AnyDSL-MD: Molecular Dynamics Simulations in Impala

Jonas Schmitt

Master Thesis
AnyDSL-MD: Molecular Dynamics Simulations in Impala

Jonas Schmitt
Master Thesis

Aufgabensteller: PD Dr. Harald Köstler
Betreuer: PD Dr. Harald Köstler
Bearbeitungszeitraum: 01.03.2017 – 28.08.2017
Erklärung:

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Der Universität Erlangen-Nürnberg, vertreten durch den Lehrstuhl für Systemsimulation (Informatik 10), wird für Zwecke der Forschung und Lehre ein einfaches, kostenloses, zeitlich und örtlich unbeschränktes Nutzungsrecht an den Arbeitsergebnissen der Master Thesis einschließlich etwaiger Schutzrechte und Urheberrechte eingeräumt.

Erlangen, den 28. August 2017 ......................................
# Contents

1 Introduction ..................................................  8
   1.1 AnyDSL ..................................................  8
   1.2 Molecular Dynamics ........................................  10

2 Background .................................................. 13
   2.1 Numerical integration ...................................... 14
      2.1.1 Time discretization .................................. 14
      2.1.2 The integration of the Newtonian equations of motion 15
   2.2 Force calculation ......................................... 17
      2.2.1 The basic algorithm .................................. 17
      2.2.2 The cutoff radius .................................... 18
      2.2.3 The cell list method .................................. 19
   2.3 Impala .................................................... 22
      2.3.1 Variables ............................................. 22
      2.3.2 Functions ............................................. 22
      2.3.3 References ........................................... 22
      2.3.4 Control structures .................................... 23
      2.3.5 Arrays ................................................. 23
      2.3.6 Records ................................................ 24
      2.3.7 Type conversions and casts ............................ 24
      2.3.8 Partial evaluation .................................... 25
      2.3.9 Interfacing C and C++ code ........................... 25

3 Implementation ............................................... 26
   3.1 Molecular dynamics in Impala ............................. 26
      3.1.1 The Velocity Verlet integration ....................... 29
      3.1.2 The Lennard-Jones potential .......................... 29
   3.2 The basic algorithm ....................................... 31
      3.2.1 Data structures ....................................... 31
      3.2.2 Algorithms ............................................ 32
   3.3 The cell list algorithm .................................... 32
      3.3.1 Data structures ....................................... 32
      3.3.2 Algorithms ............................................ 35
      3.3.3 Algorithms ............................................ 37
      3.3.4 Simulation loop ....................................... 41
   3.4 Integration into the pe physics engine .................... 42

4 Experiments .................................................. 47
   4.1 Single-node performance ................................... 47
   4.2 Multi-node performance ................................... 53

5 Conclusion and future work ................................... 55

References ..................................................... 56

Appendix ......................................................... 60
Figures

1. Deep embedding ........................................ 9
2. Shallow embedding ..................................... 9
3. Visualization of a system with 1000 particles placed in a quadratic box of size 100. .. 13
4. Lennard-Jones potential with the parameters $\varepsilon = 1$ and $\sigma = 1$. ................. 19
5. Geometric decomposition of a three dimensional simulation area into cells. .......... 20
6. A singly linked list. The sixth node represents the end of the list. ................. 21
7. Visualization of a 2D domain (white), extended with a ghost layer (gray). ............ 34
8. Two cases in which a particle is located in an inner cell (blue). The neighbors that need to be traversed for the force calculation are colored red. .................. 38
9. Two cases in which a particle is located in a ghost cell (blue). The neighbors that need to be traversed for the force calculation are colored red. .................. 40
10. Domain decomposition of a 2D domain into four quadratic subdomains. The gray cells must be communicated. ........................................ 43
11. Simulation of a uniform cubic grid of 10000 particles. ................................. 48
12. Simulation of the collision of two bodies that in total consist of 10000 particles. ... 48
13. Simulation of 1000 randomly initialized particles. ...................................... 49
14. Measured runtime with an increasing number of particles. .............................. 49
15. Single-node scaling in the grid test case. ............................................... 51
16. Single-node scaling in the collision test case. .......................................... 51
17. Single-node scaling in the random test case. ........................................... 52
18. Weak scaling in a multi-node setting. .................................................... 53
19. Simulation of $10^7$ particles that are initially positioned in a grid. ................. 54

Tables

1. Impala: Strong scaling with four processors. ........................................... 50
2. C: Strong scaling with four processors. .................................................. 50
3. Impala: Weak scaling with four processors. ............................................ 50
4. C: Weak scaling with four processors. ................................................... 50
5. Impala: Weak scaling in a multi-node setting. .......................................... 53

Algorithms

1. Basic Molecular Dynamics Algorithm ................................................... 11
2. Velocity Verlet algorithm ................................................................. 17
3. Simple force calculation ................................................................. 17
4. Improved simple force calculation ..................................................... 18
5. Cell list force calculation ................................................................. 20
Code listings

1 List iterator in Impala ........................................... 9
2 List iterator after partial evaluation ........................... 10
3 Data type declaration of a singly linked list in C .......... 21
4 List insertion and removal in C ................................. 21
5 Variables ......................................................... 22
6 Functions ......................................................... 22
7 References ....................................................... 23
8 Control Structures ................................................ 23
9 Arrays ............................................................ 24
10 Records .......................................................... 24
11 Type conversions and casts ...................................... 24
12 Partial evaluation ................................................ 25
13 Interfacing C/C++ code in Impala ............................... 25
14 Interfacing Impala code in C++ ................................. 25
15 Type definitions and constants ................................. 26
16 Iteration and index functions ................................... 27
17 Vector interface .................................................. 27
18 Vector operations ................................................ 27
19 Vector implementation .......................................... 28
20 Particle interface ................................................. 28
21 Velocity Verlet integration ..................................... 29
22 Lennard-Jones force ............................................. 30
23 Basic algorithm: Particle system interface definition ...... 31
24 Basic algorithm: Particle system implementation .......... 31
25 Basic algorithm: Integration .................................... 32
26 Basic algorithm: Force calculation ............................. 32
27 Linked list ......................................................... 33
28 Insertion into a linked list ...................................... 33
29 Removal from a linked list ...................................... 33
30 Cell list algorithm: Particle system interface definition .. 35
31 Cell list algorithm: Particle system implementation ...... 35
32 Array of pointers ................................................ 36
33 Traversal of a linked list ....................................... 36
34 Iteration over cells .............................................. 36
35 Iteration over a particle system ................................. 37
36 Iteration over a particle system with ghost layers ......... 37
37 Cell list algorithm: Integration ................................ 38
38 Cell list algorithm: Force calculation ......................... 39
39 Cell list algorithm: Cell position computation .............. 40
40 Cell list algorithm: Particle distribution .................... 41
41 Simulation loop .................................................. 41
42 Record for storing the internal data of a particle system ... 44
43 External interface for setting the coordinates of a particle with index i ................................. 44
44 External interface for obtaining the coordinates of a particle with index i ............................. 44
45 External interfaces to the components of the cell list algorithm ........................................... 45
46 pe simulation loop ............................................... 46
47 Particle implementation .......................................... 60
48 External interface for the reinitialization of a particle system ........................................ 61
49 C implementation: Data structures ............................. 62
50 C implementation: Velocity Verlet integration ................ 62
51 C implementation: Lennard-Jones force ....................... 63
52 C implementation: Particle redistribution .................... 64
53 C implementation: Integration of position and velocity ...... 64
54 C implementation: Force calculation ........................... 65
Abstract

This work presents a novel approach for the generation of high performance molecular dynamics code based on AnyDSL, a framework for the rapid development of domain specific languages. As foundation serves a high-level implementation in AnyDSL’s front-end language Impala, which permits the choice of algorithms and optimization strategies on different levels of abstraction. As a proof-of-concept, the implementation of a simple $O(N^2)$ force evaluation method and a shared-memory parallel version of the cell list algorithm is discussed. To investigate the performance of the generated code, it is benchmarked against a similar implementation, written in the C programming language, in a number of test cases. Furthermore, the interoperability with other frameworks is demonstrated by means of the integration into pe, a massively parallel framework for the simulation of rigid bodies. The resulting implementation is tested on a small-sized cluster by running simulations with up to $10^7$ particles. Finally, current limitations and necessary improvements with respect to the multi-node scalability of the implementation are discussed.
1 Introduction

In 2004, Phillip Coella identified seven numerical methods of major importance in science and engineering which became famous as the seven dwarfs of HPC\cite{1}. N-body methods which depend on computing the interactions between a large number of discrete points are one of these dwarfs. Many interesting phenomena can not be described in the continuous domain and therefore need to be modeled as a system of interacting bodies (particles), from galaxies and stars to molecules and atoms. Here, the number of particles can readily reach the order of magnitude of multiple millions or more. For example, every cubic meter of gas contains approximately $2.69 \times 10^{25}$ molecules (Loschmidt constant) and our own galaxy, the milky way, alone consists of 200 billion stars. The fundamental mathematical model for the simulation of these systems is Newton’s second law (equation 1), which describes the dependence of a particle’s acceleration from the force acting on it.

$$F = m \cdot \frac{dv}{dt} = m \cdot a$$

(1)

To simulate its development over time, this system of ordinary differential equations needs to be approximated numerically. After discretizing the time derivatives in a suitable way, this is achieved by calculating the force acting on every single particle during each time step, which results from the interactions between them. Naively, this requires the computation of $O(N^2)$ interactions during each time step, where $N$ is the number of particles. However, the potential fields responsible for these interactions can be of different range and thus, dependent on their characteristics, only a limited number of particles influence each other. To permit the simulation of large systems, the use of data structures which enable a fast and memory efficient computation of particle-particle interactions with different range is indispensable. The implementation of these methods imposes significant challenges on modern architectures. Each molecular dynamics implementation needs to be optimized with respect to the present force fields and the compute environment the implementation is supposed to be executed in. Both C/C++ and Fortran, the predominant programming languages in high performance computing are based on an inherently static and sequential model of computation. To deal with the increasing architectural complexity of modern architectures, these languages have been augmented with various extensions, most notably in form of parallel programming libraries like OpenMP, MPI or Cuda. Although, they allow targeting new architectures, their power and expressiveness is limited by the library and they are by no means extensible. In order to overcome these limitations, new programming models and tools which make software both reusable and extensible, while also meeting the performance requirements of large scale HPC systems, are necessary. AnyDSL is a framework that aims to facilitate the development of domain specific languages (DSLs). It grants the ability to create functional abstractions over multiple levels of hierarchies while the overhead is completely removed by utilizing a novel partial evaluator. Therefore, performance optimization boils down to the specialization of an implementation to the given architectural requirements. The goal of this thesis is to apply this approach to the domain of N-body methods by providing an implementation, that offers different algorithmic options while achieving comparable performance to an overhead-free C implementation, in AnyDSL’s front-end language Impala, Furthermore, this implementation should allow for an easy integration into existing parallel HPC frameworks, which is demonstrated by integrating it into the pe physics engine \cite{2}\cite{3}.

1.1 AnyDSL

AnyDSL is a framework that aims to enable the fast and easy development of domain specific languages (DSL). A domain specific language is a programming language specialized to a certain application domain. While the use of DSLs has a long tradition in certain domains, as for example markup languages for creating web pages and applications, recently there has been an increasing interest in the design of DSLs for high performance computing applications. To meet the requirements of these applications, a DSL does not only have to offer abstractions, but it must enable the generation of highly optimized (parallel) code. This usually restricts the use of HPC-DSLs to narrow domains. An example are DSLs for the domain of stencil codes \cite{4}\cite{5}\cite{6}\cite{7}, which frequently arise in the numerical solution of partial differential equations. In principle, a domain specific language can either be implemented as a stand-alone tool with a dedicated compiler, or embedded in
an already existing programming language. The first approach allows for more flexibility, because syntax and semantic of the DSL can be freely chosen, but in return it can not be easily integrated into existing programs written in other languages. In contrast, embedding a DSL into an existing language increases its usability, but puts restrictions onto its syntax. In general, there are two different approaches for embedding a domain specific language: Deep and shallow.

Deep: The DSL program is represented as data structure in the host language (figure 1). The programmer writes a program generator $p_{gen}$ in the host language that constructs the embedded DSL program $e_{spec}$ while partially evaluating it with respect to the input $s$. Next, an optimizer $opt$ transforms $e_{spec}$ to $e_{opt}$, that can then finally be compiled to target code. The availability of the embedded program as a data structure facilitates extensive optimizations. Furthermore, for the same reason, deep embedding grants the flexibility to embed any language. The drawback of its implementation is that it essentially constitutes writing a program generator within the host language [8].

Shallow: The semantics of the DSL are directly implemented as a program in the host language. The programmer writes the embedded program $e$ in the host language which is then partially evaluated with respect to the input $s$. This requires the availability of a partial evaluator $mix$ in the compiler of the host language. Both the optimizer $opt$ and the code generator $compile$ are integrated functions of the compiler of the host language. Unlike deep embedding, shallow embedding only enables embedding programs written in the host language itself. Because the embedded program is not available as a data structure, it can only be manipulated and hence optimized with respect to the syntax and semantics of the host language. In return, shallow embedding offers a better programming experience, because the programmer writes an actual program and not a generator. Although, the requirement for this approach is the availability of a partial evaluator $mix$ in the compiler of the host language.

\[
\begin{align*}
\triangleright \text{code in the host language} & \quad \triangleright \text{code in the host language’s compiler} \\
e_{\text{spec}} & := p_{\text{gen}}(s) \\
e_{\text{opt}} & := \text{opt}(e_{\text{spec}}) \\
\text{compile}(e_{\text{opt}}) & \\
\end{align*}
\]

Figure 1: Deep embedding

\[
\begin{align*}
\triangleright \text{code in the host language’s compiler} & \quad \triangleright \text{code in the host language’s compiler} \\
e_{\text{spec}} & := \text{mix}(e, s) \\
e_{\text{opt}} & := \text{opt}(e_{\text{spec}}) \\
\text{compile}(e_{\text{opt}}) & \\
\end{align*}
\]

Figure 2: Shallow embedding

AnyDSL offers the continuation-passing style based language Impala together with a novel partial evaluator. Impala [9] is an imperative and functional programming language whose syntax is inspired by Rust [10]. In Impala, domain specific constructs can be implemented as higher-order functions, which essentially constitutes writing a tagless interpreter, as described in [11]. For example, to implement a list iterator, the programmer has to write a program that "interprets" a function on all elements of a list. The program is then specialized by applying partial evaluation [12], whereas its specialized version only contains statements that depend on variables not known at compile time. This step is performed online, which means the partial evaluator acts as a interpreter for the static fraction of the source code by actually executing it while emitting code for the residual program. In Impala, partial evaluation can be triggered by annotating source code with the symbol. This delegates the decision when to apply program specialization to the programmer. In context of shallow embedding, this steps corresponds to calling $mix$ with the embedded program $e$ and the (static) input $s$ as arguments. Consider the list iterator illustrated in listing 1, assuming the list is a three element list defined at compile time, utilizing partial evaluation, the code can be specialized to the residual program shown in listing 2.

Listing 1 List iterator in Impala

```rust
fn iterate_over_list(ptr: &LinkedList, body: fn(&LinkedList) -> () -> i) { 
    if (ptr != 0 as * &LinkedList) {
        body(ptr);
        iterate_over_list((*ptr).next, body, return)
    }
}
```

9
The details of the partial evaluator integrated into Impala, including its termination properties, are described in [13]. As compiler back end AnyDSL offers Thorin, a higher-order intermediate representation based on continuation-passing style (CPS) [14]. In general, Impala code is first translated to Thorin, where a number of transformations take place that remove the overhead associated with the use of higher-order function. By exploiting the correspondence between CPS and the static single assignment (SSA) form [15], Thorin can be converted to SSA. SSA finds widespread use as a low-level intermediate representation. This facilitates the integration of Thorin into many existing compilers. By default, Thorin uses LLVM [16] as a backend. So far, AnyDSL has been used to implement a number of applications. In [17] and [18], a platform-specific optimization of stencil codes is represented. [19] extends this approach to the generation of multigrid solvers.

1.2 Molecular Dynamics

An important part of numerical simulation is based on so-called N-body or particle models. A large subset of these methods has found widespread use in the simulation of molecular structures and therefore has been grouped together under the term molecular dynamics, even if the particles do not refer to atoms or molecules. In molecular dynamics, the physical system is solely represented by a number of discrete particles and their mutual interactions. Thereby, the system’s current state and its evolution can be fully described by the physical quantities of the individual particles it consists of. Even though these particles can be considered as building blocks of an abstract model, they often correspond to actual physical entities. Depending on the simulation’s scale, they can refer to atoms and molecules or to stars and even parts of whole galaxies. The particles then exhibit properties of the corresponding entity like mass, position, velocity and charge. In many particle systems, the laws of classical mechanics are assumed, in particular Newton’s second law 1. This leads to a system of second-order ordinary differential equations which describes the dependency of a particle’s acceleration on the force acting on it. This force results from the interaction with other particles and therefore depends on the position. When the relative positions of the particles change, in general, the forces also change. Hence, by solving this system of ordinary differential equations, the trajectories of all particles can be obtained deterministically for all time points, assuming certain initial conditions. However, to accurately model the behavior of atoms and molecules, the laws of quantum mechanics need to be assumed and equation 1 is replaced by the Schrödinger equation. Unfortunately, this equation is so complex that its numerical solution is restricted to certain cases and low particle numbers, which are collected under the term ab initio molecular dynamics[20] to distinguish them from classical molecular dynamics, that is based on the laws of classical mechanics. To avoid this difficulty, classical molecular dynamics uses an approximation based on the assumption that electrons are significantly lighter than atomic nuclei. The Schrödinger equation which describes the state of electrons and nuclei is separated into two coupled equations (Born-Oppenheimer approximation) [21]. The influence of the electrons on the interactions between nuclei is described by an effective potential, which is derived as an approximation of the respective part of the Schrödinger equation. As a second approximation, it is assumed that the nuclei move according to equation 1 and a potential, that is either directly derived from the laws of quantum mechanics (potentially incorporating the influence of the electrons) or of empirical nature, is assumed. An overview of the derivation of classical molecular dynamics from the Schrödinger equation can be found in [22][23][24].
The following physical systems are typical examples for the application of particle methods:

**Materials science:** In materials science, molecular dynamics simulations are often used to analyze the properties of known or newly developed materials. Examples for simulated phenomena are temperature- or strain-induced structural changes in metals, crack formation during fracture experiments, the propagation of sonic waves, the impact of defects on the structure and strength of materials, and the analysis of elastic and plastic deformation. An overview can be found in [25].

**Fluid dynamics:** Particle methods can serve as an alternative to classical mesh-based methods in computational fluid dynamics. They allow the investigation of complex fluids, fluid mixtures and phase transitions on a microscopic scale. Examples are smoothed particle hydrodynamics [26] and the moving particle semi-implicit method [27].

**Biochemistry:** The simulation of the motion of macromolecules on an atomic level is one of the most prominent applications of molecular dynamics. The availability of an increasing amount of computational power enables the simulation of biological and chemical processes of increasing complexity. Examples are the simulation of viral assembly [28], ribosome translation [29], and photosynthesis [30].

**Astrophysics:** In astrophysics, simulations are especially used to evaluate theoretical models with regard to their reliability. For example, in a simulation of the development of the universe’s structure particles can correspond to stars or even whole galaxies [31]. Usually, the force between particles results from the gravitational potential.

In a computer simulation of particle models, the development of a system of interacting particles is approximated by integrating the equations of motion over time. Hence, the main components of the simulation of particle models are the time integration of the Newtonian equations of motion and the fast evaluation of particle interactions.

**Time integration:** During the numerical integration, the solution of the governing differential equations is computed at discrete points of time. For this purpose, the current solution is approximated from the one of the previous time step, using an appropriate integration method. Thereby, the forces acting on each particle need to be computed. This is achieved by forming the negative gradient of the system’s potential function. By assuming that $x_i$, $v_i$ and $F_i$ denote the position, velocity of a particle $i$ and the force acting on it, respectively, the basic algorithm for calculating the trajectories of $N$ particles can be formulated.

\[ t = 0 \]
\[ \text{for } i = 1, \ldots, N \text{ do} \]
\[ \text{Set } x_i \text{ and } v_i \text{ according to the initial conditions} \]
\[ \text{while } t < t_{\text{end}} \text{ do} \]
\[ \text{for } i = 1, \ldots, N \text{ do} \]
\[ \text{Compute } F_i \text{ according to the existing potential} \]
\[ \text{for } i = 1, \ldots, N \text{ do} \]
\[ \text{Compute } x_i, \text{ } v_i \text{ at } t + dt \text{ by numerical integration, using the old values of } x_i, \text{ } v_i \text{ and } F_i \]
\[ t = t + dt \]

**Force evaluation:** In principle, there exist $N^2$ mutual interactions between particles, in a system of $N$ particle, from which the interactions of particles with themselves need to be subtracted. Therefore, $O(N^2)$ force evaluations are required in each time step and doubling the number of particles results in a fourfold increase of the number of operations. Because of this unfavorable complexity, it impossible to scale the algorithm to large particle numbers. Although, by restricting the force computation to a certain accuracy, substantial complexity reductions can be achieved.

Apparently, the minimal complexity of an approximate force calculation is $O(N)$ in the number of particles, as the force for each particle needs to be computed at least once. An algorithm whose complexity for achieving a certain accuracy is proportional to $N$ is called optimal. If an algorithm’s complexity only differs from the optimal complexity by a logarithmic factor, i.e. the
complexity is $O(N \log(N)^\alpha)$ with $\alpha > 0$, it is called quasi-optimal. The procedure of designing an appropriate algorithm must be oriented towards the characteristics of the present interactions and other relevant parameters like the distribution of particles over the simulated domain. Evidently, a class of algorithms that is optimal for a certain kind of interactions can perform significantly worse in a different setting. The most obvious difference exists between potentials that drop fast with increasing distance from its center and slowly dropping ones. In the former case, a particle only significantly contributes to the force that acts on another particles if the distance between them is small. If the particles are evenly distributed over the domain, the evaluation of the forces caused by a rapidly dropping potential can be performed in $O(N)$ operations, because only interactions with neighboring particles must to be considered. In contrast, slowly dropping potentials, like the coulomb or gravitational potential, can not be ignored after a certain distance. Therefore, the utilization of data structures that allow a fast force evaluation, that exploits the characteristics of the governing potential, is one of the main challenge in the design of an efficient molecular dynamics algorithm.

In principle, algorithms are grouped into short-range and long-range interaction methods. Typical representatives of the former group are methods based on neighbor [32][33] or cell lists [34][35], that utilize dedicated data structures to track the neighborhood relations between particles. In the neighbor (or Verlet) list method, according to the cutoff distance of the present potential, a list of neighbors is stored for each particle. The cell list (or linked-cell) method subdivides the simulated domain into cells with a side length greater than or equal to the cutoff distance of the potential. Therewith, only particles in the same and neighboring cells need to be considered for the computation of all interactions. Which method performs better in a certain case depends on the potential and the performance characteristics of the target architecture. A combination of both methods is also possible and has been suggested in [36]. The most common methods for the computation of long-range interactions are either mesh- or tree-based. Mesh-based methods convert the system of particles into a density grid and then compute the potential function on this grid instead of for each individual particle. The forces are then assigned to each particle according to the grid cell it is positioned in. Most mesh-based methods use Ewald summation, where the interaction is divided into a short-range and a long-range contribution. The former one is computed in real space and the latter one in the Fourier space [37]. Representatives of mesh-based methods are the particle mesh Ewald method [38], the particle–particle particle–mesh method [39]. In contrast, tree-based methods recursively subdivide the domain into cubic cells via a tree data structure (for example an octree in a three dimensional space). Particles in nearby cells are treated individually while particles in distant cells can in sum be considered as superparticle[40].

In molecular dynamics, typically the majority of the compute time is spent on the evaluation of pairwise non-bonded interactions\(^1\) associated with intermolecular forces and electrostatic charge [41]. Intermolecular forces, including Van der Waals forces and dipole-dipole interactions, are even present in case of the absence of charged particles and thus need to be considered in most molecular dynamics simulations. The underlying potentials are often modeled as short-range interactions that can be efficiently computed using the aforementioned methods. Widely used frameworks as GROMACS[42], LAMMPS [43], NAMD[44], CHARMM[45] and AMBER[46] are implemented in C++ or Fortran. To overcome the limited expressiveness of these languages, a lot of effort has been invested in the support of different force fields and integration methods while reaching good performance on a wide range of architectures. As it has been discussed in the last section, there exist a number of techniques to implement domain specific languages embedded into a host language. The AnyDSL-framework offers the host language Impala that allows an overhead-free shallow embedding of DSLs using partial evaluation and targets Thorin, a novel higher-order intermediate representation. Hence, it aims to grant a programmer the full expressiveness of a custom DSL without sacrificing performance compared to low-level imperative languages such as C and Fortran. In the field of molecular dynamics, the concept of domain specific languages so far has not found much adaption and only a few DSLs have been implemented [47][48]. The goal of this thesis is to evaluate the suitability of the AnyDSL-framework for the generation of high performance molecular dynamics code based on short-range interactions while focusing on the cell list algorithm.

\(^1\) Non-bonded interactions are present between atoms that are not linked by covalent bounds.
2 Background

As it has been discussed in the last section, the fast computation of particle interactions is essential for the performance of a molecular dynamics simulation. Furthermore, a suitable time integration scheme must be chosen, to accurately determine the particle trajectories. Both problems are considered in detail in the following section.

Given is a system of \( N \) particles with the masses \( \{m_1, \ldots, m_N\} \), that is characterized by the positions \( \{x_1, \ldots, x_N\} \) and velocities \( \{v_1, \ldots, v_N\} \). Both the \( x_i \) as well as \( v_i \) are two or three dimensional vectors (one dimension for each direction in space) and functions of time. In the following, vectors are always printed in a bold font to distinguish them from scalars. Each point in this \( 4N \) or \( 6N \) dimensional phase space represents a possible configuration of the system. The simulation area is assumed to be rectangular with \( \Omega = [0, L_1] \times [0, L_2] \) in two dimensions and with \( \Omega = [0, L_1] \times [0, L_2] \times [0, L_3] \) in three dimensions. Figure 3 shows an example for the three dimensional case. The color corresponds to the velocity magnitude whereas red denotes a high and blue a low value.

\[
\begin{align*}
\frac{\partial x_i}{\partial t} &= \nabla_{p_i} \mathcal{H}, \\
\frac{\partial p_i}{\partial t} &= -\nabla_{x_i} \mathcal{H}, \quad i = 1, \ldots, N,
\end{align*}
\]

Figure 3: Visualization of a system with 1000 particles placed in a quadratic box of size 100.

Dependent on the problem, different conditions at the boundaries of the system are possible. For example in periodic systems, like crystals, periodic boundary conditions are a natural choice. Here, particles that leave the simulation area reenter it at the opposite side and those located at opposite sides interact with each other. In contrast, in the presence of reflecting boundary conditions, particles that approach a boundary experience a certain repulsive force. In and outflow boundary conditions are used if particles can enter or leave the simulation area. Beyond that, there are a number of problem specific boundary conditions that are not further discussed here.

We now assume that the temporal development of the system within the region \( \Omega \) is described by the Hamiltonian equations\(^2\)

\[
\begin{align*}
\frac{\partial x_i}{\partial t} &= \nabla_{p_i} \mathcal{H}, \\
\frac{\partial p_i}{\partial t} &= -\nabla_{x_i} \mathcal{H}, \quad i = 1, \ldots, N,
\end{align*}
\]

where \( \mathcal{H} \) is a Hamilton function. If the interactions between particles are described by a (conservative) potential, which is not explicitly time dependent

\[
V = V(x_1, \ldots, x_N)
\]

\(^2\) \( \nabla_Y F \) denotes the application of the nabla operator on \( F \) as a function of \( Y \)
and if Cartesian coordinates and velocities are assumed, the Hamilton function $H$ is

$$H(x_1, \ldots, x_N, p_1, \ldots, p_N) = \sum_{i=1}^{N} \frac{p_i^2}{2m_i} + V(x_1, \ldots, x_N) = E_{\text{kin}} + E_{\text{pot}}.$$  \(4\)

With $p_i = m_i v_i$, the Newtonian equations of motions can be directly derived from (2)

$$\frac{\partial x_i}{\partial t} = v_i, \quad i = 1, \ldots, N \quad (5)$$

$$m_i \frac{\partial v_i}{\partial t} = m_i \frac{\partial^2 x_i}{\partial t^2} = F_i, \quad (6)$$

whereas in case of a conservative potential, the forces only depend on the coordinates:

$$F_i = -\nabla_{x_i} V(x_1, \ldots, x_N). \quad (7)$$

If the initial positions and velocities of all particles are known, the temporal development of the system solely depends on the governing potentials. The Hamilton function (4) represents the total energy of the system and consists of two terms, the potential energy given by the evaluation of the potential $V$ at the particle positions and the kinetic energy

$$E_{\text{kin}} = \sum_{i=1}^{N} \frac{p_i^2}{2m_i} = \sum_{i=1}^{N} \frac{1}{2} m_i v_i^2. \quad (8)$$

Derivation of $H$ with respect to time leads to

$$\frac{dH}{dt} = \frac{dE}{dt} = \frac{dE_{\text{kin}}}{dt} + \frac{dE_{\text{pot}}}{dt} = \sum_{i=1}^{N} m_i v_i \frac{\partial v_i}{\partial t} + \sum_{i=1}^{N} \frac{\partial V}{\partial x_i} \frac{\partial x_i}{\partial t} + \sum_{i=1}^{N} \nabla_{x_i} V \cdot \frac{\partial x_i}{\partial t}. \quad (9)$$

For systems with potentials of the form (3), $\partial V/\partial t = 0$ applies. Furthermore, by insertion of (7), equation (9) reduces to

$$\frac{dE}{dt} = \sum_{i=1}^{N} (m_i \frac{\partial v_i}{\partial t} + \sum_{i=1}^{N} \nabla_{x_i} V) v_i$$

$$= \sum_{i=1}^{N} (m_i \frac{\partial v_i}{\partial t} + \sum_{i=1}^{N} F_i) v_i = 0.$$

Therefore, the energy $E$ is a conserved quantity and does not change during the temporal development of system. Now, before considering how the fast evaluation of the forces $F_i$ can be achieved, a suitable time integration scheme must be chosen.

### 2.1 Numerical integration

In numerical mathematics, the term discretization describes the transition from a problem in the continuous domain to a problem that is only considered on a finite number of points. Usually, this means transforming a differential equation into a system of linear equations, whose solution approximates the equation on certain points. In connection with molecular dynamics, this refers to the computation of the new positions and velocities of the particles based on the old positions, velocities and the corresponding forces.

#### 2.1.1 Time discretization

The first step is to decompose the time interval $[0, t_{\text{end}}] \subset \mathbb{R}$, on which the system of differential equations (5) and (6) is supposed to be solved, in $I$ sub-intervals of equal size $\delta t := t_{\text{end}}/I$. This results in a uniform grid with the points $t_n := n \cdot \delta t$, $n = 0, \ldots, I$, on which the differential equations are considered. According to the definition of the total derivative of a function $x: \mathbb{R} \to \mathbb{R}^k$

$$\frac{dx}{dt} := \lim_{\delta t \to 0} \frac{x(t + \delta t) - x(t)}{\delta t},$$

14
the forward difference approximation of the differential operator at a grid point \( t_n \) leads to

\[
\left[ \frac{dx}{dt} \right]_n^f := \frac{x(t_{n+1}) - x(t_n)}{\delta t}.
\]

Here, \( t_{n+1} = t_n + \delta t \) is the right neighboring grid point of \( t_n \). Taylor expansion of the function \( x \) around point \( t_{n+1} \) according to (11) reveals a discretization error of order \( O(\delta t) \) for the approximation of the first order derivative:

\[
x(t_n + \delta t) = x(t_n) + \delta t \frac{dx}{dt}(t_n) + O(\delta t^2).
\]

Therefore, by decreasing the time step size, an error reduction of the same order can be achieved.

Alternatively, the differential operator \( dx/dt \) can also be approximated by applying a central difference scheme:

\[
\left[ \frac{dx}{dt} \right]_n^c := \frac{x(t_{n+1}) - x(t_{n-1})}{2\delta t}.
\]

In this case, Taylor expansion reveals a discretization error of order \( O(\delta t^2) \) for the approximation of the first order derivative.

Finally, an approximation of the second order derivative \( d^2x/dt^2 \) can be obtained through the difference quotient

\[
\left[ \frac{d^2x}{dt^2} \right]_n := \frac{1}{\delta t^2} (x(t_n + \delta t) - 2x(t_n) + x(t_n - \delta t)).
\]

Taylor expansion around the points \( t_n + \delta t \) and \( t_n - \delta t \) up to the third order leads to

\[
x(t_n + \delta t) = x(t_n) + \delta t \frac{dx(t_n)}{dt} + \frac{1}{2} \delta t^2 \frac{d^2x(t_n)}{dt^2} + \frac{1}{6} \delta t^3 \frac{d^3x(t_n)}{dt^3} + O(\delta t^4)
\]

and

\[
x(t_n - \delta t) = x(t_n) - \delta t \frac{dx(t_n)}{dt} + \frac{1}{2} \delta t^2 \frac{d^2x(t_n)}{dt^2} - \frac{1}{6} \delta t^3 \frac{d^3x(t_n)}{dt^3} + O(\delta t^4).
\]

Inserting both terms into (13) reveals that the discretization error for the approximation of the second order derivative with finite differences is of order \( O(\delta t^2) \):

\[
\left[ \frac{d^2x}{dt^2} \right]_n = \frac{d^2x(t_n)}{dt^2} + O(\delta t^2).
\]

### 2.1.2 The integration of the Newtonian equations of motion

In case of a system of ordinary differential equations of the form (6), the differential quotient can be replaced by the difference quotient at every point \( t_n \), \( n = 1, \ldots, I - 1 \). Thus, by means of equation (13), the position at time \( t_{n+1} \) can be computed from the positions at time \( t_n \) and \( t_{n-1} \) and the force at time \( t_n \). Using the abbreviation \( f^i_n := f_i(t_n) \) for the positions \( x_i(t_n) \), velocities \( v_i(t_n) \) and forces \( F_i(t_n) \) one obtains

\[
m_i \frac{1}{\delta t^2} (x_i^{n+1} - 2x_i^n + x_i^{n-1}) = F_i^n
\]

and thereby

\[
x_i^{n+1} = 2x_i^n - x_i^{n-1} + \delta t^2 \cdot \frac{F_i^n}{m_i}.
\]

Assuming the starting positions \( x_i^0 \) and the positions during the first time step \( x_i^1 \) are known, then all following positions can be uniquely determined with equation (14), which is the standard form of the Störmer-Verlet method for the integration of the Newtonian equations of motion [49][32]. This scheme requires storing the positions at \( t_n \) and \( t_{n-1} \), as well as the force at time \( t_n \). A disadvantage of this formulation is its sensitivity to rounding errors, because a small force term \( \delta t^2 \cdot F_i^n/m_i \) is added to two significantly greater terms \( 2x_i^n \) and \( x_i^{n-1} \), that are independent of the time step size. Furthermore, equation (14) does not contain any velocities, which are for example required
for the computation of the kinetic energy (8). The velocity, as derivative of the position, can be approximated with the central difference scheme
\[ v^n_i = \frac{x^{n+1}_i - x^{n-1}_i}{2\delta t}. \]  
(15)

There exist two alternative formulations of the Störmer-Verlet method. The first one is the so-called Leapfrog integration [50], in which the velocities are evaluated at half steps \( t + \delta t/2 \). Thereby first, the velocities \( v^{n+1/2}_i \) are computed using the velocities at time \( t_{n-1/2} \) and the force at time \( t_n \) according to
\[ v^{n+1/2}_i = v^{n-1/2}_i + \frac{\delta t}{m_i} F^n_i. \]  
(16)

Then, the positions \( x^{n+1}_i \) are obtained using the previously computed velocities and the positions at time \( t_n \):
\[ x^{n+1}_i = x^n_i + \delta t v^{n+1/2}_i. \]  
(17)

Compared to the standard formulation (14), the effect of rounding errors is reduced and the velocities are explicitly calculated. However, positions and velocities are obtained at different points of time. Hence, to determine the potential and kinetic energy, the velocities at \( t_n \) must additionally be computed through averaging.

The second formulation is the so-called Velocity Verlet formulation [51]. By solving equation (15) for \( x^{n-1}_i \) and inserting the result into (14) one obtains
\[ x^{n+1}_i = x^n_i + \delta t v^n_i + \frac{F^n_i \delta t^2}{2m_i}. \]  
(18)

and from equation (14) and the central differencing approximation (15)
\[ v^n_i = \frac{x^n_i - x^{n-1}_i}{\delta t} + \frac{F^n_i \delta t}{2m_i}. \]

By adding the respective term for \( v_{i+1} \) and again utilizing (15), the following relation is derived:
\[ v^{n+1}_i = v^n_i + \frac{(F^n_i + F^{n+1}_i) \delta t}{2m_i}. \]  
(19)

In combination, equation 18 and 19 yield the Velocity Verlet method.

The standard formulation (14), (15) and the leapfrog scheme (17), (16) possess approximately the same storage requirements, whereas the velocity variant (18), (19) needs an additional field for the previously calculated forces\(^3\). The accuracy of all three variants is of second order \( (O(\delta t^2)) \). The Velocity Verlet variant has beneficial properties, it is both robust against rounding errors and the positions and velocities are available at the same time.

The algorithm for integrating the equations of motion, based on the Velocity Verlet method, is summarized in algorithm 2. Thermodynamic parameters, like the potential and kinetic energy, can be computed at the end of each time step
\[ E_{\text{kin}}^n = \frac{1}{2} \sum_{i=1}^{N} m_i (v^n_i)^2 \]
\[ E_{\text{pot}}^n = V(x^n_1, \ldots, x^n_N), \]
where \( V(\cdot) \) is the potential function of the system.

\(^3\) If the kinetic energy is not required, it is not necessary to keep track of the velocity at every time step and hence the memory consumption of all three variants is the same.
Algorithm 2 Velocity Verlet algorithm

1: \( t = 0 \)
2: for \( i = 1, \ldots, N \) do
3:   Set \( x_i \) and \( v_i \) according to the initial conditions
4: for \( i = 1, \ldots, N \) do
5:   Compute the initial forces \( F_i \)
6: while \( t < t_{\text{end}} \) do
7:   for \( i = 1, \ldots, N \) do
8:     \( x_i := x_i + \delta t (v_i + \frac{1}{2m_i} F_i \delta t) \)
9:     \( F_i^{\text{old}} := F_i \)
10: for \( i = 1, \ldots, N \) do
11:   Compute \( F_i \) according to the existing potential
12: for \( i = 1, \ldots, N \) do
13:   \( v_i := v_i + \delta t \frac{1}{2m_i} (F_i + F_i^{\text{old}}) \)
14: \( t := t + dt \)

2.2 Force calculation

After choosing a suitable time integration scheme, the last missing link for the formulation of a complete molecular dynamics algorithm is the computation of particle interactions. As it has been discussed in section 1, this step is critical for the performance of a particle simulation and requires careful assessment of data structures and algorithmic components. First, a simple force calculation scheme, which requires \( \mathcal{O}(N^2) \) force evaluations per time step, is introduced. To overcome its limitations, the cell list method, which allows computing all short-range particle interactions in \( \mathcal{O}(N) \), is explained in detail.

2.2.1 The basic algorithm

Naively, the force between each pair of particles can be computed by evaluating the corresponding potential \( U(r_{ij}) \) as a function of the distance between them:

\[
V(x_1, \ldots, x_N) = \sum_{i=1}^{N} \sum_{j=i+1}^{N} U(r_{ij}).
\]

The force induced by a potential corresponds to its gradient evaluated at the respective position. Hence, the total force acting on a particle can be obtained by computing its interaction with every other particle and adding up the resulting forces:

\[
F_i = -\nabla x_i V(x_1, \ldots, x_N) = \sum_{j=1}^{N} \left( -\nabla x_i U(r_{ij}) \right) = \sum_{j=1}^{N} F_{ij}(r_{ij}).
\]

Consequently, determining the forces of a system of \( N \) particles requires \( N \times N \) computations of \( F_{ij}(r_{ij}) \) during each time step, which yields an algorithmic complexity of \( \mathcal{O}(N^2) \). The resulting computation scheme is summarized in algorithm 3.

Algorithm 3 Simple force calculation

1: for \( i = 1, \ldots, N \) do
2:   \( F_i := 0 \)
3: for \( j = 1, \ldots, N \) do
4:   \( r_{ij} := x_j - x_i \)
5:   \( F_{ij} := -\nabla x_i U(r_{ij}) \)
6:   \( F_i := F_i + F_{ij} \)
If only pairwise interaction are considered, algorithm 3 can be further improved. According to Newton’s third law $F_{ij} + F_{ji} = 0$, the force particle $i$ causes to particle $j$ is equal to the force particle $j$ causes to particle $i$ with the opposite sign, such that the resulting external force is zero. Until now, the forces $F_i$ and $F_j$ have been determined by computing both $F_{ij}$ and $F_{ji}$ separately and including them into the respective sums. Although, if $F_{ij}$ is already known, $F_{ji}$ does not need to be computed via potential evaluation, but can be trivially determined by exploiting the relation $F_{ji} = -F_{ij}$. Consequently, only half of the previously required force evaluations are necessary, which yields the improved version of the simple force calculation summarized in algorithm 4.

**Algorithm 4 Improved simple force calculation**

1. for $i = 1, \ldots, N$ do 
2. $F_i := 0$ 
3. for $i = 1, \ldots, N$ do 
4. for $j = i + 1, \ldots, N$ do 
5. $r_{ij} := x_j - x_i$ 
6. $F_{ij} := -\nabla x_i U(r_{ij})$ 
7. $F_i := F_i + F_{ij}$ 
8. $F_j := F_j - F_{ij}$

The amount of force evaluations can hereby be reduced by a factor of two, though the algorithmic complexity of the algorithm is still $O(N^2)$ in the number of particles, which impedes the implementation of simulations with large particle numbers. To further reduce the computational complexity, the rest of this section focuses on potentials of limited range.

2.2.2 The cutoff radius

So far, for the calculation of the force on a particle, the interactions with all other particles have been considered. However, if the governing potential is of limited range and hence interactions only occur between neighboring particles, this approach involves a large number of unnecessary computations. Instead, it suffices to include only those particles whose potential contributes significantly to the total force. For potentials that drop rapidly after a certain distance a cutoff radius can be defined which limits the contribution of the resulting force to particles within a certain range.

An important example is the Lennard-Jones potential, which is extensively used to approximate the interactions between neutral atoms or molecules. Here, the potential between two particles at a distance$^5$ of $r_{ij}$ is defined as

$$U(r_{ij}) = 4 \cdot \varepsilon \left( \frac{\sigma}{r_{ij}} \right)^{12} - \left( \frac{\sigma}{r_{ij}} \right)^{6} = 4 \cdot \varepsilon \left( \frac{\sigma}{r_{ij}} \right)^{6} \left( \left( \frac{\sigma}{r_{ij}} \right)^{6} - 1 \right), \quad (20)$$

with two additional constants $\varepsilon$ and $\sigma$, that are needed to parametrize different physical behavior. Figure 4 shows the Lennard-Jones potential with $\varepsilon = 1$ and $\sigma = 1$. The potential function for $N$ particles is then defined as the sum over all pairwise potentials

$$V(x_1, \ldots, x_N) = \sum_{i=1}^{N} \sum_{j=i+1}^{N} U(r_{ij}) = 4 \cdot \varepsilon \sum_{i=1}^{N} \sum_{j=i+1}^{N} \left( \frac{\sigma}{r_{ij}} \right)^{6} \left( \left( \frac{\sigma}{r_{ij}} \right)^{6} - 1 \right). \quad (21)$$

Finally, the force $F_i$ on a particle $i$ is obtained through gradient formation after $x_i$

$$F_i = -\nabla x_i V(x_1, \ldots, x_N) = 24 \cdot \varepsilon \sum_{i=1}^{N} \sum_{j\neq i}^{N} 1 \left( \frac{\sigma}{r_{ij}} \right)^{6} \left( 1 - 2 \left( \frac{\sigma}{r_{ij}} \right)^{6} \right) r_{ij}. \quad (22)$$

---

$^4$ A function $U(r)$ is called rapidly dropping if its magnitude decreases faster than $1/r^d$, where $d$ is the dimension of the problem.

$^5$ $r_{ij} = \|r_{ij}\|_2$
As both the potential and the force drop rapidly, neglecting all corresponding terms contained in (20) and (22) introduces only a small error, which leads to the approximation

\[
U(r_{ij}) \approx \begin{cases} 
4 \cdot \varepsilon \left( \left( \frac{\sigma}{r_{ij}} \right)^{12} - \left( \frac{\sigma}{r_{ij}} \right)^6 \right) & \text{for } r_{ij} \leq r_{\text{cut}} \\
0 & \text{for } r_{ij} > r_{\text{cut}},
\end{cases}
\]  

where the potential is truncated at a radius \( r_{ij} > r_{\text{cut}} \). The artificially introduced cutoff radius \( r_{\text{cut}} \) is typically chosen as \( 2.5 \sigma \). By means of equation (23), an error in the force calculation is introduced, that slightly alters the total energy of the system. Moreover, both the potential as well as the resulting force are no longer continuous and hence the total energy is not preserved exactly. However, for sufficiently large cutoff radii, those effects are insignificant and can be ignored [22]. Under the assumption that the particles are equally distributed over the simulation area, \( r_{\text{cut}} \) can be chosen in a way that the number of non-disappearing terms is always restricted. Consequently, the effort for evaluating the potentials and computing the forces is only proportional to the number of particles \( N \), which leads to a reduced complexity of \( O(N) \). The challenge in the design of algorithms for the simulation of systems with short-range potentials, is to organize the particles in a manner that allows the fast detection of all neighboring particles within the potential’s range. The cell list method aims to accomplish this by means of a geometric decomposition of the simulation area into cells.

### 2.2.3 The cell list method

The idea of the cell list (or linked-cell) method is to subdivide the simulation area \( \Omega \) into uniform subdomains, so-called cells. If the side lengths of these cells is chosen to be greater or equal than the cutoff radius \( r_{\text{cut}} \), the interactions with respect to the truncated potentials are restricted to particles in the same and neighboring cells. Figure 5 shows an example where a three dimensional simulation area is subdivided into cells of size \( r_{\text{cut}} \times r_{\text{cut}} \times r_{\text{cut}} \). Assume \( \mathcal{N}(ic) \) describes the set of all particles in a cell \( ic \) and all of its neighboring cells. For the calculation of the force on particle \( i \) in cell \( ic \), one then obtains a sum of the form

\[
F_i \approx \sum_{\text{cell } jc} \sum_{j \in jc \setminus \{i\}} F_{ij},
\]  

Figure 4: Lennard-Jones potential with the parameters \( \varepsilon = 1 \) and \( \sigma = 1 \).
In three dimensions, every cell position can be described by three indices \((ic_1, ic_2, ic_3)\). Each interior cell possesses 26 neighbors with the indices \(jc_1 \in [ic_1 - 1, ic_1 + 1]\), \(jc_2 \in [ic_2 - 1, ic_2 + 1]\) and \(jc_3 \in [ic_3 - 1, ic_3 + 1]\). Accordingly, except in the case of periodic boundary conditions, the number of neighbors for boundary cells is reduced. If the number of cells in dimension \(d\) is defined as

\[ nc_d = \left\lfloor \frac{L_d}{r_{\text{cut}}} \right\rfloor, \]

then the resulting force calculation scheme can be summarized in algorithm 5.

**Algorithm 5** Cell list force calculation

1: for \( ic_1 = 1, \ldots, nc_1 \) do
2:   for \( ic_2 = 1, \ldots, nc_2 \) do
3:     for \( ic_3 = 1, \ldots, nc_3 \) do
4:       for each particle \( i \in \text{cell} (ic_1, ic_2, ic_3) \) do
5:         \( F_i := 0 \)
6:         for \( jc_1 = ic_1 - 1, ic_1, ic_1 + 1 \) do
7:           for \( jc_2 = ic_2 - 1, ic_2, ic_2 + 1 \) do
8:             for \( jc_3 = ic_3 - 1, ic_3, ic_3 + 1 \) do
9:               for each particle \( j \in \text{cell} (jc_1, jc_2, jc_3) \) do
10:               if \( i \neq j \) then
11:                 \( r_{ij} := x_j - x_i \)
12:                 if \( \|r_{ij}\| \leq r_{\text{cut}} \) then
13:                   \( F_i := F_i + F_{ij} \)

Despite the reduction in the computational cost for computing all pairwise interactions from \(O(N^2)\) to \(O(N)\), algorithm 5 has one drawback. If \( r_{\text{cut}} \times r_{\text{cut}} \times r_{\text{cut}} \) is chosen as the cell size, usually a significant proportion of the particles in neighboring cells is not within range of the cutoff radius of a particle and a lot of unnecessary distance calculations are performed. One method to reduce this number, that has been first suggested in [35], is to choose a cell size smaller than the cutoff radius. In [52][53][54] this approach has been further implemented and analyzed. The caveat is that the number of cells, that need to be considered for each particle, grows cubically with a reduction of the cell size. Another approach is the sorting of particles along one axis [55], such
that if a certain particle in the sorted list is already outside the cutoff radius, the same will be the case for the remaining ones and thus they do not need to be considered for the force calculation. This further reduces the number of unnecessary distance calculations, though introduces additional complexity for sorting the particles. As a proof-of-concept this thesis focuses on the conventional cell list method described in algorithm 5.

The remaining challenge lies in the design of a data structure that allows the fast access to particles contained in the same and neighboring cells. Additionally, because the particles move during the simulation, it is necessary to continuously redistribute them over the cells, such that they are always positioned within the cell that covers the respective part of the simulated domain. The singly linked list is a data structure that fulfills these requirements. Figure 6 shows a simple example of a list of five elements. Each list element consists of two components, the actual data and a pointer to the next element. Listing 3 shows the implementation of a singly linked list in the C programming language. The pointer of the last element usually possesses a special value that marks the end of the list. Because the structure of a linked list is solely determined by the order in which elements refer to each other, the list can be easily modified for the insertion or removal of elements by overwriting the pointer values. Inserting an element at a certain position within the list only requires two modifications: First, the pointer of the previous list node must be overwritten to point to the new element and second, the pointer of the new element must be set to the address of the next list element. To remove an element, the only operation that needs to be performed is setting the pointer of the previous node to the value of the pointer of the removed element. Listing 4 shows a C implementation of both operations. Finally, traversing the list corresponds to recursively dereferencing the pointers until the desired node has been reached, which is of linear complexity in the number of traversed list elements. To utilize this data structure for the cell list algorithm, for each cell a linked list is created for storing all particles that are currently positioned within its range. The data that needs to be stored in each list element then consists of the particle’s mass, its current position, velocity and force. Hence, obtaining all particles of a specific cell corresponds to traversing the respective list and moving particles from one cell to another can be realized by removing the respective element from the list of the old cell and subsequently inserting it into the list of the new one. For a more detailed description of linked lists the reader is referred to [56] and [22].

---

6 The Velocity Verlet method additionally requires storing the force of the previous time step.

**Figure 6**: A singly linked list. The sixth node represents the end of the list.

<table>
<thead>
<tr>
<th>Listing 3</th>
<th>Data type declaration of a singly linked list in C</th>
</tr>
</thead>
<tbody>
<tr>
<td>struct List {</td>
<td></td>
</tr>
<tr>
<td>Data data;</td>
<td></td>
</tr>
<tr>
<td>struct List *next;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Listing 4</th>
<th>List insertion and removal in C</th>
</tr>
</thead>
<tbody>
<tr>
<td>void insert(List **head, List *node) {</td>
<td></td>
</tr>
<tr>
<td>node-&gt;next = *head;</td>
<td></td>
</tr>
<tr>
<td>*head = node;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

| void remove(List **head) { |
|   *head = (*head)->next; |
| } |

21
2.3 Impala

Impala is a statically typed programming language with Rust-like syntax that features both imperative and functional programming and serves as front end for AnyDSL. In the following, its syntax and semantics are presented in an informal fashion.

2.3.1 Variables

Variables are immutable by default and are declared with `let` in case of local variables and with `static` in case of global variables. To make a variable mutable, the `mut` keyword must be provided. Impala offers a type inference system and types only need to be specified in case of ambiguities. In contrast to C, Impala’s type system is strict in a sense that type conversions are never performed implicitly.

Listing 5 Variables
```r
static N = 10; // a global variable

fn main() -> () {
    // local variables
    let i = 42; // i32
    let x = 42.0; // f64
    let n : u64 = 10u64; // explicit type declaration
    let mut s = "foo";
    s = "bar";
    let y : f64 = 0 as f64; // type conversions must be explicit
}
```

2.3.2 Functions

Functions can be defined with the `fn` keyword, either in global scope or inside another function. The syntax of function application is similar to that of C/C++. As a fully-featured functional programming language, Impala allows declaring higher-order\(^7\) and anonymous functions. Furthermore, as functions are first-class citizens, they can be assigned to variables and declared as data members within a record. Impala is an expression-based language and hence functions do not possess a return value, but instead the last evaluated expression is returned.

Listing 6 Functions
```r
fn add(x: f64, y: f64) -> f64 {
    x + y
}

// A nested function
fn foo() -> i32 {
    fn bar() -> i32 {return 42}
    let f = bar;
    f()
}

// A higher-order function
fn negate(f: fn(i32) -> i32) -> fn(i32) -> i32 {
    |i| {-f(i)} // anonymous function with syntax |argl,..., argn|{body}
    // returns a function that first applies the passed function
    // and then negates the result
}
```

2.3.3 References

Reference (or pointer) types are declared with the `&` keyword. To obtain a reference to a certain variable, the same keyword can be used. Just like all other variables, reference variables are

\(^7\) A higher-order function is a function which has at least one argument that is a function.
immutable by default, but can be made mutable with the `mut` keyword. Accordingly, mutable references can only be obtained from mutable objects. References are mainly useful to manage allocated memory and to interface code written in a different programming language, such as C.

### Listing 7 References
```rust
fn main() -> () {
    let x = 1;
    let p0 = &x;
    // Mutable references can only be obtained from mutable variables
    let mut b = false;
    let p1 = &mut b;

    // p0 has only read access to x
    let y = *p0;
    *p1 = true;
}
```

---

**2.3.4 Control structures**

Impala features typical imperative control structures such as `if/else` and `while`. Even though conventional imperative loops are available, Impala favors expressing iteration in a recursive way using higher-order functions. For convenience, the `for` syntax allows expressing a higher-order function call in imperative style.

### Listing 8 Control Structures
```rust
fn pow1(x: i32, n: i32) -> i32 {
    let mut i = 0;
    let mut res = 1;
    while (i < n) {
        res *= x;
        ++i;
    }
    res
}

fn range(lower: i32, upper: i32, body: fn(i32) -> ()) -> () {
    if lower < upper {
        body(lower);
        range(lower+1, upper, body, return)
    }
}

fn pow2(x: i32, n: i32) -> i32 {
    let mut res = 1;
    // the same as: range(0, n, |i| {res *= x});
    for i in range(0, n) {
        res *= x;
    }
    res
}
```

---

**2.3.5 Arrays**

Impala features both definite and indefinite arrays. Definite arrays are of type `[type * size]`, where `type` is the type of the individual elements and `size` the size of the array. Indefinite arrays are of type `[type]` and can not be declared as local variables because their size is not statically known. Array access possesses the same syntax as function application.
2.3.6 Records

Records are also available in Impala and can be created in form of a structure that contains a fixed set of data types. Because Impala is a functional programming language, records can have functions as data members. This enables the creation of functional interfaces to arbitrary data types.

Listing 10 Records

```rust
// A linked list
struct List {
    data: &u8,
    next: &List
}

// Interface to a three dimensional vector of f64
struct Vector3D {
    // Get the data
    fn getData() -> [f64 * 3],
    // Map: Apply a function to each element of the vector and return the result
    fn map(fn(f64) -> f64) -> Vector3D,
    // Reduce: Combine the elements of the vector by applying a function
    // the second argument acts as accumulator
    fn reduce(fn(f64, f64) -> f64, f64) -> f64
}
```

2.3.7 Type conversions and casts

In order to circumvent Impala’s static type system, type conversions or casts can be applied. Type conversions force the compiler to transform one type into another based on certain rules, for example a floating point value is converted to an integer by truncating the value. In contrast, a cast directly reinterprets the data stored in memory as a different type.

Listing 11 Type conversions and casts

```rust
fn float_to_int(x: f32) -> i32 { x as i32 }
fn float_as_int(x: f32) -> i32 { bitcast[i32](x) }
fn main() -> () {
    let x = 1.5f32;
    let i1 = float_to_int(x); // i1 == 1
    let i2 = float_as_int(x); // i2 == 1069547520
}```
2.3.8 Partial evaluation

Impala delegates the responsibility of performing partial evaluation to the programmer. Partial evaluation of a function is triggered with the `@` sign. To prevent Impala from partial evaluating a function the `$` sign can be employed. An example is shown in listing 12.

Listing 12 Partial evaluation

```plaintext
fn factorial(n: i32) -> i32 {
    if(n == 0) {
        1
    } else {
        n * factorial(n-1)
    }
}
fn main() -> () {
    let a = @ factorial(10); // computed at compile time
    let b = $ factorial(10); // computed at runtime
}
```

2.3.9 Interfacing C and C++ code

If the same compiler backend is used, Impala code can be easily interfaced with C or C++ code. For this purpose, the `extern "C"` keyword is used to declare externally defined functions within Impala. On the other hand, calling Impala functions externally requires their declaration in the respective file. A simple example is provided in listing 13. In C++ external functions must additionally be declared as `extern "C"` to avoid name mangling, as shown in listing 14.

Listing 13 Interfacing C/C++ code in Impala

```plaintext
extern "C" {
    fn println(&[u8]) -> ();
}

extern fn hello() -> () {
    println("Hello, world!");
}
```

Listing 14 Interfacing Impala code in C++

```plaintext
#include <stdio.h>

extern "C" void hello();

extern "C"
void println(const char* str) {
    puts(str);
}

int main(int argc, char** argv) {
    hello();
    return 0;
}
```

---

8 The example is from the AnyDSL-GitHub page: https://github.com/AnyDSL/anydsl/wiki/Tutorial
3 Implementation

The scope of this section is the implementation of molecular dynamics with short-range interactions based on the AnyDSL-framework. This is illustrated by presenting a functioning molecular dynamics code in Impala, that supports both the basic algorithm (algorithm 3), as well as the cell list method (algorithm 5). The time integration is based on the Velocity Verlet method (algorithm 2). As an application example, the Lennard-Jones potential (figure 4), which finds extensive use in molecular dynamics simulations, is implemented. Additionally, to evaluate the performance of the generated code, a baseline code written in the C programming language is provided, that offers the exact same functionality without any abstractions besides simple function calls. As a distributed memory parallelization is currently not available within AnyDSL, the implementation presented here is integrated into the pe physics engine, a massively parallel framework for the simulation of rigid bodies.

3.1 Molecular dynamics in Impala

There is a certain amount of functionality that is required in every type of molecular dynamics simulation. First, it is cumbersome to fix the floating point precision in an implementation. The caveat is that Impala does not include type polymorphism at the current state of work, though a type declaration can be provided that introduces a type real, which can be set according to the floating point precision. It is also desirable to define a type size_t, which can take on integer values large enough to index all potential array sizes and a type ptr_t for comparing pointer values. Additionally, a number of constants which set important parameters of the simulation, such as the dimensionality of the problem and the data size of a particle, are provided.

Listing 15 Type definitions and constants.

```ocaml
type real = f64;
type size_t = u64;
type ptr_t = u64;
fn epsilon() -> real {1e-12}
fn dimension() -> size_t {3 as size_t}
fn size_of_particle() -> size_t {
    (1 as size_t + (4 as size_t) * dimension()) * sizeof(real)() as size_t
}
```

Furthermore, it is useful to define functions for iterating over a certain range. For thread-parallel iteration, the function parallel from the Impala runtime can be utilized.\(^9\) In general, for reasons of efficiency, it is beneficial to map multidimensional array accesses to their one dimensional counterparts, as a one dimensional array can be efficiently represented as a single chunk of memory with consecutive addresses. This can be accomplished for arbitrary dimensions by defining a recursive function that is partially evaluated with respect to the dimension of the problem. The convention chosen here is to store multidimensional indices in a definite array of type size_t * dimension(), where the first index corresponds to the first and the last index to the last dimension. In three dimensional Cartesian coordinates, this corresponds to storing the x coordinate at position zero, the y coordinate at position one and the z coordinate at position two. Listing 16 shows the implementation of these functions.

The position, velocity and force of a certain particle is always a vector of length equal to the simulation’s dimension. Accordingly, within the time integration and force calculation, most operations are performed on these fixed-size vectors. Hence, it is useful to abstract the mathematical operations on these vectors by defining a purely functional interface, which offers the required functionality without introducing any overhead. As mentioned in the last section, such an interface can be defined by declaring a record with functions as data members. Listing 17 shows the resulting interface.

\(^9\) The parallel function currently only accepts range limits of type i32, hence an additional type conversion is necessary.
The following four functions are declared, which suffice to express all required vector operations in a functional style.

- **get**: Simply returns the raw data of the vector, in form of a definite array.
- **combine**: Combines two vectors element-wise through application of a binary function. The resulting vector is returned.
- **map**: Creates a new vector by applying a function to each element.
- **reduce**: Accumulates the elements of a vector through application of a binary function. The accumulated result is returned.

Listing 18 describes how, based on these functions, a number of common vector operations can be expressed.

The vector interface is implemented by providing a function that returns an instance of the record for a specific data set, passed in form of a definite array, as it can be seen in listing 19.
currently in Impala, the size of a definite array can not be specified with a constant parameter, the
implementation described here only covers the three dimensional case. Although, all relevant parts
are designed to be as generic as possible, such that a transition to 2D requires only minimal code
changes.

Listing 19 Vector implementation

```rust
fn get_vector(data: [real * 3]) -> Vector {
    Vector {
        get: || {data},
        combine: |f: fn(real, real) -> real, v: Vector| {
            let mut res : [real * 3];
            let v_data = v.get();
            for d in @iterate(0 as size_t, dimension()) {
                res(d) = f(data(d), v_data(d));
            }
            get_vector(res),
        },
        map: |f: fn(real) -> real| {
            let mut res : [real * 3];
            for d in @iterate(0 as size_t, dimension()) {res(d) = f(data(d));}
            get_vector(res),
        },
        reduce: |f: fn(real, real) -> real, empty: real| {
            let mut acc = empty;
            for d in @iterate(0 as size_t, dimension()) {acc = f(acc, data(d));}
            acc
        }
    }
}
```

Independent of the applied algorithm, particles are the basic data structure of a molecular
dynamics simulation. By utilizing the same functional abstraction, a particle interface can now be
defined, which allows setting and obtaining different the components of a particle (mass, position,
velocity and force). The goal is to implement this interface in a way that enables the access from
a pointer to raw data. Thereby, the user of the interface can safely access the respective fields,
without knowing any details about the underlying data representation. Furthermore, the resulting
operations are highly efficient as the compiler can replace every function call by a simple pointer
access. Every particle consists of a single real value, representing the mass, and four vectors for the
position, velocity, force and the force from the previous time step\(^\text{10}\). Hence, to represent a particle
in memory, in total \(1 + 4 \cdot \text{dimension}()\) floating point values are required. If a reference that points to
previously allocated memory is given, it is possible to statically compute the corresponding offsets
and generate accesses to the respective components by means of data reinterpretation. The details
of the offset calculation are described in listing 47 in the appendix.

Listing 20 Particle interface

```rust
struct Particle {
    getMass: fn() -> real,
    getCoordinates: fn() -> Vector,
    getVelocities: fn() -> Vector,
    getForces: fn() -> Vector,
    getForces_old: fn() -> Vector,
    setMass: fn(real) -> (),
    setCoordinates: fn(Vector) -> {},
    setVelocities: fn(Vector) -> {},
    setForces: fn(Vector) -> {},
    setForces_old: fn(Vector) -> ()
}
```

\(^\text{10}\) As required for the Velocity Verlet integration scheme.
3.1.1 The Velocity Verlet integration

If the current and previous forces are available, the Velocity Verlet integration method (algorithm 2) allows the computation of the velocity and position of each individual particle at the same point of time. Therefore, two functions \texttt{integrate\_x} and \texttt{integrate\_v}, which update a particle’s position and velocity according to equation 18 and 19, are required. Based on the previously defined vector and particle interfaces, the respective equations can be directly translated to Impala code, as shown in listing 21.

**Listing 21 Velocity Verlet integration**

```impala
fn integrate\_x(p: Particle, dt: real) -> () {
    let a = dt * 0.5 / p.getMass();
    let F = p.getForces();
    let tmp1 = scale\_vector(a, F);
    let V = p.getVelocities();
    let tmp2 = add\_vectors(V, tmp1);
    let tmp3 = scale\_vector(dt, tmp2);
    let X = p.getCoordinates();
    p.setCoordinates(add\_vectors(X, tmp3));
    p.setForces\_old(F);
}

fn integrate\_v(p: Particle, dt: real) -> () {
    let a = dt * 0.5 / p.getMass();
    let F = p.getForces();
    let F\_old = p.getForces\_old();
    let tmp1 = add\_vectors(F, F\_old);
    let tmp2 = scale\_vector(a, tmp1);
    let V = p.getVelocities();
    p.setVelocities(add\_vectors(V, tmp2));
}
```

3.1.2 The Lennard-Jones potential

Because the force calculation is the determining complexity factor in a molecular dynamics simulation, minimizing the number of arithmetic operations is of great importance. As an application, the Lennard-Jones potential, which is in widespread use, is considered here. Given a system of \( N \) particles, the total Lennard-Jones force on a particle can be obtained from equation (22). It is assumed that the cutoff radius of the potential is determined by the constant \( r_{\text{cut}} \). Hence, the force between each individual pair of particles \( i, j \) can be computed by

\[
F_{ij} = \begin{cases} 
24 \cdot \varepsilon \cdot \frac{1}{r_{ij}} \left( \frac{\sigma}{r_{ij}} \right)^6 \left( 1 - 2 \left( \frac{\sigma}{r_{ij}} \right)^6 \right) r_{ij} & \text{for } r_{ij} \leq r_{\text{cut}} \\
0 & \text{for } r_{ij} > r_{\text{cut}}.
\end{cases}
\]  

Terms that include powers of \( \varepsilon \) or \( \sigma \) and other constants can be moved out of the equation:

\[
F_{ij} = \begin{cases} 
(24 \cdot \varepsilon \cdot \sigma^6) \left( \frac{1}{r_{ij}} \right)^8 \left( 1 - (2\sigma^6) \left( \frac{1}{r_{ij}} \right)^6 \right) r_{ij} & \text{for } r_{ij} \leq r_{\text{cut}} \\
0 & \text{for } r_{ij} > r_{\text{cut}}.
\end{cases}
\]  

The euclidean distance \( r_{ij} \) between two particles in a \( D \) dimensional space is given by

\[
r_{ij} = \sqrt{\sum_{d=1}^{D} r_{ij}(d)},
\]

where \( r_{ij}(d) \) is the \( d \)th component of the vector \( r_{ij} \). The computation of the square root is an expensive operation on modern CPUs and should be avoided where possible. As equation 25 only
contains even powers of \( r_{ij} \), the computation of the square root can be avoided through substitution

\[
F_{ij} = \begin{cases} 
(24 \cdot \varepsilon \cdot \sigma^6) \left( \frac{1}{r_{ij}^2} \right)^4 \left( 1 - (2\sigma^6) \left( \frac{1}{r_{ij}^2} \right)^3 \right) r_{ij} & \text{for } r_{ij}^2 \leq r_{cut}^2 \\
0 & \text{for } r_{ij}^2 > r_{cut}^2 
\end{cases}
\]

Some final adjustments to minimize the numerical error lead to the desired optimized version of the original equation

\[
F_{ij} = \begin{cases} 
(24 \cdot \varepsilon \cdot \sigma^6) \frac{1}{(r_{ij}^2)^4} \left( 1 - (2\sigma^6) \frac{r_{ij}}{(r_{ij}^2)^{3/2}} \right) r_{ij} & \text{for } r_{ij}^2 \leq r_{cut}^2 \\
0 & \text{for } r_{ij}^2 > r_{cut}^2 
\end{cases}
\] (26)

According to equation 26, the factors \( 24 \cdot \varepsilon \cdot \sigma^6 \) and \( 2\sigma^6 \) must be only computed once during the simulation. Furthermore, the term \( 1/(r_{ij}^2)^4 \) can be computed in three steps: \( r_{ij}^2 \) is first squared twice, then the reciprocal is computed. In total, assuming \( r_{ij} \leq r_{cut} \), the computation of \( F_{ij} \) has been reduced to \( 6 + D \) multiplications, one subtraction and one division. It is now tempting to further reduce the number of operations, by moving all constant factors out of the sum in equation (22). However, this optimization can not be safely performed, because it has a negative impact on the numerical stability of the algorithm. Dependent on the cutoff radius, extreme value of the term \( 1/(r_{ij}^2)^4 \) must be assumed in certain cases. Their accumulation either leads to cancelation in case of very small values or to infinity in case of very high values. Retaining the constant factors within the sum has a dampening effect on the individual summands and is therefore crucial for the numerical stability of the algorithm. Listing 22 shows the implementation of the optimized computation scheme for the Lennard-Jones force. \( \varepsilon, \sigma \) and \( r_{cut} \) are stored in the Constants record. Additionally,

### Listing 22 Lennard-Jones force

```plaintext
struct Constants {
    r_cut: real,
    sigma: real,
    epsilon: real,
    r_cut_sqr: real, // r_cut * r_cut
    tmp1: real, // 24 * epsilon * sigma^6
    tmp2: real // 2 * sigma^6
}

fn force(p1: Particle, p2: Particle, write1: bool, write2: bool, constants: Constants) -> () {
    let sqr = |x : real| {x*x};
    let Dist = sub_vectors(p2.getCoordinates(), p1.getCoordinates());
    let Dist_sqr = square_vector(Dist);
    let r_sqr = vector_sum(Dist_sqr);
    if(r_sqr < constants.r_cut_sqr) {
        let r_8_inv = 1.0/sqr(sqr(r_sqr));
        let f = constants.tmp1 * r_8_inv * (1.0 - r_sqr * r_8_inv * constants.tmp2);
        let F = scale_vector(f, Dist);
        if(write1) {
            p1.setForces(add_vectors(p1.getForces(), F));
        }
        if(write2) {
            p2.setForces(sub_vectors(p2.getForces(), F));
        }
    }
}
```

three floating point variables tmp1, tmp2 and r_cut_sqr are included, which correspond to the constant terms in the force calculation. Again, the implementation is based on the previously defined Vector and Particle interfaces. Under certain conditions\(^\text{11}\), the force only needs to be updated

\(^{11}\) The cases where particles do not need to be updated are discussed later. Essentially, this occurs if the simulation area is partitioned into subdomains which are stored on different compute nodes.
for one of two interacting particles. Therefore, two boolean arguments, which indicate if the force of the respective particle is to be updated, are provided.

### 3.2 The basic algorithm

Since now both a time integration and a force evaluation function are available for individual particles, the implementation of molecular dynamics can be completed with an algorithm that computes all pairwise interactions. Similar to the last section, the basic algorithm, as described in listing 3, is considered first.

#### 3.2.1 Data structures

The basic algorithm implies multiple iterations with consecutive access to the same sequence of particles. For such an access pattern, a one dimensional array optimizes spatial locality and is therefore the preferred method of storage. In principle, there are two possibilities to represent an array of particles, either as a single array of records or as a record of arrays. Although a record of arrays has the advantage that it enables vectorization, this implementation chooses to use an array of records, because this allows to use the same interface for individual particle access for the basic and cell list algorithm. Therewith, both implementations are comparable and apart from that, the practicability of algorithm 3 is limited to small particle numbers, even in case of vectorization.

A system of particles can be represented as a record containing functional interfaces to all data elements relevant for its simulation. The resulting interface definition is shown in listing 23 and its implementation in 24. The access to individual particles is abstracted by providing the `getParticle` function, which returns an interface to individual particles while hiding the actual low level implementation. The ghost layer member function is only needed to support the same interface for both algorithms and can be ignored here.

#### Listing 23 Basic algorithm: Particle system interface definition

```c
struct ParticleSystem {
    np: fn() -> size_t, // number of particles
    l: fn() -> Vector, // size of the simulated domain
    start: fn() -> size_t, // beginning of the particle array
    end: fn() -> size_t, // end of the particle array
    ghost_layer: fn() -> size_t, // not required in the basic algorithm
    getParticle: |i| { get_particle(&bitcast[&[u8]](data)(i*size_of_particle())) },
    constants: |i| {constants},
    data: |i| {data}
}
```

#### Listing 24 Basic algorithm: Particle system implementation

```c
fn get_particle_system(np: size_t, l: Vector, data: &u8, constants: Constants) -> ParticleSystem {
    ParticleSystem {
        np: || {np},
        l: || {l},
        start: || {0 as size_t},
        end: || {np},
        ghost_layer: || {0 as size_t},
        getParticle: |i| {
            get_particle(&bitcast[&[u8]](data)(i*size_of_particle()))
        },
        constants: || {constants},
        data: || {data}
    }
}
```
3.2.2 Algorithms

With the availability of the required data structures, the time integration and force calculation can finally be implemented. The function `update` (listing 25) applies a function to every particle of the system within a certain range. To update position and velocity, the function `update_x` and `update_v`, respectively, has to be passed as an argument. The update of the whole particle system is then performed through application of the `iterate` function. Because the number of particles is not statically known, the partial evaluator must be prevented from touching the `iterate` function. This is achieved with the `$` sign, which forces the partial evaluator to skip subsequent code until the next `@` sign is reached. In contrast, the body of the iteration function can be specialized to the respective integrator, by turning partial evaluation on again. Similarly, based on the previously defined function for computing the force between two particles, an implementation of the improved simple force evaluation method can be provided. This is realized in the function `compute_force`, shown in listing 26, which utilizes a generic function `force` to compute the force between each pair of particles. Again, partial evaluation of the whole iteration is averted while the body of the `iterate` function is specialized to the respective potential.

3.3 The cell list algorithm

The cell list algorithm is based on a partitioning of the simulation area into quadratic cells with a side length equal to the cutoff radius of the potential. Each cell contains all particles which are positioned in the respective subdomain. The force of each particle can then be computed by considering all particles in the same and neighboring cells.

3.3.1 Data structures

The core of the cell list algorithm is the implementation of data structures that allow an efficient computation of short-range interactions. As it has been discussed in the last section, the singly linked list is a data structure that is suitable to deal with the dynamic nature of a molecular dynamics simulation, where particles continuously move through the simulated domain.
Linked lists: Listing 27 shows the implementation of a singly linked list in Impala. Each node consists of a pointer data, to the particle data and a pointer next, to the next list element. To simplify the retrieval of particles from list nodes, the function get_particle_from_node is provided. Additionally, for the insertion and removal of elements, the functions insert and remove, as shown in listing 28 and 29, are provided. To insert a new list element, a mutable reference head, to the address of the next element in the current list, is required. The insertion is then performed by first setting the next pointer of the inserted node to the old value of head and then overwriting the value of the address, to which head points, by the address of the new list element.

Listing 27 Linked list

```rust
struct ParticleList {
    data: *u8,
    next: *ParticleList
}

fn get_particle_from_node(pl: *ParticleList) -> Particle {
    get_particle((*pl).data)
}
```

Listing 28 Insertion into a linked list

```rust
fn insert(head: &mut *ParticleList, pl: &mut ParticleList) -> () {
    (*pl).next = *head;
    *head = pl;
}
```

Removal from a linked list requires a mutable reference (head) to the address of the element that is to be removed. The old value of head is set to the value of its next pointer and a pointer to the removed element is returned, which is needed for memory deallocation. In case the list is empty, a pointer with value 0 is returned.

Listing 29 Removal from a linked list

```rust
fn remove(head: &mut *ParticleList) -> &mut ParticleList {
    if (*head != 0 as *ParticleList) {
        let retval : &mut ParticleList = *head;
        *head = (**head).next;
        retval
    } else {
        0 as &mut ParticleList
    }
}
```

As in the case of the simple force calculation scheme, the next step is to define a data structure that contains the complete simulation state. Each cell can be represented as a mutable reference to a pointer to the first element of the respective linked list. Therefore, the cells can be stored efficiently in a one-dimensional array, by utilizing the function index (listing 16) to map their three dimensional index into one dimension.

Scalability. In general, the implementation presented here aims to support two levels of parallelism: Shared and distributed memory. In the former setting, concurrent processes share the same address space, which usually refers to a parallel execution of threads, that access the same main memory on a single compute node. The partitioned global address space (PGAS) model transfers a shared address space to a multi-node setting, but is not considered here. In a distributed memory setting, the system consists of processes with distinct address spaces, that are only able to exchange data by means of explicit communication. Typically, this model refers to a multi-node system, like a cluster or supercomputer, where a number of compute nodes are connected via a high throughput network. The goal of this implementation is to supports both settings on a different level. The shared memory parallelization is directly implemented in Impala using the multi-threading support.
of the AnyDSL runtime library. A distributed memory parallelization is not provided by the Impala implementation itself, but its design allows for a smooth integration into distributed computing frameworks. As a proof-of-concept, this implementation is integrated into the pe physics engine, which is described later. In a distributed memory setting, the data must be partitioned in a way that allows for an even distribution of the workload among all nodes. The simplest method of distribution is to store the complete data on every single node, such that every particle is accessible on every node. However, this method is only applicable in cases where the whole data fits on a single node and hence does not scale with the number of particles. A better approach is the so-called data partitioning, where the data in form of cells and particles is distributed among the nodes. In this work, for the sake of simplicity, only a static data partitioning is assumed, which means that the cells that are assigned to each node are fixed at the beginning of the simulation. The caveat for a multi-node parallelization is the placement of neighboring cells on different compute nodes. As a result, the computation of all interactions requires the communication of a number of particles. Furthermore, particles that leave one node’s subdomain and enter a different one’s, first must be sent to the destination node and then deleted on the source node. In case of a static data partitioning and an even distribution of particles, a geometric domain decomposition into cubic subdomains minimizes the interdomain surface area and therefore the overall communication [57]. Thereby, the domain is decomposed into cubic subdomains, whereas every subdomain is assigned to a different compute node. Hence, to compute all interactions, only particles in cells located at subdomain boundaries have to be communicated. Assuming the number of particles per compute node is of order $O(N/P)$, where $N$ is the total number of particles and $P$ the number of compute nodes, then the number of communicated particles is of order $O(\sqrt{N/P})$ in 2D and of order $O((N/P)^{2/3})$ in 3D. To store particles that are positioned within a neighboring cell on a different compute node, each subdomain is extended with an additional layer of ghost cells in each dimension. Hence, after all neighboring particles are communicated, they are placed in the respective ghost cell, either at the beginning or at end of the subdomain. If the maximum distance a particle is able to move during one time step is restricted to one cell, this layer is of size one. Figure 7 shows an example of a two dimensional domain, that is extended by an additional layer of ghost cells.

![Figure 7: Visualization of a 2D domain (white), extended with a ghost layer (gray).](image)

**High level interface:** Based on the described domain decomposition, a system of particles can be implemented in form of a record *ParticleSystem*, as it is shown in listing 30. Again, this is realized in form of a purely functional interface to a pointer to raw data, which must be passed to the `get_particle_system` function (listing 31) to obtain a *ParticleSystem* instance. The function `head` provides access to the individual cells by returning a mutual reference to the beginning of the respective linked list. To obtain the index of the first and last cell of the local domain, excluding ghost cells, the `start` and `end` functions can be employed. Additionally, indexed access to all particles is possible by utilizing the function `addresses`. It returns a record that provides access to the data of individual particles based on an index of type `size_t`. To obtain an interface to the respective particle from a pointer to data, the `get_particle` function can be employed. The respective data structure `ArrayOfAddresses` is defined and implemented in listing 32.
Listing 30  Cell list algorithm: Particle system interface definition

```rust
struct ParticleSystem {
    np: fn() -> size_t, // number of particles
    nc: fn() -> [size_t * 3], // number of cells in each dimension, including ghost cells
    ghost_layer: fn() -> size_t, // size of the ghost layer
    start: fn() -> [size_t * 3], // index of the first non-ghost cell
    end: fn() -> [size_t * 3], // end of the non-ghost cells
    l: fn() -> Vector, // domain size
    offset: fn() -> real, // domain offset of the ghost cells
    head: fn(size_t) -> &mut &ParticleList, // head of the linked list of cell i
    constants: fn() -> Constants, // potential constants
    r_cut_inv: fn() -> real, // inverse cutoff radius
    addresses: fn() -> ArrayOfAddresses, // allows consecutive access to particles
    data_cells: fn() -> &u8, // pointer to the cell data
    data_addresses: fn() -> &u8 // pointer to the address array data
}
```

Listing 31  Cell list algorithm: Particle system implementation

```rust
fn get_particle_system(np: size_t, nc: [size_t * 3], ghost_layer: size_t, l: Vector,
    data_cells: &u8, constants: Constants, data_addresses: &u8) -> ParticleSystem {
    ParticleSystem {
        np: || {np},
        nc: || {nc},
        ghost_layer: || {ghost_layer},
        start: || {
            let mut start : [size_t * 3];
            for d in @iterate(0 as size_t, dimension()) {start(d) = ghost_layer;}
            start
        },
        end: || {
            let mut end : [size_t * 3];
            for d in @iterate(0 as size_t, dimension()) {end(d) = nc(d) - ghost_layer;}
            end
        },
        l: || {l},
        offset: || {ghost_layer as real * constants.r_cut},
        head: || {s
            mut cast[&mut [&ParticleList](data_cells)(i)|,
        constants: || {constants},
        r_cut_inv: || {1.0/constats.r_cut},
        addresses: || {get_array_of_addresses(np, data_addresses)},
        data_cells: || {data_cells},
        data_addresses: || {data_addresses}
    }
}
```

3.3.2 Iterators

One of the main objectives of AnyDSL is to provide functional interfaces for multiple layers of abstractions. A fundamental operation, that needs to be performed within the cell list algorithm, is an iteration over the complete domain. To achieve the parallelization on a shared memory level, this process needs to be performed concurrently. This essentially consists of two levels: The iteration over all cells that are within the domain and the iteration over all particles in each individual cell. Parallelizing the latter is not an option, because in most cases the workload per thread would be too small. Therefore, the approach for the shared memory parallelization is similar to the distributed memory case. The process-local domain is further partitioned in a geometric way, such that each thread processes a fraction of the total number of cells. In the following discussion, a bottom up approach is taken and the linked list traversal is considered first.

To access all particles positioned within a cell, a function that abstracts the traversal of a linked list while applying an operation on every list element is needed. As linked lists are recursive data structures, this can be implemented as a recursive higher-order function, whose first argument is
Listing 32 Array of pointers

```rust
struct ArrayOfAddresses {
    size: fn() -> size_t,
    get: fn(size_t) -> &u8,
    set: fn(size_t, &u8) -> ()
}
```

```rust
fn get_array_of_addresses(size: size_t, data: &u8) -> ArrayOfAddresses {
    ArrayOfAddresses {
        size: || {size},
        get: |i| {bitcast[&[&u8]](data)(i)},
        set: |i, ptr| {
            bitcast[&mut[&u8]](data)(i) = ptr;
        }
    }
}
```

the linked list that is to be traversed and whose second argument is a function that manipulates a
single list element (listing 33). From a functional perspective, this function can be thought of as a
map operation. Since the iteration over all particles within a single cell has been accomplished, the
next step is to define a function that iterates over all cells. In three dimension, this corresponds
to three loops, whose control variables can be kept in a definite array of type [size_t * 3]. To
optimize the spatial locality of the data accesses, the last dimension, which is defined as the last
element of the index array, has to be traversed first. The next step is to choose a suitable domain
partitioning scheme for the shared memory parallelization. In contrast to the distributed memory
case, in principle all threads have access to the complete set of particles. Because the particles are
stored in a linked list, their traversal results in an irregular memory access pattern. To optimize the
data locality in terms of cell access, the one-dimensional array of cells is divided into consecutive
chunks of equal size, such that each thread processes an equal number of cells. This is achieved by
parallelizing the outermost loop, which corresponds to an iteration over the domain in z-direction.
An additional advantage of this scheme is that threads are assured to be created only once for the
whole iteration, which minimizes the overhead of parallelization. Listing 34 shows the implemen-
tation of the described approach. The parallelization of the outermost loop is performed using the
previously defined parallel_iterate function.

Listing 33 Traversal of a linked list

```rust
fn iterate_over_list(ptr: &ParticleList, body: fn(&ParticleList) -> ()) -> () {
    if((ptr) != (0 as &ParticleList)) {
        body(ptr);
        iterate_over_list((*ptr).next, body, return)
    }
}
```

Listing 34 Iteration over cells

```rust
fn parallel_iterate_over_particle_cells(start: [size_t * 3], end: [size_t * 3],
P: ParticleSystem, numthreads: int, f: fn([size_t * 3], &ParticleList) -> ()) -> () {
    for k in parallel_iterate(start(2), end(2), numthreads) {
        for j in iterate(start(1), end(1)) {
            for i in iterate(start(0), end(0)) {
                ic = [i, j, k];
                f(ic, *P.head(index(ic, P.nc())));
            }
        }
    }
}
```
Finally, since both levels of iteration have been implemented, they can be combined in a function that maps an operation onto a complete system of particles. Two cases need to be considered here. In the first case, particles in the ghost layer are completely ignored, which is for instance the case in the integration of position and velocity. The implementation of this case is straightforward and shown in listing 35. In principle, ghost particles never need to be updated, because the original particles are located on a different process. Although, when computing the forces, interactions with ghost particles must be considered and therefore their data must be accessed in a read-only fashion. To distinguish between local and ghost particles, two write flags have been included into the computation of the Lennard-Jones force in listing 22. Each cell is either a ghost cell or within the process domain. Hence, dependent on the cell index, this flag can be fixed for all particles located in the same cell. Listing 36 shows the implementation of the ghost-layer-aware iteration function. To distinguish between local domain and ghost layer, four index arrays must be passed to the function. start and end indicate the complete range of iteration while lower_boundary marks the beginning of the local domain and upper_boundary its end. Hence, particles of a cell that is positioned within the ghost layer can be detected by comparing the cell index with the beginning and end of the local domain. By setting the write flags accordingly, it is then assured that only local particles are updated.

3.3.3 Algorithms

With the availability of data structures and iterators, the algorithmic parts of the cell list algorithm can finally be implemented. The implementation of an update function for the integration of position and velocity is similar to that provided within the basic algorithm in listing 25. Instead of a one dimensional particle range, a three dimensional cell range is used and passed to the function parallel_iterate_over_particle_system, which performs the update on all respective particles in parallel. Again, because the range of cells depends on the domain size, which is not statically known, the partial evaluator is prevented from touching the iteration function.

---

**Listing 35** Iteration over a particle system

```rust
fn parallel_iterate_over_particle_system(start: [size_t * 3], end: [size_t * 3],
P: ParticleSystem, numthreads: int, f: fn([size_t * 3], &ParticleList) -> ()) -> () {
    for ic, head in parallel_iterate_over_particle_cells(start, end, P, numthreads) {
        for pl in iterate_over_list(head) {
            f(ic, pl);
        }
    }
}
```

**Listing 36** Iteration over a particle system with ghost layers

```rust
fn iterate_over_particle_system_with_ghost_layers(start: [size_t * 3], end: [size_t * 3],
lower_boundary: [size_t * 3], upper_boundary :[size_t * 3], P: ParticleSystem,
f: fn([size_t * 3], &ParticleList, bool) -> ()) -> () {
    for ic, head in iterate_over_particle_cells(start, end, P) {
        let mut write = true;
        for d in @iterate(0 as size_t, dimension()) {
            if (ic(d) < lower_boundary(d) || ic(d) >= upper_boundary(d)) {
                write = false;
            }
        }
        for pl in iterate_over_list(head) {
            f(ic, pl, write);
        }
    }
}
```
Listing 37 Cell list algorithm: Integration

```plaintext
fn update(P: ParticleSystem, start: [size_t * 3], end: [size_t * 3], dt: real,
    numthreads: int, f: fn(Particle, real) -> () -> () {    
    for ic, pl in $parallel_iterate_over_particle_system(start, end, P, numthreads) @{
        let p = get_particle_from_node(pl);
        f(p, dt);
    }
}
```

Figure 8: Two cases in which a particle is located in an inner cell (blue). The neighbors that need to be traversed for the force calculation are colored red.

The next and most important part of the cell list algorithm is the implementation of the force calculation. In principle, the implementation, that is presented in the following, corresponds to algorithm 5. After resetting the force of each particle to zero, an iteration over all particles is performed. For each particle the interactions with particles in the same and neighboring cells are computed and the force is updated accordingly. In section algorithm 3.3, it has been assumed that every interaction must be computed twice, once for each interacting particle, because the cell list algorithm includes no explicit ordering of particles. Consequently, the algorithm performs twice as many force evaluation as necessary. To reduce the number of evaluations in a similar way as in the improved simple force evaluation, the implicit ordering of the particles must be utilized. Particles correspond to actual data, allocated in the processors main memory. The (virtual) address of this data is stored in a pointer variable. Because the data resides in a linear address space, the pointer values, which address the data of every single particle, introduce an implicit ordering. Hence, the number of computed interactions can be reduced by a factor of two, by comparing the memory addresses of every pair of interacting particles and then only performing the force evaluation once while updating both particles. The resulting implementation is shown in listing 38.

To ensure scalability in a distributed memory setting, additional ghost layers, which contain cells located on a different process, have been introduced. When iterating over the complete system of particles, these ghost layers must be handled specially. Ghost cells are either located at the beginning or end of the domain. In both cases, when evaluating the interactions of a ghost particle with other particles, only the interactions with particles in non-ghost cells need to be considered. In figure 8 and 9, a two dimensional case of a grid of 25 cells is visualized. Figure 8 shows the regular case, where a cell is located in the local domain. Here, additional to the particles located in the same cell, those in all nine neighboring cells need to be considered. In contrast, if a particle is located in one of the ghost cells, as it is shown in figure 9, only particles in neighboring inner cells need to be considered and even those in the same cell can be ignored. This translates to the boundary handling implemented in listing 38. Here, `ic` is the index of the current cell and the bounds of the neighbor traversal correspond to the variables `jc_start` and `jc_end`. 

38
Listing 38 Cell list algorithm: Force calculation

```rust
fn compute_force(P: ParticleSystem,
    force: fn(Particle, Particle, bool, bool, Constants) -> (), numthreads: int) -> () {
    // set the forces of all local particles to zero
    for ic, pl1 in $ parallel_iterate_over_particle_system(P.start(), P.end(),
        P, numthreads) @{
        let pl = get_particle_from_node(pl1);
        pl.netForces(get_null_vector());
    }
    let z = 0 as size_t;
    let mut ic_start : [size_t * 3];
    for d in @iterate(0 as size_t, dimension()) {
        ic_start(d) = z;
    }
    let ic_end = P.nc();
    // iterate over the complete domain, including ghost layers
    for ic, head in $ parallel_iterate_over_particle_cells(ic_start, ic_end, P,
        numthreads) @{
        let mut jc_start : [size_t * 3];
        let mut jc_end : [size_t * 3];
        let mut write_i = true;
        for d in @iterate(0 as size_t, dimension()) {
            if(ic(d) >= P.start()(d)) {
                jc_start(d) = ic(d) - 1 as size_t;
            } else {
                jc_start(d) = ic(d) + 1 as size_t;
                write_i = false;
            }
            if(ic(d) < P.end()(d)) {
                jc_end(d) = ic(d) + 2 as size_t;
            } else {
                jc_end(d) = ic(d);
                write_i = false;
            }
        }
        for pl1 in $ iterate_over_list(head) @{
            // iterate over all neighboring cells, including ghost layers
            for jc, pl2, write_j, in $ iterate_over_particle_system_with_ghost_layers(
                jc_start, jc_end, P.start(), P.end(), P) @{
                if((pl1 as ptr_t < pl2 as ptr_t)) {
                    // compute each interaction only once
                    let pl = get_particle_from_node(pl1);
                    let p2 = get_particle_from_node(pl2);
                    // only update non-ghost particles
                    force(pl, p2, write_i, write_j, P.constants());
                }
            }
        }
    }
}
```

In case ic is located in the interior of the domain, jc_start is set to ic − 1 and jc_end to ic + 2. However, if ic is a ghost cell, dependent on whether it is located at the beginning or end
of the domain, two cases can be distinguished for each dimension. In case the cell is located at
the beginning of the domain, \( j_{c\text{ _start}} \) is set to \( i_c + 1 \) and \( j_{c\text{ _end}} \) to \( i_c + 2 \), such that only the
rightmost neighbors in the respective dimension are subsequently considered. If the cell is located
at the end of the domain, \( j_{c\text{ _start}} \) is set to \( i_c - 1 \) and \( j_{c\text{ _end}} \) to \( i_c \), such that only the leftmost
neighbors are subsequently considered. Additionally, to prevent ghost particle updates, the write
flag is set to false. Finally, to implement the neighbor traversal, the previously defined function
\texttt{iterate\_over\_particle\_system\_with\_ghost\_layers} is employed.

As a final step to complete the implementation of the cell list algorithm, a function that maps
a particle’s position to the correct cell index is required for the redistribution of moving particles.
Listing 39 shows the implementation of this function. In each dimension, the previously computed
domain offset is added to the position. The cell index is then obtained by dividing the shifted
position through the cutoff radius, followed by rounding and a type conversion to \texttt{size\_t}. Thereby,

### Listing 39 Cell list algorithm: Cell position computation

```rust
fn compute_cell_index(p: Particle, P: ParticleSystem) -> [size_t * 3] {
    let mut kc : [size_t * 3];
    let coordinates = p.getCoordinates().get();
    for d in @iterate((0 as size_t), dimension()) {
        kc(d) = math.floor((coordinates(d) + P.offset()) * P.r_cut_inv()) as size_t;
    }
    kc
}
```

the redistribution of moving particles can be accomplished with the implementation of a function
\texttt{move\_particles}, which is shown in listing 40. The function performs a complete iteration over the
system of particles. Because the internal data structures must be altered, the previously defined
iterators can not be employed. Hence, to iterate over all cells and then traverse the corresponding
linked list, four loops are required. For each particle, the position is then checked for conformity
with its current cell location. If a particle has moved out of the cell range, it is removed from
the source cell and inserted into the respective destination cell. Moreover, the boundary handling,
which repositions or removes particles that leave the global domain, is applied here.
Listing 40  Cell list algorithm: Particle distribution

```rust
fn move_particles(P: ParticleSystem) -> () { 
    let start = P.start();
    let end = P.end();
    for k in iterate(start(2), end(2)) {
        for j in iterate(start(1), end(1)) {
            for i in iterate(start(0), end(0)) {
                let ic = [i, j, k];
                let mut q : &mut & ParticleList = P.head(index(ic, P.nc()));
                let mut i : &ParticleList = *q;
                while (i != 0 as &ParticleList) {
                    let p = get_particle_from_node(i);
                    boundary(p, P);
                    let kc = compute_cell_index(p, P);
                    let mut moving_node : &mut ParticleList;
                    if (ic(0) != kc(0) || ic(1) != kc(1) || ic(2) != kc(2)) {
                        moving_node = remove(q);
                        insert(P.head(index(kc, P.nc())), moving_node);
                    } else {
                        q = &((&i).next) as &mut ParticleList;
                    }
                    i = (*q);
                }
            }
        }
    }
}
```

3.3.4 Simulation loop

Finally, based on the functionality described in this section, the Velocity Verlet time integration scheme (algorithm 2) can be directly implemented as the main loop for the simulation of a system of particles. An implementation which is both compatible with the interfaces of the basic

```rust
Listing 41  Simulation loop

fn time_integration(t_start: real, t_end: real, dt: real, 
P: ParticleSystem, numthreads: int) -> () { 
    let mut t : real = t_start;
    compute_force(P, force, numthreads);
    while(t < t_end) {
        t += dt;
        update(P, P.start(), P.end(), dt, numthreads, integrate_x);
        move_particles(P);
        compute_force(P, force, numthreads);
        update(P, P.start(), P.end(), dt, numthreads, integrate_v);
    }
}
```

the initialization are not relevant for the performance of the algorithm, they have been omitted in this section. For the sake of completion, it needs to be mentioned that all memory is allocated using wrappers to the C function malloc, though the use of other methods of allocation is alternatively possible as well. Additionally, for visualization purposes, a function that produces VTK output has been implemented. The performance evaluation within this thesis is based on a comparison with an equivalent low-level C code. The shared memory parallelization is based on OpenMP and employs

---

12 The only addition is that an empty function move_particles must be provided, whose calls are removed by the compiler anyways.
the same static scheduling as the Impala code. Because this code does not introduce new concepts, it is not further discussed in this section, though to ensure the reproducibility of this research, the performance critical parts are included in the appendix. Therefore, the goal to implement a shared memory parallel Impala code for the simulation of molecular dynamics with short-range interactions has been accomplished. Additionally, the foundations for an extension of the cell list algorithm to a distributed memory setting have been laid. In the next part of this thesis, such an extension is accomplished with the integration of the presented Impala code into the pe physics engine.

### 3.4 Integration into the pe physics engine

pe is a framework for the parallel simulation of rigid bodies [2][3]. Rigid body dynamics is a subfield of classical dynamics that deals with the interaction of bodies which can not be deformed by external forces. Similar to molecular dynamics, it can therefore be considered as a special case of an N-body method. The main difference is, that in rigid body dynamics the interactions between particles are not described by an intrinsic force field, but are treated as collisions between non-deformable bodies in a physical accurate way. Because these bodies can possess complex geometries, the treatment of collisions involves methods that are considerably more computation intensive than the treatment of short-range interactions in molecular dynamics. pe is based on a parallel version of the fast friction dynamics algorithm [58]. It is implemented as an extensible framework written in the C++ programming language. To achieve scalability on recent supercomputers the Message-Passing Interface (MPI) is utilized. Recently, pe has been integrated into the WaLBerla (Widely applicable Lattice-Boltzmann from Erlangen [59]) framework to enable the simulation of particulate flows [60]. WaLBerla is a highly parallel software framework, which aims to simulate a wide range of physical phenomenon and whose scalability on a variety of supercomputing architectures has been demonstrated [61][62].

To integrate the presented molecular dynamics code into pe, the internal collision handling must be replaced by the integration and force calculation methods presented here. This primarily boils down to the integration of the previously defined cell list data structures. The main step has been already performed by the domain decomposition\(^\text{13}\) presented in section 3.3.1. Figure 10 shows a schematic example of the two dimensional case of the domain decomposition. Particles in the boundary cells need to be exchanged and stored in the respective ghost cells. pe already features the automatic distribution of particles over a number of cubic subdomains, according to a static domain decomposition. Therefore, a pe-coupled molecular dynamics simulation proceeds as follows:

1. pe performs the initial domain decomposition and particle distribution.
2. According to the size of the subdomains, cells are created for each process.
3. Then, the following procedure is repeated on each process for a number of time steps.
   (a) To match the number of particles on the process, including ghost particles, new particles are created or existing ones deleted.
   (b) The current particle data is received from pe and the internal data structures are updated accordingly.
   (c) The integration of the position is performed.
   (d) The particles are redistributed to ensure that they are positioned in the correct cell.
   (e) First the force calculation and then the velocity integration is performed.
   (f) The new positions and velocities are passed to pe, which then performs a synchronization of the processes and a redistribution of the particles.
4. The simulation finishes and all data structures are deallocated.

Now, the remaining step is the implementation of C compatible interfaces to the Impala code. This work aims to provide a fully functioning molecular dynamics code, that is implemented completely in Impala. The caveat of this approach is, that the data structures offered by pe can not be used within

\(^\text{13}\) As a reminder it is mentioned here that this work is restricted to a static domain decomposition, hence a dynamic load balancing is out of scope.
Figure 10: Domain decomposition of a 2D domain into four quadratic subdomains. The gray cells must be communicated.

the time integration and force calculation. Hence, to perform a synchronization, at the beginning of each time step, all necessary data must be transferred from pe to the Impala data structures and at the end of each time step, the same data transfer must be performed the other way around. The overhead has been already reduced by only creating or deleting particles when they move from one subdomain to another, though the data of all particles must still be synchronized. Despite these drawbacks, this approach is sufficient for a first proof-of-concept implementation. A further reduction of the overhead could be achieved, by either directly accessing the pe data structures within Impala, or by associating pe particles with Impala particles. Both approaches require an adaption of the data structures described in section 3.3.1 and are therefore not considered here.

The main challenge that lies within the implementation of interfaces, is the requirement that all function arguments must be compatible with Impala’s type system. It therefore restricts the set of possible arguments to types that have an equivalent representation in C++ and Impala. For instance, the functional interfaces, that have been presented in this section, do not fall into this category. As a consequence, a ParticleSystem can not be maintained over multiple external function calls and the data required for its creation must be stored in between. For this purpose, an additional record ParticleSystemData is provided, as shown in listing 42. The global state of the simulation can then be stored in a mutable static variable $P_{\text{data}}$.

Furthermore, it is necessary to define a set of functions, that exchange data between the pe and Impala data structures, which is best performed by means of simultaneous iteration over all particles. So far, it was only necessary to access particles with respect to their placement in a certain cell. The cell list algorithm only provides an indexed access to the cells, but not to individual particles.

As it has been described in listing 30, an additional function addresses, that returns a record ArrayOfAddresses, has been included into a ParticleSystem. This function can be utilized to get an integer-indexed access to the data of individual particles. Thereby, external functions that grant access to the mass, position, velocity and force can be implemented. Listing 43 shows the external function for setting the position of a particle with index $i$ and listing 44 the function for obtaining its
Listing 42 Record for storing the internal data of a particle system

```c
struct ParticleSystemData {
    np: size_t,
    nc: [size_t + 3],
    ghost_layer: size_t,
    l: [real + 3],
    constants: Constants,
    data_cells: &u8,
    data_addresses: &u8,
}
```

```c
def get_particle_system_from_data(P_data: ParticleSystemData) -> ParticleSystem {
    get_particle_system(P_data.np, P_data.nc, P_data.ghost_layer, get_vector(P_data.l),
    P_data.data_cells, P_data.constants, P_data.data_addresses)
}
```

```c
def get_data_from_particle_system(P: ParticleSystem) -> ParticleSystemData {
    ParticleSystemData {
        np: P.np(),
        nc: P.nc(),
        ghost_layer: P.ghost_layer(),
        l: P.l().get(),
        constants: P.constants(),
        data_cells: P.data_cells(),
        data_addresses: P.data_addresses()
    }
}
```

```c
static mut P_data_: ParticleSystemData;
```

Listing 43 External interface for setting the coordinates of a particle with index i

```c
extern fn set_coordinates(X: &[real], i: size_t) -> () {
    let P = get_particle_system_from_data(P_data_); 
    let p = get_particle(P.addresses().get(i));
    p.setCoordinates(get_vector([X(0), X(1), X(2)]));
}
```

Listing 44 External interface for obtaining the coordinates of a particle with index i

```c
extern fn get_coordinates(X: &mut [real], i: size_t) -> () {
    let P = get_particle_system_from_data(P_data_);
    let p = get_particle(P.addresses().get(i));
    let tmp = p.getCoordinates();
    for d in @iterate(0 as size_t, dimension()) {
        X(d) = tmp.get()(d);
    }
}
```

position. Similar functions for mass, velocity and force are also defined. After the particle data has
been obtained from pe, the next step is the actual time integration. A number of C++ interfaces to
the different steps of the cell list algorithm, the integration of position, velocity, the force calculation
and the redistribution of moving particles, are required. Listing 45 shows the implementation of
these interfaces. For now, it is assumed that the parallelization is solely performed by pe and
hence no additional threads are created, though an extension to a hybrid approach, combining a
shared and distributed memory parallelization, can be achieved by setting the number of threads
accordingly.

With the required interfaces available, a distributed version of the cell list algorithm is imple-
mented in C++. Listing 46 shows the essential parts of this implementation. At the beginning,
both in the pe and the Impala part of the simulation, all data structures and state variables must
be allocated and initialized. Accordingly, the same data must be deallocated at the end of the simulation. Both operations entail a constant overhead, that is insignificant for the overall complexity of the simulation and therefore are omitted here. Next, the main loop follows, which repeatedly executes the described simulation kernel for a number of time steps. First, the local `pe` instance iterates over all process-local blocks. `pe` manages its local data in form of a block forest. Because a static domain decomposition is employed, the blocks of a certain process are all located within the same cubic subdomain. Under the given conditions, each process possesses only one block, that includes its complete domain. However, the algorithm presented here, can in principle also handle multiple blocks per process by automatically treating the additional ghost particles as regular particles and creating enough cells that the combined subdomain of all blocks is covered. The drawback is that, even though only the updates of particles that are in the interior of the current block are propagated to `pe`, the cell list algorithm internally treats particles that are shared between multiple blocks as regular particles and hence updates them once per block. A solution would be the creation of one `ParticleSystem` instance per block. Although, this introduces further complication and is only reasonable in case of a dynamic load balancing, where blocks can be migrated to other processes. Because, as a proof-of-concept, this work focuses on a static domain decomposition, the case of multiple blocks per process is not further examined here. The particle system is first reinitialized to the number of particles contained in the current block by creating new or removing old particles. This is accomplished by the function `reinitialize_particle_system`. For the details of its implementation, the reader is referred to listing 48 in the appendix. Next, the actual data exchange takes place, by means of a simultaneous iteration over all local and ghost particles contained in the current `pe` block and over those internally stored in the Impala implementation of the cell list algorithm. For each particle, the data is first received from `pe` and then updated in the corresponding Impala particle. After all data has been exchanged, the time integration is performed, using the previously defined interfaces to the cell list algorithm. Now, a second data exchange takes place, where the updated particle data is propagated from the cell list data structures to the `pe` block. As ghost particles do not need to be updated, this involves only particles positioned in the interior of the current block. Finally, `pe` performs a synchronization of the global simulation state by redistributing particles that have moved out of their current subdomain.

Therewith, the discussion of the implementation of molecular dynamics in Impala is finished. In the next section, the performance of the presented code is examined in a number of cases. The mentioned C code serves as baseline for the single-node performance.
Listing 46 pe simulation loop

```c
int main(int argc, char **argv) {
    /* Initialization */
    ...
    for (size_t i = 0; i < simulationSteps; ++i) {
        for (auto blkIt = forest->begin(); blkIt != forest->end(); ++blkIt) {
            IBlock& currentBlock = *blkIt;
            Storage * storage = currentBlock.getData< Storage >();
            BodyStorage localStorage = *(storage)[0];
            BodyStorage shadowStorage = *(storage)[1];
            size_t np_local = localStorage.size() + shadowStorage.size();
            reinitialize_particle_system(np_local);
            for (auto bodyIt = localStorage.begin(); bodyIt != localStorage.end();
                 ++bodyIt) {
                /* pe -> Impala: Exchange local particle data */
                ...
            }
            for (auto bodyIt = shadowStorage.begin(); bodyIt != shadowStorage.end();
                 ++bodyIt) {
                /* pe -> Impala: Exchange ghost particle data */
                ...
            }
            // In the first iteration an initial distribution
            // and force calculation is necessary
            if (i == 0) {
                sort_particle_system();
                force_update();
            }
            position_update(dt);
            sort_particle_system();
            force_update();
            velocity_update(dt);
            for (auto bodyIt = localStorage.begin(); bodyIt != localStorage.end();
                 ++bodyIt) {
                /* Impala -> pe: Exchange local particle data */
                ...
            }
            pe::syncNextNeighbors<BodyTuple>(*forest, storageID, &tt, real_c(0.0), false);
        }
    }
    /* Deallocation */
    ...
```
4 Experiments

In the following, AnyDSL’s capabilities to generate parallel molecular dynamics simulation code are evaluated in a number of cases. The following setting is used in all simulations:

- Integration method: Velocity Verlet
- Potential: Lennard-Jones with $\varepsilon = 5$, $\sigma = 1$, $r_{cut} = 2.5$
- Particle mass: $m = 1$
- Boundary condition: Outflow
- Number of time steps: 1000
- Time step size: $10^{-3}$ in the single node and $10^{-5}$ in the multi-node setting
- Compiler backend: LLVM 3.8.1
- Optimization flags: -O3, -march=native
- Thorin revision: 4bedc364865e002d03633af20143563243581a84
- Impala revision: 3df271a42e110dfcdc09ae2c739e9eb31694b9a2
- AnyDSL runtime revision: b0d9c50e743d0b53116f01bbcd64d9f1998e60c4
- WaLBerla revision: be2839e1be79916ce56b5d3786e5ae9db46b56b9

For the single-node performance measurements, the following machine is used:

- Intel(R) Xeon(R) CPU E3-1275 v5 @ 3.60GHz (Skylake)
- Main memory: 64260 MB
- Caches: 32 kB (L1), 256 kB (L2), 8 MB (L3)

For the multi-node performance measurements, a cluster consisting of 8 nodes is employed, where each node consists of the following hardware:

- 4 x Intel(R) Xeon(R) CPU E7-4830, 2.13 GHz - 2.4GHz (max. turbo) (8 cores + SMT)
- Cache: 24 MB shared
- Main memory: 256 GB
- Connection via QDR Infiniband network

4.1 Single-node performance

To investigate the performance on a single node machine, three test cases have been implemented.

**Grid:** The particles are placed on a uniform cubic grid with spacing $2 \cdot r_{cut}$. The velocity components are randomly chosen from the interval $[-1,1]$, normalized and then multiplied with another random number from the interval $[-10,10]$. The domain size is adjusted to contain the complete grid. An example is shown in figure 11.

**Body collision:** The particles are arranged in form of two colliding bodies. Within both bodies, particles are placed on a uniform cuboid grid with a spacing of $2^{1/6}\sigma$ (Minimum of the Lennard-Jones potential). The upper body is initialized with a velocity of $(0, -10, 0)$ while all velocity components of the lower body are set to zero. The simulation takes place in a cubic domain of size 250. An example is shown in figure 12.

**Random:** The positions and velocities of all particles are randomly chosen using the method described in the grid case, with $\|x\|_2 \in [0, 100]$ and $\|v\|_2 \in [-10,10]$ and placed in a cubic domain of size 100. An example is shown in figure 13.
Figure 11: Simulation of a uniform cubic grid of 10,000 particles.

Figure 12: Simulation of the collision of two bodies that in total consist of 10,000 particles.

The color of the visualized particles corresponds to their velocity in y-direction, whereas red denotes a high (positive) and blue a low (negative) value.

Particles can leave the simulated domain.
First, the complexity of the two variants, the basic and the cell list algorithm, is investigated in an experimental setting. For this purpose, the sequential runtime is measured for an increasing number of particles. According to the derivation in section 2, the runtime of the basic algorithm increases quadratically with the number of particles while the cell list algorithm only shows a linear scaling. Figure 14 shows the measured runtime for the simulation of particle numbers between 200 and 2000. Both variants show the expected behavior of a quadratic and linear increase, respectively, with the number of particles. As a result of its higher complexity, in the given case, the basic algorithm is only faster for systems with particle numbers smaller than 400.

Next, the strong and weak scalability of the Impala implementation of the cell list algorithm is benchmarked in the three test cases. As a baseline, the mentioned C implementation is used, whose performance critical parts can be found in the appendix. In the strong scaling setup, the number of particles is fixed to 5000 while the number of (physical) processors is increased. In contrast, the weak scaling setup simulates particle numbers that are proportional to the number of

Figure 14: Measured runtime with an increasing number of particles.
processors. To prevent threads from switching between processors, the tool *Likwid* [63] is employed. The average runtime of 100 test runs is plotted in the figures 11, 12 and 13. The error bars denote the measured standard deviation. Additionally, the tables 1 and 2 show the runtime and parallel efficiency in the strong scaling and the tables 3 and 4 in the weak scaling case. Even though the same compiler back end is used, in all three test cases, the Impala implementation is slightly faster than the corresponding C code. The parallel efficiency of the Impala code ranges between 30 % and 65 %, which is due to the static domain decomposition, that has been employed so far. It can be concluded that, in the investigated cases, the particles are not always uniformly distributed over the domain, which leads to a load imbalance between processors. A dynamic load balancing is therefore indispensable to achieve a better parallel efficiency in arbitrary cases. Beyond that, the parallel performance of the cell list algorithm can hardly be predicted, because it highly depends on the physical characteristics of the simulated system.

<table>
<thead>
<tr>
<th>Test case</th>
<th>Average (s)</th>
<th>Standard Deviation (ms)</th>
<th>Parallel efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grid</td>
<td>0.5907</td>
<td>1.687</td>
<td>41.99 %</td>
</tr>
<tr>
<td>Collision</td>
<td>5.750</td>
<td>6.415</td>
<td>35.74 %</td>
</tr>
<tr>
<td>Random</td>
<td>0.6259</td>
<td>1.168</td>
<td>41.24 %</td>
</tr>
</tbody>
</table>

Table 1: Impala: Strong scaling with four processors.

<table>
<thead>
<tr>
<th>Test case</th>
<th>Average (s)</th>
<th>Standard Deviation (ms)</th>
<th>Parallel efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grid</td>
<td>0.6873</td>
<td>1.061</td>
<td>45.80 %</td>
</tr>
<tr>
<td>Collision</td>
<td>9.760</td>
<td>21.90</td>
<td>31.05 %</td>
</tr>
<tr>
<td>Random</td>
<td>0.7226</td>
<td>5.234</td>
<td>45.31 %</td>
</tr>
</tbody>
</table>

Table 2: C: Strong scaling with four processors.

<table>
<thead>
<tr>
<th>Test case</th>
<th>Average (s)</th>
<th>Standard Deviation (ms)</th>
<th>Parallel efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grid</td>
<td>1.075</td>
<td>2.586</td>
<td>46.01 %</td>
</tr>
<tr>
<td>Collision</td>
<td>10.66</td>
<td>56.44</td>
<td>63.75 %</td>
</tr>
<tr>
<td>Random</td>
<td>1.060</td>
<td>5.907</td>
<td>63.73 %</td>
</tr>
</tbody>
</table>

Table 3: Impala: Weak scaling with four processors.

<table>
<thead>
<tr>
<th>Test case</th>
<th>Average (s)</th>
<th>Standard Deviation (ms)</th>
<th>Parallel efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grid</td>
<td>1.369</td>
<td>45.93</td>
<td>44.80 %</td>
</tr>
<tr>
<td>Collision</td>
<td>11.65</td>
<td>14.47</td>
<td>82.84 %</td>
</tr>
<tr>
<td>Random</td>
<td>1.320</td>
<td>42.88</td>
<td>64.47 %</td>
</tr>
</tbody>
</table>

Table 4: C: Weak scaling with four processors.
Figure 15: Single-node scaling in the grid test case.

Figure 16: Single-node scaling in the collision test case.
(a) Strong scaling with 5000 particles.  
(b) Weak scaling with 2500 particles per processor.

Figure 17: Single-node scaling in the random test case.
4.2 Multi-node performance

To evaluate the scalability of the pe coupled implementation of the cell list algorithm, a grid test case is investigated in the following weak scaling setup:

- 32 MPI processes per node
- 1000 particles per process
- Each velocity component is randomly chosen from the interval \([-100, 100]\).

The results are summarized in table 5 and have been additionally visualized in figure 18. In accordance with the single-node measurements, the parallel efficiency drops with an increasing number of processes. The parallel efficiency declines even faster, because of the synchronization of the nodes, which includes the additional communication that is required for the exchange of ghost and moving particles. An investigation of the runtime of the individual components of the simulation shows the following picture. If only one node is employed, about 35% of the time is spent in the execution of the cell list algorithm and about 60% in the synchronization. In contrast, when eight nodes are employed, the share of the cell list algorithm drops to 15% and now 80% of the time is spent for the synchronization. Therefore, the synchronization currently represents a significant overhead, that limits the scalability of the current implementation. Currently, the largest system that has been simulated includes \(10^7\) particles and has been visualized in figure 18. However, to simulate even larger systems on recent supercomputers, the synchronization overhead needs to be reduced.

<table>
<thead>
<tr>
<th>#Nodes</th>
<th>Average (s)</th>
<th>Standard Deviation (s)</th>
<th>Parallel efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>16.39</td>
<td>0.1110</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>27.41</td>
<td>0.1014</td>
<td>59.80%</td>
</tr>
<tr>
<td>3</td>
<td>33.39</td>
<td>0.1160</td>
<td>49.09%</td>
</tr>
<tr>
<td>4</td>
<td>48.56</td>
<td>0.1619</td>
<td>33.75%</td>
</tr>
<tr>
<td>5</td>
<td>56.84</td>
<td>0.8016</td>
<td>28.84%</td>
</tr>
<tr>
<td>6</td>
<td>57.93</td>
<td>0.1957</td>
<td>28.30%</td>
</tr>
<tr>
<td>7</td>
<td>73.01</td>
<td>0.2667</td>
<td>22.45%</td>
</tr>
<tr>
<td>8</td>
<td>82.49</td>
<td>0.2703</td>
<td>19.87%</td>
</tr>
</tbody>
</table>

Table 5: Impala: Weak scaling in a multi-node setting.

Figure 18: Weak scaling in a multi-node setting.
Figure 19: Simulation of $10^7$ particles that are initially positioned in a grid.
5 Conclusion and future work

This thesis has provided a parallel implementation of molecular dynamics with short-range interactions in Impala, that permits the specification of algorithmic choices on a high level of abstraction. For the given use case, it has been shown that the AnyDSL framework is able to generate efficient code, whose performance is on par with a comparable low-level implementation in the C programming language. Furthermore, it has been demonstrated that the given implementation can be integrated into existing distributed parallel frameworks with minimal effort.

However, the given implementation, so far, only represents a proof-of-concept and therefore has various limitations. As it is based on a vanilla implementation of the cell list algorithm, certain modifications and extensions, from which a few have been mentioned in this thesis, can be considered to improve its performance and scope of application. The neighbor list algorithm is another method for the fast calculation of short-range interactions, that is employed in many molecular dynamics codes and hence has to be considered as an alternative. Furthermore, vectorization is a key prerequisite for optimal performance on modern architectures, that has been ignored so far. AnyDSL incorporates RV [64], a unified region vectorizer for LLVM, that produces vectorized code that can be as fast as hand-optimized versions using low-level intrinsics [65]. The utilization of automatic vectorization necessitates the storage of data in a continuous chunk of memory and therefore requires a modification of the data structures of the cell list algorithm. Another important aspect is the scalability of the given implementation. So far, the parallelization is based on a static domain decomposition. As the performance evaluation in section 4 shows, the assumption that the particles are evenly distributed over the domain does not generally apply and a dynamic load balancing is a necessary condition to achieve good scalability on large multi-node systems. Ultimately, the given implementation acts as a prototype, but as such, lacks important features required for its deployment as fully functioning molecular dynamics software. Even though many intermolecular forces can be modeled with short-range interactions, most real-world applications involve potentials that do not drop rapidly, as for example the electrostatic potential. The presence of these potentials makes the treatment of long-range interactions, with either a mesh or tree based approach, indispensable. Furthermore, the incorporation of ab initio methods, which are rooted in quantum mechanics, could be considered to improve the physical accuracy in certain parts of the simulated system. Achieving the mentioned enhancements and extensions comprises a significant implementation effort, which is illustrated by the code size of popular molecular dynamics frameworks, that have been developed in decades of research. WaLBerla is a parallel multi-physics framework, that achieves excellent scalability on some of the largest supercomputers available [66]. It has been extended to the domain of rigid body methods by incorporating the pe physics engine. An extension by molecular dynamics therefore is in reach as well. WaLBerla already offers the data structures necessary for a treatment of long-range interactions in a distributed-parallel setting. The combination of these highly optimized data structures with the strength of AnyDSL’s code generation capabilities, could open up the increasingly important field of molecular dynamics to WaLBerla.
References


**Appendix**

**Listing 47** Particle implementation

```rust
fn get_particle(data: &u8) -> Particle {
    Particle {
        getMass: || { bitcast[b\[real\]](data)(0) },
        getCoordinates: || {
            let mut res : [real * 3];
            for d in @iterate(0 as size_t, dimension()) {
                res(d) = bitcast[b\[real\]](data)(1 as size_t + d);
            }
            get_vector(res)
        },
        getVelocities: || {
            let mut res : [real * 3];
            for d in @iterate(0 as size_t, dimension()) {
                res(d) = bitcast[b\[real\]](data)(1 as size_t + dimension() + d);
            }
            get_vector(res)
        },
        getForces: || {
            let mut res : [real * 3];
            for d in @iterate(0 as size_t, dimension()) {
                res(d) = bitcast[b\[real\]](data)(1 as size_t + 2 as size_t * dimension() + d);
            }
            get_vector(res)
        },
        getForces_old: || {
            let mut res : [real * 3];
            for d in @iterate(0 as size_t, dimension()) {
                res(d) = bitcast[b\[real\]](data)(1 as size_t + 3 as size_t * dimension() + d);
            }
            get_vector(res)
        },
        setMass: |m| { bitcast[\mut real](data)(0) = m; },
        setCoordinates: |v| {
            for d in @iterate(0 as size_t, dimension()) {
                bitcast[\mut real](data)(1 as size_t + d) = v.get()(d);
            }
        },
        setVelocities: |v| {
            for d in @iterate(0 as size_t, dimension()) {
                bitcast[\mut real](data)(1 as size_t + dimension() + d) = v.get()(d);
            }
        },
        setForces: |v| {
            for d in @iterate(0 as size_t, dimension()) {
                bitcast[\mut real](data)(1 as size_t + 2 as size_t * dimension() + d) = v.get()(d);
            }
        },
        setForces_old: |v| {
            for d in @iterate(0 as size_t, dimension()) {
                bitcast[\mut real](data)(1 as size_t + 3 as size_t * dimension() + d) = v.get()(d);
            }
        }
    }
}
```
Listing 48 External interface for the reinitialization of a particle system

extern fn reinitialize_particle_system(np: size_t) -> () {
    let P = get_particle_system_from_data(P_data_);
    let np_old = P.np();
    if(np_old != np) {
        let constants = P.constants();
        let data_cells = P.data_cells();
        let ghost_layer = P.ghost_layer();
        let l = P.l();
        let nc = P.nc();
        deallocate(P.data_addresses());
        // A new array of pointers to the particle data addresses is required
        let data_addresses = allocate(np * sizeof[u8]() as size_t);
        let P_new = get_particle_system(np, nc, ghost_layer, l,
                                       data_cells, constants, data_addresses);
        if(np_old > np) {
            let mut end = l as size_t;
            for d in iterate(0 as size_t, dimension()) {
                end *= nc(d);
            }
            let mut j = np_old as size_t;
            let mut c = 0 as size_t;
            while (j > np as size_t && c < end) {
                let head = P_new.head(c);
                let node = remove(head);
                if(node != 0 as &mut ParticleList) {
                    deallocate_particle_node(node);
                    --j;
                } else {
                    ++c;
                }
            }
        } else {
            for j in iterate(np_old, np) {
                let node = allocate_particle_node();
                let p = get_particle_from_node(*node);
                p.setCoordinates(get_null_vector());
                // inserts the new list element into an interior cell
                push_node(node, P_new);
            }
            // Initializes the ArrayOfAddresses such that it points to the particle data
            init_addresses(P_new);
            P_data_ = get_data_from_particle_system(P_new);
        }
    }
}
#define DIM 3
#define EPS 1e-12

typedef double real;

typedef struct Particle {
    real m;
    real x[DIM];
    real v[DIM];
    real F[DIM];
    real F_old[DIM];
} Particle;

typedef struct ParticleList {
    Particle p;
    struct ParticleList *next;
} ParticleList;

void insertNode(ParticleList **head, ParticleList *pl) {
    pl->next = *head;
    *head = pl;
}

void removeNode(ParticleList **head) {
    *head = (*head)->next;
}

typedef ParticleList* Cell;

typedef struct ParticleSystem {
    size_t np;
    size_t nc[DIM];
    size_t ghost_layer;
    size_t start[DIM];
    size_t end[DIM];
    real l[DIM];
    real r_cut_inv;
    real offset;
    Cell *grid;
    Constants constants;
    uintptr_t *addresses;
} ParticleSystem;

Listing 50 C implementation: Velocity Verlet integration

void update_x(Particle * restrict p, real const dt) {
    real const a = dt * 0.5 / p->m;
    for(size_t d = 0; d < DIM; ++d) {
        p->x[d] += dt * (p->v[d] + a * p->F[d]);
        p->F_old[d] = p->F[d];
    }
}

void update_v(Particle * restrict p, real const dt) {
    real const a = dt * 0.5 / p->m;
    for(size_t d = 0; d < DIM; ++d) {
        p->v[d] += a * (p->F[d] + p->F_old[d]);
    }
}
 typedef struct {
    real r_cut;
    real sigma;
    real epsilon;
    real r_cut_sqr;
    real tmp1;
    real tmp2;
} Constants;

 real sqr(real const x) {
    return x * x;
}

 void force(Particle *restrict p1, Particle *restrict p2, bool const write1,
            bool const write2, Constants const constants) {
    real r = 0.0;
    real dist[6];
    for(size_t d = 0; d < DIM; ++d) {
        dist[d] = p2->x[d] - p1->x[d];
        r += sqr(dist[d]);
    }
    if(r < constants.r_cut_sqr) {
        real const r8_inv = 1.0/sqr(sqr(r));
        real const f = constants.tmp1 * r8_inv * (1.0 - r*r8_inv*constants.tmp2);
        real F[6];
        for(size_t d = 0; d < DIM; ++d) {
            F[d] = f * dist[d];
        }
        if(write1) {
            for(size_t d = 0; d < DIM; ++d) {
                p1->F[d] += F[d];
            }
        }
        if(write2) {
            for(size_t d = 0; d < DIM; ++d) {
                p2->F[d] -= F[d];
            }
        }
    }
}
Listing 52 C implementation: Particle redistribution

```c
void compute_cell_position(Particle *p, ParticleSystem *P, size_t jc[DIM]) {
    for (size_t d = 0; d < DIM; ++d) {
        jc[d] = (size_t)floor((p->x[d] + P->offset) * P->r_cut_inv);
    }
}

void move_particles(ParticleSystem P) {
    for (size_t k = P.start[2]; k < P.end[2]; ++k) {
        for (size_t j = P.start[1]; j < P.end[1]; ++j) {
            for (size_t i = P.start[0]; i < P.end[0]; ++i) {
                ParticleList *q = &(P.grid[i + P.nc[0] * (j + P.nc[1] * k)]);
                ParticleList *pl = *q;
                size_t jc[3];
                while (pl != NULL) {
                    compute_cell_position(&pl->p, P, jc);
                    if (i != jc[0] || j != jc[1] || k != jc[2]) {
                        removeNode(q);
                        insertNode((P.grid[jc[0]+P.nc[0]*jc[1]+P.nc[1]*k]), pl);
                    } else {
                        q = q->next;
                    }
                    pl = *q;
                }
            }
        }
    }
    move_particles(P);
}
```

Listing 53 C implementation: Integration of position and velocity

```c
void update_coordinates(ParticleSystem P, real const dt) {
    #pragma omp parallel for schedule(static)
    for (size_t k = P.start[2]; k < P.end[2]; ++k) {
        for (size_t j = P.start[1]; j < P.end[1]; ++j) {
            for (size_t i = P.start[0]; i < P.end[0]; ++i) {
                ParticleList *restrict pl = P.grid[i + P.nc[0] * (j + P.nc[1] * k)];
                if (pl != NULL) {
                    update_x(pl->p, dt);
                }
            }
        }
    }
    move_particles(P);
}

void update_velocities(ParticleSystem P, real const dt) {
    #pragma omp parallel for schedule(static)
    for (size_t k = P.start[2]; k < P.end[2]; ++k) {
        for (size_t j = P.start[1]; j < P.end[1]; ++j) {
            for (size_t i = P.start[0]; i < P.end[0]; ++i) {
                ParticleList *restrict pl = P.grid[i + P.nc[0] * (j + P.nc[1] * k)];
                if (pl != NULL) {
                    update_v(pl->p, dt);
                }
            }
        }
    }
}
```
Listing 54 C implementation: Force calculation

```c
void compute_force(ParticleSystem P) {
    #pragma omp parallel for schedule(static)
    for (size_t k = P.start[2]; k < P.end[2]; ++k) {
        for (size_t j = P.start[1]; j < P.end[1]; ++j) {
            for (size_t i = P.start[0]; i < P.end[0]; ++i) {
                ParticleList *restrict pl=P.grid[i+P.nc[0]*(j+P.nc[1]*k)];
                if (pl != NULL) {
                    for (size_t d = 0; d < DIM; ++d) {
                        pl->p.F[d] = 0.0;
                    }
                }
            }
        }
    }
    size_t ic_start[DIM], ic_end[DIM];
    for (size_t d = 0; d < DIM; ++d) {
        ic_start[d] = 0;
        ic_end[d] = P.nc[d];
    }
    #pragma omp parallel for schedule(static)
    for (size_t k1 = ic_start[2]; k1 < ic_end[2]; ++k1) {
        for (size_t j1 = ic_start[1]; j1 < ic_end[1]; ++j1) {
            for (size_t i1 = ic_start[0]; i1 < ic_end[0]; ++i1) {
                size_t k2_start, j2_start, i2_start;
                size_t k2_end, j2_end, i2_end;
                bool write_i = true;
                if (i1 >= P.start[0]) {i2_start = i1 - 1;}
                else {i2_start = i1 + 1; write_i = false;}
                if (i1 < P.end[0]) {i2_end = i1 + 2;}
                else {i2_end = i1; write_i = false;}
                if (j1 >= P.start[1]) {j2_start = j1 - 1;}
                else {j2_start = j1 + 1; write_i = false;}
                if (j1 < P.end[1]) {j2_end = j1 + 2;}
                else {j2_end = j1; write_i = false;}
                if (k1 >= P.start[2]) {k2_start = k1 - 1;}
                else {k2_start = k1 + 1; write_i = false;}
                if (k1 < P.end[2]) {k2_end = k1 + 2;}
                else {k2_end = k1; write_i = false;}
                for (ParticleList * restrict pl1=P.grid[i1+P.nc[0]*(j1+P.nc[1]*k1)];
                    pl1 != NULL; pl1 = pl1->next) {
                    for (size_t k2 = k2_start; k2 <= k2_end; ++k2) {
                        for (size_t j2 = j2_start; j2 <= j2_end; ++j2) {
                            for (size_t i2 = i2_start; i2 <= i2_end; ++i2) {
                                bool write_j = true;
                                if (i2 < P.start[0]) || i2 > P.end[0] || j2 < P.start[1] || j2 > P.end[1] || k2 < P.start[2] || k2 > P.end[2]) {
                                    write_j = false;
                                }
                                for (ParticleList * restrict pl2 = P.grid[i2+P.nc[0]*(j2+P.nc[1]*k2)];
                                    pl2 != NULL; pl2 = pl2->next) {
                                    if (pl1 < pl2) {
                                        force(&pl1->p, &pl2->p, write_i, write_j, P.constants);
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}
```
Short Biography

Jonas Schmitt was born in Forchheim, Upper Frankonia, where he went to first and secondary school. After studying materials science for two semesters, he completed a bachelor and master degree in computer science at the University of Erlangen. As a side project, he worked on the simulation and optimization of micromanagement within the real-time strategy game StarCraft II. He currently is a Ph.D. student at the chair of system simulation and works in the area of code generation and metaprogramming for large-scale simulations.