FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Implementation and Evaluation of Additional Particle Simulation Types with AutoPas

Raphael Penz

FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Implementation and Evaluation of Additional Particle Simulation Types with AutoPas

Implementierung und Evaluierung von zusätzlichen Partikel Simulationsarten mit AutoPas

Author:	Raphael Penz
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisor:	Fabio Alexander Gratl, M.Sc.
Date:	15.10.2021

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, 15.10.2021 Raphael Penz

Acknowledgements

I would like to express my gratitude to my supervisor, Fabio Gratl, who guided me throughout this project with constant support and valuable feedback.

Abstract

The simulation of particles plays a massive role in today's world. From the simulation of water currents in hydrodynamics to the simulation of molecules in molecular dynamics, particle-based simulation techniques find use in many areas. Each simulation needing to fulfill its own specific purposes paired with the broad field of particle simulation applications has led to the creation of a multitude of simulation approaches. This thesis implements and evaluates an additional particle simulation type for AutoPas, an open-source C++ library for delivering optimal node-level performance for particle simulations. The new particle is implemented using the discrete element method and then compared to the pre-existing implementation of a molecular dynamics particle.

Zusammenfassung

Die Simulation von Partikeln spielt eine massive Rolle in der Welt von heute. Von der Simulation von Strömungen im Bereich der Hydrodynamik zu der Simulation von Molekülen im Bereich der molekular Dynamik, Partikel basierte Simulationstechniken finden einen Nutzen in vielen Bereichen. Dass jede Simulation einen spezifischen Zweck erfüllen muss, gekuppelt mit dem weiten Feld der Partikel Simulationen, hat zu der Entwicklung einer Vielzahl von Simulationsmethoden geführt. Diese Bachelorarbeit implementiert und evaluiert einen zusätzlichen Partikel-Simulationstypen für AutoPas, eine Open-Source C++ Bibliothek für das Liefern von optimaler node-level Performance für Partikel Simulationen. Das neue Partikel wurde implementiert mithilfe der Diskrete-Elemente-Methode und danach verglichen mit der sich bereits vor befindlichen Implementation eines Partikels der molekular Dynamik.

Contents

Part I.

Introduction and Background

1. Introduction

There are various fundamental approaches for particle simulation and even more algorithms and their implementations for each of these approaches. These methods often differ significantly in their functionality and in what they are used to simulate. For example, using molecular dynamics(MD), a system can be created to observe the motion and interaction of particles at a molecular level or using the discrete element method(DEM), a system can be designed to simulate the interactions between granular particles with differing shapes.

This thesis is developed in the context of the AutoPas library. AutoPas is a node-level auto-tuned particle simulation library developed in the context of the TaLPas project. [1] The C++ library AutoPas is intended to be used as a black-box particle container to deliver optimal node-level performance for particle simulations by using auto-tuning to dynamically select the optimal combination algorithms and configuration options at runtime. Such algorithms and configuration options include particle storage algorithms, neighborhood search and interaction algorithms, parallelization strategies, and so forth.[GST$^+$19]

The goal of this thesis is to expand the AutoPas libraries example proxy applications with particles and their corresponding pairwise force calculations to widen the range of pre-implemented particle simulation methods. In this project, we implement a particle and force functor for the simulation of spherical DEM particles with an accompanying simulation. In order to provide performance data, example simulations of these particles are created and integrated into the example application "md-flexible," previously only used for the simulation of molecular dynamics. To obtain more insight into this simulation technique and how AutoPas handles it, a comparison between the new DEM implementation and the previous MD implementation is made.

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

2. Theoretical Background

In this chapter, relevant background knowledge about the discrete element method and molecular dynamics is provided, and also an introduction of key components of the AutoPas library.

2.1. Discrete Element Method

2.1.1. General

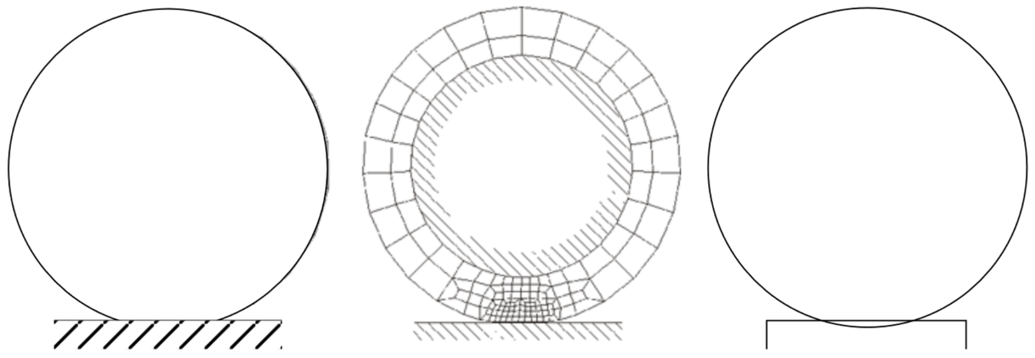


Figure 2.1.: Collision comparison between the physical situation (left, soft sphere being deformed on contact), simulation via finite element method (middle, sphere's body splitting into smaller finite pieces for force modelling) and simulation via discrete element method (right, shapes overlapping, overlap used for force modelling).
Source: [MC14]

The discrete element method is a method that models the forces between colliding particles based on a materials elasticity parameters and the overlap of rigid particle shapes. Figure 2.1 gives a short overview and comparison between the physical situation, the finite element method, and the discrete element method. As shown, in a real physical situation(left) a sphere would be slightly deformed during a collision, while in a simulation using the finite element method(middle) the force modelling requires the sphere's body to split into smaller finite pieces and using the discrete element method the contacting bodies would overlap. This overlap can be understood as the amount of space the two objects would physically occupy during the deformation the actual physical bodies would experience.[MC14] Like most particle simulation methods, it is based on Newton's laws of motion. However, it differs from, for example, molecular dynamics by being completely reliant on its geometry for physical collisions, while in MD particles are described using force fields and potentials.

As such, DEM is a well-suited particle method for modeling granular material behavior in chemical, mining, pharmaceutical, food, and more industries. Its applications can be categorized into three classes, particle packing, particle flow, and particle fluid interaction. Particle packing processes define how particles fill certain spaces. These processes entail the deposition of particles, vibration after deposition of particles, and compaction. Particle flow applications on the other hand describe the regular movement of particles under gravity and other driving forces. The class of particle fluid interaction describes the movement of particles within a fluid flow, during wavelike motion, and during fluidization.[CS79]

2.1.2. Common DEM Particles

Sphere Particle

The most basic particle used in three-dimensional DEM simulations is the sphere particle. With their very basic geometry, defined solely by the position of their center and their radius, sphere particles offer the easiest and most efficient contact detection method. They also offer an accurate and fast calculation of the contact overlap between two particles, supporting a fast and reliable calculation of the inter-particle forces during a collision. As easy to use and implement as spherical particles may be, in reality, it is only rarely the case that granular materials are perfectly spherical, so more complex particle structures are often necessary.[MC14]

Multisphere Particles

Multisphere particles are, as the name suggests, clusters of multiple sphere particles with rigid bonds connecting them. They enable the modeling of particle irregularities while mostly preserving the efficiencies of singular spheres. The contact detection between two instances of multi spheres can be broken down to the level of the component spheres, and the same goes for the calculation of the contact overlap. There is also the option to enable the creation and breaking of bonds between particles if desired. For these reasons, they are one of the most common types of particles used in DEM simulations.[KERWS08][ZZYY07]

Polyhedral Particles

In terms of accuracy, polyhedral particles are the best choice for simulating the properties of granular particles , as they can depict the edges that actual granular materials have in the greatest detail. While close relation to reality is very desirable for simulations, it also comes hand-in-hand with a higher computational time cost for contact detection and contact forces and a higher difficulty for implementing the necessary algorithms and particle geometry. Furthermore, more accurately than with multi-sphere particles, the option to have particles break into smaller particles is available.[ZZYY07]

2.1.3. Force interactions

Contact mechanics is one of the essential ingredients for DEM simulations. In DEM, a particle can be affected by a multitude of contact interactions. Collisions can occur between two similar particles, between two particles of differing types, between particles and rigid surfaces, or between multiple particles at once. Several fundamental theories exist which model the forces in these collisions, and for this project, the Hertzian theory of non-adhesive elastic contact is utilized.[Her82]

Hertzian theory of non-adhesive elastic contact

The main focus of Hertz's classical theory of contact is on non-adhesive contacts where no tension force is allowed to occur within the area of contact. The following assumptions are made in determining the solutions of Hertzian contact problems[Her82]:

1. The two interacting bodies are assumed to be of an elastic, isotropic and homogenous material
2. The surface is assumed to be perfectly smooth such that no shear stresses occur in the interacting surfaces
3. It is assumed that only a relatively small part of the total surfaces is in contact.

In a simulation, the forces that geometric objects exert on each other are determined by the affected parties' contact area and collision depth. The theory of contact between two elastic bodies is used to determine the contact area and collision depth. There are multiple types of solutions for differing geometric object pairs, but we will only be looking at the Hertz normal contact between spheres in this thesis.

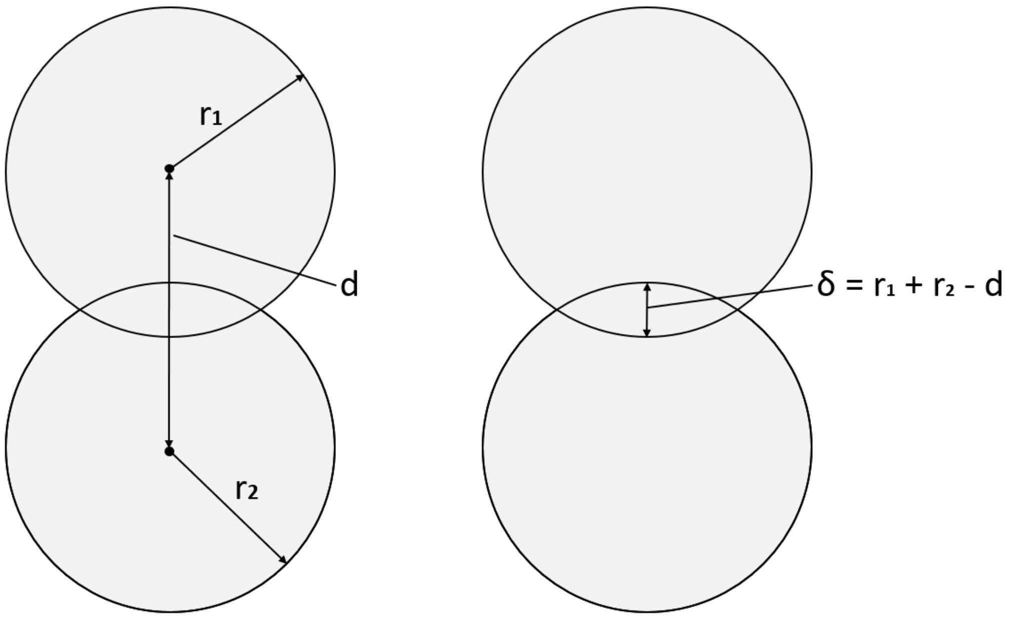


Figure 2.2.: Collision between two spheres and approach distance calculation

Once the contact between two particles has been determined, meaning approach distance δ is smaller than zero, see Figure 2.2, several geometrical and elasticity parameters are required to determine the inter-particle forces for the Hertz elastic solution. The required formulas for two contacting spherical particles are determined as follows[CS79]:

$$F = \frac{4}{3}E^{*}\sqrt{R}\sqrt{\delta^{3}} \tag{2.1}$$

where

$$R = \frac{R_1 R_2}{R_1 + R_2} \tag{2.2}$$

and

$$\frac{1}{E^*} = \frac{1-\nu_1^2}{E_1} + \frac{1-\nu_2^2}{E_2} \tag{2.3}$$

E1 and E2 are the Young modulus, v1 and v2 are the Poisson's ratio, and R1 and R2 are the radii of the two contacting particles, respectively. The Young modulus, also called the modulus of elasticity in tension or compression, is a mechanical property that measures the tensile or compressive stiffness of a solid material. The Poisson's ratio is a gauge of the Poisson effect, which describes how a material tends to expand in the direction perpendicular to the direction of compression. These parameters need to be specified during particle creation.[CS79]

2.2. Molecular Dynamics

Molecular dynamics simulations are utilized to examine the forces and interactions between particles on a molecular level. While in DEM simulations, physical contact between particles is necessary to create particle-to-particle interactions, in molecular dynamics, these interactions happen over range using intermolecular pair potentials.[GST+19]

2.2.1. The Lennard-Jones-12-6-Potential

In MD simulations, calculating the force that two particles exert on each other depends on two elements, the chosen underlying potential, and their relative positions to each other. The in AutoPas pre-existing implementation for MD-Simulations uses the Lennard-Jones-12-6-Potential, which simulates short-range interactions of electronically neutral atoms and molecules created by van der Waals forces and Pauli repulsion. As the most extensively studied potential under the intermolecular potentials, it is generally considered to be a classic exemplar for accurate and easy-to-implement molecular particle interactions. The formula used to calculate the potential between two molecules is as follows[GST+19]:

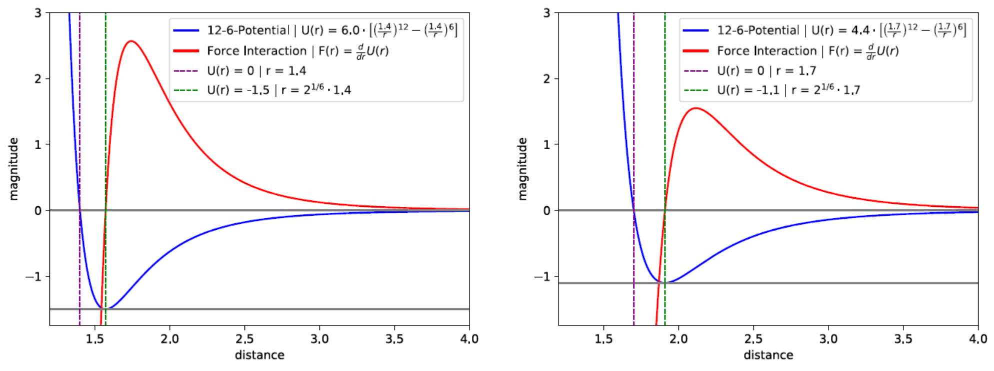
$$U\left(r_{ij}\right) = 4\epsilon\left(\left(\frac{\sigma}{r_{ij}}\right)^{12} - \left(\frac{\sigma}{r_{ij}}\right)^{6}\right) \tag{2.4}$$

And to gain the actual forces interacting between both particles, one can use the following formula:

$$F(r) = -\frac{\partial U(r)}{\partial r} = 48\epsilon\left[\frac{\sigma^{12}}{r^{13}} - \frac{\sigma^6}{r^7}\right] \tag{2.5}$$

Here r_{ij} is the distance between the two particles, ϵ signifies the dispersion energy, and σ determines at which distance the potential energy between the two particles is equal to zero.

As shown in Figure 2.3 the Lennard-Jones-Potential basically describes the fundamental force interaction between molecules. At close distances smaller than σ two particles repel each other, at a distance of exactly σ repulsion and attraction balance each other out to zero and at further distances than σ they attract each other, yet the attraction rapidly converges to zero towards infinite distance. Because of this a cut-off radius is often applied, which determines at which distance the resulting potential can be ignored.



(a) $\epsilon = 1.5 \quad \sigma = 1.4$ (b) $\epsilon = 1.1 \quad \sigma = 1.7$

Figure 2.3.: Two Lennard-Jones-12-6 Potential curves and the subsequent force curves in relation to the distance between two particles and the parameters of dispersion energy ϵ and size parameter σ.

2.3. Störmer-Verlet Algorithm

For all types of particle simulations, once the forces interacting between particles have been calculated, a method is necessary to apply these forces to them and so update their velocities and positions for the end of a simulation step. To fulfill this purpose in this thesis, the Störmer-Verlet algorithm is used for both DEM and MD simulations.[HLW03]

The formulation for the Störmer-Verlet algorithm is as follows:

$$\begin{aligned} x_i\left(t^{n+1}\right) &= x_i\left(t^n\right) + \Delta t \cdot v_i\left(t^n\right) + (\Delta t)^2 \frac{F_i\left(t^n\right)}{2m_i} \\ v_i\left(t^{n+1}\right) &= v_i\left(t^n\right) + \Delta t \frac{F_i\left(t^n\right) + F_i\left(t^{n+1}\right)}{2m_i} \end{aligned} \tag{2.6}$$

Here $x(t)$ and $v(t)$ signify the position and velocity of a particle respectively at time step t, and m stands for the mass of a particle. To be able to use the Störmer-Verlet algorithm, it is necessary to have the velocity and position of the previous time step saved for the calculation of the current time step, as can be seen from the formulation of the algorithm.

2.4. AutoPas

2.4.1. Particle Containers

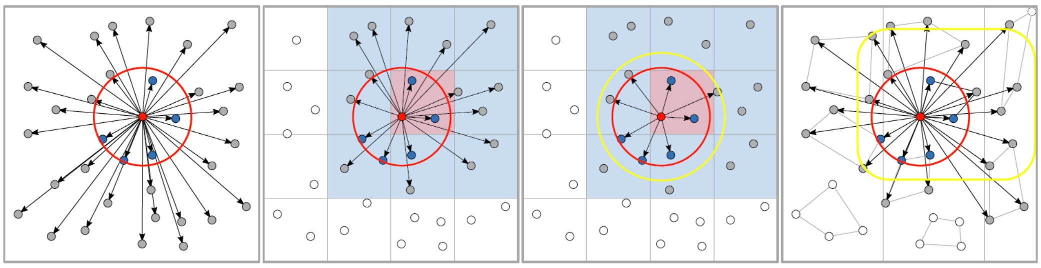


Figure 2.4.: Comparison between particle containers Direct Sum(a), LinkedCells(b), Verlet Lists(c) and Verlet Cluster Lists(d)
Source: [GSBN21]

One of the most fundamental aspects of any particle simulation is the particle neighbor identification algorithm so as to enable efficient computation of all pairwise forces. The basic premise of such an algorithm is to find for every particle in the simulation all neighboring particles in their effective range of significant interaction. The workflow of the chosen algorithm also greatly influences the data structures and layout of how particles are stored. AutoPas currently offers four different basic types of neighbor identification algorithms, which will be presented in the following.

Direct Sum

The Direct Sum algorithm offers the most trivial solution to this problem, as it simply works by calculating the distances between all pairs of particles in the whole simulation. If the calculated distance is smaller than the specified cut-off radius, the algorithm considers particles for the force calculation, as shown in Figure 2.4 (a), where the red circle signifies the cut-off radius. Due to its straightforward approach of simply comparing all particles to each other, this method has a complexity of $O(N^2)$, where N is the total number of particles in the system. To compensate for its high complexity, it offers the advantage of not needing any additional memory overhead for complex data structures.[GST+19]

Linked Cells

Moving up in terms of implementation complexity follows the Linked Cells, also called cell list algorithm. With the goal in mind to improve the scalability of the simulation, this intuitive algorithm works in a way so that it divides the spatial domain of the simulation into a regular cartesian grid of cells. Once the grid has been set up, all particles are sorted into their respective cells, as can be seen in Figure 2.4(b). To identify a particles(red particle) neighbors, it gets matched with all particles in its own cell(red cell) and all particles in its neighboring cells(blue cells). Afterward, the algorithm calculates the pairwise distances to all matched neighbors, and if the distance is smaller than the cut-off distance, considers the pair for force calculation. To ensure that the algorithm misses no actual particle interaction, the mesh size of the Cartesian grid needs to be bigger or equal to the cut-off radius. Otherwise the particles of more neighbor cells need to be taken into account. Using this technique, the time complexity can be reduced to O(N) in the case of homogeneous particle distributions, which is tremendously better than Direct Sum in case of a large enough number of particles.[GST$^+$19] Another advantage offered by the linked cells algorithm is that particles that are processed in sequence are also in sequence in memory, making vectorization and cache prefetching easier.

Verlet Lists

Once again, moving up in terms of implementation complexity follows neighborhood identification using Verlet Lists. Unlike the previous two algorithms, the Verlet list algorithm does not rely on spatial information but rather uses a less intuitive approach by keeping a record of a particle's previous interaction partners. These records are called Verlet Lists, and they consist of references to all neighboring particles within their cut-off distance. To improve the reusability of Verlet Lists, particles that are just barely out of range of the cut-off radius are also included in them. This extension creates the so-called Verlet Skin, as shown in Figure 2.4(c) indicated by the yellow circle. To determine the relevant pairwise force calculations is now relatively easy, as it is only necessary to calculate the distances to all particles within the Verlet Skin, which considerably decreases the number of neighbor distance evaluations. Since the generation of Verlet Lists is computationally expensive, it is recommended not to have to fall back to a Direct Sum pattern for the generation of these records. A better alternative is to use Verlet Lists in combination with the Linked Cells algorithm to fall back on, especially for large numbers of particles. One of the main disadvantages of Verlet Lists is that the algorithm has a huge memory overhead, consisting of a list of references for each particle. The second main disadvantage of the method is created by the data structure for particles leading to a lack of data locality. Because of this, it is hard to gauge if two particles are close in memory, which in turn makes the loading from memory result in bad cache behavior and complicated vectorization.[GST$^+$19]

Verlet Cluster Lists

The last and also least intuitive approach for neighborhood identification implemented in AutoPas is Verlet Cluster Lists. An extension upon Verlet Lists, this approach works with the idea that the Verlet Lists of neighboring particles are most often quite similar. So in this algorithm, particles with similar lists are combined into a Verlet Cluster List. Such a cluster is visualized in Figure 2.4(d). In comparison to Verlet Lists, this method is capable of significantly reducing the number of lists while also decreasing the overall size since it is no longer necessary to track individual particles. Another main advantage created by Verlet Cluster Lists is that it enables easier vectorization again, since it is possible to load a whole cluster into vector registers. While this method offers advantages in some areas, it also creates disadvantages in others. The first one is that the number of distance calculations increases since when two clusters need to interact, the search radius for a single particle increases to that of both clusters. Secondly, the algorithm creating Verlet Cluster Lists is significantly harder to implement compared to the previous algorithms.[GST$^+$19]

2.4.2. Data and Memory Layout

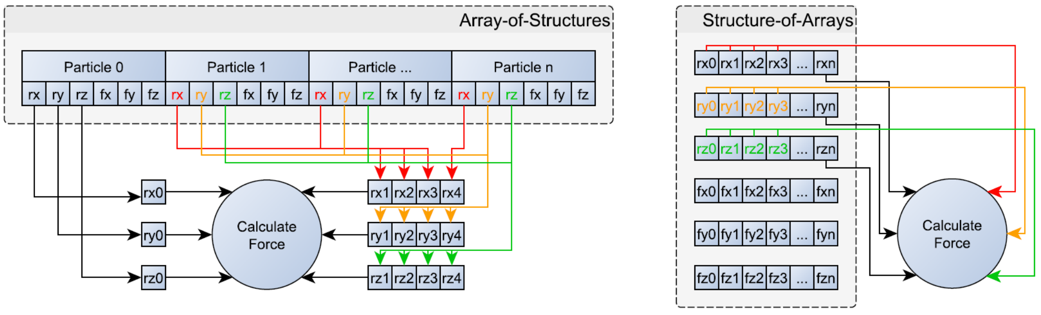


Figure 2.5.: Loading of particle data in AoS and SoA layout.[GST+19]

Another important aspect of any simulation is the data structures used to store the particle data. For this purpose, AutoPas offers two kinds of data layouts called Array-of-Structures (AoS) and Structure-of-Arrays (SoA).

AoS

The AoS layout can be described as an array-like container, often similar to a C++ `std::vector`, and is used as the default data layout in AutoPas. A practical method for force calculations in particle simulations is using SIMD (Single Instruction, Multiple Data) vectorizations, which offer the option to carry out an operation on a whole vector of data as a single instruction. To be able to use SIMD vectorization, it is required that all instances of a particle's attribute are stored contiguously. However, in the AoS layout, a particle's data structure is stored contiguously, offering the advantage of easily loading a single particle. The AoS layout proves disadvantageous for force calculations when loading multiple particle together since the required particle properties need to be loaded individually from each particle's own memory location into the needed vector registers. This leads to the program having to make jumps to reach the relevant part of a particle's data, invoking a multitude of slow memory accesses. For an easier understanding of this issue, an illustration of the AoS layout is shown on the left side of Figure 2.5[GST+19].

SoA

A more sustainable solution to the issue of data loading is offered by first converting the data structure from the AoS data layout to the SoA data layout. In SoA, a structure is created that stores multiple arrays, each containing all instances of a single particle attribute. Using this data layout, all necessary data for the force calculation can be efficiently loaded into memory since it is already stored contiguously. An illustration of this situation is shown on the right side of Figure 2.5. While the SoA layout is more efficient in terms of caching and vectorization, it is also less intuitive than AoS. Another disadvantage is created by the fact that AutoPas stores data in AoS format, so if one chooses to use calculations in the SoA format, AutoPas has to first convert the entire data structure from AoS to SoA and after force calculations again back to AoS. Keeping all advantages and disadvantages in mind, the SoA format can be considered to be the more efficient data layout for Linked Cells and simulations with huge amounts of particles.[GST+19]

2.4.3. Newton's third law

Before we talk about the next chapter, traversals, it is necessary to mention how AutoPas handles Newton's third law of motion, which states that for every action, there is an equal and opposite reaction. So in the case of interaction between a particle i and a particle j, the resulting force F needs to be applied to both particles but in opposite directions, meaning:

$$\mathrm{F}ij = -\mathrm{F}ji$$

Now to apply this to the subject of particle simulations, this leads us to the optimization, further on called Newton3 optimization, which allows us to calculate the force between two particles only once and apply it to both particles in opposite directions.[GST+19]

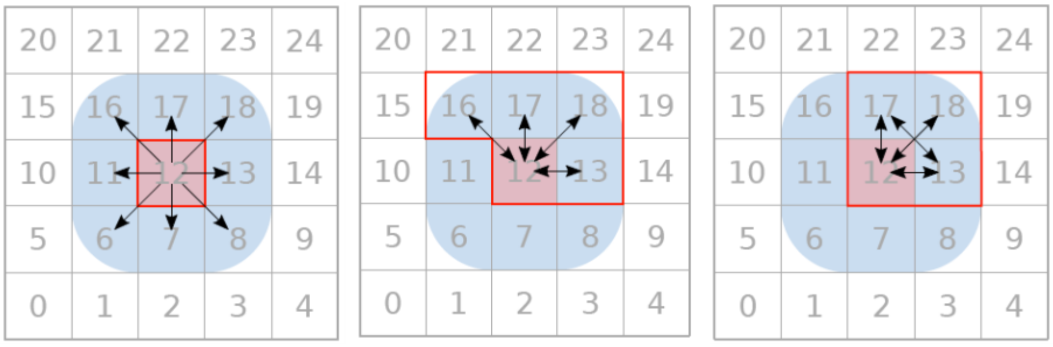
2.4.4. Traversals

Another key component in particle simulations is the traversal of the domain and its shared-memory parallelization. Implementing a traversal that allows evenly distributing the simulations workload onto multiple threads can significantly increase its runtime. In AutoPas, the chosen traversal option decides the order in which particles are iterated over and the parallelization of the force calculation. The available traversal methods are grouped based on the underlying container structure. This is because the choice of container dictates the underlying data structure and a traversal dictates how to navigate the data structure.[GST+19]

This thesis will focus only on the sequential traversal option for Direct Sum "ds_sequential" and the c01, c18, and c08 base steps for Linked Cells.

Traversal for Direct Sum

The traversal "ds_sequential" can arguably be considered the most basic traversal option. It processes all particle pairs sequentially in the order they were passed to the container.



(a) c01 base step (b) c18 base step (c) c08 base step

Source: [GSBN21]

Figure 2.6.: Base steps currently implemented in AutoPas. The currently traversed cell, also called base cell, is signified by its red coloring. The cells with which the particles of the base cells interactions need to be taken into consideration are marked with blue coloring. The red outlined bounding box signifies which cells must be protected from race conditions during the traversed cells base step.

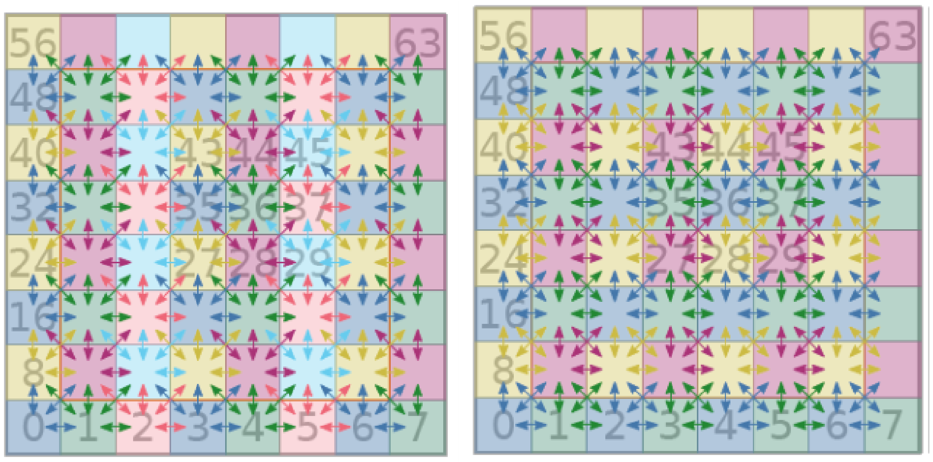
Traversals For Linked Cells

Three base step methods have been implemented to enable shared parallel traversals over Linked Cells in AutoPas. To discuss how these base steps influence a domain's parallelization, we will introduce domain coloring. Each cell gets a color assigned, and cells with matching colors get processed in parallel. This ensures protection against race conditions since the coloring eliminates the possibility of a thread accessing data, which is potentially being modified by another thread.[GSBN21]

c01 base step - Depicted in Figure 2.6a, the c01 base step is the most intuitive and easy to implement base step. It works by computing all interactions of a cell with all its neighbors without modifying its neighbors. Because of this, it can not make use of the Newton3 optimization.[GSBN21]

c18 base step - Depicted in Figure 2.6b, this option only computes interactions with neighbors of a greater cell index. It makes use of Newton3 optimization but, in turn, also increases the box of cells, highlighted by a red border, that other threads can not modify in parallel to guard them against race conditions. Aside from the utilization of Newton3, another advantage this method offers is that the number of cells relevant for one base cells force calculation is decreased to five in 2D and 18(3x3x2) in 3D, making the method more cache efficient. Illustrated in Figure 2.7a, the domain coloring of a traversal using the c18 base step can be found. In 2D, six colors need to be used, and in 3D, this number jumps up to 18.[GSBN21]

c08 base step – Building upon the idea of the c18 base step, the c08 base step decreases the size of the box to guard against race conditions by replacing the computation of the forward diagonal interaction with the forward interaction of the cell next to it. This can be observed in Figure 2.6c where the interaction between cells 12 and 16 has been replaced by the interaction between cells 17 and 13. The same can be applied to a 3D box as well. By doing so, the number of locked cells gets further decreased, which again offers better cache usage and faster parallelization. Illustrated in Figure 2.7b, the domain coloring of a traversal using the c08 base step can be found. In 2D, four colors need to be used, and in 3D, the number increases to eight.[GSBN21]



(a) c18 base step traversal in 2D (b) c08 base step traversal in 2D

Figure 2.7.: Domain colorings using different base steps

Several traversal patterns can be conceived by using these base steps in combination with the underlying respective data structure. This thesis will not go into further detail concerning the possible traversal patterns since they are already discussed in great detail in [GSBN21] and the official AutoPas documentation[1].

[1]`https://autopas.github.io/`

Part II.

Implementation

3. DEM Sphere Particle

In this chapter, we discuss the necessary features that need to be implemented to add a new custom particle and its respective pairwise force calculation to the AutoPas library and how such a particle and force calculations were created in the scope of this thesis. The class in which the pairwise force calculation is implemented will be called functor in the following. Afterward, we will discuss how the necessary features needed to run a simulation with this particle have been added or changed in the pre-existing example code ”md-flexible”.

3.1. Particle Implementation

Every particle used in an AutoPas simulation has the option to inherit from the particle base class autopas::Particle. This base class is used to define the minimally required parameters and functions by AutoPas. These parameters consist of the particle’s position as an array of 3D coordinates, its id, its ownership state, and the velocity and force of the particle as arrays of 3D vectors. The functions inherited by the base class include the accompanying setters and getters of all mentioned parameters and additionally functions for adding and subtracting the particle’s force, position, and velocity. With these parameters already taken care of, the next consideration is what additional parameters need to be implemented to expand the particle implementation. Necessary for our DEM-Sphere particle are the particle’s radius, mass, Poisson ratio, Young modulus, and the force the particle experienced in the previous iteration, which we will from now on call OldForce. The first four of these parameters are needed for the force calculation, as already discussed in Subsection 2.1.3. The parameter OldForce, however, is necessary for the implementation of the Störmer-Verlet timestep algorithm, which will be later discussed in the functor implementation section. With all parameters taken care of, our particle class, including all inherited parameters, can be viewed in Listing 3.1.

Listing 3.1: Structure of a DEM Sphere.

```
class DEMParticle{
    public:
        std:array<double, 3> position;
        std:array<double, 3> velocity;
        std:array<double, 3> force;
        std:array<double, 3> oldForce;
        int id;
        OwnershipState ownershipstate;
        double radius;
        double mass;
        double poissonRatio;
        double youngModulus;
}
```

Listing 3.1: Structure of a DEM Sphere.

AutoPas uses a system involving dynamically sized SoA to store these parameters, so it is required to implement an enumeration of individual ids for all parameters, together with a definition of a matching SoAArraysType, as seen in Listing 3.2.

Listing 3.2: DEM particle parameter enumeration and SoAArraysType

```
enum AttributeNames : int { ptr, id, posX, posY, posZ, forceX, forceY, forceZ, rad, poisson, young, mass, typeId, ownershipState };

using SoAArraysType =
      typename autopas::utils::SoAType<DEMParticle<floatType> *, size_t /*id*/, floatType /*x*/, ..., OwnershipState /*ownershipState*/>::Type;
```

Listing 3.2: DEM particle parameter enumeration and SoAArraysType

The definition of the SoAArraysType is necessary for AutoPas so that its simulation system has precise knowledge of the types and contents of all the parameters used for the desired particle type. As the last step for finishing our particle, we implemented two functions to be used as the particles handler for the get and set interactions of all attributes. These are necessary for enabling AutoPas to access an attribute without having to call it by its getter or setter, but rather by using their respective enum attribute name.

Listing 3.3: DEM particle get and set functions

```
constexpr typename std::tuple_element<attribute, SoAArraysType>::type::value_type get() {
    if constexpr (attribute == AttributeNames::ptr) {
      return this;
      ... }
  }

constexpr void set(typename std::tuple_element<attribute, SoAArraysType>::type::value_type value) {
    if constexpr (attribute == AttributeNames::id) {
      setID(value);
    ... }
}
```

Listing 3.3: DEM particle get and set functions

3.2. Functor Implementation

For calculating the interactions between our sphere particles, we implement our functor using the in Subsection 2.1.3 introduced Hertzian theory of non-adhesive elastic contact's solution for force calculation between spheres. Here again, a base class called autopas::Functor is available to inherit from. It implements virtual functions to be re-defined in the user's functor implementation, a function for converting the AoS data of a given cell into the SoA layout, and a function for converting it back into AoS format. The most important part of a functor is the force calculation, which can be adapted to fit both AoS and SoA layouts. AutoPas allows to implement only one of the two layouts, but for this thesis, both were implemented.

3.2.1. AoS force calculation

According to the AoS data layout, the parameters needed for this function consist of two particles directly and additionally a boolean that indicates if the Newton3 optimization is to be applied. Because one only has to handle two particles simultaneously, the AoS force calculation is more intuitive and was implemented first. As a first step, the algorithm checks if one of the two particles is a particle not meant for force calculations called a dummy particle. If none of them are, it calculates the penetration depth to determine if the two spheres are colliding using the particle's positions and radii. If a collision has been confirmed, the force is calculated according to Equation 2.1.3. Afterward, the resulting force vector gets applied to either one or both particles according to the Newton3 boolean.

3.2.2. SoA force calculation

Compared to the AoS data layout, the SoA data layout makes for a more challenging force calculation implementation due to its more complex structure. As mentioned in Section 2.4.2, an SoA consists of multiple arrays, each containing the singular parameters of multiple particles in order. For the force calculation in SoA format, AutoPas requires three specific functions for implementing different cases. Firstly, a function for the force calculations for all particles within a single SoA; secondly, a function for the force calculation of particles between two different SoAs ; and thirdly, a function for the force calculation for SoAs for neighbor lists. Since the force calculations for a single SoA and between two SoAs is pretty similar, we implemented a helper function to handle both of them. This function receives two SoAs, for the first variant the same one twice, a boolean called single to differentiate between both variants, and again a boolean Newton3. Its functionality goes as follows: As a first step, pointers for each particle parameter, using the parameters defined attribute names, get set to the beginnings of all SoA arrays within both SoA. Using these pointers as starting points, we iterate over each particle in the first array. For each particle we iterate over, an iteration over particles in the second array gets triggered. If the boolean single is set to true, the second iteration starts with a shift to avoid a particle interacting with itself. If set to false, the iteration goes over the entire array. The force calculation happens again according to Equation 2.1.3. However, since we use the SoA data layout, we can use SIMD vectorization via OpenMP for increased calculation efficiency. As previously mentioned, the last necessary force calculation is for SoA for neighbor lists. This function's implementation is again mostly the same, however, to enable efficient vectorization, we divide the neighbor list into fragments the size of a fixed parameter called vecsize. So in each iteration step, we calculate the interactions of the current particle with the number of vecsize particles in the neighbor list of said particle.

3.2.3. Further functions

With the main part of the functor taken care of, there are only a few remaining functions that AutoPas needs for the functor to be fully implemented. The first two of these are the functions `initTraversal()` and `endTraversal()`, which, as the name suggests, are called at the start and end of each traversal. The next two are used to get all input and output variables of the force calculation via `getComputedAttr()` for all output variables and `getNeededAttr()` for all input variables. And the last two functions which we implemented are used to tell AutoPas whether the functor allows Newton3 optimization via `allowsNewton3()` and whether the functor is relevant for tuning via `isRelevantForTuning()`.

Listing 3.4: Implemented functions of DEM sphere functor

```
class DEMFunctor{
    public:
        void AoSFunctor(Particle &i, Particle &j, bool newton3) final {...}

        void SoAFunctorSingle(SoAView<SoAArraysType> soa, bool newton3) final {...}

        void SoAFunctorPairImpl(SoAView<SoAArraysType> soa1, SoAView<SoAArraysType> soa2) {...}

        void SoAFunctorCalc(SoAView<SoAArraysType> soa1, SoAView<SoAArraysType> soa2,  bool single, bool newton3){...}

        void SoAFunctorVerlet(SoAView<SoAArraysType> soa, const size_t indexFirst,
                        const std::vector<size_t, autopas::AlignedAllocator<size_t>> &neighborList,
                        bool newton3) final {...}

        constexpr static auto getNeededAttr() {...}

        constexpr static auto getComputedAttr() {...}

        bool isRelevantForTuning() final { return relevantForTuning; }

        bool allowsNewton3() final { return useNewton3 == FunctorN3Modes::Newton3Only or useNewton3 == FunctorN3Modes::Both; }
}
```

Listing 3.4: Implemented functions of DEM sphere functor

4. Simulation

AutoPas is shipped together with a few example codes, one of them being md-flexible. Its primary purposes are to showcase AutoPas to first-time users, to provide developers with a fast and easy way to run actual simulation code for testing purposes and to show how one can configure all of AutoPas's parameters from outside. Primarily md-flexible is a simple molecular dynamics simulation using the Lennard-Jones 12-6 potential [Equation 2.2.1] for force calculations between molecular particles. This section describes how we adapted and expanded the example code md-flexible to additionally enable the simulation of our newly created DEM sphere particle.

4.1. Simulation preparation

Before the simulation loop can be started, it is first necessary to load the configuration and afterward initialize the simulation. Since there are several differences between simulating particles using the discrete element method and molecular dynamics, the simulation contains some features not relevant for this thesis which will, as such, mostly be omitted. As the first preparation step, the configuration of the simulation is loaded using either a YAML-file parser or command line arguments. Both of these loading options remain mostly untouched, with the exception of the addition of DEM-sphere relevant options, being the radius, Young modulus and Poisson ratio, to parse. We then set all the configuration options for AutoPas and initialize all the objects in the simulation. For generating these objects, md-flexible offers a number of objects generators for different purposes. These have also been slightly adapted with the necessary DEM parameters, to be able of creating DEM-sphere particles. With all preparations complete, the simulation is ready to start.

Algorithm 1: Simulation Initialization

```
Input:  simulation configuration, autoPas container
Output: configured autoPas container
Function init_Simulation(config, autoPasContainer):
    autopas.set_Values()
    for auto object : config → varObjects do
        object.generate(autopas)
```

Figure 4.1.: Simulation initialization

Algorithm 2: Simulation Loop

```
   Input:    autoPas container, functor
   // Main Loop
1  Function do_Simulation(autoPasContainer):
2      for autoPas.needsMoreIterations() do
3          calculatePositions(autopas)
4          updateContainer(autopas)
5          autopas.iteratePairwise(functor)
6          calculateVelocity(autopas)

   // update Container function
7  Function updateContainer(autopas):
8      wrapPositionsAroundBoundaries(autopas)
9      addEnteringParticles(autopas)
10     updateHaloParticles(autopas)
```

Figure 4.2.: Simulation loop

4.2. Simulation loop

The simulation begins by opening the main loop for the entire simulation. In each iteration, the first step consists of calculating the new positions of all particles using the first part of the Störmer-Verlet timestep algorithm in Equation 2.3. Our next course of action is to handle the movements of particles in the underlying data structure by updating the container inside AutoPas. This step consists of determining which particles left their domain box, applying boundary conditions on them, and afterward again add them to the container in the correct domain box. It also includes identifying and handling halo particles, which are particles situated outside the bounding box of the domain for handling boundary conditions. With particle movement completed for the iteration step, the force calculation for all particles is next in line. To iterate over all particles with the correct functor, we call the function autoPas.iteratePairwise(functor). The last step of each iteration is the calculation of the new velocities using the second part of the Störmer-Verlet timestep algorithm.

Part III.

Results

5. Testing Setup

All performance measurements were run on a system with an AMD Ryzen 3700X 8-Core Processor(3593MHz), 16 GB DDR4 memory(1497 MHz). The CPU's theoretical single-precision performance in 64-bit accuracy is approximately 1.6896. GFLOPS[1] For an accurate comparison between the performance of the MD particle and the performance of the DEM particle, the respective simulations comparing the two have been run with the same configuration options wherever possible and with the most similar configuration options otherwise. Shared configuration options include all AutoPas tuning-related configurations, such as data layout, data structure, traversal patterns, the tuning strategy, and general simulation options, such as particle distribution, the number of particles, and the time-step size, and the number of iteration steps. The non-shared options consist of options related to the used particle methods, such as a particles dispersion energy parameter ϵ and size parameter σ for MD and a particles Poisson ratio and Young modulus for DEM. Simulation results will be compared by two approaches. Each simulation will be compared by two approaches, by the performance results of each simulation and by a simulation's physical results, alongside the visualization of the simulation at certain time-steps.

[1]`https://www.techpowerup.com/cpu-specs/ryzen-7-3700x.c2130`

6. Particle Funnel Simulation

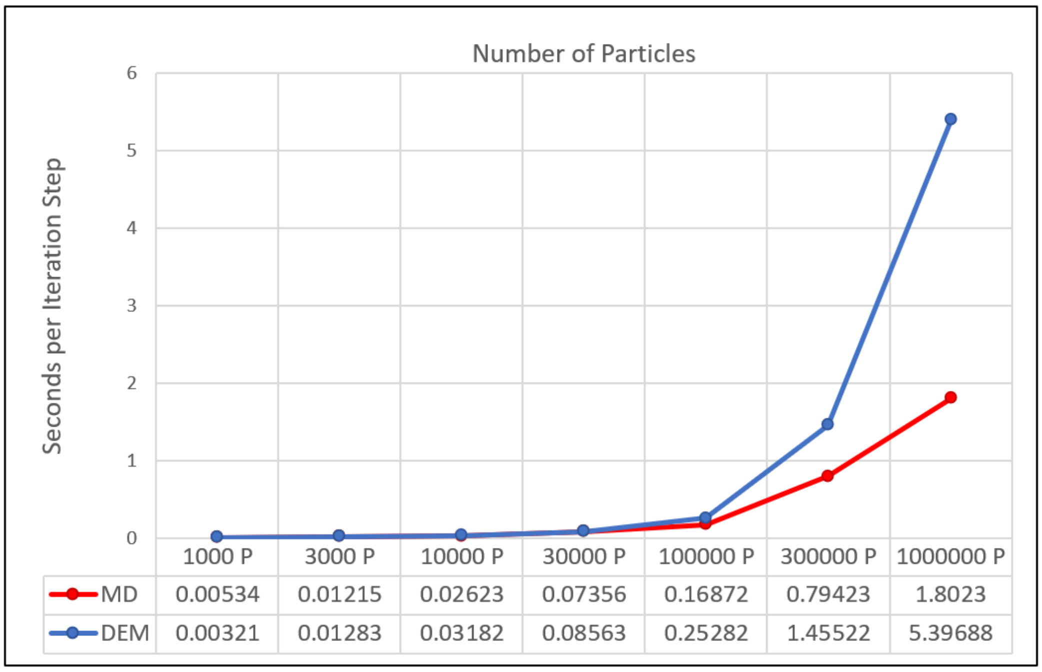
For the first simulation, we constructed a simulation in which loose particles in the form of a sphere fall into a funnel made out of four trapezoid particle grids and four square particle grids. Afterward, the particles fall into a particle casing without a roof, made out of five square particle grids, as seen in Figure 6.2c. A global force is applied to all particles to simulate gravity. The configuration for this simulation allows all types of particle containers, data and memory layouts, traversals, and the Newton3 optimization to be on or off. Furthermore, periodic boundaries are turned on, meaning that if particles leave the simulation boundaries, they get re-inserted at the other side of the simulation box. Using periodic boundaries requires the creation of halo particles, which are also relevant for calculations. Each calculation allowed up to eight threads to be used and the cut-off radius was set to three, the Verlet-skin-radius to 0.3 and the verlet rebuild frequency to ten, for all simulations.

6.1. Performance Comparison

To evaluate the performance of both simulation methods, we compare them based on the time of a single iteration step for an exponentially increasing amount of particles, ranging from 1.000 to 1.000.000. Both particles mostly had the same auto-tuned configurations for each simulation: The used data layout was AoS from 1.000 to 3.000 particles and SoA from 10.000 to 1.000.000 particles. The chosen particle container was VerletLists from 1.000 to 10.000 particles and Verlet Cluster Lists for the remaining amount of particles. The performance comparison between the two Particles can be seen in Figure 6.1. Ranging from 1.000 to 30.000 particles, the MD and the DEM particle share similar iteration times, however, from 30.000 to 1.000.000 particles, the MD particle gains the advantage and at 1.000.000 particles has nearly three times faster calculation per iteration.

6.2. Physical Comparison

For the comparison of the physical result, it is relevant to know the physical parameters describing the particles. The goal we set was for the particles to behave like hard rubber, so the parameters were chosen accordingly in the DEM particles case. The Poisson ratio is set to 0.45, and the Young modulus is set to five for all moving particles, which would roughly translate to the actual parameters of hard rubber at 20 Degree Celsius. Since MD particle parameters are usually used to describe physical properties on a molecular level, the choice of parameters to mimic the desired material was not possible, and so the parameters from AutoPas's standard simulation were kept, meaning $\epsilon = 1$ and $\sigma = 1$. The snapshots were taken after 10000 iteration steps with a time-step size of 0.0005 seconds.

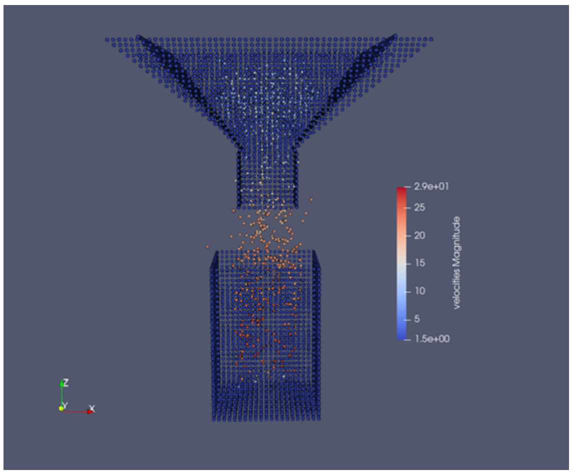


	1000 P	3000 P	10000 P	30000 P	100000 P	300000 P	1000000 P
MD	0.00534	0.01215	0.02623	0.07356	0.16872	0.79423	1.8023
DEM	0.00321	0.01283	0.03182	0.08563	0.25282	1.45522	5.39688

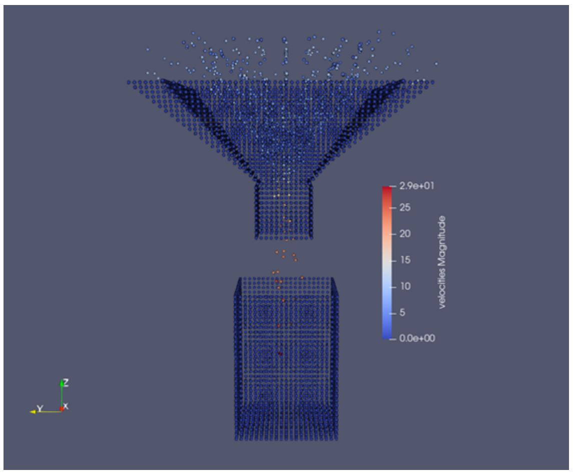
Figure 6.1.: Performance comparison of particle funnel simulation. The performance is measured by the amount of time it takes to pass a single time-step, for various amount of particles. At lower particle counts both MD and DEM have similiar performances but MD outperforms DEM at higher particle counts by being nearly three times as fast

As illustrated in Figure 6.2a, the particles mostly behave as they would in reality. After landing in the funnel, they bounce off the walls and gather in the middle of the funnel before dropping through its hole at the bottom. They then fall through the casing and bounce off its bottom. While in reality, all particles would fall through the funnel, in this DEM simulation, a few outliers clipped through the funnel. This situation is because by simulating walls out of spherical particles, spaces between particle centers are created where particles could slip through with enough velocity or force. To prevent this situation, one could either increase the funnel particles radius or decrease the spacing between its particles. However, both of these options would increase the simulations calculation effort. A more fitting solution would be to use a simulation wall or a more complex particle model like multi spheres or polyhedral particles for a more accurate simulation.

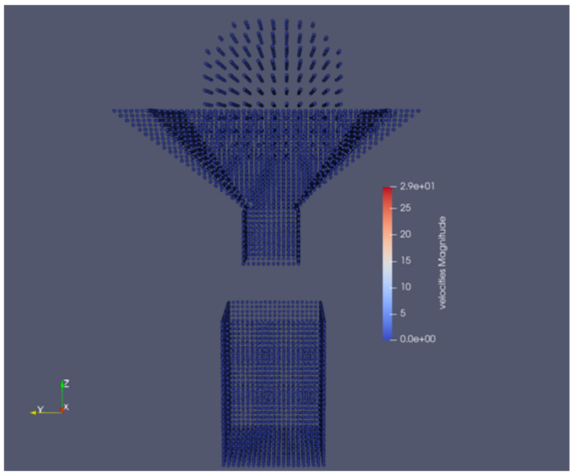
Figure 6.2b shows us the results of the particle funnel simulation using MD particles. It can be seen that a few particles slowly trickle down the funnel, while the majority of them are suspended in the space above the funnel structure. The reason for this situation is that the particles are not colliding since they have no physical body, but are instead pushing against each other stronger than gravity. If we let the simulation run for more iterations, all particles would slowly move towards the bottom of the casing due to gravity. However, particles would not only travel on the inside but also outside the simulation structure since they are overflowing.



(a) Particle funnel - result DEM



(b) Particle funnel - result MD



(c) Particle funnel at start of Simulation

Figure 6.2.: Visualization of particle funnel simulation results for DEM(a), MD(b) and of starting position for both(c). The particles color is based on their velocity, from not not moving(dark blue) to very fast moving(dark red). In the results for DEM the particles of the sphere drop into the funnel, bounce of its walls, fall down through the funnel's bottom hole and collect in the casing at the bottom. The MD results show how some particle fall down through the funnel's hole, but the majority gets pushed upward by MD's inter-particle forces.

7. FallingDrop Simulation

For the second simulation, we chose md-flexibles standard simulation example. This simulation is to show how a spherical droplet made out of multiple particles would drop into a pool of liquid particles. Once again, a global force is applied to all particles to simulate gravity. Same as in the particle funnel simulation, the configuration for this simulation allows all types of particle containers, data and memory layouts, traversals, and the Newton3 optimization to be on or off. Periodic boundaries are once again enabled. To stop liquid particles from dropping through the bottom of the simulation, a layer of fixed particles is added underneath. Again, each calculation allowed up to eight threads to be used and the cut-off radius was set to three, the Verlet-skin-radius to 0.3 and the verlet rebuild frequency to ten, for all simulations.

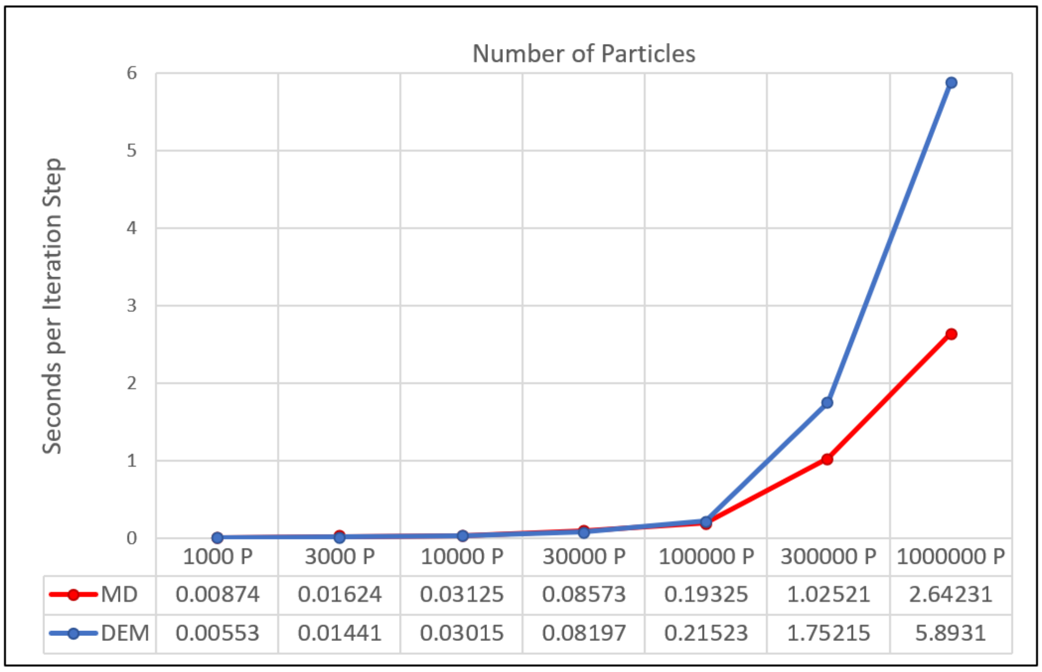
7.1. Performance Comparison

Same as before, to gain the performance of both simulation methods, we compare them based on the time of a single iteration step for an exponentially increasing amount of particles, ranging from 1.000 to 1.000.000. The auto-tuned configurations are nearly identical to the previous simulation, with the exception that the chosen particle container was Verlet Cluster Lists for all amounts of particles. The comparison between the two Particles can be seen in Figure 7.1. This time ranging from 1.000 to 10.000 particles, the DEM particle has a slight advantage over the MD-Particle. However, from 30.000 to 100.000 particles, both are again in similar time ranges, and from 300.000 to 1.000.000 particles, the MD particle again gains the advantage with about half the time-cost per iteration.

7.2. Physical Comparison

The visual goal of this simulation was to show how a drop of liquid falls into a sea of liquid. Since the discrete element method is generally not used for the simulation of fluids, we kept the same particle property values as before, meaning Poisson ratio = 0.45 and Young modulus = 5. All particle parameters were kept the same as they were initially in the pre-existing simulation file, meaning again $\epsilon = 1$ and $\sigma = 1$. The snapshots were taken after 10000 iteration steps with a time-step size of 0.0005 seconds.

At the start of the DEM simulation, the liquid main body of particles and the sphere particles start dropping due to gravity. First, the liquid particles crash into the bottom layer, where most of them are stopped and bounce back up slightly, before being pushed back down by the force of the particles above, followed by the sphere's particles. A layer of particles that due to immense forces crashed through the bottom layer is re-inserted at the top of the simulation, where the particles continue to accelerate. Figure 7.2a shows us the state at the end of the DEM simulation, where it can be seen how even though the bottom

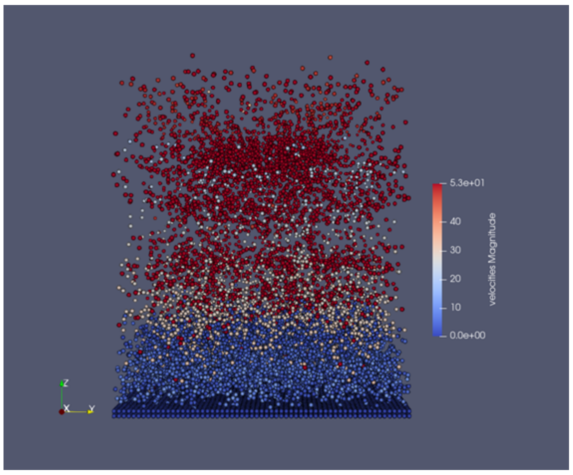


	1000 P	3000 P	10000 P	30000 P	100000 P	300000 P	1000000 P
MD	0.00874	0.01624	0.03125	0.08573	0.19325	1.02521	2.64231
DEM	0.00553	0.01441	0.03015	0.08197	0.21523	1.75215	5.8931

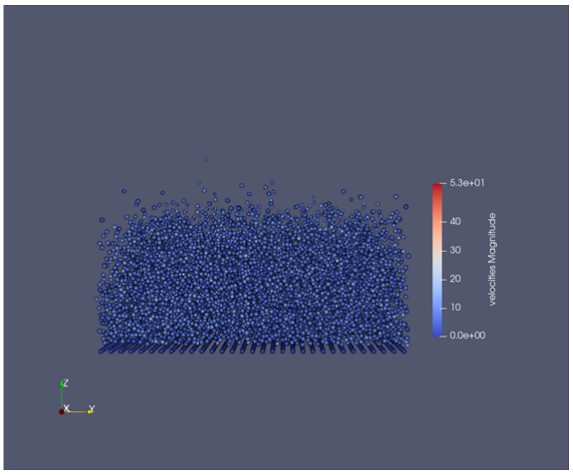
Figure 7.1.: Performance comparison of falling drop simulation. The performance is measured by the amount of time it takes to pass a single time-step, for various amount of particles. At lower particle counts both MD and DEM have similiar performances but MD outperforms DEM at higher particle counts by being more than double as fast.

two layers of the simulation were fixed in place, many particles penetrated them and are now still falling at rapid speeds.

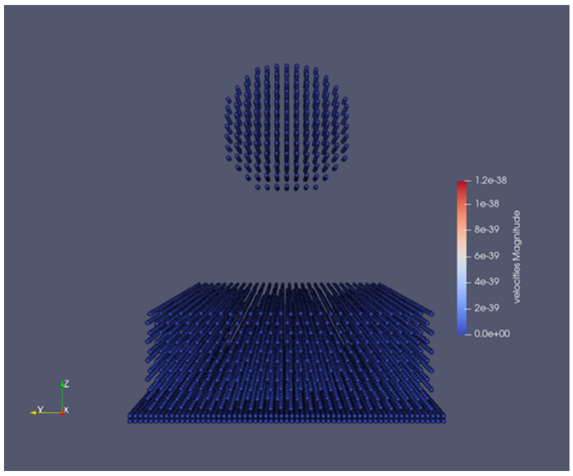
In the MD simulation, the liquid particles crash into the bottom layer, again followed by the sphere's particles. However, other than in DEM, the particles bounce back even up to the top layer of particles. In Figure 7.2b, the results of the MD simulation are shown, where it can be observed that, unlike the DEM Simulation, no particles crossed through the bottom layer. The results show how the particle sphere has immersed itself entirely in the body of liquid particles and how the force of the impact still affects the body of particles.



(a) FallingDrop - result DEM



(b) FallingDrop - result MD



(c) FallingDrop at start of Simulation

Figure 7.2.: Visualization of falling drop simulation results for DEM(a), MD(b) and of starting position for both(c). The particles color is based on their velocity, from not not moving(dark blue) to very fast moving(dark red). In the results for DEM the particles drop through the bottom layer and get re-inserted at the top, while the results for MD show how the sphere drop has mixed with the rest of the particles, but the results of the impact are still visible.

8. Conclusion and Future Work

As an auto-tuner and as a particle simulation software, AutoPas is a powerful and helpful tool for creating simulations of all kinds. While one can easily use AutoPas to create a simulation by using md-flexible as an example, implementing a new particle simulation method definitely takes more time and effort. However, since everything is well documented, both in the code and in various other documents, not limited to [GSBN21][GST$^+$19], one can nicely learn about the structure of AutoPas, how it works and what needs to be implemented for creating a new particle type and simulation. Throughout the course of this thesis, we have implemented a particle using the discrete element method, adapted it to the requirements of AutoPas and adapted md-flexible to the requirements of the particle. We successfully created simulations that showed the strengths and weaknesses of our particle and different simulation methods. Furthermore, the performance of the particle was tested and evaluated for its optimum simulation configurations using AutoPas's auto-tuning ability and compared the performance with the pre-existing molecular dynamics particle.

Possibilities and opportunities for future work involve the implementation of different or complex particles and simulation methods. Examples of this are multi-sphere or polyhedral DEM particles, or particles using approaches like Smoothed Particle Hydrodynamics or the Finite Element Method. Additionally, one could implement a simulation using multiple different particle types simultaneously, as many more extensive DEM Simulations already have.

Part IV.

Appendix

List of Figures

Bibliography

[CS79] P. A. Cundall and O. D. L. Strack. A discrete numerical model for granular assemblies. *Géotechnique*, 29(1):47–65, 1979.

[GSBN21] Fabio Alexander Gratl, Steffen Seckler, Hans-Joachim Bungartz, and Philipp Neumann. N ways to simulate short-range particle systems: Automated algorithm selection with the node-level library autopas. 2021.

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

[Her82] Heinrich Hertz. Ueber die berührung fester elastischer körper. 1882(92), 1882.

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

[KERWS08] H Kruggel-Emden, S Rickelt, S Wirtz, and V Scherer. A study on the validity of the multi-sphere discrete element method. *Powder Technology*, 188(2):153–165, 2008.

[MC14] Hans-Georg Matuttis and Jian Chen. Understanding the discrete element method: Simulation of non-spherical particles for granular and multi-body systems. *Understanding the Discrete Element Method: Simulation of Non-Spherical Particles for Granular and Multi-body Systems*, 05 2014.

[ZZYY07] HP Zhu, ZY Zhou, RY Yang, and AB Yu. Discrete particle simulation of particulate systems: theoretical developments. *Chemical Engineering Science*, 62(13):3378–3396, 2007.