Hybrid Acceleration of a Molecular Dynamics Simulation Using Short-Ranged Potentials

JULIAN HORNICH

Master of Science Thesis
Stockholm, Sweden 2013
Hybrid Acceleration of a Molecular Dynamics Simulation Using Short-Ranged Potentials

JULIAN HORNICH

Master’s Thesis in Scientific Computing (30 ECTS credits)
Master Programme in Scientific Computing (120 credits)
Royal Institute of Technology year 2013
Supervisor at KTH was Michael Schliephake
Examiner was Michael Hanke

TRITA-MAT-E 2013:36
ISRN-KTH/MAT/E--13/36--SE

Royal Institute of Technology
School of Engineering Sciences

KTH SCI
SE-100 44 Stockholm, Sweden

URL: www.kth.se/sci
Abstract

Molecular dynamics simulations are a very useful tool to study the behavior and interaction of atoms and molecules in chemical and bio-molecular systems. With the fast rising complexity of such simulations hybrid systems with both, multi-core processors (CPUs) and multiple graphics processing units (GPUs), become more and more popular. To obtain an optimal performance this thesis presents and evaluates two different hybrid algorithms, employing all available compute capacity from CPUs and GPUs. The presented algorithms can be applied for short-range force calculations in arbitrary molecular dynamics simulations.
Referat

Hybrid acceleration av en molekulärdynamisk simulation för potentialer med kort räckvidd

Molekulärdynamiska simulationer är ett mycket lämpligt verktyg för att studera beteendet och växelspelet av atomer och molekyler inom kemiska och biomolekylära system. Hybrid datorsystem som innehåller såväl processorer med multipla kärnor (CPU:er) som flera grafikprocessorer (GPU:er) blir populära i växande utsträckning. Vi undersöker två olika hybrida algoritmer som använder den hela beräkningsförmågan av CPU:er och GPU:er för att få ut den optima beräkningsprestandan. De presenterade algoritmerna kan användas i godtyckliga molekulärdynamiska simulationer med potentialer av kort räckvidd.
# Contents

## I Introduction

1 Introduction ........................................ 1

1.1 Existing Implementations .......................... 3
  1.1.1 GROMACS ...................................... 4
  1.1.2 NAMD ........................................... 5
  1.1.3 LAMMPS ......................................... 6

## 2 Theoretical Background

2 Mathematical and Physical Background ................. 7
  2.1.1 The Schrödinger Equation ..................... 8
  2.1.2 The Derivation of Classical Molecular Dynamics 10
  2.1.3 Analytical Potentials ......................... 11
  2.1.4 The Lennard-Jones Potential ................. 12
  2.2 Algorithmic Background .......................... 13
    2.2.1 Time-Discretization ......................... 13
    2.2.2 Initial Conditions .......................... 17
    2.2.3 NVE- and NVT-Ensembles .................... 19
    2.2.4 The Linked-Cell Method ...................... 20
    2.2.5 Boundary Conditions ......................... 23
    2.2.6 Reduced Units ................................ 25
  2.3 Computer Architecture Background ................. 26
    2.3.1 CPUs ......................................... 27
    2.3.2 GPUs ......................................... 28
    2.3.3 Cluster Computers ........................... 29

## II Algorithms

3 Implementation ....................................... 31
  3.1 Basic Algorithm ................................. 33
    3.1.1 Algorithmic Complexity and Operation Count 34
  3.2 Data Structures ................................. 35
    3.2.1 Memory Consumption .......................... 36
3.3 Vectorization ........................................... 37
3.4 Parallelization ....................................... 41
  3.4.1 Multi-Threading .................................. 41
  3.4.2 Utilization of Multiple Nodes .................... 43
  3.4.3 Combining OpenMP and MPI ....................... 48
3.5 Basic GPU Algorithm ................................. 49
3.6 Hybrid Implementations ............................... 52
  3.6.1 Stopwatch Algorithm .............................. 52
  3.6.2 Domain Decomposition Algorithm ................ 55

III Results 61

4 Benchmarks 63
  4.1 Performance Measures ............................... 63
    4.1.1 Speedup and Efficiency ........................ 63
    4.1.2 Scalability .................................... 64
  4.2 Setup .............................................. 64
    4.2.1 Hardware Setup .................................. 64
    4.2.2 Compiler and Compiler Options .................. 64
    4.2.3 Simulation Setup ................................ 65
  4.3 Single Node Performance ............................ 66
    4.3.1 Basic vs. Reduced Algorithm .................... 66
    4.3.2 Atomic Problems ................................ 67
    4.3.3 OpenMP Performance ............................. 68
    4.3.4 GPU and Hybrid Performance .................... 69
    4.3.5 Multi-GPU Performance ........................... 72
  4.4 Multi-Node Performance ............................. 73

5 Discussion and Conclusion 79

6 Outlook 81

Appendices 82

List of Figures 82

List of Tables 85

Bibliography 89
Part I

Introduction
Chapter 1

Introduction

The main goal of molecular dynamics is to understand effects on an atomic and sub-atomic level by simulating complex systems of atoms. To solve such problems a mathematical representation of the physical system is needed. For macroscopic objects such a representation is given by Newton’s three laws of motion. Those laws give accurate approximations for phenomena of a wide range, but lack accuracy when they are applied on an atomic level. To express movements and interactions on such a tiny scale the theory of quantum mechanics allows more accurate predictions. Its roots go back to scientists like Christiaan Huygens and Leonhard Euler, who tried to describe experiments with light and came up with their wave theory of light. Max Planck – who is today considered to be the initiator of quantum theory – used Ludwig Boltzmann’s proposal of discrete energy states of physical systems and published his quantum hypothesis in the year 1900. Employing Werner Heisenberg’s uncertainty principle, in 1926 Erwin Schrödinger presented an equation to describe the behavior of such a quantum system. It no longer predicts the exact motion of discrete particles, but employs probabilities. Since the computational effort to solve systems with an interesting number of atoms or molecules rises exponentially when using Schrödinger’s equation directly, different assumptions and simplifications have to be made, with respect to the specific use case of the simulation. During this thesis a derivation of a classical molecular dynamics simulation from the Schrödinger equation will be given.

With the ever rising desire to simulate larger physical systems with higher accuracy, faster and larger computer systems have to be employed. Modern compute clusters and even desktop systems are no longer simply using more capable multi-core processors (CPUs), but also include graphics processing units (GPUs). To utilize all the available computing power of modern computer systems, it is necessary to adapt existing algorithms. In some cases the adaption is trivial but quite often it is necessary to use completely new or reformulated algorithms. Especially when simulations are run on large cluster computers, it is necessary to get the most performance out of the system, because compute-time on such systems is scarce and precious. But not only with respect to rare compute-time on clusters, a higher
efficiency of the used algorithm should be the aim. Also a lower energy consumption through shorter execution time of a simulation should be a challenging incentive. With energy consumptions in the range of up to 10 MegaWatt for modern clusters this should definitely be of interest [15].

The goal of this thesis is to implement an efficient algorithm for a molecular dynamics simulations using short-ranged potentials with hybrid acceleration. In the following section 1.1 existing implementations of other molecular dynamics frameworks will be outlined. Chapter 2 will cover the theoretical background including the derivation from quantum mechanical principles and basic algorithm. Furthermore a short overview over computer architecture will be given in section 2.3 to define the used terms in later chapters. Chapter 3 comprises details about the specific implementation and optimizations employed in this thesis, including the two hybrid algorithms in the sections 3.6.1 and 3.6.2. The benchmark results of the presented algorithms can be found in chapter 4.

1.1 Existing Implementations

It is interesting to have a look at approaches by other molecular dynamics (MD) codes. Because one is most interested in the way those implementations handle the parallelization and interaction of CPUs and GPUs, the focus of this section will be on this topic. Since there exist a wide range of different MD implementations, representatively three big molecular dynamics frameworks with parallelization for clusters and GPU support were selected, namely GROMACS, NAMD and LAMMPS have been picked out. Those three frameworks will now be presented with respect to their parallelization and hybrid implementation.

1.1.1 GROMACS

The “GROningen MAchine for Chemical Simulations” (GROMACS) performs simulations of classical molecular dynamics and is usually employed for proteins and lipids. It is capable of handling millions of particles [8]. It supports the parallelization framework MPI for a long time and uses OpenMP for multi-threading. Vectorization with SSE and AVX instructions is used to exploit all the computing power on each node. Since version 4.5 (Sept. 2010) GPU computations are supported, but all computations are done on the GPUs, leaving the CPUs only with communication work and giving away a lot of computing power. Version 4.6, which was released just before the end of this thesis, introduces CPU-GPU parallelization but is not yet optimal. The different possible parallelization techniques of GROMACS can be summarized as the following:

- Pure MPI parallelization: Is using neutral-territory domain decomposition which supports fully automatic dynamic load balancing. Particle decomposition is also supported with MPI. Multithreading with thread-MPI is also
1.1. EXISTING IMPLEMENTATIONS

available, it can be run on machines without MPI and has proven to be a little faster.

- MPI + OpenMP: Multi-level parallelism aims to address the NUMA and communication related issues by employing efficient intra-node parallelism, typically multithreading.

- Hybrid/heterogeneous acceleration: To efficiently use all compute resources available, CPU and GPU computation is done simultaneously, overlapping with the OpenMP multithreading. This is currently under development and load balancing is not yet optimal. Similar to the LAMMPS approach, which will be described later, the overlapping CPU-GPU computation utilizes neighbor lists on the GPU to minimize computational complexity.

In the online documentation [1] of the latest release version 4.6, problems with the hybrid acceleration techniques are mentioned. The load balancing between GPU and CPU does not seem to give the desired results, leaving one of the devices idle until the other has finished its work.

1.1.2 NAMD

The second MD code considered is NAMD, “Not (just) Another Molecular Dynamics program”, an application designed specifically for the simulation of large biomolecular systems using special object-oriented techniques. The parallel programming model Charm++ is used, which invokes methods on objects, so called “chares”, that may reside locally or on another machine. A message is sent to the target chare, which can also be a collection of chares, with the corresponding command to be executed. Computation does not follow a set order, instead it is driven by the incoming messages. Short and long ranged interactions such as 2-, 3-, and 4-body interactions, electrostatic interactions, and van der Waals interactions are supported [12]. NAMD uses a spatial domain decomposition into so called “patches”, as well as measurement-based load distribution, where calculation of the non-bonded forces can be assigned to any processor. The first 200 iterations are used to figure out suitable load distribution and hence slower than subsequent iterations, with applied load distribution [10]. CPU-GPU parallelization in NAMD is done per patch using CUDA and distributes the work as the following:

- GPU: all short range forces are computed from the particle positions on the GPUs
- CPU: position updates, long range interactions and bonded forces are computed on the CPU

This is quite similar to the implementations of GROMACS and LAMMPS, but in [9] it is stated that this method does not map perfectly to the object oriented NAMD approach, but a speed up is still observed, however not on a satisfying level.
First computing “off-patch” forces, whose new positions might need to be commu-
cicated, and then computing “on-patch” forces, results in an optimized behavior and
less idling. Benchmarks from [9] show, that the speed-up decreases with increasing
numbers of cores & GPUs, which is definitely not desired.

1.1.3 LAMMPS

The “Large-scale Atomic/Molecular Massively Parallel Simulator” (LAMMPS) is a
molecular-dynamics code that also uses MPI for parallel communication. The simu-
lation domain is split in small three-dimensional cubes by spatial-decomposition [14].
The code is optimized for short-ranged potentials which makes it most efficient, if the
particle density is distributed uniformly. GPU (CUDA and OpenCL) and OpenMP
support exists for many code features. In [16] a very promising implementation for
CPU-GPU is presented and is based on the following idea: The time integration is
completely done on the CPU, while the short ranged potentials are computed on
the GPU and CPU together. Load balancing of particles between CPU and GPU is
done by evaluating the execution time of both every $\sim 10$ steps. This way idle time
of any of the compute units is reduced. To reduce the idle time of the GPUs further,
all cores of one node share all GPUs and submit execution commands concurrently.
The Lennard-Jones-Potential uses the linked cell algorithm as well as neighbor lists
to minimize the computational complexity from $O(N^2)$ to $O(N)$. The neighbor list
computation is done completely on the GPUs which, according to [16], has proven
to be faster, especially on recent GPUs with larger caches.

This algorithm is very similar to the implementation in the GROMACS frame-
work and will be the base for one of the two hybrid implementations in this thesis.
Chapter 2

Theoretical Background

Before specific implementations and algorithmic optimizations are described, it is important to give a description of the mathematical and physical background of the considered problem. This will be done in section 2.1. The last part of this chapter will then give a short overview over current computer architecture, such as modern CPU and GPU architectures and fast networks as they are used in recent clusters (section 2.3). With that knowledge the implementation details and optimizations of this thesis will then be presented in chapter 3.

2.1 Mathematical and Physical Background

From basic physics classes in school everyone is familiar with Isaac Newton’s three laws of motion. Those formulas have been proven to be applicable in a wide range of fields. The motion of general, macroscopic objects with given initial positions and velocities can be calculated utilizing Newton’s formulas for any later point in time. But around the year 1900 physicists like Max Planck, Albert Einstein, Werner Heisenberg and Erwin Schrödinger developed the theory of quantum mechanics. This theory started off as a description of wave phenomena, such as light. But with the discovery of the wave-particle duality it was possible to state a formula that could describe and predict those quantum mechanical effects. This formula is known as the Schrödinger equation and until today the most important formula to describe physical systems of quantum scale.

The following section will derive Newton’s laws of motion from the Schrödinger equation and hence describe the validity of the later presented algorithm, which uses Newton’s laws of motion on an atomic level. The subsequent sections will follow the description of this topic, as they are presented in [7] and [2] and probably many more.
2.1.1 The Schrödinger Equation

Erwin Schrödinger, an Austrian physicist, published his formula in 1926. In contrast to Newton’s laws of motion it no longer predicted the movement of objects uniquely. By using Werner Heisenberg’s uncertainty principle it was now necessary to describe the motion of atoms, molecules and electrons in terms of probabilities, since the precision of the measurement of both, the position and impulse of a particle, is no longer arbitrarily accurate on that scale. The Schrödinger equation describes this phenomenon with a wave function $\Psi$ of an observed system. This wave function is dependent of the observed point in time $t$ and the position of all the nuclei and electrons in the system. If we consider a system with $N$ nuclei with positions $X$ and $M$ electrons with positions $x$, the wave function can be written as:

$$\Psi = \Psi(X_1, ..., X_N, x_1, ..., x_M, t)$$ (2.1)

From this representation the degrees of freedom $d$ of such a system can be estimated as $d = 3(N + M)$. The function space of the wave function is a Hilbert space with $\mathcal{H} = L^2(\mathbb{R}^d)$. From the dimension of this function space the problem for numerical simulations can already be observed. Due to its fast growth with an increasing number of involved particles, numerical methods will soon become very complex and time consuming. For one simple water molecule the dimension parameter is already $d = 39$ and for complex molecules this can quickly become of order $d > 10^5$.

The time-dependent Schrödinger equation, which is a partial differential equation, can be written as:

$$i\hbar \frac{\partial}{\partial t} \Psi = H\Psi$$ (2.2)

Here $i$ is the imaginary unit, $\hbar$ is the reduced Planck’s constant, which represents the quantization of matter and light and can be used as a measure for subatomic scales. The fact that the electrons in an atom form standing waves, in this context also known as the atomic orbitals, can be exploited. Equation 2.2 can then be reformulated to represent those standing waves. The time-independent form of the Schrödinger equation is a linear eigenvalue equation and can be written as:

$$H\Psi = E\Psi$$ (2.3)

Here $H$ represents a Hamilton operator, which depends on the positions and impulses of the nuclei and electrons of the system, and characterizes the total energy of the system. For a molecular system the Hamiltonian operator $H$ consists of a potential and kinetic part:
2.1. MATHEMATICAL AND PHYSICAL BACKGROUND

\[ H = - \sum_{i=1}^{N} \frac{1}{2m_i} \Delta_i + \sum_{j=1}^{M} \frac{1}{2} \Delta_j \]  
\[ \quad - \sum_{i<j}^{M} \frac{1}{|x_i - x_j|} + \sum_{i}^{N} \sum_{j}^{M} z_j \frac{1}{|x_i - X_j|} + \sum_{i<j}^{N} z_i z_j \]  
\[ = T_n + T_e + V_{e,e} + V_{e,n} + V_{n,n} \]  
\[ (2.4) \]

The kinetic part (2.4) is split up into a nuclei kinetic energy and electronic kinetic energy. The potential part (2.5) is split up into three parts: the interactions between nuclei \( V_{n,n} \), the interactions between nuclei and electrons \( V_{e,n} \) and the electron-electron interaction \( V_{e,e} \), which is also known as the Coulomb interaction. Here the so called \textit{atomic unit system} is used with \( m_e = e = \hbar = \frac{1}{4\pi\epsilon_0} = 1 \), which results in a nice simplification of the above equation and only leaves the nuclei masses \( m \) and their charges \( z \).

The last unknown part in equation 2.3 is \( E \) which represents the energy state of the observed system. The smallest eigenvalue of equation 2.3 is also called the ground-state and defined by the scalar product:

\[ E_0 = \inf \{ \langle \Psi_0, H \Psi_0 \rangle : \Psi_0 \in \mathcal{H}, \| \Psi_0 \|_{L^2} = 1 \} \]  
\[ (2.7) \]

The reason why we restrict ourselves to the ground state is the simple fact, that this energy state is the most stable, since there is no state with a lower energy the system could switch to.

The Born-Oppenheimer Approximation

We are still left with the problem that for simulations with reasonable numbers of atoms the computational effort is much too high. To tackle this fact another approximation is used, the \textit{Born-Oppenheimer approximation}. This approximation uses the fact, that the system is composed out of some heavy nuclei and a number of lighter electrons. Due to the big inertia of the nuclei compared to the electrons, the electrons almost instantaneously adapt to new nuclei configurations from the standpoint of the nuclei. And from the standpoint of the electrons, the nuclei are nearly standing still. This lets us split up our wave function into a product of a nuclei part \( \Theta \) and an electronic part \( \Phi \):

\[ \Psi_{mol}(X,x) = \Phi(X,x)\Theta(X) \]  
\[ (2.8) \]

This also allows us to split up the Schrödinger equation into two parts. We can then solve the electronic Schrödinger equation first and use the electronic eigenstates and energies in a second step in the nuclear Schrödinger equation. The
nuclei movement is not influenced by the movement of the electrons, but the nuclei feel the eigenenergy of the electronic state. This results in an separate potential for every nuclei. The original time-independent Schrödinger equation can now be reformulated in the following steps:

\[
\begin{align*}
(T + V_{e,n} + V_{n,n}) \Phi_{h}(X, x) &= E_{h}(X) \Phi_{h}(X, x) \quad (2.9) \\
(T_{n} + E_{h}(X)) \Theta_{h,k}(X) &= E_{h,k} \Theta_{h,k}(X) \quad (2.10)
\end{align*}
\]

Ab initio methods versus Analytical Methods

From the equations (2.9) and (2.10) one can also see that the Hamiltonian has been split up into two parts: \( H = H_{e} + H_{n} \). To solve those equations one can now use so called \textit{ab initio} methods. Those methods solve the electronic Schrödinger equation approximately to obtain the effective potential energy of the nuclei. The currently used methods can be split up into two main categories, the \textit{Wave Function Methods} and the \textit{Density Functional Theory Methods}:

- \textbf{Wave Function Methods}: Hartree-Fock and Self-Consistent Field methods
- \textbf{Density Functional Theory}: Hohenberg-Kohn, Kohn-Sham and Thomas-Fermi methods

A more detailed description of those methods can be found in \cite{2}. Those methods are still very time consuming and computational costly but also very accurate. For this reason another approach is chosen in this thesis. Instead of using an \textit{ab initio} method where usually only small systems are simulated and which are based just upon the original laws of quantum mechanics without further approximations, another class of methods can be used. Those methods are called empirical methods and most commonly used in molecular dynamics simulations. Here approximations about the potential are made and an analytic form is used. This way one can omit the costly computation of the electronic Schrödinger equation (2.9) and approximate it by the chosen potential.

2.1.2 The Derivation of Classical Molecular Dynamics

With the original goal being the derivation of classical molecular dynamics with Newton’s laws of motion from quantum mechanical principles, there is only one last step left. As just stated before instead of an approximative method for the potential energy of the nuclei an analytical potential \( V(x) \) can be chosen. The time-independent Schrödinger equation can then be written as:
2.1. MATHEMATICAL AND PHYSICAL BACKGROUND

\[
\left( \frac{|p|^2}{2m} + V(x) \right) \Psi(x) = E \Psi(x)
\]  

(2.11)

With the nuclei wave function \( \Psi(x) \), the ground state energy \( E \) and the Hamiltonian \( H(p,x) = \left( \frac{|p|^2}{2m} + V(x) \right) \). It consists of a kinetic part \( \frac{|p|^2}{2m} \) and a potential part \( V(x) \). From now on a system of \( i = 1, ..., N \) particles is considered, each with mass \( m_i \). Their positions are denoted by \( x_i \) and their momenta by \( p_i \), both of \( \mathbb{R}^3 \).

The evolution in time of the particles can now be represented by the Hamiltonian system for all particles:

\[
\begin{align*}
\dot{x}_i &= \nabla_{p_i} H(p,x) \\
\dot{p}_i &= -\nabla_{x_i} H(p,x)
\end{align*}
\]  

(2.12)

Here \( \nabla_{p_i} H(p,x) \) and \( \nabla_{x_i} H(p,x) \) describe the partial derivative of the Hamiltonian w.r.t. \( p_i \) and \( x_i \). With \( p_i = m_i v_i \) this Hamiltonian system can directly be transformed into Newton’s equations of motion:

\[
\begin{align*}
\dot{x}_i &= v_i \\
m_i \ddot{x}_i &= F_i
\end{align*}
\]  

(2.13)

Here the second equation describes Newton’s second law of motion with the force term \( F_i = -\nabla_{x_i} V(x) \) with the analytical potential \( V(x) \).

2.1.3 Analytical Potentials

To describe the interaction between particles it is important to choose an appropriate analytical potential. There exist a variety of different potentials, all with different properties and application fields. It is also possible to use different potentials at once:

\[
V(x_1, ..., x_N) = \frac{1}{2} \sum_{i,j} U(x_i, x_j) + \frac{1}{6} \sum_{i,j,k} U(x_i, x_j, x_k) + ... + \frac{1}{N!} \sum_{i...n} U(x_i, ..., x_n)
\]  

(2.14)

Depending on the application and the current problem appropriate potentials have to be chosen. In this thesis only so called pair potentials are considered, which means potentials, that only depend on the distance \( r_{i,j} = |x_i - x_j| \) between two particles. Common pair potentials are:

- **Gravitational Potential**: Describes the gravitational field between two particles with masses \( m_i \) and \( m_j \) and the gravitational constant \( G \).

  \[
  U(r_{i,j}) = -G \frac{m_i m_j}{r_{i,j}}
  \]
CHAPTER 2. THEORETICAL BACKGROUND

- **Coulomb Potential**: Describes the potential between two bodies with electrical charges \( q_1 \) and \( q_2 \).
  \[
  U(r_{i,j}) = \frac{1}{4\pi \epsilon_0} \frac{q_i q_j}{r_{i,j}}
  \]

- **Van der Waals Potential**: Describes the weak intermolecular or inter-atomic forces involving dipole interactions.
  \[
  U(r_{i,j}) = -a \left( \frac{1}{r_{i,j}} \right)^6
  \]

- **Lennard Jones Potential**: Describes the interaction of neutral, non-linked atoms or molecules. It is a combination of the van der Waals potential and dipole-dipole interactions.
  \[
  U(r_{i,j}) = \alpha \epsilon \left[ \left( \frac{\sigma}{r_{i,j}} \right)^{12} - \left( \frac{\sigma}{r_{i,j}} \right)^6 \right]
  \]

- **Morse Potential**: Used for modeling the vibrational spectra of molecules and solids.
  \[
  U(r_{i,j}) = D(1 - e^{-a(r_{i,j} - r_0)})^2
  \]

The different potentials can furthermore be subdivided into short- and long-range potentials. The gravitational and the Coulomb potential are examples for long-range potentials because they decay very slowly with increasing distance of the two bodies and the van der Waals, Lennard-Jones and Morse potential are examples for short range potentials decaying a lot more quickly with the distance of the bodies.

In this thesis the Lennard-Jones potential will be used exclusively, but it could easily be replaced by any other (short-ranged) pair potential.

### 2.1.4 The Lennard-Jones Potential

The Lennard-Jones potential (LJ) is usually used in its \((12,6)\)-form, where the \( m = 6 \) represents the van der Waals contribution. The parameter \( n = 12 \) is chosen for simplicity, but can be changed to simulate different physical and chemical properties. The potential is good for modeling inert gases and their liquid and solid states. For two particles \( i \) and \( j \) the potential has the form:

\[
U_{i,j}^{LJ} = 4\epsilon \left[ \left( \frac{\sigma}{r_{i,j}} \right)^{12} - \left( \frac{\sigma}{r_{i,j}} \right)^6 \right]
\]  

(2.15)

It is parametrized by \( \sigma \) and \( \epsilon \). The former specifying the zero crossing of the potential, which means that for particles with distance \( \sigma \) the potential is zero. And the later specifying the maximal depth of the potential, i.e. the maximum attraction.
2.2. ALGORITHMIC BACKGROUND

between two particles. This maximum attraction lies at \( r_m = 2^{\frac{1}{6}} \sigma \) and is \(-\epsilon\). For two particles with distance \( r_{i,j} < \sigma \) the potential will result in very high repulsive force and for distances \( r_{i,j} > \sigma \) the resulting force will be attractive. A graphical representation of this potential can be seen in fig. 2.1.

![Figure 2.1: The Lennard-Jones-(12,6)-potential](image)

The Lennard-Jones potential belongs to the class of short-range potentials which means that it decays very quickly to zero with increasing distance. Often distances \( r_{i,j} > 2.5\sigma \) are assumed to be zero, which allows a huge saving in computations. This fact will be exploited later. From the Hamiltonian system from equation 2.13 can be seen that we are not exactly interested in the potential \( U_{i,j}^{LJ} \) itself but in its partial derivative, i.e. the resulting force \( F_{i,j} \). The derivation of the LJ potential with respect to the coordinate \( x_i \) then leads to:

\[
F_{i,j} = -\nabla_{x_i} V(x) = 24\epsilon \left( \frac{1}{r_{i,j}^{12}} \right) \cdot \left( \frac{\sigma}{r_{i,j}} \right)^6 \left( 1 - 2 \cdot \left( \frac{\sigma}{r_{i,j}} \right)^6 \right) d_{i,j} \tag{2.16}
\]

With \( d_{i,j} = x_j - x_i \) being the direction vector between the two particles.

2.2 Algorithmic Background

After describing the mathematical and physical background needed for a molecular dynamics simulation it is now possible to formulate an algorithm processing Newton’s equation of motion.

2.2.1 Time-Discretization

To evolve a differential equation in time on a computer a suitable numerical integration method is needed. The most elementary numerical integrator is the Euler
method. But both, the implicit and explicit Euler scheme lack one fundamental property, we are interested in physical simulations: Conservation of energy. To fulfill this property we need a so called symplectic integrator. The Velocity Störmer-Verlet integrator is such a method and can for our case be derived in the following way: From the first part of equation 2.13 we get:

\[
\dot{x}^n_i = \frac{\partial x^n_i}{\partial t} \approx \frac{x^{n+1}_i - x^{n-1}_i}{2\Delta t} = v^n_i
\]  

(2.17)

This is the central finite difference of the first time derivative of \(x^n_i\). From now on the index \(i = 1, ..., N\) represents the particle index and the index \(n\) corresponds to the current time step \(n\) with \(t_{n+1} = t_n + \Delta t\), where \(\Delta t\) is a sufficiently small step size.

The second part of equation 2.13 can be reformulated via the second order finite difference of the time derivative of \(x^n_i\) to:

\[
\ddot{x}^n_i = \frac{\partial^2 x^n_i}{\partial t^2} \approx \frac{x^{n+1}_i - 2x^n_i + x^{n-1}_i}{\Delta t^2} = \frac{1}{m_i} F^n_i
\]

(2.18)

\[
x^{n+1}_i = 2x^n_i - x^{n-1}_i + \frac{\Delta t^2}{m_i} F^n_i
\]

(2.19)

To involve only contributions from the current time step \(n\) to compute the value \(x^{n+1}_i\) of the new time step, we insert equation 2.17 into equation 2.19, this leads to:

\[
x^{n+1}_i = x^n_i + \Delta t v^n_i + \frac{\Delta t^2}{2m_i} F^n_i
\]

(2.20)

Since we need to update the velocities too, we have to derive a suitable update formula. In a first step we insert equation 2.19 into equation 2.17:

\[
v^n_i = \frac{x^n_i - x^{n-1}_i}{\Delta t} + \frac{F^n_i \Delta t}{2m_i}
\]

(2.21)

Now we add the corresponding equation for \(v^{n+1}_i\):

\[
v^{n+1}_i + v^n_i = \frac{x^{n+1}_i - x^{n-1}_i}{\Delta t} + \frac{(F^{n+1}_i + F^n_i) \Delta t}{2m_i}
\]

(2.22)

To get rid of the contribution from time step \(n - 1\) we insert equation 2.17:

\[
v^{n+1}_i = v^n_i + \frac{(F^{n+1}_i + F^n_i) \Delta t}{2m_i}
\]

(2.23)

Equations 2.20 and 2.23 together result in the Velocity Störmer Verlet method. This integrator has all desired properties:

- **Simplicity**: The integration method should give a good trade off between computational effort and memory requirements and accuracy. It is always possible to construct a higher order method by involving more terms of the Taylor expansion, like in the Runge-Kutta methods but this always comes together with an increased computational effort.
2.2. ALGORITHMIC BACKGROUND

- **Time-reversibility**: An integrator should lead to the same results if it was applied “forward” or “backward” in time. This can be checked by running a simulation in the usual manner (“forward”). The result of the last time step will then be used as initial values for a second simulation with just the signs of the velocities changed in the other direction. At the end of the second simulation the result should only differ in small amount from the initial values of the first simulation.

- **Conservation of energy**: This means that the total energy of the system \( E = E_{\text{kin}} + E_{\text{pot}} \) is approximately constant over time. Even though both, the kinetic and the potential energy, may change over time, their sum has to stay approximately constant.

The Basic Algorithm

After the mathematical derivation of the used integration method it is almost possible to write down a first simple algorithm for a molecular dynamics simulation. The updates of position and velocity vectors have already been covered, what is still missing is the force computation. As it was already said in section 2.1.3, to compute the forces between particles the Lennard-Jones potential is used. The force between every particle pair can be computed by the spatial derivative of the potential as stated in equation 2.16, \( F_{i,j} = -\nabla_x V(x_i, x_j) \). To compute the complete force acting on one particle \( i \) all the influences of all the other particles have to be taken into account, resulting in the following formula:

\[
F_i = \sum_{j=1, i \neq j}^{N} F_{i,j} = 24 \epsilon \sum_{j=1, i \neq j}^{N} \left( \frac{1}{r_{i,j}^2} \right) \cdot \left( \frac{\sigma}{r_{i,j}} \right)^6 \left( 1 - 2 \cdot \left( \frac{\sigma}{r_{i,j}} \right)^6 \right) d_{i,j} \tag{2.24}
\]

Here \( d_{i,j} = x_j - x_i \) is the direction vector between the two particles \( i \) and \( j \), \( r_{i,j} \) is the length of the direction vector of the two particles and \( \sigma \) and \( \epsilon \) are parameters of the Lennard-Jones potential.

But this is not optimal. The computation of the forces of all particles will result in a computational complexity of \( \mathcal{O}(N^2) \), since every particle iterates over all the other particles. For large numbers of particles this will become very time consuming. From Newton’s third law of motion we know that \( F_{i,j} = -F_{j,i} \). This can be exploited to halve the computational complexity of equation 2.24 and results in algorithm 1 to compute the forces of all particles. From equation 2.23 we know that we need two force vectors, one for the current time step \( n \) and one for the new time step \( n + 1 \). For this reason, before the forces for the new time step are computed, the old values have to be saved to a temporary vector \( F^{\text{old}} \) that corresponds to \( F^n \) in equation 2.24 an the updated forces correspond to \( F^{n+1} \).
CHAPTER 2. THEORETICAL BACKGROUND

Algorithm 1 Computation of the Lennard-Jones potential

\[ F^{\text{old}} = F \]

for \( i = 1, \ldots, N \) do
  for \( j = i + 1, \ldots, N \) do
    \( d_{i,j} = x_j - x_i \)
    \( r_{i,j} = \text{length}(d_{i,j}) \)
    \( t = \left( \frac{\sigma}{r_{i,j}} \right)^6 \)
    \( F_i = F_i + 24\epsilon \left( \frac{1}{r_{i,j}^6} \right) \cdot t \cdot (1 - 2t) \cdot d_{i,j} \)
    \( F_j = F_j - 24\epsilon \left( \frac{1}{r_{i,j}^6} \right) \cdot t \cdot (1 - 2t) \cdot d_{i,j} \)
  end for
end for

Now it is possible to assemble a first basic algorithm for a molecular dynamics simulation. The Velocity Störmer Verlet method is used for time-integration and the Lennard-Jones potential to compute the forces between particles:

Algorithm 2 Velocity Störmer Verlet algorithm

setup particles with initial conditions

for \( t_n = t_0, \ldots, t_{\text{end}} \) do
  \( x_i^{n+1} = x_i^n + \Delta t v_i^n + \frac{\Delta t^2}{2m} F_i^n \) // update positions
  \( F^{\text{old}} = F^n \) // backup old forces
  \( F^{n+1} = 0 \)
  for \( i = 1, \ldots, N \) do // update forces
    for \( j = i + 1, \ldots, N \) do
      \( d_{i,j} = x_j^{n+1} - x_i^{n+1} \)
      \( r_{i,j} = ||d_{i,j}|| \)
      \( F_{i}^{n+1} = F_{i}^{n+1} + 24\epsilon \left( \frac{1}{r_{i,j}^6} \right) \cdot \left( \frac{\sigma}{r_{i,j}} \right)^6 \cdot (1 - 2 \left( \frac{\sigma}{r_{i,j}} \right)^6) \cdot d_{i,j} \)
      \( F_{j}^{n+1} = F_{j}^{n+1} - 24\epsilon \left( \frac{1}{r_{i,j}^6} \right) \cdot \left( \frac{\sigma}{r_{i,j}} \right)^6 \cdot (1 - 2 \left( \frac{\sigma}{r_{i,j}} \right)^6) \cdot d_{i,j} \)
    end for
  end for
  \( v_i^{n+1} = v_i^n + \frac{\Delta t}{2m} (F_{i}^{n+1} + F_{i}^{\text{old}}) \) // update velocities
end for

The first step is initializing the particle data. Here it is only necessary to set positions and velocities to appropriate initial values, section 2.2.2 will describe this in more detail. Afterwards the time integration will be started for the desired amount of time steps \( t_n \). After the positions, forces and velocities are updated in each iteration, it is possible to write the current state of the system to disk, if visualization is desired, and statistics, like the kinetic and potential energy or temperature, can be computed.
2.2. ALGORITHMIC BACKGROUND

2.2.2 Initial Conditions

Initial particle positions and velocities have to be provided at the start of the simulation. One way to provide such initial conditions is to use the state of a previous simulation at a specific time step, but this is a rather trivial case. The second way is to create a new particle setup from scratch. The concepts of the setup of initial particle positions and velocities will now be described.

Particle Positions

The particle positions in gases underlie no special rules or restrictions, as they do in crystals or other solids. So the initial positions of our setup could be chosen randomly. But this would probably yield two undesired problems, one physical and one technical. The first would be particles possibly being placed too close, resulting in unphysical, extreme initial forces resulting from the used potential making the integration unstable. The second problem is rather a computer science aspect. With a large number of particles the generation of three random numbers, for all the three spatial coordinates, per particle is extremely costly and would take a huge amount of time, just to setup the system. To avoid those two problems some special properties from the potential can be used. To avoid extreme forces right at the start of the simulation very small particle distance should be preempted. The Lennard-Jones potential, as it is used in this thesis, results in a very high force for distances smaller then $\sigma$, every distance larger than this level results in a relatively reasonable force. So this is a constraint one would keep in mind. The second problem can be circumvented by using a regular particle setup, even though this is not particularly physical. But random initial velocities and a few initial time integration steps would

![Figure 2.2: Setup of a face-centered cubic system.](image)
result in a nice setup of particle positions. One commonly used regular setup is the so called face-centered cubic (fcc) system. This system is built up by using unit cubes with edge length \(a\). Every cube contains four particles. One at the origin of the cube and one at the center of each of the adjacent faces of the origin. This setup is demonstrated in figure 2.2a. A special property of this setup is, that the distance of all particles is chosen in such a way that the total force of each particle sums up to zero, since in every direction and distance there is always another particle at the exactly opposite direction but same distance.

For the complete setup, the whole computation domain is decomposed into such fcc cells. Resulting in a more or less closely packed system, depending on the choice of the edge length \(a\), as it can be seen in figure 2.2b.

**Particle Velocities**

The most common approach to model particle speeds in gases is to use the Maxwell-Boltzmann distribution. It is basically a zero-mean normal distribution with variance \(\frac{k_B T}{m}\) where \(k_B\) is the Boltzmann constant, \(T\) the temperature and \(m\) the particle mass. For the velocity vector \(v = (v_x, v_y, v_z)\) of a particle the probability density function is:

\[
p_v(v_x, v_y, v_z) = \left(\frac{m}{2\pi k_B T}\right)^{\frac{3}{2}} e^{-\frac{m(v_x^2 + v_y^2 + v_z^2)}{2k_B T}}
\]

(2.25)

Figure 2.3 shows the Maxwell-Boltzmann distribution for Helium, Neon and Argon particles at room temperature.

![Figure 2.3: The Maxwell-Boltzmann distribution for the noble gases Helium, Neon and Argon at room temperature (≈ 25°C)](image-url)
2.2. ALGORITHMIC BACKGROUND

This formula is due to the fact that the distribution of the velocity of the particles (and thus their kinetic energy) directly causes the temperature of the material. For every particle in the initial setup a unique velocity vector is sampled from equation 2.25.

To set up an initial system of particles those two steps are the only ones that need to be taken. The initial forces can be computed by the previously described algorithm, to allow the start of the time discretization.

2.2.3 NVE- and NVT-Ensembles

As it is typical for statistical physics a microcanonical ensemble is used to analyze an isolated thermodynamic system. This ensemble is also called *NVE-ensemble*, which describes a system with fixed number of particles $N$, fixed volume $V$ and constant energy $E$ of the total system. The first two points are easy to implement by using a constant computation domain with reflecting and/or periodic boundary conditions where no existing particle is “destroyed” or a new one “created” during the simulation. To keep the total energy of the system constant a symplectic integrator has to be used.

Another commonly used microcanonical ensemble is the *NVT-ensemble*. Here the number of particles and the volume are handled in the same way as in the *NVE-ensemble* but as a third point the temperature $T$ is fixed. To achieve this a thermostat is used. A thermostat is a means to control the temperature of the system of particles. The temperature is defined as proportional to the mean kinetic energy of all the particles $T = \frac{2}{3k_B} \bar{E}_{\text{kin}}$. Thus changing the temperature effectively means changing the kinetic energy of particles, which of course corresponds to changing their velocities. Hence a thermostat can be implemented as a multiplicative constant for all particle velocities in each time step, which might of course change over time:

$$\vec{v}_{\text{new}} = c_t \vec{v}_{\text{old}}$$

Here $c_t$ is the thermostat multiplier at time $t$. Various approaches are available in the literature how to construct this multiplier, the most basic one being $c_t = \sqrt{\frac{T_0}{T}}$, where $T_0$ is the target temperature and $T$ is the current temperature. This thermostat, however, shows a quite drastic interference into the simulation. Since one usually is interested in a smoother change of temperature, a different formula for $c_t$ was used:

$$c_t = \sqrt{1 - \frac{\Delta t}{\tau} \left(1 - \frac{T_0}{T}\right)}$$

Here $\Delta t$ is the simulation time step and $\tau$ is a constant which allows to adjust how aggressive the thermostat will enforce the change of temperature. In this project a value of $\tau = 0.1$ turned out to be a good value.

The implementation of this thermostat is quite straightforward: It can be implemented by expanding the kernel for the computation of the new velocities by a calculation of the kinetic energies first in order to know the current temperature and then adding the thermostat factor $c_t$ into the calculation of the new velocities.
CHAPTER 2. THEORETICAL BACKGROUND

2.2.4 The Linked-Cell Method

If we have a look at algorithm 2 and compute the complexity of the components, both the velocity and position updates lead to \( O(N) \) and the force update still leads to \( O(N^2) \). Even though we already approximately halved the complexity of the force computation, it would be desirable to reduce it further. If we look at figure 2.1, we see that at \( r = 2.5\sigma \) the gradient of the potential and therefore the force almost reaches zero. Hence we can exploit this fact and ignore all particle contributions with \( r_{i,j} > r_{\text{cut}} = 2.5\sigma \). This distance \( r_{\text{cut}} \) is called the cut-off radius and the potential at this point is:

\[
V(r_{\text{cut}}) = 4\epsilon \left[ \left( \frac{1}{2.5} \right)^{12} - \left( \frac{1}{2.5} \right)^6 \right] \approx -0.0163\epsilon \quad (2.26)
\]

All neglected particles will hence lead to an error < 2% and we can now approximate the Lennard-Jones force term as:

\[
F_{i,j} = \begin{cases} 
24\epsilon \left( \frac{1}{r_{i,j}} \right) \cdot \left( \frac{\sigma}{r_{i,j}} \right)^6 \cdot \left( 1 - 2 \left( \frac{\sigma}{r_{i,j}} \right)^6 \right) \cdot d_{i,j} & \text{if } r_{i,j} \leq 2.5\sigma \\
0 & \text{if } r_{i,j} > 2.5\sigma 
\end{cases} \quad (2.27)
\]

We now only need to compute the forces of \( N_{\text{neighbor}} \) particles within the cut-off radius. Especially for large domains and numbers of particles the complexity of the force computation reduces from \( O(N^2) \) to \( O(N \times N_{\text{neighbors}}) \) which is \( \sim O(N) \) with only a small constant of proportion, since the number of particles with \( r < r_{\text{cut}} \) becomes very small with rising domain sizes. But we still need to iterate over all particles and check if they are above or below the cut-off radius. To make this process more efficient we make use of the cut-off radius. We subdivide our computation domain \( \Omega \) into cells of edge length \( \geq r_{\text{cut}} \). Here it is not important to have a cell length of exactly \( r_{\text{cut}} \) but some domains may require the cell length to be slightly larger to avoid half cells at the end of the domain.
2.2. ALGORITHMIC BACKGROUND

Figure 2.4: The figure shows a domain decomposed into $3 \times 3$ “linked cells” and the cut-off radius of a particle (black dot). All particles outside of this radius are assumed to have no noticeable force-influence on the particle.

Figure 2.4 visualizes this domain subdivision with all the particles being scattered over the whole domain. If we now consider one of the particles and draw an imaginary circle around it with radius $r_{\text{cut}}$, we see that no particle outside the directly neighboring cells and the particles cell itself is inside the cutoff radius and hence does not contribute to the force of the particle. It is now possible for every particle to just iterate over all the 27 neighboring cells (in the three dimensional case) and the containing particles and therefore use the full savings introduced by the cut-off radius. It is now important to find a good data structure to handle this linked cell algorithm. Every cell of the domain is now assigned an index $k \in \mathbb{N}_{\text{cells}}$ with $N_{\text{cells}} = N_{\text{cells}}^x \times N_{\text{cells}}^y \times N_{\text{cells}}^z$. Furthermore an array of size $27 \cdot N_{\text{cells}}$ is created storing all the 27 neighbors of each cell. This is possible due to the fact that we will not change the cell size or number of cells during the simulation, so the neighbors of each cell constant during the whole time. This cellNeighbor array can now be used to efficiently iterate over all the neighboring cells of each particle:

$$F_i = \sum_{k \in \text{cellNeighbors}} \sum_{j \in k} F_{i,j}$$ \hspace{1cm} (2.28)

In a second step we need to improve iteration over all particles within a cell. Therefore two arrays are created, one “cell list” (cells) of size $N_{\text{cells}}$ and a “particle list” (particleIDs) with one entry for every particle. Both lists are initialized by $-1$, which represents the state “empty” or “end of list”. Now for every particle the associated cell index is computed. Then the value of the cell list at the computed
Figure 2.5: The figure shows the two arrays of the linked cell list data structure, \textit{cells} and \textit{particleIDs}. When all particles in a cell should be traversed, the value in the \textit{cells} list at the required cell position is read, containing the ID of the first particle in the cell. The next particle in the cell can always be found at the position of the current particle in the \textit{particleIDs} array. If a $-1$ is encountered the end of this cell list is reached.

index is copied to the index of the particle in the \textit{particle list} and afterwards the index of the current particle is written in the cell list at the computed index. This way a very simple but effective linked list is created, that can be quickly traversed. This procedure is presented in algorithm 3. During the force computation every particle now simply computes the cell index it belongs to, fetches the first particle index from the cell list, evaluates the force and looks up the next particle in the particle list at the previously fetched particle index. This is repeated until a fetched particle index (note that this can already happen in the cell list) has the value $-1$. Then the force computation for this cell is completed and the next cell can be processed. Since the particle positions will be updated every time step and particles will move from one cell to another quite often, this cell list has to be recomputed in every time step. Figure 2.5 visualizes such a list.

The algorithm for the force computation using the just described cell lists can be seen in algorithm 4.

\begin{algorithm}[H]
\caption{Inserting the particles into cells}
\begin{algorithmic}
\State set \textit{cells} and \textit{particleIDs} to $-1$
\For {every particle $i = 1, ..., N$}
\State compute cell index $k$ of particle $i$
\State $\text{particleIDs}[i] = \text{cells}[k]$
\State $\text{cells}[k] = i$
\EndFor
\end{algorithmic}
\end{algorithm}
2.2. ALGORITHMIC BACKGROUND

Algorithm 4 Computation of the Lennard-Jones potential using cell lists

\[ F^{\text{old}} = F \]

\begin{verbatim}
for every particle \( i = 1, \ldots, N \) do
    compute cell index of particle \( i \)
    for all 27 neighbor cells \( k \) do
        particle index \( j = \text{cellList}[k] \)
        while \( j \neq -1 \) do
            compute force between particles \( i \) and \( j \)
            \( j = \text{particleList}[j] \)
        end while
    end for
end for
\end{verbatim}

2.2.5 Boundary Conditions

So far we have ignored the behavior at the boundaries of our domain \( \Omega \). For a molecular dynamics simulation with a constant number of particles there are basically two different ways to handle the domain boundaries, reflecting or periodic boundary conditions (BC).

Reflecting Boundary Conditions: Every particle that would leave the computation domain \( \Omega \) bounces back from the boundary as if the domain was surrounded by imaginary walls. To implement this kind of BC it is necessary to introduce an imaginary particle during the force computation, that is placed on the other side of the reflecting wall with the same distance to the wall as the original particle. This way a force will be introduced to the particle that will reflect it away from the wall. The reflection process with the imaginary particle is shown in figure 2.6.

![Reflecting Boundary Conditions](image)

Figure 2.6: Reflecting boundary conditions: The black particle is close to the border and will be reflected by using an imaginary particle (dashed) to create a repelling force, the reflected particle will then be at the position of the gray particle.
Periodic Boundary Conditions: If a particle leaves the computation domain it will reenter the domain on the exactly opposite side of the domain. Here no additional work during the force computation has to be performed. But when the particle positions are updated for every spatial coordinate a modulo operation has to be performed with the length of the domain. This way the particles will automatically be moved to the right position. Figure 2.7 illustrates such a periodic boundary.

Figure 2.7: Periodic boundary conditions: A particle (black) leaving the domain at the bottom, will reenter the domain at the top (gray), with the same coordinates but the one it has left the domain shifted by the domain length.

Figure 2.8: Force Computation with periodic boundaries. All cells at the boundary of the domain have now virtual neighbors, that have to be taken into account, when updating the force.
2.2. ALGORITHMIC BACKGROUND

Additionally to moving particles leaving the domain another aspect has to be covered. Since the domain is now periodic, cells at the boundary have a neighbor cell on the other side. This results in a simulation of a virtually larger domain. For this process no particles or imaginary cells have to be introduced, only during the computation of the cell neighbors the corresponding neighbors on the other side of the domain have to be chosen.

In this thesis periodic BC were implemented. The reason is that with the Lennard-Jones potential usually gases are simulated and we would like to simulate as large domains and numbers of particles as possible. With periodic BC the domain is virtually extended in every direction resulting in the simulated domain embedded into a larger domain.

2.2.6 Reduced Units

In molecular dynamics simulations physical quantities are often expressed dimensionless or in reduced units [6]. The idea is to place all numerical values in the same order of magnitude, usually around unity, in order to reduce possible errors introduced by floating-point arithmetic. Furthermore (computational) errors are easier to spot, since very small or large values indicate error-prone computations. There are four basic units: Length $\sigma$, energy $\epsilon$, mass $M$ and the Boltzmann constant $k_B$.

<table>
<thead>
<tr>
<th>Unit</th>
<th>Real World Value</th>
<th>Reduced Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length $\sigma$</td>
<td>$3.4505 , \text{Å}$</td>
<td>1</td>
</tr>
<tr>
<td>Energy $\epsilon$</td>
<td>$119.8 , \text{J}$</td>
<td>1</td>
</tr>
<tr>
<td>Mass $M$</td>
<td>$39.948 , \text{u} \times 6.634 \times 10^{-23} , \text{kg}$</td>
<td>1</td>
</tr>
<tr>
<td>Boltzmann Constant $k_B$</td>
<td>$1.381 \times 10^{-23} , \text{J} / \text{K}$</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 2.1: Reduced Units for Argon gas

All other variables like velocity or force can be expressed via dimensionless variables derived from the basic units.
CHAPTER 2. THEORETICAL BACKGROUND

<table>
<thead>
<tr>
<th>Meaning</th>
<th>Symbol</th>
<th>Dimensionless Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distance</td>
<td>$r$</td>
<td>$\hat{r} = r/\sigma$</td>
</tr>
<tr>
<td>Energy</td>
<td>$E$</td>
<td>$\hat{E} = E/\epsilon$</td>
</tr>
<tr>
<td>Temperature</td>
<td>$T$</td>
<td>$\hat{T} = k_B T/\epsilon$</td>
</tr>
<tr>
<td>Density</td>
<td>$\rho$</td>
<td>$\hat{\rho} = \rho \sigma^3$</td>
</tr>
<tr>
<td>Time</td>
<td>$\tau$</td>
<td>$\hat{\tau} = \tau/ (\sigma \sqrt{M/\epsilon})$</td>
</tr>
<tr>
<td>Velocity</td>
<td>$v$</td>
<td>$\hat{v} = v/\sqrt{\epsilon/M}$</td>
</tr>
<tr>
<td>Force</td>
<td>$F$</td>
<td>$\hat{F} = F\sigma/\epsilon$</td>
</tr>
<tr>
<td>Pressure</td>
<td>$p$</td>
<td>$\hat{p} = F\sigma/\epsilon$</td>
</tr>
</tbody>
</table>

Table 2.2: Dimensionless Variables

After converting the values of the initial conditions to reduced units, the used integration scheme and force model can be simplified to the following equations:

\[
\begin{align*}
\hat{x}_{i}^{n+1} &= \hat{x}_{i}^{n} + \hat{v}_{i}^{n} \Delta \hat{t} + \frac{\Delta \hat{t}^2}{2} \hat{F}_{i,j}^{n} \\
\hat{v}_{i}^{n+1} &= \hat{v}_{i}^{n} + \frac{\Delta \hat{t}}{2} (\hat{F}_{i,j}^{n} + \hat{F}_{i,j}^{n+1}) \\
\hat{F}_{i,j}^{n+1} &= \frac{24}{\hat{r}_{ij}} \left( \frac{1}{\hat{r}_{ij}} \right)^6 \left[ 1 - 2 \left( \frac{1}{\hat{r}_{ij}} \right)^6 \right] \hat{x}_{ij}
\end{align*}
\]

The above equations will be used throughout the remaining thesis and all computations are carried out with reduced units. If real world values are needed for some reason, table 2.2 can be used to convert the reduced quantities.

2.3 Computer Architecture Background

Obtaining fast and efficient algorithms is not only a theoretical and mathematical process. Depending on the available resources of the used computers it might be necessary to alter an algorithm slightly to gain better performance. To understand those steps it is necessary to know about the features of the used hardware. In this section a general description of computer architecture, that is needed to understand the later optimizations of the described algorithm, will be given. As it was already mentioned, processing large numbers of particles is the goal of this thesis. To do so it is necessary to parallelize the algorithm on different levels. Section 2.3.1
2.3. COMPUTER ARCHITECTURE BACKGROUND

describes CPU-level parallelizations, section 2.3.2 describes the integration of GPUs and finally section 2.3.3 will introduce the features of cluster-computers.

2.3.1 CPUs

The central processing unit (CPU) is the basic computational unit in a computer performing arithmetic operations. It performs all logical and arithmetical work and also processes the input and output operations to and from the system. The arithmetic operations are performed within the arithmetic logical unit (ALU) and the floating-point unit (FPU), the former performing integer arithmetic and the later floating-point arithmetic. To increase the performance of a CPU the vendors, mainly Intel and AMD, have followed two strategies, one is increasing the number of transistors to make more ALUs and FPUs available and the other is increasing the clock speed to perform more operations in the same time. Both strategies worked well for some time but an increasing transistor density and a higher energy consumption through higher clock speeds resulted in thermal problems. To encounter this problem, for some years now, the clock speed of CPUs has no longer been increased. Instead multiple cores within one processor were introduced, allowing the parallel execution of programs. Today processors usually have four to eight cores. To increase the performance of a program different ways are possible from a CPU perspective. There are two main types, that will now be presented.

Data Level Parallelism  Modern CPUs are no longer restricted to serial execution but do now have the possibility to execute the same instruction (addition, subtraction, multiplication, ...) on more than one data element at once. This is called vectorization, since a whole vector of input data is processed at once. Usually this kind of parallelization is called data level parallelism or single instruction, multiple data (SIMD). The most common way to utilize this feature on modern CPUs is the Streaming SIMD Extensions (SSE) instruction set [3]. It allows vectorized loading and storing of data as well as all kinds of arithmetic operations. The main benefit of this approach is the increase of (floating-point) operations per second (FLOPs), which is commonly used as a performance measure across all kinds of computation devices and can be computed by the formula:

\[
\text{FLOPs} = \text{cores} \cdot \text{clock speed} \cdot \frac{\text{operations}}{\text{clock cycle}}
\]  

A modern CPU with four cores, a clock speed of 2.5 GHz and four operations per clock cycle hence yields a theoretical maximum performance of 40 GFLOPs.

Thread Level Parallelism  Another way to increase the performance of a system is the utilization of multiple threads. Every thread then operates on a different subset of the data. This is especially efficient if the CPU has more than one core and/or allows hyper-threading. The different threads are then executed in parallel which in theory allows a reduction of the execution time to \(\frac{1}{n}\)-th of the original,
if $n$ threads are used. This kind of parallelization is hence called thread level parallelism or multiple instructions, multiple data (MIMD). A simple way to use this MIMD approach is to use the OpenMP API [13], where different parts of a program, especially loops, can easily be parallelized.

2.3.2 GPUs

Although CPUs gained a huge performance increase over the time, for mathematical and physical simulations their performance still leaves a lot to be desired. This is mostly due to the fact that the fraction of FPUs is relatively low. This does not make CPUs bad computing devices, but for simulations, where most computations are done in floating-point arithmetic, other architectures might be more beneficial. At this point graphics processing units (GPU) come in to play. As their name already reveals they were originally designed to process graphical content. This has the background that there are a lot and similar floating-point operations to be performed on every data element. Additional to the high computational intensity, processing graphical content in modern applications also requires large amounts of memory. Those two facts make GPUs the perfect computing devices for most physical simulations. A schematic view of a GPU can be seen in figure 2.9. There is a large memory, usually larger than 2 GByte on modern GPUs, and a relatively large number, around 16 on modern GPUs, of so called multiprocessors (MP). Every MP has his own cache to store intermediate data and several processing units (PU), usually 32 per MP. All PUs of one MP execute the same instruction in one cycle,

![Figure 2.9: Schematic view of a GPU. The connection to the host is via the PCIe Bus. The memory modern GPUs is in the range of some GByte. Every multiprocessor contains its own cache for data that can be accessed by all the compute units of this multiprocessor.](image-url)

28
2.3. COMPUTER ARCHITECTURE BACKGROUND

which means that a maximum performance can only be reached if the operations performed on the data are uniform. Here different application fields of CPUs and GPUs can be seen. If a program has a lot of branches and different operations on every data element, CPUs are to be preferred, but if the program has a mainly uniform execution, GPUs might be the better choice.

**General-purpose computing on GPUs** Under the name *General-Purpose Computing on GPUs* (GPGPU) all programs and simulations utilizing GPUs as their main source of computations are summarized. The main fields of application of GPGPU can be summarized in two categories, *data intensive* computations, such as simulation, modeling, data mining and image- and video-processing, and *computational intensive* applications such as numerical and iterative methods and computations in the financial sector. To utilize GPUs for computations there are two main frameworks:

- **CUDA**: The GPU vendor Nvidia started the CUDA framework in 2007 [4]. The goal was to make the hardware available to a wider field of applications. It provides both, a high level API, to control the GPU from within the original program, and a low level API for the computation kernels on the GPUs themselves. CUDA is now available in version 5.0 and probably the most commonly used framework for GPGPU, but it is restricted to the hardware from Nvidia.

- **OpenCL**: The *Open Computing Language* (OpenCL) [11] allows heterogeneous computations on several devices, such as CPUs, GPUs and other accelerators. It follows a similar approach to Nvidia’s CUDA framework. A high-level API controls the execution of kernels on selected devices. The kernels themselves are written in a C dialect and compiled at runtime for the specific devices. OpenCL is developed by a number of companies under the lead of the Khronos Corporation.

In this thesis the CUDA framework has been used, simply to the fact that the used systems only consisted of Nvidia GPUs and due to better optimizations of the compilers would result in a better performance.

2.3.3 Cluster Computers

To increase the available computing power and allow larger computations *cluster computers* are used. They can range from a collection of a few desktop-computers, often named Beowulf clusters [5], to large dedicated computer systems with a total processor count in the range of hundred thousands to millions. The fastest supercomputer at the moment, as it is listed in the “Top500” list from November 2012 [15], is the “Titan” at Oak Ridge National Laboratory with 560640 cores and a performance of 17.6 PFLOPs but also a power consumption of 8.2 MWatt. This last number makes clear why the trend, when building new supercomputers, is not
just higher performance but also and more and more important energy efficiency, namely a better FLOPs/Watt ratio. Lowering the operational costs of cluster computers is an important factor, not only for such large systems but also for smaller company and university systems in the range of hundreds or thousands of cores. A second important factor that should not be neglected to lower the operational costs is the usage and implementation of more efficient programs on clusters. This is also the motivation of this thesis. Increasing the efficiency of the simulation by using all of the available resources.

Fast Network Interconnects  The usage of a multitude of machines introduces also a new problem. Data has to be synchronized and transferred between the compute nodes. Most systems do no longer employ regular Gigabit-Ethernet but use special network hardware to minimize the transmission latency and maximize the bandwidth. A widely used network interconnect (NIC) is Infiniband. It allows to lower the network latency from $\sim 100 \mu s$ down to about $10 \mu s$ with a bandwidth of some GB/s. To reach this latency and bandwidth special NIC hardware is used together with optical fibers and an own transmission protocol. Another nice feature of Infiniband is the possibility to skip parts of the operating system while transferring data from one node to another and copying the data of one application directly in the memory of another application.

The Message Passing Interface  When a program is parallelized by a shared memory approach such as the previously described OpenMP, no special communication between the different threads is needed since they all share the same memory. But cluster computers are usually distributed memory systems, making it necessary to communicate between the different nodes and processes. The Message Passing Interface (MPI) is such a commonly used communication API, which introduces a common interface on top of the used NIC hardware, making it easier to run a program on different hardware. As the name already reveals, MPI uses messages to communicate between applications, which are usually the same for all processes, which is also called single program, multiple data (SPMD) approach. Every process is assigned a unique id and all, or also just subsets, of the processes are controlled in communicators. In addition to the point to point messages between two processes, there are also collective operations like all-to-all-, broadcast- or gather-operations. There exist different MPI implementations depending on the field of application and used hardware.
Part II

Algorithms
Chapter 3

Implementation

In the following chapter the specific implementation and optimization details will be discussed. Since this thesis is aiming at an optimal hybrid-acceleration performance, the first step is optimizing the CPU implementation. Starting with the basic algorithm (section 3.1) vectorization (section 3.3) will be applied, followed by two different parallelizations techniques: multi-threading (section 3.4.1) and multi-node utilization (section 3.4.2). Afterwards section 3.5 describes the basic GPU implementation, where all the work is done on the GPU without any help from the CPU. Finally the last two sections 3.6.1 and 3.6.2 will then present the two hybrid-acceleration techniques.

3.1 Basic Algorithm

The basic implementation follows algorithm 2 but in addition to the theoretical algorithm a few further steps are necessary, mainly to keep the data structures up to date and for communication between processes. The extended basic algorithm then reads as the following:

<table>
<thead>
<tr>
<th>Algorithm 5 Basic Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setup data structures, MPI-topologies, ...</td>
</tr>
<tr>
<td>read or generate initial particle data</td>
</tr>
<tr>
<td>for $t_0, ..., t_{\text{end}}$ do</td>
</tr>
<tr>
<td>Update positions</td>
</tr>
<tr>
<td>Update cells</td>
</tr>
<tr>
<td>Exchange particles with neighboring processes</td>
</tr>
<tr>
<td>Update forces</td>
</tr>
<tr>
<td>Update velocities</td>
</tr>
<tr>
<td>Compute Statistics if necessary</td>
</tr>
<tr>
<td>Write particles to disk if necessary</td>
</tr>
<tr>
<td>end for</td>
</tr>
</tbody>
</table>
CHAPTER 3. IMPLEMENTATION

The first step is the setup of all basic data structures, this includes reading the simulation parameters, allocating memory for the particle data structures, obtaining and computing topology information according to the used numbers of processes and decomposition of the domain. The update formulas of position, velocity and force vectors have already been described in sections 2.2.1 and 2.2.4 and are implemented without any further modifications at this point. The communication and particle exchange between different processes follows the ideas presented later in section 3.4.2 and can be put aside for the moment. The computation of simulation statistics is optional and not necessary to the simulation itself. It includes the computation of potential and kinetic energy and the current temperature if a thermostat is used, as mentioned in section 2.2.3. Together with writing the current simulation status to disk, the statistics computation is only used to check the correct behavior of the implementation and can be turned off or only performed after a desired number of iterations.

3.1.1 Algorithmic Complexity and Operation Count

To get a measure for how an algorithm will or should perform it is necessary to determine the complexity of the algorithm. In chapter 2 the complexities of the different parts of the algorithm were already mentioned and after some modifications to the force update formula, the complexity of all parts of the algorithm are of \( O(N) \), even though the force update formula has a potentially large constant in front of it. Additional to the complexity itself, it is interesting to know how many arithmetic and load/store operations are necessary. Table 3.1 gives an overview over all parts of the algorithm including complexity, operation count and load/store-operations.

<table>
<thead>
<tr>
<th>Function</th>
<th>Complexity</th>
<th>Operations per Particle</th>
<th>Load/Store per Particle</th>
</tr>
</thead>
<tbody>
<tr>
<td>position</td>
<td>( O(N) )</td>
<td>( 3 \cdot 7 )</td>
<td>( 4/1 )</td>
</tr>
<tr>
<td>velocity</td>
<td>( O(N) )</td>
<td>( 3 \cdot 6 )</td>
<td>( 4/1 )</td>
</tr>
<tr>
<td>forces</td>
<td>( O(N) )</td>
<td>( 23 \cdot \frac{1}{2} N_{nbr} )</td>
<td>( N + \frac{1}{2} N_{nbr}/\frac{1}{2} N_{nbr} )</td>
</tr>
<tr>
<td>cells</td>
<td>( O(N) )</td>
<td>( 3 \cdot 2 + 5 )</td>
<td>( 3/2 )</td>
</tr>
</tbody>
</table>

Table 3.1: Complexity and operation count of the different parts of the algorithm

From this table it is possible to determine the overall work, in terms of operations, that has to be done in one iteration:

\[
\text{Work} \sim \left( 21 + 18 + 11 + 23 \cdot \frac{1}{2} N_{nbr} \right) \cdot N = \left( 50 + \frac{23}{2} \cdot N_{nbr} \right) \cdot N \quad (3.1)
\]

The last missing part is to determine the average order of neighbor particles \( N_{nbr} \). The number of neighbors of each particle correlates directly to the particle
3.2. DATA STRUCTURES

density. By choosing different values for the face-centered cell length $a$ (see section 2.2.2) the density can directly be manipulated. To obtain the average number of neighbors, the number of potential computations in each iteration can be counted and divided by the number of particles. This was done for two different setups, the first is a closely packed setup where the fcc length was chosen to be $\sqrt{2}\sigma$ resulting in an initial distance of all particles of $\sigma$. The second setup is less dense with an fcc length of $2\sigma$ resulting in a particle distance of $2\sqrt{2}\sigma$. With a linked cell length of $2.5\sigma$ in each direction the first setup results in $\rho_1 = \frac{21.43}{\text{particles/cell}}$ and the second setup results in $\rho_2 = \frac{7.8125}{\text{particles/cell}}$. Table 3.2 shows the results for this measurement with $N_{nbr}$ for each setup, which corresponds directly to the number of counted potentials per particle and the resulting work per iteration:

<table>
<thead>
<tr>
<th>Density</th>
<th>$N_{nbr}$</th>
<th>Work</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\rho_1 = 21.4375$</td>
<td>83.861</td>
<td>$\approx 1000 \cdot N$</td>
</tr>
<tr>
<td>$\rho_2 = 7.8125$</td>
<td>32.521</td>
<td>$\approx 425 \cdot N$</td>
</tr>
</tbody>
</table>

Table 3.2: Results for counting the number of potential computations per particle and the resulting work per iteration. Note: the number of potentials per particle is the total number $N_{nbr}$, not $\frac{1}{2}N_{nbr}$

From the table it can be seen that between the density and $N_{nbr}$ there is almost exactly a constant of proportionality of 4. This means it is possible to write out the previous work-equation without unknown factors:

$$\text{Work} \approx (50 + \frac{23}{2} \cdot 4 \cdot \rho) \cdot N = (50 + 46 \cdot \rho) \cdot N$$

(3.2)

Note that this formula is only valid for an equal density distribution over the whole domain, as it assumed for this implementation.

3.2 Data Structures

Up until now the used data structures were not mentioned yet, except for the linked cell lists. The basic approach for molecular dynamics simulations is to store all the data of one particle in one “particle-object” and to use doubly linked lists or arrays of objects to connect the particles. Since during this thesis also GPUs will be used a decision against the usual molecular dynamics data structures was made. The data for each of the quantities (masses, positions, velocities, forces and previous forces) is stored in a separate array. The anticipated benefit of this data structure is that for the position and velocity updates the data can be processed in a cell-independent fashion, since every particle only depends on its own data here. Hence the advantages of coalesced memory accesses better caching can be exploited. The
cell information is only needed for the force update. Here the already introduced linked-cell lists come into play allowing cell-based access to particles. The force update has a relatively random memory access pattern for both, the list and array based data structures, resulting in no drawbacks when arrays are used. Hence the advantages of better memory access and coordinate based access are combined in this data structure. Deleting and inserting new particles in a efficient way is a little bit more tricky and it is only needed when particles can leave or enter the domain, like described in section 3.4.2. The following operations are needed:

- **Removing particles**: When a particle needs to be removed the only thing that happens is that its ID is saved to a separate removedParticles array. Resulting in almost no costs to remove a particle.

- **Inserting particles**: When a particle needs to be inserted, the last element of the removedParticles array will be extracted and the new particle data will be copied to the corresponding places in the arrays. If the removedParticles array is empty the new particle will be appended at the end.

- **Clean Up**: Since the removed particles are not actually removed yet it is important to clean up the data structures from time to time. During this process the removedParticles array will be walked through and particles from the end of the data structure will be copied to the places where the removed particles where previously at.

Since the particle density is assumed to be constant over the whole domain the numbers of particles to be removed and inserted in every step are almost the same, meaning that the “clean up” operation does not move a large amount of particles, resulting in only very little effort.

### 3.2.1 Memory Consumption

It is also interesting to know about the memory consumption of a simulation. The main factor here is the data structure of the particles. For $N$ particles the mass array has a length $N$ and the positions, velocities, force and oldForce arrays are of size $3 \cdot N$. If single precision floating point (FP) data types are used, the consumed memory sums up to:

$$\left(1 + 3 + 3 + 3 + 3\right) \cdot N \cdot 4 \text{ Byte} = 52 \cdot N \text{ Byte}$$

For a million particles this results in $\approx 50$ MByte. The linked cell lists were neglected here, since they are of a much smaller order compared to the number of particles. Furthermore additional space for ghost particles is needed, if the simulation is parallelized (see section 3.4.2). This additional space can be neglected for large domains, since the fraction of ghost particles gets smaller with rising size of the domain.
3.3 Vectorization

To improve single core performance the number of processed particles per time has to be increased, which can be done by reducing the number of arithmetic operations per particle. To obtain this goal SSE instructions were utilized [3]. SSE instructions allow to process 128 Bit of data in on SSE register at once (see figure 3.1). For single precision floating point values of 32 Bit length this results in four operations that can be processed at once. For double precision floating point numbers, two operations can be processed at once. This already shows a problem for the implementation, since most of the particle data, except for the mass, has three components per particle. Hence either some operations have to be discarded, or multiple particles have to be processed at once. Depending on the use case one of those two approaches has to be applied. In the following the vectorization will be explained for single precision floating point, but it can be adapted for double precision almost analogously.

Figure 3.1: SSE allows to process 4 single precision floating point values at once reducing the operations by \( \frac{1}{4} \)

An additional point that has to be taken into account when using SSE instructions with single precision floating points is the loading and storing of data. For both operations to variants exist, one for aligned and one for unaligned data. If the data that is supposed to be loaded or stored is aligned, the operations can be processed much faster than for unaligned data. Alignment in this case means, that the starting address of the processed data is a multiple of 16 Byte. Otherwise the data is unaligned. Unaligned loads and stores are slower by a factor of \( \sim 2 \), but can be used for all data, independent if they are aligned or not. This means, aligned loads should be preferred, if possible. But if an alignment of 16 Bytes can not be ensured, unaligned loads have to be taken. For the particles quantities like the force this means that only every second particle will be aligned, since every particle consists of three values, each 32 Bit long.
Vectorized Position and Velocity Update

Vectorizing the position and velocity updates is relatively simple. Since all of the arithmetic operations only require data from the same particle and even only from the same (x,y,z)-component, they are easily vectorized. To take full advantage of the vectorization always four particles are processed at once, resulting in three SSE registers, as it can be seen in figure 3.2.

Algorithm 6 shows how the position update is vectorized. It is split into two parts. This is necessary, since always 4 particles (or 12 vector components) are processed at once. If the number of particles is not a multiple of 4 the last remaining particles have to be processed in a regular fashion, without vectorization. First the value of the time step is loaded into all four parts of one SSE register. Then the loop over the particles is started. The data for positions, velocities and forces of 4 particles will be loaded into 3 registers each. Another 3 registers will be loaded with the factor \( \frac{\Delta t^2}{2m} \) that is multiplied with the force later, here the order of the registers has to be taken into account, since always three consecutive mass values per particle have to be loaded (see figure 3.2). When all the required data is loaded into registers the arithmetic operations are performed, according equation 2.20. The last step is storing back the new position data.
3.3. VECTORIZATION

The vectorization of the velocity update is almost the same as the position update. Only the factor this time is $\frac{\Delta t}{2m}$ and instead of the time step the value of the thermostat, if a thermostat is to be used, will be loaded. The resulting algorithm 7 then looks like:

Algorithm 7 Vectorized Velocity Computation

\begin{verbatim}
thermostat = _mm_set1_ps()
for i = 1, ..., N - (N%4) do
    factor[1...3] = _mm_set_ps()
    velocity[1...3] = _mm_load_ps()
    force[1...3] = _mm_load_ps()
    forceOld[1...3] = _mm_load_ps()
    velocity[1...3] = _mm_mul_ps( thermostat, velocity[1...3])
    force[1...3] = _mm_add_ps( force[1...3], forceOld[1...3])
    force[1...3] = _mm_mul_ps( force[1...3], factor[1...3])
    velocity[1...3] = _mm_add_ps( velocity[1...3], force[1...3])
    _mm_store_ps(velocity[1...3])
end for
for i = N - (N%4), ..., N do
    compute velocities in a regular fashion
end for
\end{verbatim}

In both cases all load and store operations are aligned to 64 Bit, which means that unaligned loads can be avoided.

Vectorized Force Update

The vectorization of the force update is a little more tricky. The data accesses are no longer continuous but scattered over the whole array of particles. This comes from the fact that during the force update particle data from the surrounding cells is needed and that data can be distributed all over the array. This makes it necessary to only use one SSE register per particle and accept that only three of the four parts of the register will be needed afterwards. But this also results in an algorithm, that is very similar to the original algorithm 4. For every particle the position data has to be loaded by an unaligned load operation and a register for the temporary force is assigned to zero. Then the cell ID of the current particle is computed and all the particles of the neighboring cells are processed. All neighboring particles with an array index $j$ less then the particle index $i$ will be discarded, resulting in the described saving of 50% of the operations. For the remaining neighbors the position data is loaded into a register and the distance and length of the distance is computed. The computation of the latter is a little bit tricky, but by computing the dot product with shuffling the data, taking the square root and extraction of the first part of the register this can be done solely with SSE instructions, returning a single floating point value:
Algorithm 8 Length of a three dimensional vector with only SSE instructions

\[ \text{\texttt{mm_cvtxs_f32(m_m_sqrt_ss(m_m_dp_ps(distance, distance, 0x71))}}} \]

Algorithm 9 Vectorized Force Computation

for \(i = 1, \ldots, N\) do
\[ \text{forceOld = force} \]
end for

for \(i = 1, \ldots, N\) do
    Compute CellID
    position\(_i\) = \text{\texttt{mm_loadu_ps}}()
    tmpForce = \text{\texttt{mm_set_ps(0,0,0,0)}}
    for all 27 neighboring cells \(k\) do
        for all particles \(j\) in cell \(k\) do
            Skip particle if \(j \leq i\)
            position\(_j\) = \text{\texttt{mm_loadu_ps}}()
            distance = \text{\texttt{mm_sub_ps(position\(_j\), position\(_i\))}}
            length = \text{\texttt{mm_cvtxs_f32(m_m_sqrt_ss(m_m_dp_ps(distance, distance, 0x71))}}}
            if \(\text{length} \leq r_{\text{cut}}\) then
                \(\text{tmp1} = (1 / \text{length})^6\)
                \(\text{tmp1} = (24 / \text{length}^2) \cdot \text{tmp1} \cdot (1 - 2 \cdot \text{tmp1})\)
                \(\text{tmp2} = \text{\texttt{mm_set_ps(0, tmp1, tmp1, tmp1)}}\)
                \(\text{tmpForce} = \text{\texttt{mm_add_ps(tmpForce, tmp2)}}\)
                \(\text{force\(_j\)} = \text{\texttt{mm_loadu_ps}}()\)
                \(\text{force\(_j\)} = \text{\texttt{mm_sub_ps(force\(_j\), tmp2)}}\)
                \(\text{\texttt{mm_storeu_ps(force\(_j\))}}\)
            end if
        end for
    end for
\end{verbatim}
\[ \text{\texttt{mm_storeu_ps(force\(_i\))}} \] // store tmpForce to force\(_i\)
end for

If the distance is less than \(r_{\text{cut}}\) the force between the two particles can be computed. First the first part of the force term is computed without SSE instructions, since it is the same for all three vector components and then assigned to a SSE register, where one part is assigned to zero, due to the fact that when the value is later stored back to the array only the three components of the involved particle are changed and the fourth register component (of another particle) stays the same. Then the complete absolute force is computed by multiplying with the distance vector. This value is the added to the temporary force register and subtracted from the current force value of the neighboring particle. When all neighboring cells and
3.4. PARALLELIZATION

particles are processed, the temporary force value can be stored back to the particle array. The described vectorized force computation can be seen in algorithm 9.

Complexity and Operation Count Since the vectorization introduces a reduced number of arithmetic operations it is interesting to see the impact of this optimization on the operation count and resulting work per iteration:

<table>
<thead>
<tr>
<th>Function</th>
<th>Complexity</th>
<th>Operations per Particle</th>
<th>Load/Store per Particle</th>
</tr>
</thead>
<tbody>
<tr>
<td>position</td>
<td>$\mathcal{O}(N)$</td>
<td>$\frac{16}{4}$</td>
<td>4/1</td>
</tr>
<tr>
<td>velocity</td>
<td>$\mathcal{O}(N)$</td>
<td>$\frac{16}{4}$</td>
<td>4/1</td>
</tr>
<tr>
<td>forces</td>
<td>$\mathcal{O}(N)$</td>
<td>$13 \cdot \frac{1}{2} N_{nbr}$</td>
<td>$N + \frac{1}{2} N_{nbr} + \frac{1}{2} N_{nbr}$</td>
</tr>
<tr>
<td>cells</td>
<td>$\mathcal{O}(N)$</td>
<td>2 + 5</td>
<td>3/2</td>
</tr>
</tbody>
</table>

Table 3.3: Complexity and operation count of the different parts of the algorithm using SSE instructions

The resulting work per iteration is now reduced by a factor of two to:

$$\text{Work} \approx \left( \frac{16}{4} + \frac{16}{4} + 7 + \frac{13}{2} \cdot 4 \cdot \rho \right) \cdot N = (15 + 26 \cdot \rho) \cdot N \quad (3.3)$$

3.4 Parallelization

Up until now the simulation is only running on a single core of the processor. As described in section 2.3.1 modern CPUs ship with multiple independent cores within one processor. To exploit all the available computing power of the CPU all those cores should be used to reduce the total simulation time. This can be done via the OpenMP API and will be described in section 3.4.1. If the simulation gets more complex and the memory consumption rises, it might no longer be possible to run the simulation on one machine. Then it is necessary to split the simulation into parts and distribute it over multiple computer nodes. A way to do this is to make use of the MPI protocol, as it was already introduced in section 2.3.3. The specific MPI implementation in this thesis is presented in section 3.4.2.

3.4.1 Multi-Threading

To utilize multiple cores of one CPU a common approach is to use different threads, each running on another CPU core and processing a different task. A simple way to do this is to use the OpenMP API which allows easy control over creation and synchronization of multiple threads [13]. The principle of OpenMP is quite simple, during the execution of a program a specified number of threads can be spawned
and the following computations can be split up between those threads. Since those threads can run on different CPU cores, the computations will be performed in parallel resulting in an optimal speedup of this part of the application by \( n \), where \( n \) is the number of used threads. A common use case of this kind of parallelization is to split up loops into parts and let them be computed by different threads. In this simulation the position, force and velocity updates are such loops that can be easily parallelized by multiple threads. Figure 3.3 shows which parts of the algorithm can be parallelized by OpenMP and which parts are processed by only one process.

The parallelization of the position and velocity updates can be done without any changes to the previous code. The loops will simply be split up between the number of threads and every thread then computes the updated values of the particles assigned to it. Multi-Threaded parallelization of the force update requires some adaption. The loop over all particles will be split up just like for the position and velocity updates. But during the potential computation of two particles the previously introduced optimization using Newton's third law of motion (\( F_{i,j} = -F_{j,i} \)) results in problems. When a particle on one thread adds its force contribution to another particle it is not sure if that particle is not currently processed by another thread. This makes it necessary to either use (slow) atomic operations during this process or to revert this optimization. Which of the two possibilities performs better will be evaluated in the benchmark chapter 4.

As it can be seen from figure 3.3 some parts of the algorithm can/will not be processed in parallel. This has the reason, that those parts would require a lot of synchronization between threads, which is a very slow/costly process. An example for this is the update of the linked cell lists. Since all threads would possibly operate on the same data elements of the cell lists those operations would have to be atomic operations. Those operations need synchronization between all threads slowing down the computations by a significant factor. Hence those expansive atomic operations should be omitted if the parts of the algorithm requiring them are of low complexity.

Additionally it has to be taken into account that creating threads anew at every loop will actually slow down the simulation a little, since the thread creation
3.4. PARALLELIZATION

Process takes some time. To avoid this behavior, the threads will be spawned at the beginning of the time integration. The parallel parts will then be computed by all those threads and the serial parts will be processed only by a single master process, while the others will be in an idle state. This way the threads only have to be created once reducing the overhead to a minimum.

![Figure 3.4: NUMA setup of a system, the two CPUs are connected to their own memory. Any memory access to the memory of the other CPU has to be transferred over the Hyper Transport Link (HTL).](image)

Furthermore the system setup has to be taken into account when using threads. A widely used system configuration on clusters is to use a dual-socket setup. Meaning that every node has two CPUs to work with including memory specifically assigned to each of the CPUs. Since accessing the memory of the other CPU is not only a logic process but also a physical process, it is a relatively slow process compared to usual memory accesses. This kind of setup is called a non-uniform memory access design. Accessing the memory of the other CPU does not require additional functions, but is done implicitly over the hyper transport link between the two CPUs. Such a NUMA setup can be seen in figure 3.4. The implications for threads on this setup is, that it has to be controlled that the threads are spawned on the CPU, where the master thread resides, such that they are closest to the memory to be operated on. This can be done via OpenMP variables, but is specific to the used system and has to be adapted for each used system.

3.4.2 Utilization of Multiple Nodes

It was already mentioned a few times in this thesis, that for most simulations a large number of particles has to be processed and a rising number of particles is accompanied by a rise in computation time and memory requirement. If the simulation time, even with the just introduced multi-threaded approach, becomes too high and/or the memory consumption of the simulation reaches a level where one compute node is no longer enough to process the amount of data, the simulation has to be split up between multiple compute nodes.

When a molecular dynamics simulation is parallelized this way, there are two factors that have to be considered: The distribution of work and the data exchange between processors. In this thesis work distribution is done by decomposing the total computation domain \( \Omega \) into equally large sub-domains depending on the number of available nodes, this is described in section 3.4.2. If the domain does not reside on a single processor anymore, particles, moving from one cell to another, may have to
be transferred from one processor to another, here it is important to have a fast data exchange algorithm, such that the computation is not unnecessarily slowed down. The implemented approach is presented later in this section. More background knowledge on the used hardware can be found in section 2.3.

**Domain Decomposition**

To allow a parallel computation of a molecular dynamics simulation the work has to be distributed equally over all the compute units. Work in this case means particles. The position and velocity updates are embarrassingly parallel, meaning that the update of the quantities of one particle does not require information from other particles and/or processors. But due to the choice of the LJ potential, during the force update position informations from other particles are needed. It would not be a good idea to simply distribute the particles in a random fashion, such that every processor has the same amount of particles and thus computational work. If not all required particles for the force computation reside on one compute unit it would be necessary to communicate between processors multiple times per particle, which is simply too expensive for an efficient parallel computation. It is also not a good idea to simply synchronize the whole particle data between all nodes after every time step and let every processor compute an equal number of particles, since this is much too time consuming for large numbers of particles.

Hence an effective way to distribute the particles over the different processors

![Figure 3.5: Decomposition of a two dimensional domain. The original domain is split up into six sub-domains ordered in a 2 × 3 grid. Those sub-domains can now be distributed over different nodes.](image-url)
3.4. PARALLELIZATION

with a minimal need for data exchange has to be found. Recalling the linked
cell approach from section 2.2.4 tells us, that our domain is already (logically)
subdivided into small parts, the linked cells. Using this fact we can divide
the computation domain into equal sub-domains aligned at the previously introduced
linked cells. Our original computation domain $\Omega = l_x \times l_y \times l_z$ is now subdivided into
$n$ sub-domains of size $\Omega_{\text{sub}} = l_{\text{sub}}^x \times l_{\text{sub}}^y \times l_{\text{sub}}^z$, where $n$ is the number of available
processors and $l_{\text{sub}}^x$, $l_{\text{sub}}^y$ and $l_{\text{sub}}^z$ describe the length of the new sub-domain. The
processors will be aligned in a three-dimensional Cartesian grid and every processor
is assigned a unique coordinate within this grid. For example three processors would
result in a grid of size $3 \times 1 \times 1$ and 30 processors would result in a grid of size
$5 \times 3 \times 2$. Depending on the number processors $n$ different grids setups might be
possible. Such a decomposition is visualized in figure 3.5.

In this thesis an equal particle density over the whole computation domain is
assumed, meaning that over the whole domain there are approximately the same
number of particles per volume, i.e. cell. This allows a static domain decomposition
and makes a dynamic adaption of the sub-domain sizes depending on the particle
density unnecessary. But to make sure that every process has approximately the
same number of particles it might be necessary to adjust the linked cell size a little
to allow an equal number of cells over all nodes in each direction. The cells will be
at least of size $r_{\text{cut}}$ to allow the linked cell algorithm to be applied and might be
extended in each direction, following the formula:

$$l_{\text{cell},i} = \frac{l_i/n_i}{\lfloor (l_i/n_i)/r_{\text{cut}} \rfloor}$$ (3.4)

Here the index $i$ represents one of the three spatial dimensions. The length of
one cell is denoted by $l_{\text{cell},i}$, the number of processes in direction $i$ is denoted by $n_i$
and the length of the whole domain in this direction is $l_i$. With this linked cell size
every processor has the same total amount of cells, resulting in approximately the
same amount of work.

But to be able to update the particles in the cells at the boundary of one
processor, at least the positions of the particles from neighboring cells, have to be
accessible. This makes it necessary to introduce ghost particles and ghost cells.
Ghost cells describe a layer of cells around the actual sub-domain of one processor,
where the data of neighboring cells needed for the force update is stored. Since we
only need the current position of the particles in those ghost cells we do not need the
complete particle data in those ghost cells but it is sufficient to synchronize only so
called ghost particles which contain only information about the particles’ position.
Hence every processor does not only have to store the particles in its sub-domain but
extend the sub-domain by one layer of cells around the whole sub-domain, where
the ghost particle data from neighboring processors is stored.
Data Exchange

To communicate between processes the Message Passing Interface (MPI) is used. It simplifies the communication between processes to a point where only the sending and receiving process has to be known and the amount of data, the rest is done by MPI. For a deeper understanding networks and network communication is treated in more detail in section 2.3.3.

In this section the data exchange pattern used in the implementation will be treated. To make a parallel molecular dynamics simulation possible the different processes have to exchange two kinds of particles, one are the particles leaving the sub-domain of one process and entering another and the second are the ghost particles within the afore mentioned ghost cells. Since we are treadng a three-dimensional domain every process has in general 26 neighbors, six direct neighbors, with two in every dimension, and 20 diagonal neighbors. But establishing a communication to a neighbor over the network takes quite a lot of time compared to the computation. So it is desirable to reduce the number of communications to a minimum. It is possible to decrease the number of data exchanges with neighbors per time-step from 26 to only 6, namely only the direct neighbors in every direction. To allow this reduction the data exchange is split into three parts, one for each dimension. Further more we have two different types of data exchanges, one for the particles leaving the sub-domain to another process and a second for the ghost particles. The first step needs to transfer the current position, velocity and force of each particle and the second step only needs to transfer the positions, i.e. ghost particles.

Particle Exchange  In the first step the particles leaving the sub-domain will be transferred. To make sure every particle reaches its destination process within only 3 communication steps special rules have to be followed. The whole procedure is visualized in figure 3.6. In the first step communication only takes place in the z-direction. All particles that are found in the ghost cells at the top and bottom of the sub-domain after position update will now be transfered to the direct neighbor. To do so all the ghost cells at the top and bottom will be traversed and the particles there will be stored in one array for exchange to the “top” neighbor and one array to the “bottom” neighbor. At this point it is important to note, that the particles will only be copied, not deleted, why will be explained later. After all particles are collected they can be sent to the respective neighbor. Here two MPI_Send operations are needed, one to notify the neighbor about the amount of data that will be sent and one for the actual particle data. After the data is sent in both z-directions, every process waits until it receives the data from his neighbor and stores the received particles into the data structures. Note that in this step not only particles in the “inner” part of the sub-domain are received, but also particles in the ghost cells. Those are the particles, that will later reach the diagonal neighbors. After a process has sent, received and stored all data in z-direction, it can move on to the exchange in y-direction. The exchange here is somewhat similar to the one
3.4. PARALLELIZATION

Figure 3.6: Particle data exchange pattern. The first exchange includes all ghost cells in this direction. The second exchange leaves out the already exchanged cells at the borders of the first exchange direction. The last exchange step only includes particles from the “inner” ghost cells that were not yet exchanged. (Figure taken from [7])

in z-direction. With the only difference that not all particles in the ghost cells in “north” and “south” are traversed. The rows at the top and bottom will be left out, since they were already treated in the previous step. After those, a little smaller, ghost cell slices are traversed and the particles are sent received and stored, the last step can be started. Here the communication will take place only in x-direction. The procedure is the same as before and the ghost cells in the “east” and “west” direction are traversed. This time the rows at the very top, bottom, north and south will be left out. After all the data in x-direction is exchanged, all particles that have left their sub-domain have been sent to their new sub-domains and processes.

Ghost Particle Exchange After the first data exchange part, it is now necessary to exchange the ghost particles and to fill the ghost cells, to make the force update possible. The followed procedure is the same as for the “regular” particles with two differences. The first difference is, that the exchange will be carried out in the reverse direction. Starting with the x-direction exchange and ending with the z-direction exchange. The second difference is, that now instead of the ghost cells the first inner layer of cells in each direction is processed. Since we only need to exchange ghost particles the amount of data per particle will be smaller, only the positions are now needed.

The exchange of ghost particles is shown in figure 3.7. The first step is the x-direction exchange to the neighbors in the “east” and “west”. The particles to be sent come strictly from the interior of the sub-domain slices. The second step is then the y-direction exchange. Here the particles are gathered from the first and last interior slices in “north” and “south” direction, but including the first and last ghost cell rows in x-direction. After the y-direction exchange is finished, the exchange in
Figure 3.7: Ghost particle data exchange pattern. The procedure follows the same rules as the regular particle exchange, only in reverse direction. That way even the particles at the corner ghost cells will reach their destination process. (Figure taken from [7])

z-direction can be started. Now all the particles from the interior slices in “top” and “bottom” direction including the first and last rows in x- and y-direction. By this procedure all particles that would have needed to be sent to a diagonal neighbor will reach this neighbor without additional transfers.

**Periodic Boundaries**  Up until now only data exchanges within the domain have been described. To treat the periodic boundary conditions, communications with virtual neighbors, that are actually at the other end of our Cartesian grid have to be performed. This is not done in an extra step but within the original data exchange, since every process has calculated its neighbors before the simulation starts. The only difference for those boundary exchanges is, that the particle positions in the current exchange direction have to be adjusted. This is due to the fact that the periodic boundary conditions simulate a virtually larger domain, but our computation domain is actually bounded. So every particle and ghost particle that is transferred over the boundaries has to be adjusted by a factor of $\pm l_i$ where $i$ is the exchange direction. The sign for transfers from the start to the end of the domain is positive and in the other direction it is negative. This has to be done for both, particles and ghost particles.

**3.4.3 Combining OpenMP and MPI**

When the algorithm is parallelized by both, OpenMP and MPI, it is necessary to make a trade-off between the number of MPI processes on one compute node and the number of OpenMP threads per process. It is desired to utilize all available cores of the CPU and one would naturally use one MPI process per node and use as many threads per process as cores are available. Using more MPI processes or more
OpenMP threads both have their advantages and drawbacks, listed in table 3.4. To make a decision here only from theoretical deliberations is very difficult and hence a good trade-off between the two factors can be easily obtained from running multiple simulations with different thread/process setups and evaluating their benchmark results. This will be done in chapter 4.

<table>
<thead>
<tr>
<th>MPI parallelization</th>
<th>OpenMP multi-threading</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ the whole algorithm is parallelized</td>
<td>+ data exchange can be omitted</td>
</tr>
<tr>
<td>− possibly expensive data exchange</td>
<td>− only force, position and velocity updates can be parallelized</td>
</tr>
</tbody>
</table>

Table 3.4: Comparison between the advantages and drawbacks between the parallelization using MPI and OpenMP

### 3.5 Basic GPU Algorithm

Up until now the whole computation is done on the CPU. But as it was already mentioned in section 2.3.2 modern computers and clusters also include GPUs with an enormous theoretical computing power. The difference between the peak computing power of a current CPU (up to 100 GFLOPs) and a modern GPU (some TFLOPs) is quite remarkable. But it has to be taken into account that the architecture of the two is very different and require special approaches to get all the performance out of the device. Performance optimization of the CPU algorithm has already been discussed in the previous sections. In this section the basic GPU algorithm will be explained. The three computationally expensive parts, the velocity, position and force updates, will be computed on the GPU and the CPU will handle the data exchange with other MPI processes.

The difference of the GPU programming approach compared to CPU programming approach comes from the architecture. A GPU is subdivided into a number of multiprocessors, usually 16 or 32, each consisting of some compute units, which will perform the actual computations. All compute units within one multiprocessor will perform the same operations at the same time. This is both the advantage and the pitfall of GPU programming. The computations performed on a GPU have to be almost the same for all compute units, to make sure that only a minimum number of compute units will be idle and wait for the others during execution. The kernels (programs or functions) that are executed on a GPU are written in a way that they describe what computations one compute unit would perform. Communication between compute units is possible by grouping them in so called blocks or via global atomic operations. When such a kernel is called from the host program the number
of blocks and threads per block has to be specified, where one thread corresponds to one compute unit.

When using the CUDA platform it is possible to use predefined vector data types, in our case with three components. This allows to simplify the code a lot since the operations on the three vector elements of one particle are usually the same. The used data structures on the GPU are exactly the same as described in section 3.2, except for the use of vector data types. Additionally to predefined data types, CUDA also comes with some common mathematical functions for those data types, for example to compute the length of a vector or the dot product between two vectors.

Data Transfers Since the GPU can not directly access the main memory of the system, all data needed during computation has to be transferred to the GPU before executing a kernel and if the data from the GPU is needed on the host it has to be transferred back again. This is often a bottleneck, especially when large amounts of data have to be transferred. There exist two versions of memory copy operations between host and device, synchronous and asynchronous. The first is blocking, meaning that the host has to wait until all data is transferred, where the latter allows the host to continue right after enqueuing the copy operation to the device. The second operation has the advantage that the time while the copy operation takes place can be used for other computations but the data to be copied can, or rather should, only be modified after the copy operation has finished. But in the case of this simulation only synchronous copy operations can be used, since there are no computations to be done other than the ones on the data that is copied. This creates an additional slow down when using GPUs in the code, which means that the computations on GPUs have to be faster to a degree that they make up for this additional time, spent to transfer data to and from the device.

Position and Velocity Updates Executing the position and velocity updates on the GPU is quite simple. Every vector element can be processed by one thread and no communication between different compute units is needed. The algorithms 10 and 11 show the kernels for both updates. The only thing that is new is the check if the current thread is within the range of particles. This is necessary to make sure that the computations only take place within the allocated memory and comes from the way kernels on the GPU are called from the CPU, where a fixed number of threads per block have to be specified. The total number of threads may then be a little larger than the actually needed number.

Force Update Just like the velocity and position updates, the force update can be ported to the GPU almost without alterations. The only modification is that Newtons’ third law of motion will not be used. This means every particle will have to compute all forces acting on it by itself. This has the reason that, just like in the multi-threaded CPU implementation, global atomic operations would be needed.
3.5. BASIC GPU ALGORITHM

Algorithm 10 GPU Position Kernel
compute \( id \) of current thread
if \( id < N \) then
\[
\begin{align*}
\text{tmp} &= timestep \cdot \text{timestep}/(2 \cdot \text{mass}[id]) \\
\text{position}[id] + &= \text{timestep} \cdot \text{velocity}[id] + \text{force}[id] \cdot \text{tmp}
\end{align*}
\]
end if

Algorithm 11 GPU Velocity Kernel
compute \( id \) of current thread
if \( id < N \) then
\[
\begin{align*}
\text{tmp} &= timestep/(2 \cdot \text{mass}[id]) \\
\text{velocity}[id] &= \text{thermostat} \cdot \text{vel}[id] + (\text{force}[id] + \text{forceOld}[id]) \cdot \text{tmp}
\end{align*}
\]
end if

The slowdown from stalled multiprocessors due to atomic operations is much more significant than the factor of \( \frac{1}{2} \) we would save. So the additional work is acceptable.

Algorithm 12 GPU Force Computation
compute \( id \) of current thread
if \( id < N \) then
\[
\begin{align*}
\text{pos} &= \text{position}[id] \\
\text{tmpForce} &= (0, 0, 0) \\
\text{compute cellID} \\
\text{for all 27 neighboring cells } k \text{ do} \\
\text{otherParticle} &= \text{first particle of this cell} \\
\text{while } \text{otherParticle} \neq -1 \text{ do} \\
\text{distance} &= \text{position}[\text{otherParticle}] - \text{pos} \\
\text{length} &= \text{length}(\text{distance}) \\
\text{if } \text{length} \leq r_{\text{cut}} \text{ then} \\
\text{tmp} &= (1/\text{length})^6 \\
\text{tmpForce} &= \text{tmpForce} + (1/(\text{len} \cdot \text{len})) \cdot \text{tmp} \cdot (1 - 2 \cdot \text{tmp}) \cdot \text{distance}
\end{align*}
\]
end if
\text{otherParticle} = \text{next particle in cell}
end while
end for
\text{forceOld}[id] = \text{force}[id]
\text{force}[id] = 24 \cdot \text{tmpForce}
end if
3.6 Hybrid Implementations

Now that both, the optimized CPU and basic GPU implementation have been discussed, it is time to present the hybrid implementations. The goal of hybrid implementations is to exploit every available compute power, be it on the CPU or on the GPU. Two different algorithms have been implemented. The first algorithm is the so called *Stopwatch*-algorithm, distributing the computations between CPU and GPU by measuring the elapsed time of both and shifting particles if necessary. The second hybrid algorithm is quite different from the first. Its goal is to minimize expensive data transfers and decomposes the domain of each MPI process into an inner part solely for the GPU and an outer part for the CPU. The performance of both algorithms will later be compared to each other and the basic CPU/GPU implementations in chapter 4.

### 3.6.1 Stopwatch Algorithm

The main idea of the *Stopwatch* algorithm can be explained very quickly. Computations are performed on the CPU and GPU in parallel. After a specified number of iterations the execution time of the GPU force update and the CPU force update will be measured. From those two measurements a factor $q$ will be computed specifying the number of particles being processed on the GPU in the next iterations, the remainder will be computed on the CPU, see figure 3.8.

$$0 \quad \frac{q \cdot N}{N_{GPU}} \quad \frac{N}{N_{CPU}}$$

Figure 3.8: Symbolic distribution of the particles between CPU and GPU, the ratio $q$ will be recomputed at a regular frequency

The factor $q$ is computed by normalizing the execution time of the kernels of both devices to one particle ($t_{1, CPU}$ and $t_{1, GPU}$) and then computing the ratio for particles to be computed on the GPU, such that the expected execution time of the two kernels is the same in the next iteration.

$$t_{1, CPU} = \frac{t_{CPU}}{N_{CPU}} \quad t_{1, GPU} = \frac{t_{GPU}}{N_{GPU}} \quad (3.5)$$

$$q = \frac{t_{1, CPU}}{t_{1, CPU} + t_{1, GPU}} \quad (3.6)$$

$$N_{GPU} = q \cdot N \quad N_{CPU} = N - N_{GPU} \quad (3.7)$$
3.6. HYBRID IMPLEMENTATIONS

Since particles will enter and leave the sub-domain during the simulation, the last step has to be performed in every iteration, to adapt to the new number of particles. This ratio will be used for all three kernels, position, velocity and forces. Initially the ratio will be set to $q = 0.9$, resulting in 90% of the particles being computed on the GPU and 10% on the CPU. The value of this initial ratio is simply a guess obtained from test runs of the algorithm. The recomputation frequency of the factor $q$ can be set at will, but a recomputation every 25 iterations has shown to be a good value. This algorithm was first introduced in [16] where the factor will not be taken purely from the measured times but as a mean between the previous and the new factor. Test runs of this algorithm have shown, that the factor will reach a steady value after only a few updates, making the mean computation dispensable.

The used GPU kernels are the same as described in the previous section and for the CPU kernels both vectorization and multi-threading are used to obtain the best performance. Since the computation on the GPU happens asynchronously, both computations overlap with almost no idle time for any of the two devices. Algorithm 13 describes how one iteration of the simulation is processed when the Stopwatch algorithm is applied.
CHAPTER 3. IMPLEMENTATION

Algorithm 13 StopWatch Algorithm

Update position on GPU for particles \((1, ..., N \cdot q)\)
Update position on CPU for particles \((N \cdot q, ..., N)\)

copy particle data \((1, ..., N \cdot q)\) to CPU
Exchange particles with neighboring processes

copy complete particle data to GPU
if update ratio then
    Start time measurement for GPU
end if
Enqueue force computation on GPU of particles \((1, ..., N \cdot q)\)
if update ratio then
    Stop time measurement for GPU
    Start time measurement for CPU
end if
Compute forces on CPU for particles \((N \cdot q, ..., N)\)
if update ratio then
    Stop time measurement for GPU
    \( t_{1\cdot CPU} = \text{CPU time normalized to 1 particle} \)
    \( t_{1\cdot GPU} = \text{GPU time normalized to 1 particle} \)
    \( q = t_{CPU}/(t_{CPU} + t_{GPU}) \)
end if

Update velocity on GPU for particles \((1, ..., N \cdot q)\)
Update velocity on CPU for particles \((N \cdot q, ..., N)\)

The only drawback of this algorithm is that in every iteration the whole particle data has to be synchronized between host and device. For large numbers of particles this can be quite expensive and time consuming. But on the other hand, this algorithm has the advantage that it will find the optimal ratio between CPU and GPU disregarding what kind of hardware is used. This means on a cluster with different hardware setups the optimal ratio per node will be found.

From the initial guess for the ratio it can be seen that most of the work will be done on the GPU due to a higher performance of the device. To avoid idle times of the GPU, while the CPU exchanges data with neighbors, it might be a good idea to let multiple processes submit work to the GPU. This will of course result in a shift of the ratio towards the CPU but comes with better occupancy of the GPU. In the benchmark chapter 4 the behavior of the algorithm under different processes per GPU proportions will be evaluated.
3.6. HYBRID IMPLEMENTATIONS

3.6.2 Domain Decomposition Algorithm

The second hybrid algorithm takes a whole different approach to distribute the work between CPU and GPU. In this approach the domain of each MPI process is split up into two parts, an inner part and an outer part. The inner part will be completely computed by the GPU and the outer part solely by the CPU, as it can be seen from figure 3.9. On the boundary between the two newly introduced sub-domains, ghost cells need to be introduced to allow complete force computation by each device separately. The advantage of this algorithm is, that a lot less data has to be transferred between host and device, since only the particles, that moved from one sub-domain to another, and the ghost particles, have to be transferred.

![Diagram showing domain decomposition between CPU and GPU during the hybrid decomposition algorithm](image)

Figure 3.9: Domain decomposition between CPU and GPU during the hybrid decomposition algorithm

Additional steps have to be added between position and force update of the original algorithm, to synchronize particles that moved to another sub-domain. Since no longer the whole arrays with particle data are exchanged between host and device, algorithms have to be implemented to select the particles to be exchanged. Algorithm 14 shows those additional steps, together with the information whether each step has to be performed by CPU, GPU or both.

To make the exchange with other processes possible, after the position update the particles from the GPU that moved to the CPU sub-domain, have to be exchanged. After the CPU has finished the exchange with neighboring processes, all relevant particles have to be sent to the GPU and sorted into the data structures there. In the following, the different steps that have to be added to the original algorithm will be described in the order as they are mentioned in algorithm 14.
CHAPTER 3. IMPLEMENTATION

Algorithm 14 Decomposition Algorithm

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ALL]</td>
<td>Update positions</td>
</tr>
<tr>
<td>[GPU]</td>
<td>Collect particles to be sent to CPU</td>
</tr>
<tr>
<td>[ALL]</td>
<td>Copy particles to CPU</td>
</tr>
<tr>
<td>[CPU]</td>
<td>Exchange particles with neighboring processes</td>
</tr>
<tr>
<td>[CPU]</td>
<td>Collect particles to be sent to GPU</td>
</tr>
<tr>
<td>[ALL]</td>
<td>Copy particles to GPU</td>
</tr>
<tr>
<td>[GPU]</td>
<td>Insert particles into data structure</td>
</tr>
<tr>
<td>[ALL]</td>
<td>Update cells</td>
</tr>
<tr>
<td>[ALL]</td>
<td>Update forces</td>
</tr>
<tr>
<td>[ALL]</td>
<td>Update velocities</td>
</tr>
</tbody>
</table>

Collect particles to be sent to CPU  After the positions have been updated on the GPU all particles in the inner ghost layer of the CPU and the particles that have moved from the GPU sub-domain to the CPU sub-domain have to be exchanged. This has to be done at this point, since, depending on the choice of the domain sizes, the particles that have to be exchanged by the CPU to other processes via MPI might include particles from the GPU. To do so, the current ghost particles on the GPU are discarded and for all (previous) inner particles of the GPU algorithm 15 is performed. It computes the current cell of each particle and checks if the particle has to be transferred. If so, a unique id (out) is fetched, via an atomic operation to make sure no two particles are copied to the same position of the outgoingParticles array. The data of the current particle will then be copied to the outgoingParticles array at position out. Additionally it is necessary to save the id of each particle that is removed by this kernel, for later clean up of the data structures.

After the outgoing particles are collected, they can be transferred to the host. Meanwhile the data structures on the GPU can be cleaned up. This follows the same principle as explained in section 3.2. Basically the created “holes” in the array

Algorithm 15 Collect outgoing particles to CPU

```c
id = blockIdx.x * blockDim.x + threadIdx.x;
if id < N then
    cell = ( position[id] - domainMin ) / cellLength
    cellID = cell.x + nCells.x * cell.y + nCells.x * nCells.y * cell.z
    if particle is in the inner CPU ghost layer or GPU ghost layer then
        out = atomicAdd( nRemovedParticles, 1 )
        removedParticles[out] = id
        outgoingParticles[out] = "particle[id]"
    end if
end if
```

56
3.6. HYBRID IMPLEMENTATIONS

through removing particles in algorithm 15 need to be filled, to obtain continuous arrays of particle data again. For every removed particle, that is within the new array size (newInner) a unique particle at the end is fetched, by decreasing the swapPtr, which initially points to the last inner particle of the previous iteration. Until the fetched particle is not a removed particle, this process is repeated. When a particle is found, that was not removed, it is copied to the position of the removed particle.

After kernel 16 is executed all particles on the GPU are only within the very inner part of the GPU sub-domain excluding both ghost layers of CPU and GPU. This is done to avoid the handling of special cases on the GPU. It results in a little more work, but allows a more efficient execution on the GPU.

**Update Cells on the GPU** After the particles that need to be sent to the CPU are collected and removed, the cell lists on the GPU need to be updated. Updating the cells on the GPU is basically the same as it is on the CPU, except for the fact that due to the inherent parallel execution on the GPU atomic operations are needed to make sure, the cell list is in a consistent state after the update. Algorithm 17 shows the kernel to update the cells on the GPU. The important part is the last atomic operation, making sure that the right value (the id of the particle) is written in the cells array and the current value of the cells array at the computed position cellID is copied to the particleIDs array. Those two operations form an atomic block together, because if another process writes at the same position in between, the whole data structure will be invalid. Hence the id is first written to the particleIDs array and then an atomic exchange of the that

```
Algorithm 16 Clean up data structure on the GPU

id = blockIdx.x * blockDim.x + threadIdx.x
if id < nRemovedParticles then
    target = removedParticles[id]
    newInner = N_old_inner - nRemovedParticles
    // all particles > newInner are already at the right place
    if target < newInner then
        source = -1
        do
            source = atomicSub( swapPtr, 1 )
            // look for particles not marked for deletion
            if particle source was not removed then
                copyParticle( target, source )
                source = -1
            end if
        while source > -1
        end if
    end if
end if
```
value with the current value in the `cells` array is performed.

**Algorithm 17 Update Cells Kernel**

```plaintext
id = blockIdx.x * blockDim.x + threadIdx.x
if id < N then
    cell = ( position[id] - domainMin ) / cellLength
    cellID = cell.x + nCells.x * cell.y + nCells.x * nCells.y * cell.z
    particleIDs[id] = id
    particleIDs[id] = atomicExch( &cells[cellID], particleIDs[id])
end if
```

**Collect particles to be sent to GPU** After the CPU has exchanged data with neighboring processes, relevant particles have to be sent to the GPU. This step is relatively trivial. The only thing that has to be done on the CPU is to iterate over all inner cells of the GPU and all ghost cells of the GPU and collect all particles in those cells. The particle data is only copied, since it is needed for the force computation on the CPU.

**Algorithm 18 Collect Particles to be sent to the GPU**

```plaintext
for All inner GPU cells do
    collect particles that moved to the GPU sub-domain
end for
for All GPU ghost cells do
    collect GPU ghost particles
end for
```

**Insert particles into data structure** After the CPU has collected and transferred the particles to the GPU in algorithm 18, those particles have to be inserted into the GPU data structures. Inserting the particles on the GPU is also simple and the kernel is almost the same as the `update cells` kernel. The only addition is that before the `cellID` is computed, the new particle has to be copied from the `incomingParticles` array to the GPU data structures.
### Algorithm 19 Insert Particles on the GPU

```plaintext
id = blockIdx.x * blockDim.x + threadIdx.x
if id < nIncommingParticles then
    target = N + id
    insertParticle( incommingParticles[id], target )
    cell = ( position[id] - domainMin ) / cellLength
    cellID = cell.x + nCells.x * cell.y + nCells.x * nCells.y *
              cell.z
    particleIDs[id] = id
    particleIDs[id] = atomicExch( &cells[cellID], particleIDs[id] )
end if
```

After algorithm 19 has been completed, the force, velocity and position updates can be carried out in parallel.

The advantage of this hybrid algorithm compared to the Stopwatch algorithm is, that a significantly lower number of particles has to be transferred between the devices. The data transfer cannot be overlapped with other computations and hence should be minimized as much as possible. The drawback from this approach is, the additional work that has to be performed as it can clearly be seen from the above additional kernels. Hence it is expected, that this algorithm will only outperform the Stopwatch algorithm, if the number of particles per process is large enough, that the additional computations outperform the large data transfers from the Stopwatch algorithm. Another drawback of this algorithm is, that the number of particles computed by the CPU and GPU is almost fixed, since the sizes of the sub-domains are not changed during the simulation. If one of the devices needs longer to compute its part of the particles, the other one will be idle for some time and therefore performance will be lost.
Part III

Results
Chapter 4

Benchmarks

In this chapter the performance of the different implementations and optimizations will be evaluated. The goal of this chapter is to find an optimal hybrid algorithm. First the used performance measures to evaluate the performance of the different algorithms will be outlined in section 4.1 and the used simulation setup, hardware and procedure will presented in section 4.2. Afterwards the first part of the performance evaluation will determine the single node performance, involving the optimizations from section 3.3 and 3.4.1 and a comparison of the optimized CPU implementation to the basic GPU and hybrid implementations implementation. Afterwards the scaling of the implementation on multiple nodes will be analyzed.

4.1 Performance Measures

To make qualified and consistent statements about the performance of the different algorithms and optimizations presented in this thesis, several performance measures will be applied. They may seem fairly obvious, but for the sake of completeness and to avoid ambiguities they will be presented shortly in this section.

4.1.1 Speedup and Efficiency

When a program is parallelized one usually is interested in the performance gain. To evaluate this two different measures can be applied, the speedup and the efficiency. The speedup $S(n)$ compares the sequential execution time $T(1)$ to the parallel execution time $T(n)$ when $n$ cores/processors are utilized.

$$\text{Speed Up} = \frac{\text{sequential execution time}}{\text{parallel execution time}} \Rightarrow S(n) = \frac{T(1)}{T(n)} \quad (4.1)$$

In the best case the speedup for a parallelization with $n$ cores becomes $n$, but will usually be a little lower, due to communication and sequential parts in the algorithm.
The efficiency $E(n)$ of the parallelization is essentially the speedup normalized to the number of involved cores.

\[
\text{Efficiency} = \frac{\text{Speed Up}}{\text{Number of Cores}} \quad \Rightarrow \quad E(n) = \frac{S(n)}{n} \quad (4.2)
\]

### 4.1.2 Scalability

When multiple nodes are involved in the simulation it is interesting how the performance scales with different numbers of nodes. In particular one is interested in speedup and efficiency by adding a specific number of additional nodes to solve the same problem. This is often also called *strong scaling*. It identifies how the time to solution varies, when the global problem size it fixed and the number of processes is varied and basically indicates the optimal number of processes to solve a given problem. Since usually additional exchange between the nodes is introduced, when more nodes are added, the optimal number of nodes is not necessarily the largest number of nodes.

### 4.2 Setup

All following benchmarks where executed on the same cluster and used the same simulation setup and procedure, as it will be described in the following section.

#### 4.2.1 Hardware Setup

The used cluster to evaluate the performance of the implementations was the *Zorn* cluster of the PDC center for high-performance computing of KTH. The utilized nodes consist of the following hardware components:

<table>
<thead>
<tr>
<th>Component</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>2 Intel Xeon E5620 (Nehalem/Westmere) CPUs with 4 cores and 8 threads each at 2.4 GHz</td>
</tr>
<tr>
<td>Memory</td>
<td>60 GByte RAM</td>
</tr>
<tr>
<td>GPU</td>
<td>3 NVIDIA Tesla M2090</td>
</tr>
<tr>
<td>Network</td>
<td>QDR Infiniband Interconnect</td>
</tr>
</tbody>
</table>

#### 4.2.2 Compiler and Compiler Options

The program was compiled using the Intel C++ compiler (icpc) in version 13.1. To compile the GPU code only the Kernel calls were put into separate functions and compiled with the *NVidia CUDA compiler* (nvcc) in version 5.0 and for compatibility reasons with the GNU compiler (g++) in version 4.6.3. As MPI-implementation
4.2. SETUP

\textit{mvapich2} in version 1.8 was used. All computations were carried out using single floating-point precision. The used compiler flags were:

```
-std=c++11 -Wall -Werror -Wextra -O3 -march=native -msse4 -DUSE_FLOAT
```

4.2.3 Simulation Setup

The used simulation setup was using the previously mentioned “relaxed” fcc configuration together with the physical parameters for \textit{Argon} gas, which is a very well suited gas to be simulated by the Lennard-Jones potential. The exact simulation parameters are summarized in the following table:

<table>
<thead>
<tr>
<th>parameter</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenMP threads</td>
<td>(varies)</td>
</tr>
<tr>
<td>\textit{timestep}</td>
<td>0.001</td>
</tr>
<tr>
<td>\textit{t}_{\text{end}}</td>
<td>2.0000</td>
</tr>
<tr>
<td>(\epsilon)</td>
<td>119.8</td>
</tr>
<tr>
<td>(\sigma)</td>
<td>3.4505</td>
</tr>
<tr>
<td>(T_{\text{start}})</td>
<td>2.0</td>
</tr>
<tr>
<td>FCC type</td>
<td>relaxed</td>
</tr>
<tr>
<td>boundaries</td>
<td>periodic</td>
</tr>
<tr>
<td>domain size</td>
<td>(varies)</td>
</tr>
</tbody>
</table>

Table 4.1: Parameters of the Benchmark Simulations

Except for \(\sigma\) and \(\epsilon\) all numeric values are expressed in reduced units (see section 2.2.6). Translated into real units, the total simulation time is 63 picoseconds, with a time step of 31.5 femtoseconds. The start temperature used in the Maxwell-Boltzmann distribution corresponds to 239.8 Kelvin.

The physical properties themselves are not that important for the benchmarks, the only important values are the time step length and the starting temperature, since they directly influence the stability of the simulation and need to be chosen appropriately.

To get reliable results all benchmarks were run five times and always the fastest result was used.
4.3 Single Node Performance

All hybrid algorithms described in chapter 3 are using a single GPU per MPI process, which means it is important to evaluate the single node performance first. To get a good hybrid performance, both parts, the CPU and GPU implementation, should use the best suited algorithm. Hence this benchmark section will first evaluate the CPU performance utilizing vectorization and multi-threading. Then the basic GPU algorithm will be tested together with the two hybrid implementations.

4.3.1 Basic vs. Reduced Algorithm

The first two optimizations that are applied to the CPU implementation are the vectorization, described in section 3.3, and the force update with reduced complexity, due to Newton’s third law of motion. Figure 4.1 shows the performance of the reduced and basic force update algorithm with and without SSE vectorization.

As expected, the figure shows that the vectorized version of the reduced force yields the best performance. For both force update algorithms, the vectorization yields a constant performance gain as proposed in section 3.3. The difference of both regular and the vectorized versions is not a factor of $\approx 2$ as expected for the used “relaxed” setup in this benchmark, but this is probably due to partial automatic vectorization applied by the compiler. But nevertheless, manual vectorization still yields the best performance gain.
4.3. SINGLE NODE PERFORMANCE

Furthermore, the fact that this benchmark yields almost perfectly straight lines shows that the complexity is in fact linear with the number of particles, or $\sim O(N)$.

4.3.2 Atomic Problems

When multi-threading is applied together with the reduced force update algorithm, atomic operations are necessary, to make sure only one thread at a time writes to the same particle. Atomic operations are usually slow and should be avoided as much as possible. The following benchmark shows the resulting performance of the basic force update (without atomic operations) with and without vectorization, when multi-threading with 8 OpenMP-threads is applied, together with the reduced force update (utilizing atomic operations). The reduced force algorithm was only evaluated for the non-vectorized version, since OpenMP only allows atomic operations for simple arithmetic operations, not function calls, which have to be employed when using SSE instructions. Since the results of this benchmark are quite clear and the difference is by far larger than a constant factor of $\approx 2$, as shown in the previous section, this procedure already allows to draw the necessary conclusions.

![Single Node Atomic Performance (8 threads)](image)

Figure 4.2: The figure shows performance problems when using atomic operations in the reduced force kernel compared to the regular force kernel.

As it can be seen from the figure the atomic operations in the reduced force algorithm result in a huge slow down. Hence from now on the basic force update algorithm with SSE instructions is used for the following benchmarks to obtain the best performance.

67
4.3.3 OpenMP Performance

To utilize all computing power of the CPU multi-threading via OpenMP is applied. When \( n \) threads are used, the optimal performance gain (or speedup) is \( n \), but only if the whole code can completely be executed in parallel. As described in section 3.4.1 not all parts of the simulation can be parallelized. This will result in lower than optimal speedup, but this cannot be avoided. The following benchmark evaluates the performance, speedup and efficiency of the simulation for different numbers of particles in a “relaxed” setup.

As it can be seen from figure 4.3, the execution time is reduced with an increasing number of employed threads. But a look at the speedup graph 4.4 shows that the speedup reaches a steady level for all numbers of threads relatively quick. This comes from the previously mentioned fact that some parts of the algorithm have to be executed sequentially. Also the efficiency plot 4.5 encourages this and shows a lower efficiency when more threads are utilized. For 8 threads the speedup reaches a level of \( \approx 4 \) which is almost exactly half the optimal speedup and results in an efficiency of \( \approx 0.5 \). For 16 threads the reached level is lower than \( \frac{1}{2} \) of the optimal performance, which can be explained by the fact that in the used system each node consists of 2 CPUs with 4 cores and 8 threads each. Hence, when more than 8 threads are used, data from the memory of one CPU has to be transferred to the other, resulting in additional performance slowdown. The reason, why all numbers of threads reach the maximum speedup quite quick comes from the fact that the force kernel is memory bound, which means that the ratio between arithmetic operations and load/store memory operations has a tendency to the memory.
4.3. SINGLE NODE PERFORMANCE

operations. Hence the maximum speedup and efficiency will already be reached for a small number of particles.

Despite the relatively low speedup and efficiency, using all of the 16 available threads in the used system will result in the best performance.

![Single Node OpenMP Speed Up](image)

Figure 4.4: OpenMP Speedup Comparison for different numbers of threads

![Single Node OpenMP Efficiency](image)

Figure 4.5: OpenMP Efficiency Comparison for different numbers of threads

4.3.4 GPU and Hybrid Performance

The previous sections have evaluated the CPU performance and it is now time to compare the CPU performance to the GPU performance together with the hybrid algorithms. As CPU algorithm the vectorized multi-threaded implementation with 16 threads was used. The basic GPU algorithm was used exactly as presented in section 3.5. For the hybrid algorithms, the Stopwatch algorithm was also used as explained in section 3.6.1 with starting ratio \( q = 0.9 \) and ratio updates every 25 iterations. The domain Decomposition algorithm was used as explained in section 3.6.2 with the domain decomposition scheme, that the CPU domain consisted only of one
layer of cells all around the domain (and the additional ghost cells) and the GPU domain consisted of the inner part minus one cell in each direction. The remaining simulation parameters were the same for all four algorithms, as explained earlier.

Figure 4.6 shows the performance of the four algorithms for a varying number of particles and figure 4.7 shows the according speedup, always compared to the CPU implementation. As expected, all algorithms utilizing GPUs show a worse performance than the CPU performance for a small number of particles, which comes from the fact that the data transfers to and from the GPU create an additional overhead that is not present in the CPU implementation. But this advantage of the CPU lasts only for very small numbers of particles.

The basic GPU implementation and the two hybrid implementations show a very similar behavior, when the number of particles is increased. And as it can be seen from the graphs, their curves are almost parallel only shifted by certain factors. The Stopwatch algorithm has a small advantage over the basic GPU implementation. This behavior is not surprising, since the Stopwatch algorithm is basically the same as the basic GPU implementation, with the only difference, that the CPU will support the GPU and compute a part of the particles, resulting in a reduced execution time. Hence the curve is very similar to the one of the GPU with the advantage of the CPU support. This is already a first positive point that can be noted.

The hybrid Decomposition algorithm on the other hand shows a worse performance than the basic GPU implementation. This is a little disappointing, but can

![Image of a graph showing comparison between CPU and GPU implementations](image-url)
4.3. SINGLE NODE PERFORMANCE

![GPU vs CPU - Speed Up (relaxed)](image)

Figure 4.7: Speedup comparison of CPU, GPU and hybrid implementations

be explained. Due to the large overhead, especially for small domain sizes (and hence numbers of particles), introduced by the internal data exchange, the performance is initially pretty bad even compared to the CPU version. But with an increasing size of the computation domain, the overhead gets smaller and the GPU takes over more and more work, resulting in a better performance. Compared to the CPU the Decomposition algorithm shows a good speedup, but compared to the other two implementations it still misses out some performance.

Figure 4.8 shows the evolution of the ratio used in the Stopwatch algorithm with increasing numbers of particles. As stated in section 3.6.1 for a large number

![Final Stopwatch Ratio](image)

Figure 4.8: Trend of the Stopwatch ratio with rising number of particles
of particles the ratio will level in at around 0.9, hence this value is used as an initial guess, when the simulation is started.

4.3.5 Multi-GPU Performance

The benchmarks up until now all only utilized one MPI process with one GPU. Since modern systems do not only contain a single GPU, benchmarks, utilizing all available GPUs are necessary. Since this implementation only supports a single GPU per MPI process, multiple MPI processes have to be used per node to make use of all GPUs of the system. When more than one MPI process is run on one node, the number of OpenMP threads has to be adapted, since using more threads than available cores results in a huge performance drop. The nodes used during this benchmark contain two CPUs (with 8 cores and 16 available threads) and three NVidia Tesla M2090 GPUs, which means three MPI process per node are necessary with 5 OpenMP threads each. Furthermore to test whether the GPUs are well utilized or not, another test series will be run, where 6 MPI processes are used with 2 threads each. This will assign 2 processes to each GPU. Due to the use of several MPI processes, the total computation domain will now be split up into sub-domains for every process. For three processes the domain will be split into $3 \times 1 \times 1$ sub-domains and for six processes there will be $3 \times 2 \times 1$ sub-domains. In this benchmark total domain size was fixed, resulting in 512000 particles.

Figure 4.9 shows the results for this benchmark with 1, 3 and 6 MPI processes per node. The CPU algorithm becomes a little worse with the rising number of processes which is mostly due to the fact that the total number of used threads is no longer the maximum 16 but a little less. Hence when the pure CPU algorithm is used, one MPI process per node with the maximum number of threads will yield the best performance.

Just like before the GPU and Stopwatch algorithm show a very similar behavior with a slight advantage for the hybrid algorithm. When three GPUs are used instead of just one the speedup is close to two for both instead of the optimal three. This has most likely the reason that now in each time step additional work for exchanging particles with other processes has to be performed, which is exclusively done on the CPU. The fact that the performance stays the same when using one or two threads per GPU indicates that the GPU accelerated parts perform a little better, but nevertheless due to data exchange between the different processes this advantage is lost again. Using more than one process per GPU does not give a better or worse performance, which means, that the GPU is already well utilized with one process supplying it with work.
4.4 Multi-Node Performance

The hybrid Decomposition algorithm stands out again, but shows the same behavior as before. With a rising number of processes, the domain will be decomposed into more, smaller domains. This yields additional work, since the proportion of the CPU sub-domain compared to the GPU sub-domain on each process rises with the number of processes. Hence more work has to be done to exchange particles and a larger fraction is computed on the slower CPU. The additional compute power through the extra GPUs cannot be used effectively.

4.4 Multi-Node Performance

When the number of particles is increased further more, the previous benchmarks have shown, that all of the algorithms will not be able to increase their speedup any further. In order to speed up the simulation anyway, more than one node can be used for the simulation. The computation domain will then be equally distributed over all used nodes, as described in section 3.4.2. The following section will evaluate the scaling of the different implementations. To obtain an optimal performance, the results of the previous single node performance benchmarks will be taken into account, resulting in the following setups:

- **CPU algorithm**: Every node will use one MPI process with the maximum number of OpenMP threads available.
- **GPU algorithm**: There will be three MPI processes on every node, to utilize all available GPUs.
• **Stopwatch algorithm**: Just like the GPU implementation, three processes per node will be used, together with the maximum number of OpenMP threads distributed over all three processes.

• **Decomposition algorithm**: A single MPI process will be used on every node to maximize the domain size per process and decrease the overhead introduced by the internal data exchange. The number of threads will be distributed over all processes on each node.

Please note, that the focus of this thesis was to implement hybrid algorithms for molecular dynamics simulations. Since those hybrid algorithms only apply to a single node, the scalability test does not really test the performance of the hybrid algorithms. Only for the Decomposition algorithm a change is visible, when more nodes are used, since the algorithm is directly linked to the domain of each process. Hence with a change in domain size per node, the algorithm will perform differently as seen before.

The scalability benchmark was performed on all 4 available nodes on the cluster. Each of the four algorithms was run on 1 to 4 nodes with four different numbers of particles, ranging from 250 thousand to 1.3 million particles. The results of the benchmark can be seen in figures 4.10, 4.11 and 4.12. For large numbers of particles all algorithms show a good scalability and the data exchange between different processes does not reduce the performance by a relevant factor. The pure GPU and

![Figure 4.10: Scalability performance of the four implemented algorithms for different numbers of particles.](image-url)
4.4. MULTI-NODE PERFORMANCE

Figure 4.11: Scalability speedup of each of the four implemented algorithms for different numbers of particles.

Figure 4.12: Scalability efficiency of the four implemented algorithms for different numbers of particles.
Stopwatch algorithm perform best, as previous benchmarks have already shown. Again with a small benefit for the Stopwatch algorithm. The Decomposition algorithm also shows a similar behavior as observed on a single node. Especially in the first benchmark for 256000 particles it can be seen that for four nodes, the internal management overhead becomes very large and results in a worse performance. But with larger numbers of particles and larger domains, this overhead becomes smaller and smaller and the Decomposition algorithm performs better. It still cannot compete with the Stopwatch algorithm, but with further work on this algorithm, this gap could be closed.

The speedup graphs show a almost linear behavior for large numbers of particles, which means that the data exchange does not take over with a rising number of particles. The efficiency graphs encourage this conclusion.

Additionally to the performance of the algorithms themselves, it is interesting to have a look at how the different parts of the algorithm perform. Both hybrid algorithms have very different approaches to achieve their hybrid partitioning of the computations. The Stopwatch algorithm chooses a simple approach, by just measuring execution times, which takes vanishingly little time, but accepts possible, large data transfers. The Decomposition algorithm partitions the domain and only exchanges the necessary data at the boundaries between CPU and GPU. Here the data transfer accounts a lot less to the total time, but selecting the particles to be exchanged, introduces additional work. Figure 4.13 shows how the different parts of the algorithm contribute to the total execution time. Both hybrid algorithms were run on a single node and the times of the different parts of the algorithm were measured for simulations of 250 thousand and 1.3 million particles.

![Figure 4.13: Shares of the different parts of the hybrid algorithms for different numbers of particles. “Exchange” contains all data transfers between host and device. “Other” includes all data management, treating of periodic boundaries and updating the cell lists.](image)

As the figure shows, the contributions of the position and velocity updates can be neglected and the force update accounts for most of the work in both algorithms.
4.4. MULTI-NODE PERFORMANCE

But the difference in the fraction of the data exchange is significant. Indeed the Decomposition algorithm introduces only very little data transfer of $\approx 5\%$ and on the other hand the Stopwatch algorithm spends around 20\% on those transfers between host and device. The fraction listed as “other” contains all data management, like cell updates, treating of the periodic boundaries and for the Decomposition algorithm all functions gathering the data for the transfers. The “other”-share of the Stopwatch algorithm also sums up to about 20\% as the data transfers, but contains only the necessary data structure updates for the simulation itself. For the Decomposition algorithm this share is a lot larger and accounts for 30 – 35\%. To make the Decomposition algorithm perform better, the selection of the particles to be exchange between host and device would need to be improved. But this is left for further research.
Chapter 5

Discussion and Conclusion

During this thesis effective and fast algorithms for molecular dynamics simulations for short-ranged potentials were developed. The CPU optimizations have shown, that vectorization by hand still yields a better performance gain than relying on the automatic vectorization by the compiler. Also the use of multi-threading has been shown to be very effective. To obtain the best possible performance while using multi-threading the algorithmic optimization of the reduced force computation has to be skipped, since the necessary use of atomic operations make the promised performance gain void and even results in a worse performance. The same has to be done for the GPU implementation, since there would also be atomic operations necessary.

Using GPUs instead of CPUs for the computation promises a huge performance gain at first but since a lot of data has to be transferred between the host and the device in each iteration, this performance gain is not as huge as expected. But yet a good increase in performance by a factor of 3-4 can be obtained with GPUs and with the use of multiple GPUs per node, the performance can be increased by another factor of 2-3.

But since the sole use of only CPUs or GPUs results in the other being idle and giving away possible performance. Hence algorithms are needed that utilize all available computing devices. In this thesis two different hybrid approaches have been presented. The first “Stopwatch” algorithm can be categorized as an adaptive hybrid algorithm since it automatically adapts the number of particles processed on the GPU and CPU, such that an optimal performance is obtained. The performance results of this algorithm show, that it performs similar to the pure GPU implementation but slightly faster, due to the support of the CPU. This algorithm was first proposed in [16]. For a similar setup they obtain a speedup of $\approx 20$ compared to the implementation on a single core without optimization. The algorithm implemented in this thesis confirms this speedup and is even able to reach a slightly better performance, but this is probably due to the fact that this algorithm was especially implemented for this task and the proposed algorithm was embedded in the LAMMPS framework [16]. Also the ratio between CPU and GPU is almost the
same in the proposed algorithm and this implementation. Another advantage of this algorithm is the simple implementation. When GPU and CPU implementation are given, only a time measurement has to be introduced.

The second hybrid implementation was a new kind of hybrid algorithm and can be categorized as static hybrid algorithm. It aims at the reduction of the costly data transfers between host and device by subdividing the domain of each process further into a sub-domain solely computed by the CPU and a second sub-domain for the GPU. Since in every time step only the data on the boundaries of those sub-domains have to be exchanged, the slow down due to data transfers between host and device can be reduced. But to manage those data transfers additional steps have to be introduced to the algorithm. In order to be effective, those additional steps have to be less expensive than the saving due to the reduced transfer time. As it can be seen from the performance results, this algorithm does indeed perform worse compared to the Stopwatch algorithm when a low number of particles is simulated. But with a rising domain size and number of particles the performance increases significantly. The final performance is still not competitive to the other hybrid algorithm. But with additional work this last gap could be closed and especially for very large domains a better performance could be obtained.

Additionally it should be noted, that the speedups of the different algorithms may seem smaller, compared to the speedups in some other papers. The reason for this is that instead of an “unrealistic” comparison of the best, optimized version with the worst, single core implementation on the CPU, a more “realistic” approach was chosen. To see how the implemented algorithms perform, they where always compared to the best other implementation. This way the real speedup when switching from one algorithmic optimization to another can be seen, instead of misleading speedup statements.

Summarizing all those results a conclusion of this thesis is, that molecular dynamics simulations can benefit a lot by the use of hybrid algorithms. With the Stopwatch algorithm an already very good performing solution is available. The newly developed hybrid Decomposition algorithm showed good performance for large domains, but still leaves some performance to be desired, that could be obtained by further work.
Chapter 6

Outlook

With a shift to more hybrid platforms in modern hardware, the effective use of those devices is necessary and no computing power should be neglected. As the previous chapter implies, the hybrid “Stopwatch” algorithm already performs very good and yields a significant speed up. The second “Decomposition” algorithm still leaves some space for additional performance gains. One approach to optimize this algorithm could be to make the additional data management steps more effective. Especially on the CPU those steps are very costly, since they are done in a sequential fashion. Hence an efficient parallel data management algorithm would help a lot to obtain a better performance. But not only for the Decomposition algorithm a parallel data management would be beneficial, also for the other parts, like the data exchange.

To make the Decomposition algorithm perform better on smaller domains and more nodes, the domain decomposition could be performed adaptively through out the simulation. This way better utilization of GPU and CPU could be reached.

A further performance gain could be reach by sorting the particles by cell and by one coordinate. This way the memory access pattern for both, GPU and CPU could be greatly improved. This optimization was not implemented during this thesis, due to the lack of time.
List of Figures

2.1 The Lennard-Jones-(12,6)-potential .................................................. 13
2.2 Setup of a face-centered cubic system. ................................................... 17
2.3 The Maxwell-Boltzmann distribution for the noble gases Helium, Neon and Argon at room temperature ($\approx 25^\circ C$) ............................................................... 18
2.4 The figure shows a domain decomposed into $3 \times 3$ “linked cells” and the cut-off radius of a particle (black dot). All particles outside of this radius are assumed to have no noticeable force-influence on the particle. ........... 21
2.5 The figure shows the two arrays of the linked cell list data structure, cells and particleIDs. When all particles in a cell should be traversed, the value in the cells list at the required cell position is read, containing the ID of the first particle in the cell. The next particle in the cell can always be found at the position of the current particle in the particleIDs array. If a $-1$ is encountered the end of this cell list is reached. .......................................................... 22
2.6 Reflecting boundary conditions: The black particle is close to the border and will be reflected by using an imaginary particle (dashed) to create a repelling force, the reflected particle will then be at the position of the gray particle. ................................................................. 23
2.7 Periodic boundary conditions: A particle (black) leaving the domain at the bottom, will reenter the domain at the top (gray), with the same coordinates but the one it has left the domain shifted by the domain length. ................................................................. 24
2.8 Force Computation with periodic boundaries. All cells at the boundary of the domain have now virtual neighbors, that have to be taken into account, when updating the force. .................................................. 24
2.9 Schematic view of a GPU. The connection to the host is via the PCIe Bus. The memory modern GPUs is in the range of some GByte. Every multiprocessor contains its own cache for data that can be accessed by all the compute units of this multiprocessor. ................................. 28
3.1 SSE allows to process 4 single precision floating point values at once reducing the operations by $\frac{1}{4}$ ................................................................. 37
### List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2</td>
<td>SSE allows to process 4 single precision floating point values at once reducing the operations by $\frac{1}{4}$</td>
<td>38</td>
</tr>
<tr>
<td>3.3</td>
<td>Parallelized and serial parts of the algorithm: only the position, force and velocity updates are parallelized (here with four threads), the rest is processed by the master thread.</td>
<td>42</td>
</tr>
<tr>
<td>3.4</td>
<td>NUMA setup of a system, the two CPUs are connected to their own memory. Any memory access to the memory of the other CPU has to be transferred over the Hyper Transport Link (HTL).</td>
<td>43</td>
</tr>
<tr>
<td>3.5</td>
<td>Decomposition of a two dimensional domain. The original domain is split up into six sub-domains ordered in a $2 \times 3$ grid. Those sub-domains can now be distributed over different nodes.</td>
<td>44</td>
</tr>
<tr>
<td>3.6</td>
<td>Particle data exchange pattern. The first exchange includes all ghost cells in this direction. The second exchange leaves out the already exchanged cells at the borders of the first exchange direction. The last exchange step only includes particles from the “inner” ghost cells that have not yet exchanged. (Figure taken from [7])</td>
<td>47</td>
</tr>
<tr>
<td>3.7</td>
<td>Ghost particle data exchange pattern. The procedure follows the same rules as the regular particle exchange, only in reverse direction. That way even the particles at the corner ghost cells will reach their destination process. (Figure taken from [7])</td>
<td>48</td>
</tr>
<tr>
<td>3.8</td>
<td>Symbolic distribution of the particles between CPU and GPU, the ratio $q$ will be recomputed at a regular frequency</td>
<td>52</td>
</tr>
<tr>
<td>3.9</td>
<td>Domain decomposition between CPU and GPU during the hybrid decomposition algorithm</td>
<td>55</td>
</tr>
<tr>
<td>4.1</td>
<td>Comparison of basic versus reduced algorithm with and without SSE vectorization</td>
<td>66</td>
</tr>
<tr>
<td>4.2</td>
<td>The figure shows performance problems when using atomic operations in the reduced force kernel compared to the regular force kernel</td>
<td>67</td>
</tr>
<tr>
<td>4.3</td>
<td>OpenMP Performance Comparison for different numbers of threads</td>
<td>68</td>
</tr>
<tr>
<td>4.4</td>
<td>OpenMP Speedup Comparison for different numbers of threads</td>
<td>69</td>
</tr>
<tr>
<td>4.5</td>
<td>OpenMP Efficiency Comparison for different numbers of threads</td>
<td>69</td>
</tr>
<tr>
<td>4.6</td>
<td>Comparison of the CPU implementation with the pure GPU and the two hybrid algorithms</td>
<td>70</td>
</tr>
<tr>
<td>4.7</td>
<td>Speedup comparison of CPU, GPU and hybrid implementations</td>
<td>71</td>
</tr>
<tr>
<td>4.8</td>
<td>Trend of the Stopwatch ratio with rising number of particles</td>
<td>71</td>
</tr>
<tr>
<td>4.9</td>
<td>Utilizing multiple GPUs per node</td>
<td>73</td>
</tr>
<tr>
<td>4.10</td>
<td>Scalability performance of the four implemented algorithms for different numbers of particles.</td>
<td>74</td>
</tr>
<tr>
<td>4.11</td>
<td>Scalability speedup of each of the four implemented algorithms for different numbers of particles.</td>
<td>75</td>
</tr>
<tr>
<td>4.12</td>
<td>Scalability efficiency of the four implemented algorithms for different numbers of particles.</td>
<td>75</td>
</tr>
</tbody>
</table>
4.13 Shares of the different parts of the hybrid algorithms for different numbers of particles. “Exchange” contains all data transfers between host and device. “Other” includes all data management, treating of periodic boundaries and updating the cell lists. 

76
List of Tables

2.1 Reduced Units for Argon gas .............................................. 25
2.2 Dimensionless Variables .................................................. 26

3.1 Complexity and operation count of the different parts of the algorithm . 34
3.2 Results for counting the number of potential computations per particle and the resulting work per iteration. Note: the number of potentials per particle is the total number $N_{nbr}$, not $\frac{1}{2}N_{nbr}$ ................................. 35
3.3 Complexity and operation count of the different parts of the algorithm using SSE instructions ......................................................... 41
3.4 Comparison between the advantages and drawbacks between the parallelization using MPI and OpenMP .......................... 49

4.1 Parameters of the Benchmark Simulations ................................. 65
List of Algorithms

1. Computation of the Lennard-Jones potential ........................................ 16
2. Velocity Störmer Verlet algorithm ......................................................... 16
3. Inserting the particles into cells ........................................................... 22
4. Computation of the Lennard-Jones potential using cell lists ................... 23
5. Basic Algorithm ....................................................................................... 33
6. Vectorized Position Computation ............................................................ 38
7. Vectorized Velocity Computation ............................................................. 39
8. Length of a three dimensional vector with only SSE instructions ............. 40
9. Vectorized Force Computation ................................................................. 40
10. GPU Position Kernel ............................................................................... 51
11. GPU Velocity Kernel ............................................................................... 51
12. GPU Force Computation ......................................................................... 51
13. Stopwatch Algorithm ............................................................................... 54
14. Decomposition Algorithm ....................................................................... 56
15. Collect outgoing particles to CPU .......................................................... 56
16. Clean up data structure on the GPU ...................................................... 57
17. Update Cells Kernel ................................................................................ 58
18. Collect Particles to be sent to the GPU ................................................ 58
19. Insert Particles on the GPU ................................................................... 59
Bibliography


