79] Jun Nakashima and Kenjiro Taura. MassiveThreads: A thread library for high productivity languages. In *Concurrent Objects and Beyond*, pages 222–238. Springer, 2014.

[80] Philipp Neumann, Hans-Joachim Bungartz, Miriam Mehl, Tobias Neckel, and Tobias Weinzierl. A coupled approach for fluid dynamic problems using the PDE framework peano. *Communications in Computational Physics*, 12(1):65–84, 2012.

[81] Christoph Niethammer, Stefan Becker, Martin Bernreuther, Martin Buchholz, Wolfgang Eckhardt, Alexander Heinecke, Stephan Werth, Hans-Joachim Bungartz, Colin W Glass, Hans 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.

[82] Michael Obersteiner. Parallel cluster detection in nucleation scenarios. Bachelor's thesis, Institut für Informatik, Technische Universität München, sep 2014.

[83] Michael Obersteiner. Parallel implementation of the fast multipole method. Master's thesis, Institut für Informatik, Technische Universität München, 2016.

[84] Michael Obersteiner, Nikola Tchipev, Philipp Neumann, and Hans-Joachim Bungartz. A highly scalable MPI parallelization of the Fast Multipole Method, 2017.

[85] Szilárd Páll and Berk Hess. A flexible algorithm for calculating pair interactions on SIMD architectures. *Computer Physics Communications*, 184(12):2641–2650, 2013.

[86] Simon J Pennycook, Chris J Hughes, Mikhail Smelyanskiy, and Stephen A Jarvis. Exploring SIMD for molecular dynamics, using Intel® Xeon® processors and Intel® Xeon Phi coprocessors. In *2013 IEEE 27th International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1085–1097. IEEE, 2013.

[87] José M Pérez-Jordá and Weitao Yang. A concise redefinition of the solid spherical harmonics and its use in fast multipole methods. *The Journal of chemical physics*, 104(20):8003–8006, 1996.

[88] Ludwig Peuckert. Entwicklung einer dual tree traversal-basierten baumstruktur für die fast multipole methode. Bachelor's thesis, Institut für Informatik, Technische Universität München, 2015.

[89] James C Phillips, Rosemary Braun, Wei Wang, James Gumbart, Emad Tajkhorshid, Elizabeth Villa, Christophe Chipot, Robert D Skeel, Laxmikant Kale, and Klaus Schulten. Scalable molecular dynamics with NAMD. *Journal of computational chemistry*, 26(16):1781–1802, 2005.

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

[91] E L Pollock and Jim Glosli. Comments on P3M, FMM, and the Ewald method for large periodic Coulombic systems. *Computer Physics Communications*, 95(2-3):93–110, 1996.

[92] Antonio Ragagnin, Nikola Tchipev, Michael Bader, Klaus Dolag, and Nicolay J Hammer. Exploiting the space filling curve ordering of particles in the neighbour search of Gadget3. In *Advances in Parallel Computing, Volume 27: Parallel Computing: On the Road to Exascale*, pages 411–420, 2016.

[93] Dennis C Rapaport and Dennis C Rapaport Rapaport. *The art of molecular dynamics simulation*. Cambridge university press, 2004.

[94] Michael Riesch, Nikola Tchipev, Hans-Joachim Bungartz, and Christian Jirauschek. Numerical simulation of the quantum cascade laser dynamics on parallel architectures. In *Proceedings of the Platform for Advanced Scientific Computing Conference*, page 5. ACM, 2019.

[95] Michael Riesch, Nikola Tchipev, Sebastian Senninger, Hans-Joachim Bungartz, and Christian Jirauschek. Performance evaluation of numerical methods for the Maxwell-Liouville-von Neumann equations. *Optical and Quantum Electronics*, 50(2):112, 2018.

[96] Sascha Sauermann. Implementation and optimization of the midpoint method in ls1-mardyn. Bachelor's thesis, Institut für Informatik, Technische Universität München, sep 2017.

[97] Thomas Schilling. Neighbor lists in molecular dynamics simulations: Implementation and parallelization. Bachelor's thesis, Institut für Informatik, Technische Universität München, aug 2017.

[98] Steffen Seckler, Nikola Tchipev, Hans-Joachim Bungartz, and Philipp Neumann. Load balancing for molecular dynamics simulations on heterogeneous architectures. In *2016 IEEE 23rd International Conference on High Performance Computing*, pages 101–110, 2016.

[99] Steffen Seckler, Nikola Tchipev, Matthias Heinen, Fabio Gratl, Hans-Joachim Bungartz, and Philipp Neumann. MPI-OpenMP load balanced simulation of inhomogeneous particle systems in ls1 mardyn at extreme scale. In *SIAM CSE 2019 Spokane*, feb 2019.

[100] David 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.

[101] David E Shaw, Ron O Dror, John K Salmon, J P Grossman, Kenneth M Mackenzie, Joseph A Bank, Cliff Young, Martin M Deneroff, Brannon Batson, Kevin J Bowers, et al. Millisecond-scale molecular dynamics simulations on Anton. In *Proceedings of the conference on high performance computing networking, storage and analysis*, page 39. ACM, 2009.

[102] David E Shaw, J P Grossman, Joseph A Bank, Brannon Batson, J Adam Butts, Jack C Chao, Martin M Deneroff, Ron O Dror, Amos Even, Christopher H Fenton, et al. Anton 2: raising the bar for performance and programmability in a special-purpose molecular dynamics supercomputer. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*, pages 41–53. IEEE Press, 2014.

[103] Dorth Sølvason and Henrik G Petersen. Error estimates for the fast multipole method. *Journal of statistical physics*, 86(1-2):391–420, 1997.

[104] Volker Springel. The cosmological simulation code GADGET-2. *Monthly notices of the royal astronomical society*, 364(4):1105–1134, 2005.

[105] The OpenMP 3.0 Complete Specifications. `https://www.openmp.org/wp-content/uploads/spec30.pdf`.

[106] The OpenMP 4.0 Complete Specifications. `https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf`.

[107] Kenjiro Taura, Jun Nakashima, Rio Yokota, and Naoya Maruyama. A task parallel implementation of fast multipole methods. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 617–625. IEEE, 2012.

[108] Nikola Tchipev, Steffen Seckler, Matthias Heinen, Jadran Vrabec, Fabio Gratl, Martin Horsch, Martin Bernreuther, Colin W Glass, Christoph Niethammer, Nicolay Hammer, Bernd Krischok, Michael Resch, Dieter Kranzlmüller, Hans Hasse, Hans-Joachim Bungartz, and Philipp Neumann. TweTriS: Twenty trillion-atom simulation. *International Journal of High Performance Computing Applications*, 33(5):1094342018819741, 2019.

[109] Nikola Tchipev, Amer Wafai, Colin W Glass, Wolfgang Eckhardt, Alexander Heinecke, Hans-Joachim Bungartz, and Philipp Neumann. Optimized force calculation in molecular dynamics simulations for the Intel Xeon Phi. In *European Conference on Parallel Processing*, pages 774–785. Springer, 2015.

[110] Loup Verlet. Computer “experiments” on classical fluids. I. Thermodynamical properties of Lennard-Jones molecules. *Physical review*, 159(1):98, 1967.

[111] P B Visscher and D M Apalkov. Simple recursive implementation of fast multipole method. *Journal of Magnetism and Magnetic Materials*, 322(2):275–281, 2010.

[112] Jadran Vrabec, Martin Bernreuther, Hans-Joachim Bungartz, Wei-Lin Chen, Wilfried Cordes, Robin Fingerhut, Colin W Glass, Jürgen Gmehling, René Hamburger, Manfred Heilig, Matthias Heinen, Martin T Horsch, Chieh-Ming Hsieh, Marco Hülsmann, Philip Jäger, Peter Klein, Sandra Knauer, Thorsten Köddermann, Andreas Köster, Kai Langenbach, Shiang-Tai Lin, Philipp Neumann, Jürgen Rarey, Dirk Reith, Gábor Rutkai, Michael Schappals, Martin Schenk, Andre Schedemann, Mandes Schönherr, Steffen Seckler, Simon Stephan, Katrin Stöbener, Nikola Tchipev, Amer Wafai, Stephan Werth, and Hans Hasse. SkaSim — Scalable HPC Software for Molecular Simulation in the Chemical Industry. *Chemie Ingenieur Technik*, 90(3):295–306, 2018.

[113] H Y Wang and R LeSar. An efficient fast-multipole algorithm based on an expansion in the solid harmonics. *The Journal of chemical physics*, 104(11):4173–4179, 1996.

[114] Michael S Warren and John K Salmon. Astrophysical N-body simulations using hierarchical tree data structures. In *Supercomputing'92: Proceedings of the 1992 ACM/IEEE Conference on Supercomputing*, pages 570–576. IEEE, 1992.

[115] Stephan Werth. *Molecular Modeling and Simulation of Vapor-liquid Interfaces*. Technische Universität Kaiserslautern, Laboratory of engineering thermodynamics, 2016.

[116] Stephan Werth, Martin Horsch, and Hans Hasse. Molecular simulation of the surface tension of 33 multi-site models for real fluids. *Journal of Molecular Liquids*, 235:126–134, 2017.

[117] Christopher A White and Martin Head-Gordon. Derivation and efficient implementation of the fast multipole method. *The Journal of Chemical Physics*, 101(8):6593–6605, 1994.

[118] Christopher A White and Martin Head-Gordon. Fractional tiers in fast multipole method calculations. *Chemical Physics Letters*, 257(5-6):647–650, 1996.

[119] Christopher A White and Martin Head-Gordon. Rotating around the quartic angular momentum barrier in fast multipole method calculations. *The Journal of Chemical Physics*, 105(12):5061–5067, 1996.

[120] John A White. Lennard-Jones as a model for argon and test of extended renormalization group calculations. *The Journal of chemical physics*, 111(20):9352–9356, 1999.

[121] Rio Yokota. An FMM based on dual tree traversal for many-core architectures. *Journal of Algorithms & Computational Technology*, 7(3):301–324, 2013.

[122] Rio Yokota and L A Barba. Parameter tuning of a hybrid treecode-fmm on GPUs. In *The First International Workshop on Characterizing Applications for Heterogeneous Exascale Systems, Tucson, Arizona*, 2011.

[123] Rio Yokota, George Turkiyyah, and David Keyes. Communication complexity of the fast multipole method and its algebraic variants. *Supercomputing Frontiers and Innovations: an International Journal*, 1(1):63–84, 2014.

[124] Darrin M York, Tom A Darden, and Lee G Pedersen. The effect of long-range electrostatic interactions in simulations of macromolecular crystals: A comparison of the Ewald and truncated list methods. *The Journal of Chemical Physics*, 99(10):8345–8348, 1993.

[125] Wen Zhang and Stephan Haas. Adaptation and performance of the Cartesian coordinates fast multipole method for nanomagnetic simulations. *Journal of Magnetism and Magnetic Materials*, 321(22):3687–3692, 2009.