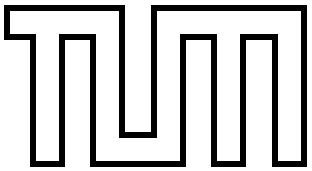


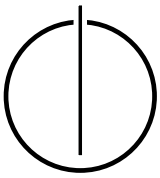
FAKULTÄT FÜR INFORMATIK

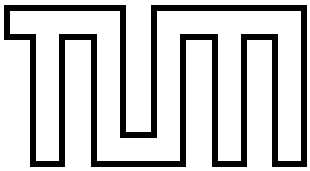
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

# Implementation of the Fast-Multipole-Method Using AutoPas

Joachim Marin





FAKULTÄT FÜR INFORMATIK

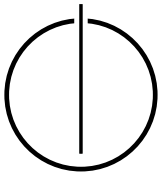
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

# Implementation of the Fast-Multipole-Method Using AutoPas

# Implementierung der Fast-Multipol-Methode unter Verwendung von AutoPas


| | |
|---|---|
| Author: | Joachim Marin |
| Supervisor: | Univ.-Prof. Dr. Hans-Joachim Bungartz |
| Advisor: | Steffen Seckler and Fabio Alexander Gratl |
| Date: | January 15, 2020 |

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

Munich, January 15, 2020 Joachim Marin

# Abstract

AutoPas is a library aimed to provide efficient calculation of particle interactions in N-body problems for short-range forces. These forces converge quickly to zero for higher distances, such that only interactions between particles with short distances need to be calculated. This massively reduces the number of required calculations and therefore enables very fast computation of the forces between the particles. However, there are also long-range forces, which do not decay as fast, so that even interactions between particles that are far away from each other need to be evaluated. The Fast Multipole Method is an algorithm that can approximate these long-range forces accurately in linear time.

This thesis describes two approaches of how the Fast Multipole Method can be used with AutoPas and analyzes them in terms of performance and accuracy. The Fast Multipole Method will be implemented for the Coulomb potential, which is an example of such a long-range force.

# Contents

# Part I.

# Introduction and Background

# 1. Introduction

Particle simulations are an important part of modern science and are used in many fields, such as astrophysics[Spu97], biology[KP90] or chemistry[vGB90]. In astrophysics a particle may represent a celestial object, whereas in chemistry a particle is generally only a small object like an atom or molecule. Many particle systems that are analyzed contain so many particles, that analytically evaluating the interactions between the particles is not feasible. Here, particle simulations are used to replicate the behaviors of the particles.

Directly performing interactions for every particle pair leads to quadratic run-time. This approach is called Direct Sum algorithm and yields exact results. In order to get faster run-times and therefore make even larger simulations possible, several algorithms that approximate the results have been developed over time.

In the Barnes-Hut algorithm, named after Josh Barnes and Piet Hut, particle clusters are combined to a single imaginary particle at the center of mass of the original cluster. Then particles that are far away from the cluster only interact with the imaginary particle instead of all particles in the cluster, reducing the number of interactions. The Barnes-Hut algorithm is usually used with an octree similar to the Fast Multipole Method (see Subsection 2.3.1) to build a hierarchical structure of clusters. This algorithm achieves run-times in $O(n \log n)$[BH86], which is a noticeable improvement over $O(n^2)$, considering the number of particles used in modern simulations. With the help of supercomputers, even particle numbers of multiple trillions have been achieved[GK08][EHB+13].

The Fast Multipole Method picks up on the idea of the Barnes-Hut algorithm and also uses clusters of particles that are usually given by an octree. However, here also imaginary particles interact with each other, so that for two clusters that are far away only a single interaction between their imaginary particles is required. As a result, the Fast Multipole Method offers even better asymptotic run-times than the Barnes-Hut algorithm and calculates the particle interactions in $O(n)$[Gre87]. However, the algorithm consists of multiple expensive steps to set up these imaginary particles, so for smaller numbers of particles the algorithm is quite slow. Since the algorithm is in $O(n)$ it will always perform better than the Barnes-Hut algorithm for large enough $n$, so for big particle simulations the Fast Multipole Method is preferred over the Barnes-Hut algorithm.

AutoPas is a library intended for solving N-body problems of short-range forces. They are called short-range, because they decay quickly with increased distance, such that they are basically zero for larger distances. This enables a very efficient algorithm, which focuses on only calculating interactions between particles that are close to each other. However, not all forces decrease as fast for higher distances, so applying the same algorithm to these forces would lead to very inaccurate results. In this thesis it will be analyzed, how the Fast Multipole Method can be used to allow AutoPas to also handle long-range forces and produce accurate results with good performance. The Fast Multipole Method will be implemented for the Coulomb potential, which is a simple example for a long-range force.

# 2. Theoretical Background

In this chapter the necessary background information for this thesis will be introduced. This includes basics about particle simulations and the Coulomb potential, as well as a detailed explanation of the Fast Multipole Method. Finally, an overview of the relevant aspects of the AutoPas library will be given.

## 2.1. Particle Simulation

In this thesis only pairwise particle simulations will be covered. That means, every particle applies a force to every other particle and this force only depends on the properties of these two particles. So in order to calculate the force that acts on one particle $P$, only interactions between $P$ and all other particles need to be considered.

The particles in a particle simulation are contained in a domain, that will usually be cuboid. For $n$ particles in the domain there are a total of $n(n-1)$ particle pairs, resulting in an $O(n^2)$ algorithm, if all forces are calculated exactly. This approach is called Direct Sum, since the forces between all other particles and $P$ are added up to the total force that acts on $P$.

## 2.2. Coulomb Potential

The Coulomb potential describes the electric potential induced by a point charge at a certain distance in vacuum[Tam79]:

$$\Phi = \frac{1}{4\pi\epsilon_0}\frac{Q}{r} \tag{2.1}$$

Here, $Q$ is the magnitude of the point charge, $r$ is the distance to the point charge, and $\epsilon_0$ is a physical constant called vacuum permittivity. In the particle simulation for the Coulomb potential every particle is a point charge with a certain charge. The potential at a particle depends on the sum of potentials induced by all other point charges. In a particle system with $n$ particles at positions $p_1, ..., p_n$ with charges $q_1, ..., q_n$ the potential at the $j$-th particle can be calculated with the following formula:

$$\Phi_j = \sum_{i \in \{1,..,n\} \setminus j} \frac{1}{4\pi\epsilon_0}\frac{q_i}{r_{ij}} \tag{2.2}$$

Here, $r_{ij}$ denotes the distance from the $i$-th particle to the $j$-th particle.

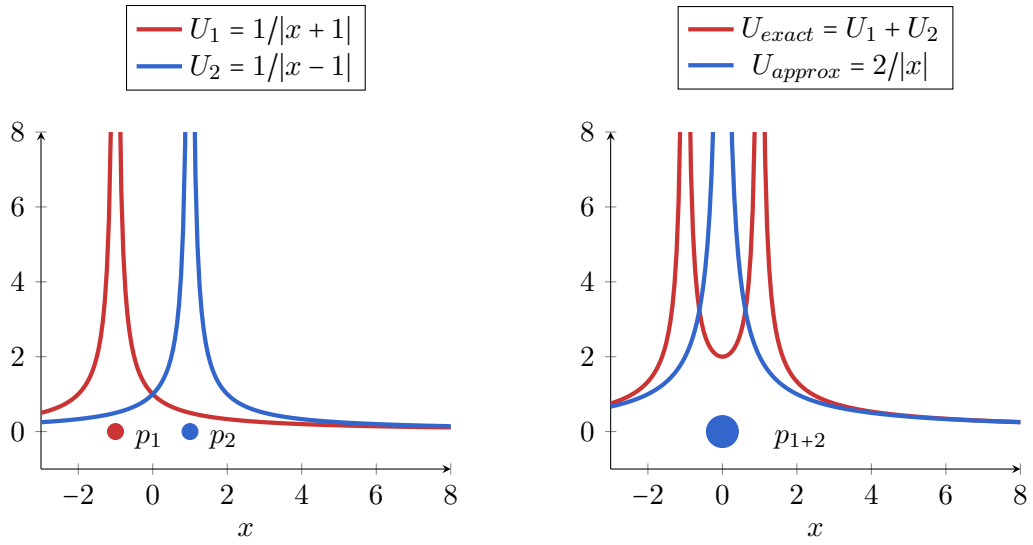
In order to calculate the potential at every particle with the Direct Sum algorithm, Equation 2.2 needs to be evaluated for every particle. Since Equation 2.1 has constant run-time, Equation 2.2 takes linear time. Evaluating this formula for every particle leads to quadratic complexity for the Direct Sum algorithm.

Good performance is very important for particle simulation algorithms, because they need to be able to simulate many particles to be useful. As a result, the Direct Sum algorithm is usually too slow and better algorithms have been developed.

Some potentials converge to zero very quickly for higher distances, such that only forces between nearby particles need to be evaluated. But there are also potentials like the Coulomb potentials that do not converge fast enough to zero. For these potentials even forces between far away particles need to be calculated. Here, the Fast Multipole Method offers an efficient way to calculate these forces.

## 2.3. Fast Multipole Method

The Fast Multipole Method (FMM) is an algorithm to calculate the forces between particles in linear complexity. It distinguishes between short range and long range interactions. Short range interactions are calculated directly like in the "Direct Sum" algorithm. For long range interactions clustered particles are combined to a bigger imaginary particle and then only interactions between these imaginary particles are calculated. The short range area around a particle is constant, so that the near field calculation for every particle takes constant time, resulting in linear time, if the near field is calculated for every particle.




(a) Potentials of two particles $p_1$ and $p_2$ in one dimension.

(b) Combined potential of the two particles using the approximation and the exact result in one dimension.

Figure 2.1.: Combining the potentials of two particles to the potential of a single imaginary particle.
Source: Based on a similar figure by Fabio Gratl[Gra17].

Figure 2.1 shows the potential around point charges in one dimension. In Figure 2.1a two different particles $p_1$ and $p_2$ induce the potentials $U_1$ and $U_2$. In order to get the potential that these two particles cause for a third particle, their potentials need to be added. Figure 2.1b shows the sum of the two potentials $U_{exact} = U_1 + U_2$ and how this sum

can be approximated using an imaginary particle $p_{1+2}$. In the area around this imaginary particle, the blue and red plot are very different, but further away from this particle both plots overlap.

The Fast Multipole Method is based on this technique. If clusters of particles are far away, their interactions can be approximated using imaginary particles. The distance between two clusters is large enough, if each cluster is contained inside its own sphere with radius $r$ and the distance between the centers of these spheres is at least $3r$[Gre87]. Then the clusters can interact with each other and the clusters are called "well separated". It is not necessary, that all particles inside a sphere also belong to the respective cluster.

In order to get a correct approximation, the algorithm needs to ensure that each particle interacts with every other particle exactly once. This can either happen in a short range interaction or in a long range interaction between two clusters, each containing one of the particles. To reduce the number of necessary calculations and therefore increase the performance of the algorithm, two particles should interact using big clusters rather than small clusters, if possible. This effect can be seen in Figure 2.2.

Because of that, the Fast Multipole Method is generally used with space-partitioning trees. Particles inside a cell of the tree are considered a cluster. Going from the root of the tree to the leaves ensures that bigger clusters are considered first. If a bigger cluster interacts with another cluster, their children must not interact with each other.

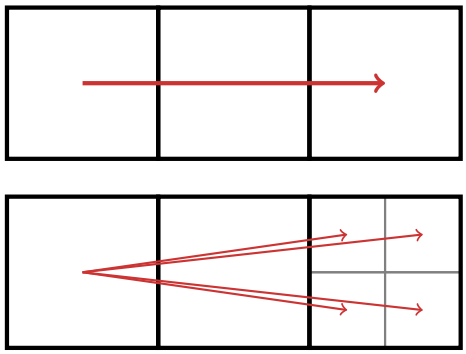


Figure 2.2.: Interaction with a large cluster(top) compared to interaction with four small clusters (bottom). While both cases cover the same areas, using larger clusters reduces the number of necessary interactions.

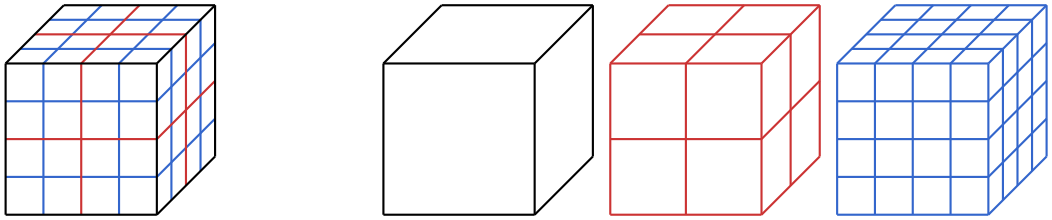
### 2.3.1. Octree

An octree is a simple space-partitioning tree for three dimension. It is created by recursively splitting cells into eight smaller child cells. Figure 2.3 depicts the three levels of an octree with a fixed depth of two. Level zero is the first and highest level of the octree and it contains only the root cell, which covers the entire domain. A cell that is not split into smaller cells corresponds to a leaf in the octree and will therefore be called leaf cell. In a general octree leaves may be on different levels, so that some parts of the domain are divided into smaller cells. If all leaves are on the same level, the octree will be called octree with fixed depth, because the tree is generated by recursively dividing the domain until a certain depth is reached. In Figure 2.3 the octree has a fixed depth of two, so all leaves are on level two.

Because of that, level two is called the leaf level and it is the deepest or lowest level in the tree.

If the domain is a cube and the tree has a fixed depth, the structure of the octree can be utilized to easily figure out, which cells need to interact with each other. Therefore for this example, it will be assumed that the domain is a cube and that all leaves are on the same level. This ensures that every level of the octree contains the entire domain in a regular grid.

For the sake of visual clarity, most figures will show two-dimensional examples.



(a) All levels of the Octree.

(b) Separated levels of the octree. The octree has a depth of two, so level two is the leaf level and the deepest or lowest level in the tree.

Figure 2.3.: An octree with a fixed depth of two and its three levels. The cell borders are black for level zero, red for level one, and blue for level two.

### 2.3.2. Algorithm

At first, the octree is built for the domain. As already said, the leaves will all be on the same level, so the octree has a fixed depth $d$ with $8^d$ cells on the last level. The Fast Multipole Method consists of two steps, called upward pass and downward pass[Gre87].

#### Upward Pass

The upwards pass starts at the leaf level and continues upwards the octree until the root is reached. At first, the so called multipole expansion is formed for every leaf cell of the octree. This multipole expansion represents the long range forces that are induced by the particles in the cell. The Particle-to-Multipole (P2M) operator is used to create the multipole expansion from the particles in a cell. This operator along with the other operators used in the Fast Multipole Method is explained in detail in Subsection 2.3.3.

With the Multipole-to-Multipole (M2M) operator the multipole expansions of child cells are combined to the multipole expansion of their parent cell. This process starts at the leaf level of the tree and goes up to the root, whose multipole expansion covers the entire domain. Now every cell in the octree has a multipole expansion.

#### Near Field List and Interaction List

Now that all multipole expansions are known, the cells can interact with other cells. Every cell has a list of nodes it needs to interact with called interaction list. These cells need to be well separated. Interactions between cells that are too close to each other must be covered by the near field calculation. For these cells the near field list is used.

Even though near field forces are only calculated on the leaf level, defining a near field list for higher levels helps to define the interaction list, which is needed on every level. A cell that is already in the near field list of a cell cannot be in its interaction list as well, because then the interaction between these cells would be calculated twice. So naturally, the near field list can be used to exclude cells from the interaction list.

As mentioned before, if the domain is a cube and all leaves are on the same level, the near field and interaction list of a cell can easily be determined.

The following definitions will be used for a cell $c$ in the octree:

$$A(c) = \{t : t \text{ is a neighbor of } c\} \tag{2.3}$$

$$C(c) = \{t : t \text{ is a child of } c\} \tag{2.4}$$

Two cells are considered neighbors, if they are on the same level in the octree and share at least one boundary point[BG]. Because of that, the neighbors of a cell are the cells on the same level in the $3 \times 3 \times 3$ cube around the cell. Additionally, every cell is considered a neighbor to itself, so a cell has up to $3 \cdot 3 \cdot 3 = 27$ neighbors. Then in an octree with fixed depth the near field list $N(c)$ and interaction list $I(c)$ of a cell $c$ with parent $p$ are defined by:

$$N(c) = A(c) \tag{2.5}$$

$$I(c) = \bigcup_{t \in N(p)} C(t) \setminus N(c) \tag{2.6}$$

$$\tag{2.7}$$

The near field list is simply the neighbor list of the cell. These are the cells that are so close, that they are not well separated, so their interactions need to be handled by the near field calculation. The interaction list of $c$ is the set of children of the neighbors of the parent of $c$ without the cells in the near field list of $c$. Obviously the cells in the near field list are handled by the near field calculation, so they must not be in the interaction list, otherwise interactions with these cells would be calculated twice. Cells that are further away than the neighbors of the parent of $c$ are far enough away from the parent, so that they would already be in an interaction list of one of the ancestors of $c$.

Figure 2.4 shows a common interaction list in the octree. The blue cell is the cell of interest and its interaction list consists of the green cells. The red cells are neighbors of the blue cell and therefore not in the interaction list, but in the near field list. The big white cells are already in the interaction list of the blue cell's parent.

This definition of the interaction list allows interactions between cells, that are not well separated according to the definition from Section 2.3. Two cells can interact, even though only a single cell is between them. If the side length of one cell is $a$, then the radius of the surrounding spheres is $r = \frac{\sqrt{2}}{2}a$ and the distance between the centers of the spheres is only $2a = 4r/\sqrt{2} \approx 2.828r$. The definition from Section 2.3 required a distance of at least $3r$. However, according to [Gre87], the desired error bounds are still fulfilled. Because of that and the simplicity of this definition of the interaction lists, it is used for cubic octrees.

The algorithm explained here can easily be adapted to other space-partitioning trees by adjusting the definitions of the near field and interaction lists. In Part II and Part III more advanced tree structures with more complicated near field and interaction lists will be used.

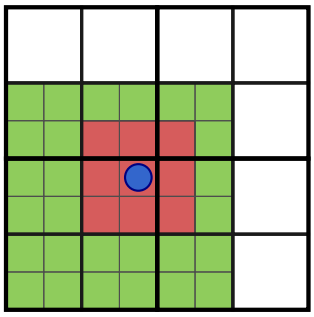


Figure 2.4.: Interaction list (green) of the blue node. The red cells are neighbors and therefore in the near field list. The white cells are not in the interaction list, because they are already in the interaction list of blue's parent.

#### Downward Pass

The downward pass starts with interactions between cells on the same level. The Multipole-to-Local (M2L) operator is used to generate the so called local expansion using all cells in the interaction list. After this operator has been applied to all cells, every cell has a local expansion, which contains influences from the particles in its interaction list.

With the Local-to-Local (L2L) operator the influences are propagated from parent cells to child cells, starting at the root. After that, every leaf has local expansions, which contain influences from all other leaf cells, except cells in its near field list.

With the Local-to-Particle (L2P) operator the far field forces are calculated for all particles. Finally the near field needs to be evaluated by calculating forces between each cell and its near field list. Combining near field and far field forces yields the result of the Fast Multipole Method.

### 2.3.3. Operators

In this section the exact operators for the Coulomb potential are introduced. The theory about the operators is explained in detail in [Gre87] and [BG]. The formulas presented in this thesis are based on [BG], though they are mathematically equivalent to the ones in [Gre87].
The operators shown here do not rely on the tree being an octree with all leaves on the same level. The only requirement is that the near field and interaction lists of every cell are set up correctly.

Every cell of the tree stores the coefficients $M_n^m$ and $L_n^m \in \mathbb{C}$, which describe the multipole and local expansion respectively. The order of expansion $p$ is a measurement for how accurate the approximation by the Fast Multipole Method is. The subscript $n$ ranges from 0 to $p$ and the superscript $m$ ranges from $-n$ to $n$. In order to get the coefficient for every valid combination of $n$ and $m$, the operators need to be evaluated for all these combinations.

Furthermore the potential calculated here assumes a simplified form of the Coulomb potential, which omits all constants and is therefore only $Q/r$. The final result from the Fast Multipole Method can be multiplied by $\frac{1}{4\pi\epsilon_0}$ to get the actual Coulomb potential as defined in Section 2.1.

### Spherical Harmonics

The operators make use of spherical harmonics, which require associated Legendre Polynomials. More information on the mathematical background of these function can be found in [Gre87]. Here it will only be shown how these functions can be calculated.

The associated Legendre polynomials $P_n^m(x)$ of degree $n$ and order $m$ are defined for $0 \leq m \leq n$ and $-1 \leq x \leq 1$. They can be expressed using derivatives of the ordinary Legendre polynomials[BG]:

$$P_n^m(x) = (-1)^m (1-x^2)^{m/2} \cdot \frac{d^m}{dx^m}(P_n(x)) \tag{2.8}$$

The ordinary Legendre polynomials $P_n(x)$ are the solution to Legendre's equation[DLMF]:

$$(1-x^2)\frac{d^2y}{dx^2} - 2x\frac{dy}{dx} + n(n+1)y = 0 \tag{2.9}$$

However these definitions are of little use for an implementation, since they do not offer a direct and efficient way to calculate the associated Legendre polynomials. For that purpose the following recurrence relation from [KP] will be used:

$$P_0^0(x) = 1 \tag{2.10}$$

$$P_m^m(x) = (-1)^m (2m-1)!!(1-x^2)^{m/2} \tag{2.11}$$

$$P_{m+1}^m(x) = x(2m+1)P_m^m(x) \tag{2.12}$$

$$P_n^m(x) = \frac{x(2n-1)P_{n-1}m(x) - (n+m-1)P_{n-2}^m(x)}{n-m} \tag{2.13}$$

The double exclamation mark $n!!$ denotes the double factorial that is defined as the product of all integers from 1 to $n$ that have the same parity as n. Since the associated Legendre polynomials $P_n^m(x)$ are only defined for $0 \leq m \leq n$, this recurrence relation is complete and leads to a recursive algorithm, as described in Algorithm 1.

**Algorithm 1:** Recursive algorithm to compute associated Legendre polynomials.

```
1 Function P(m,n,x):
2     if n = m then
3         if n = 0 then
4             return 1
5         return (−1)^m · (2 · m − 1)!! · (1 − x^2)^{m/2}
6     if n = m + 1 then
7         return x · (2m + 1) · P(m, m, x)
8     return ← (x · (2n − 1) · P(m, n − 1, x) − (n + m − 1) · P(m, n − 2, x)) · 1/(n−m)
```

The spherical harmonics of degree $n$ and order $m$ can then be expressed like this [BG]:

$$Y_n^m(\theta,\phi) = \sqrt{\frac{(n-|m|)!}{(n+|m|)!}} \cdot P_n^{|m|}(\cos\theta)e^{im\phi} \tag{2.14}$$

There are different definitions of the spherical harmonics with different normalization factors. The standard definition of the spherical harmonics includes a normalization factor of $\sqrt{(2n+1)/4\pi}$. In the operators for the Coulomb potential only the definition given in Equation 2.14 will be used, omitting the normalization factor of the standard definition[Gre87][BG].

### Particle-to-Multipole

The Particle-to-Multipole operator is the first operator and calculates the initial $M_n^m$ coefficients for every leaf cell. In all operators $c$ will be the cell, for which coefficients are computed and all positions are given in spherical coordinates relative to the center of $c$.

In this operator the cell $c$ has $k$ particles at spherical positions $(\rho_1, \theta_1, \phi_1), ..., (\rho_k, \theta_k, \phi_k)$. Then the coefficient $(M_c)_n^m$ for this cell can be calculated with Equation 2.15.

$$(M_c)_n^m = \sum_{j=1}^{k} q_j \cdot \rho^n \cdot Y_n^{-m}(\theta_j, \phi_j) \tag{2.15}$$

### Multipole-to-Multipole

The coefficient $A_n^m$ is used in multiple operators and is defined in Equation 2.16.

$$A_n^m := \frac{(-1)^n}{\sqrt{(n-m)!(n+m)!}} \tag{2.16}$$

For all operators the spherical position of the center of a cell $t$ is denoted as $(\rho_t, \theta_t, \phi_t)$. The operators use complex numbers and for that $i$ will refer to the imaginary unit, which solves $i^2 = -1$.

This operator is used to compute the coefficients $M_n^m$ for every non-leaf cell. Since it is part of the upward pass, the coefficients are calculated for lower levels first, so that they can be used in the next higher level. Here $c$ is a non-leaf cell with children $C(c)$. The coefficients $(M_c)_n^m$ of cell $c$ are calculated based on the coefficients of its children by summing up the influences from all child cells, as shown in Equation 2.17.

$$(M_c)_n^m = \sum_{t \epsilon C(c)} \sum_{j=0}^{n} \sum_{k=-j}^{j} \frac{(M_t)_{n-j}^{m-k} \cdot i^{|m|-|k|-|m-k|} \cdot A_j^k \cdot A_{n-j}^{m-k} \cdot \rho_t^j \cdot Y_j^{-k}(\theta_t, \phi_t)}{A_n^m} \tag{2.17}$$

### Multipole-to-Local

Since the local expansion is calculated in both the Multipole-to-Local and the Local-to-Local operator for the same cell, the coefficients that result in this step are labeled $(L^{m2l})_n^m$ and will be used in the next step to get the final coefficients $L_n^m$. For a cell $c$ the coefficients are computed using the coefficients $M_n^m$ of every cell in the interaction list $I(c)$ of $c$, as shown in Equation 2.18.

$$(L_c^{m2l})_n^m = \sum_{t \in I(c)} \sum_{j=0}^{p} \sum_{k=-j}^{j} \frac{(M_t)_j^k \cdot i^{|m-k|-|m|-|k|} \cdot A_j^k \cdot A_n^m \cdot Y_{n+j}^{k-m}(\theta_t, \phi_t)}{(-1)^j A_{n+j}^{k-m} \cdot \rho_t^{n+j+1}} \tag{2.18}$$

#### Local-to-Local

For the root cell, which does not have a parent, the local expansion is zero, since it does not have any interaction partners: $(L_{root})_n^m = 0$

For any other cell $c$ with parent $t$ the coefficients $L_n^m$ are calculated using Equation 2.19. This operator is used in the downward pass, so the coefficients are calculated for parent cells first, so that they can then be used to compute the coefficients in the child cells.

$$(L_c)_n^m = (L_c^{m2l})_n^m + \sum_{j=n}^{p} \sum_{k=-j}^{j} \frac{(L_{P(c)})_j^k \cdot i^{|k|-|k-m|-|m|} \cdot A_{j-n}^{k-m} \cdot A_n^m \cdot Y_{j-n}^{k-m}(\theta_t, \phi_t) \cdot \rho_t^{j-n}}{(-1)^{j+n} \cdot A_j^k} \tag{2.19}$$

#### Local-to-Particle

Finally the far field potential can be evaluated for a particle $P$ in cell $c$ at spherical position $(\rho, \theta, \phi)$. The local expansion $L_n^m$, which now contains influences from all cells, except cells in the near field list of $c$, is used to calculate the far field potential $\Phi_{far}$ with Equation 2.20.

$$\Phi_{far}(P) = \sum_{n=0}^{p} \sum_{m=-n}^{n} (L_c)_n^m \cdot Y_n^m(\theta, \phi) \cdot \rho^n \tag{2.20}$$

#### Near Field

The near field potential for a particle $P$ in cell $c$ with near field list $N(c)$ is computed using the Direct Sum algorithm as explained in Equation 2.2. Let $B(c)$ be the particles in cell $c$, $Q(P)$ the charge of a particle $P$ and $R(P_1, P_2)$ the distance between the particles $P_1$ and $P_2$. Then the near field potential $\Phi_{near}$ is given by Equation 2.21.

$$\Phi_{near}(P) = \sum_{t \in N(c)} \sum_{P' \in B(t) \setminus P} \frac{Q(P')}{R(P, P')} \tag{2.21}$$

The Fast Multipole Method concludes by combining far field and near field potential: $\Phi(P) = \Phi_{far}(P) + \Phi_{near}(P)$.

### 2.3.4. Complexity and Error Bounds

In theory, multipole and local expansions are series with an infinite number of addends. In order to get an algorithm, the number of addends is limited to the order of expansion. A higher order of expansion leads to a more accurate result at the cost of a longer run-time.

The error bounds of the operators from Subsection 2.3.3 are explained in detail in [Gre87]. The complexity depending on the order of expansion is in $O(p^4)$, as the Multipole-to-Multipole, Multipole-to-Local, and Local-to-Local operators each require four nested sums

with $O(p)$ addends per sum. The Fast Multipole Method is linear in the number of particles, though it requires a large number of particles to be faster than the Direct Sum algorithm. A more detailed analysis on the complexity is given in [Gre87].

## 2.4. AutoPas

AutoPas[1] is an open source particle simulation library written in C++. In AutoPas, an AutoPas-object is used as interface for particle simulations. This object has a container that stores particles in a cuboid domain. AutoPas offers functions to add particles to the domain and perform pairwise iterations on the particles. AutoPas is intended to be used with short range forces, that are so small for larger distances, that only interactions between nearby particles need to be calculated. Interactions between particles whose distance is larger than the so called cutoff radius will be ignored. Since the cutoff radius is usually a lot smaller than the domain size, this leads to a lot less interactions and potentially better performance. AutoPas uses auto-tuning to select the best container and algorithms to ensure fast run-times for different scenarios[GST+].

### 2.4.1. AutoPas Containers

AutoPas uses different containers to store the particles, which result in different algorithms for particle interactions. The containers are explained in detail in [GST+] and only a short introduction will be given here. The purpose of the containers is to offer efficient ways to find particle pairs with distances within the cutoff radius. The containers are visualized in Figure 2.5.

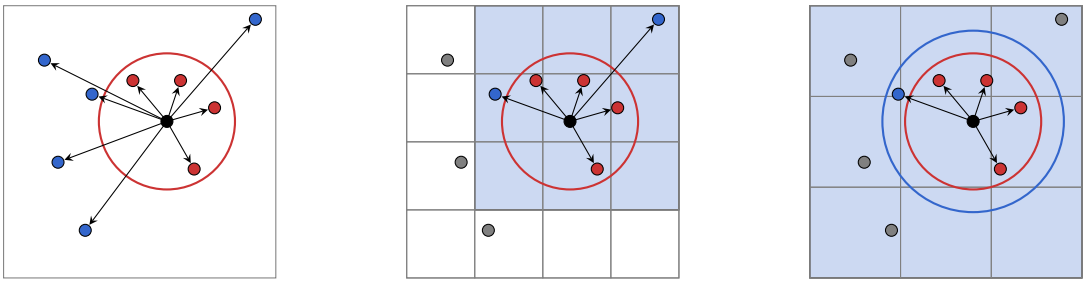
#### Direct Sum

The Direct Sum container is a basic container that stores all particles in the domain without any additional steps. As a result, the container offers no quick method to find out which particle pairs are within the cutoff radius. This leads to an $O(n^2)$ algorithm that iterates over all particle pairs, checks the distance for every pair and only performs calculations, if the distance is not larger than the cutoff radius.

#### Linked Cells

The Linked Cells container divides the domain into cells, such that the sides of the cells are at least as long as the cutoff radius. This means that for every particle only the containing cell itself and its neighbors can have particles inside the cutoff radius. Because of this, only a constant number of cells need to be considered for every particle. For instance if the cutoff radius is 1 and the domain size is $4.8 \times 8 \times 6.6$, each cell will be $1.2 \times 1 \times 1.1$. This can be seen in Figure 2.5 (b). Here only the distances to particles in blue cells need to be evaluated.

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



(a) Direct Sum (b) Linked Cells (c) Verlet Lists

Figure 2.5.: Interactions of the black particle with other particles. The cutoff radius is shown by the red circle and the verlet-skin radius by the blue circle. For particles connected by an arrow (red and blue particles), the distance to the black particle is evaluated. However only the red particles are inside the cutoff radius, so that forces are only calculated for the red particles. There are no interactions with gray particles. The blue cells are used to find particles that may be inside the cutoff radius in (b) and inside the verlet-skin radius in (c).
Source: Based on a figure from [GST+].

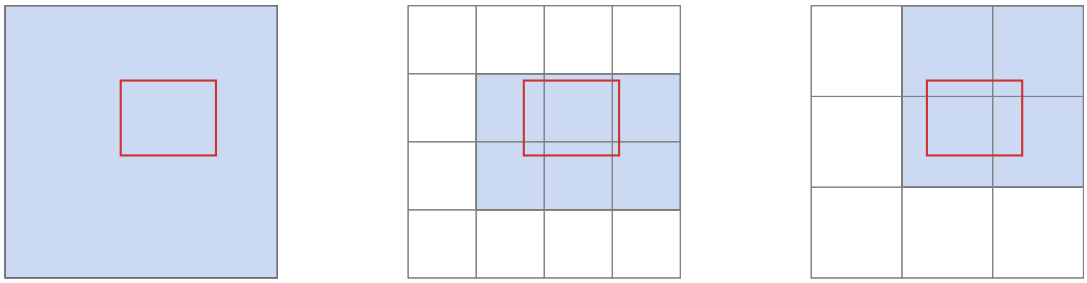
### Verlet Lists

The cuboid of the neighbor cells from the Linked Cells container is still much larger than the sphere, which is generated by the cutoff radius. If $r$ is the cutoff radius, the volume of the cuboid is at least $(3r)^3 = 27r^3$, whereas the sphere only has a volume of $4/3\pi r^3 \approx 4.19r^3$. As a result, the distance to a lot of particles will be calculated, that are not inside the cutoff radius. The Verlet Lists container tries to solve this problem by keeping a list of partner particles for every particle. However, as particles move, these lists would need to be rebuilt, resulting in a lot of overhead. Because of that, a verlet-skin radius is introduced to create a bigger sphere, so that the partner lists also include particles that are slightly further away than the cutoff radius. As a result, the lists do not need to be rebuilt as frequently, but the sphere generated by the verlet-skin radius is still much smaller than the cuboid from the Linked Cells container, so that less distances need to be evaluated. The partner lists are built using the same technique as in the Linked Cells containers. Here every cell needs to have longer side lengths than the verlet-skin radius and then again only the neighbor cells and the cell itself need to be considered for the particles inside the verlet-skin radius. Figure 2.5 (c) shows the verlet-skin as blue circle.

## 2.4.2. Iterators

All AutoPas containers offer iterators to iterate over all the particles in them. There are also region iterators that can be used to iterate over the particles within a cuboid part of the domain. Finding the particles within this area is similar to finding particle pairs with a distance smaller than the cutoff radius in that the performance depends on the container used.

The general approach is, that the search region will be expanded to the smallest accessible area that fully covers the search region and then the algorithm will search through all particles in that accessible area and only return the particles, which are inside the search region. This means, the larger the expanded area is compared to the search region, the



(a) Direct Sum (b) Linked Cells (c) Verlet Lists

Figure 2.6.: Region iterators for the different container types. The red rectangle indicates the search region. The smallest accessible region that fully contains the search region is shown by the blue area.

more time will be spent iterating over particles that are not in the search region. Figure 2.6 visualizes the search region compared to the smallest accessible region for the different container types.

For the Direct Sum container, the smallest accessible region is the entire domain, meaning that the region iterator will always go through all particles and then return the ones, which are inside the search region.

For Cell based containers such as the Linked Cells and Verlet Lists containers, the iterator will only consider particles in cells that are covered by the region. The algorithm iterates over all cells to find which cells are covered by the region and will then iterate over the particles in these cells to find the particles within the search region. Since the regions usually do not match with the cell borders, a cell might be partially covered by the region, such that only some of its particles are in the search region. As a result, the algorithm might still iterate over some particles, that are not in the search region, but a lot less compared to the Direct Sum container. By overlapping the search regions with cell borders the algorithm will only iterate through particles which are also in the search region, resulting in optimal performance.

Especially for very small search regions, the smallest accessible areas will generally be larger, so that the particle access becomes slow. The Fast Multipole Method needs to access particles within its leaf cells, so good performance for iterating over particles in certain regions is required. Because of that, the Linked Cells and Verlet Lists containers will be used for the Fast Multipole Method here. In Part III the leaf cells will only contain single AutoPas container cells, so that the search region overlaps with the cells, resulting in efficient particle access.

# Part II.

# External Fast Multipole Method

# 3. Implementation

In this chapter an implementation of the Fast Multipole Method is presented that uses AutoPas only to store and access particles. The type of the AutoPas container, which is used to store the particles, does not matter for the most part, since the particles are only accessed with region iterators, which exist for every container. However the region iterators will be considerably faster for the Linked Cells container compared to the Direct Sum container, so here the Linked Cells container will be preferred. The AutoPas container will be considered as blackbox, since its structure is not relevant here.

The particles that are stored in the AutoPas container have a certain charge and represent point charges in an electric field. This implementation builds an octree on top of the container and each cell has information about which area of the domain it covers. Using that information, the particles in that area can be accessed with the region iterators of AutoPas.

In order to work well with arbitrary particle distributions an adaptive octree is used. Similar to the octree from Subsection 2.3.1, the adaptive octree will only work for cubic domains. Due to the use of an octree, this implementation will also be referred to as octree implementation.

In this chapter it will be explained how the octree is generated and how the near field and interaction lists are built for it.

## 3.1. Adaptive Octree

The adaptive octree used here will divide areas with higher particle densities into more and smaller leaves with higher depth, so that the near field calculations in these areas do not become overly expensive. The near field calculations are done using the naive $O(n^2)$, however here the $n$ is only the number of particles in a cell and its near field list, which is usually so small compared to the total number of particles, that the algorithm is still fast.

For non-uniform particle distributions a big portion of the particles could be concentrated within a small area, such that the $n$ in that part is much larger and the $O(n^2)$ algorithm applied to that area will slow down the algorithm.

The adaptive octree tries to prevent that by having much more and smaller leaves in these areas, so that the $n$ in the $O(n^2)$ part of the algorithm can never be too big.

### 3.1.1. Tree Generation

Algorithm 2 describes how the adaptive octree is generated. The current cell is divided recursively into eight equally sized child cells, as long as the number of particles in the current cell exceeds `maxParticlesPerCell`. The function is then applied to all eight child cells.

Cells with at most `maxParticlesPerCell` particles will not be divided further, making them leaf cells. By doing that, the number of particles in a cell is limited and the number of

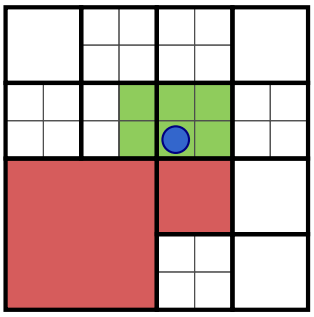


Figure 3.1.: The marked cell does not have neighbors on all sides that are on the same level. The red cells are leaf cells and neighbors, but not on the same level.

particles inside the near field list of a cell will also not be that high.

**Algorithm 2:** Adaptive Octree

```
1 Function split_cell(cell, maxParticlesPerCell):
2     if cell.numberOfParticles > maxParticlesPerCell then
          // Create 8 children
3         for i ← 0 to 7 do
4             child ← create_cell (i)
5             split_cell (child, maxParticlesPerCell)
```

The implementation of the adaptive octree offers two additional parameters to set the minimum and maximum depth for all cells. If a cell's depth is below the minimum depth, it will be divided regardless of the number of particles in it. Similarly, if a cell's depth reached the maximum depth, it will not be divided further, even if it contains more than `maxParticlesPerCell` particles. These parameters can also be used to generate an octree with fixed depth, by setting both these parameters to the desired depth.

### 3.1.2. Neighbor List

As this is an adaptive octree a cell may not have a neighbor cell on the same level. As seen in Figure 3.1, the marked cell does not have neighbors on the same level on all sides.

If a cell does not have a neighbor on the same level, the lowest level ancestor of the non-existing cell will be used as neighbor instead. In Figure 3.1 the blue cell does not have same level lower left, lower middle, and lower right neighbors. The red cells are the lowest level ancestors of the non-existing same level neighbors, so they will be used as neighbors instead. The lower right red cell is ancestor of both the lower middle and lower right same level neighbors, but it is only once in the neighbor list, because the same area of the domain must not be included several times in the same list. Same as for the octree with fixed depth, every cell is always a neighbor of itself.

### 3.1.3. Near Field List and Interaction List

Since neighbors do not necessarily have the same level, the interaction list as defined in Equation 2.7 cannot be used here. Consider the blue cell in Figure 3.1. The upper right, white cell is a neighbor of the blue cell's parent, but it has no children that can be added to the blue cell's interaction list. Additionally, the upper right cell is so big, that the blue cell cannot interact with it directly, so it cannot be in the interaction list of the blue cell. This means that the interaction between these two cells must be covered by the near field calculation. Here it becomes obvious, that the near field calculation cannot only consider direct neighbors.

Algorithm 3 describes how the near field and interaction lists are created for the adaptive octree. Even though the near field is only used directly on the leaf level, it helps in order to define the interaction list. For every cell $p$, the near field list covers the area, that is so close, that no interactions on this level are possible. The rest of the domain is far enough away, so that interactions with it are either handled by $p$ or its ancestors. This means the descendants of $p$ only need to handle interactions with the area covered by the near field list of $p$, since interactions with the rest are already handled.

For every child cell $c$ of $p$, the area of the near field list of $p$ must be partitioned into the interaction list of $c$ and the near field list of $c$. Then interactions with this area are handled by $c$ and its descendants. This has the effect, that the area covered by the near field list gets smaller, the deeper the cell is in the tree, since in every level some part of the near field list is split off to build the interaction list. Finally, on the leaf level the area covered by the near field list is very small, such that most interactions in the domain are handled by far field interactions.

The algorithm works similar to the non-adaptive octree. The children of the cells in the near field list of $p$ are considered and added to the interaction list of $c$, if they are not already in the near field list of $c$ (Algorithm 3 line 9). As a result, every child cell is either in the interaction list or the near field list of $c$. However, it may happen, that cells in the near field list of $p$ do not have any children. In that case the cell is directly added to the near field list of $c$ (Algorithm 3 line 11).

On the leaf level there are no children, so interactions with the near field list cannot be handled by the descendants of the cell. Instead, the interactions are handled by the near field calculation.

Since the interaction list and the near field list of $c$ depend on the near field list of the parent $p$, the algorithm is used on the root first and then recursively on all children. For the root cell, the neighbor list contains only the root cell itself, so also the near field list contains the root cell. Because of that, all interactions in the domain will be handled by the the descendants of the root cell.

**Algorithm 3:** Generation of the near field and interaction list.

```
1  Function init_cell(cell):
2      cell.nearFieldList ← cell.neighborList
3      cell.interactionList ← ∅
       // The root cell has no cell in its near field or interaction
          list.
4      if cell.depth > 0 then
5          for parentNearFieldCell ∈ cell.parent.nearFieldList do
               // If the parent's near field cell has children, they can be
                  added to the interaction list.
6              if parentNearFieldCell.children ! = ∅ then
7                  for child ∈ parentNearFieldCell.children do
                       // Cells in the near field list are excluded from the
                          interaction list.
8                      if child ∉ cell.nearFieldList then
9                          cell.interactionList ← cell.interactionList ∪ child
10             else
                   // The cell could not be added to the interaction list,
                      so it must be in the near field list instead.
11                 cell.nearFieldList ← cell.nearFieldList ∪ parentNearFieldCell
```

## 3.2. Optimizations

All five operators can directly be implemented using nested loops. The operators will be surrounded by two additional loops in order to calculate the coefficients for every $n$ from 0 to $p$ and every $m$ from $-n$ to $n$.

However there are a lot of optimizations that can be made to improve performance. Functions like factorial and double factorial and coefficients like $A_n^m$, which are frequently used in the operators can be cached.

The same cannot easily be done for spherical harmonics or associated Legendre polynomials, as they have continuous parameters. If Algorithm 1 is used to calculate $P_n^m$, it also has to calculate $P_{n-1}^m$ and $P_{n-2}^m$ for $n \geq m+2$ and $P_{n-1}^m$ for $n = m+1$. This means it will always calculate $P_n^m(x), P_{n-1}^m(x), ..., P_m^m(x)$.

This can be used to improve the run-time of the operators. If the order $m$ and $x$ stay constant and only the degree $n$ changes, the algorithm can be used to calculate $P_n^m(x)$ for the maximum $n$ and it will automatically also compute the result for all other degrees. So instead of using the algorithm whenever spherical harmonics are used, the algorithm is used earlier to cache the result for a certain order $m$ and a certain $x$.

Algorithm 4 shows how the Multipole-to-Local operator can be implemented using nested loops. The innermost loop body is executed for many different combinations of $(n, m, j, k)$. Since the innermost loop iterates over k, only the order of the spherical harmonics changes in the last loop. This means, the optimization from above cannot be used here. However, the order, in which the coefficients are summed up and for which combination of $m$ and $n$ the coefficients are calculated, can be changed.

**Algorithm 4:** Using for loops to implement Multipole-to-Local.

```
1  Function m2l(c,p):
2      for n ← 0 to p do
3          for m ← -n to n do
               // Initialize the coefficients with 0.
4              (L_c^{m2l})_n^m ← 0
5              for t ∈ I(c) do
6                  for j ← 0 to p do
7                      for k ← -j to j do
8                          product ← (M_t)_j^k
9                          product ← product · Y_{n+j}^{k-m}(θ_t, φ_t)
10                         ...
11                         (L_c^{m2l})_n^m ← (L_c^{m2l})_n^m + product
```

In Algorithm 5 the loop order was changed, such that the order of the spherical harmonics no longer depends on the two inner loops. Because of that the expensive function to calculate the associated Legendre polynomial can be used early on to cache the values for this order and a certain $\theta$ and $\phi$, so that in the innermost loop body only cached values need to be accessed instead of calculating them all the time. The cache is built using degree $2p$, because

in the innermost loop a degree of $n + j$ is required and both $n$ and $j$ are at most $p$.

**Algorithm 5:** Using for loops to implement Multipole-to-Local.

```
1  Function m2l(c,p):
2      for t ∈ I(c) do
3          for m ← -p to p do
4              for k ← -p to p do
                   // The order of the spherical harmonics k − m and its
                      parameters θ_t, φ_t will not change in the inner loops.
                   // Here the cache for Y_{2p}^{k−m} can be built, so it can be
                      accessed in the innermost loop.
5                  for n ← |m| to p do
                       // Initialize the coefficients with 0.
6                      (L_c^{m2l})_n^m ← 0
7                      for j ← |k| to p do
8                          product ← (M_t)_j^k
                           // Now only the cache is accessed.
9                          product ← product · Y_{n+j}^{k−m}(θ_t, φ_t)
10                         ...
11                         (L_c^{m2l})_n^m ← (L_c^{m2l})_n^m + product
```

# 4. Results

In this chapter the results of the implementation are analyzed, focusing on the performance and accuracy of the implementation. All tests use uniformly distributed particles, unless specified otherwise.

## 4.1. Performance

The run-times of the near field and far field calculations of the Fast Multipole Method are independent for the most part. The far or near field could take significantly more time than the other one, depending on the input parameters. For that reason not only the total run-time is of interest, but also how the run-time is split into near field and far field calculations. The parameters that are used in the implementation are the number of particles $n$, the order of expansion $p$ and also the depth of the octree. For the adaptive octree the depth of the octree depends on the particle distribution and the maximum number of particles per cell. If the octree is not adaptive, all leaves have the same depth $d$.

### 4.1.1. Order of Expansion

The order of expansion is only used for the far field calculations and will therefore not affect the run-time of the near field calculations. A higher order of expansion results in higher accuracy at the cost of a longer run-time. The operators that were introduced in Subsection 2.3.3 use four nested loops, with each loop having $O(p)$ iterations. As a result, the far field calculations are in $O(p^4)$, which is also the complexity shown in [Gre87].

In order to see, whether the far field calculations of this implementation are in $O(p^4)$, the far field run-time has been measured for different orders of expansion. In the first test, an octree with fixed depth $d = 3$ was chosen with $n = 4000$ particles in the domain. The result of this test can be seen in Figure 4.1. The function $p^4/50$ was added to show, that the run-time does not increase faster than $p^4/50$.

In the second test, the octree depth was reduced to two in order to get a significantly faster run-time. This made it possible to use much higher orders of expansions and still have the program finish within a reasonable time. Since the far field run-time is shorter due to less interacting cells in the octree, it is only compared to $p^4/1000$ this time. In Figure 4.2, it can be seen that the run-time does not grow faster than $p^4/1000$.

In both tests, the far field run-time was below $cp^4$ for some $c$ and $p > 12$, indicating that this far field implementation is in $O(p^4)$. This is expected, considering that the operators were implemented with four nested loops and $O(p)$ iterations per loop.

As already mentioned, the order of expansion does not affect the near field calculation. This can easily be seen in Figure 4.3, which shows the near field calculation time for different orders of expansion. All run-times deviate at most 4% from the average run-time across the different orders of expansion.

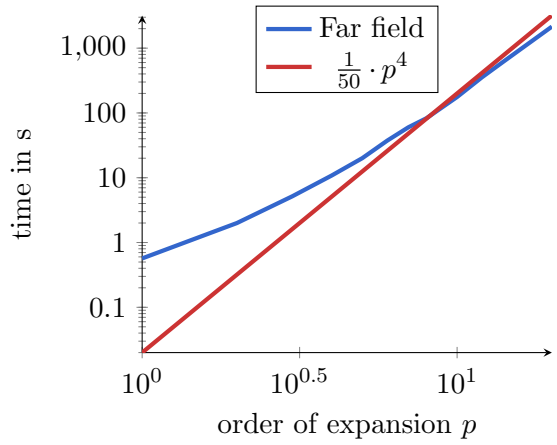



Figure 4.1.: Far field run-time depending on the order of expansion for 4000 particles in a non-adaptive octree with depth 3.

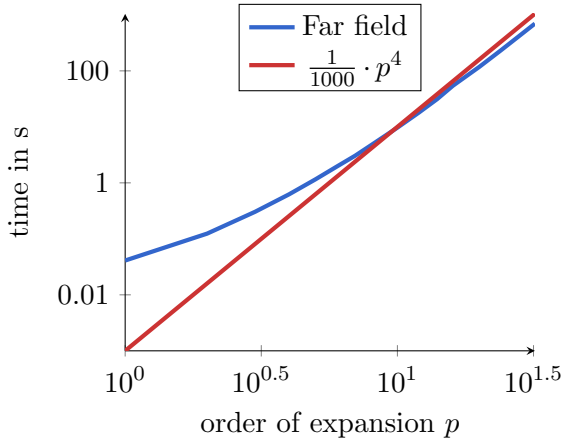



Figure 4.2.: Far field run-time depending on the order of expansion for 1000 particles in a non-adaptive octree with depth 2.

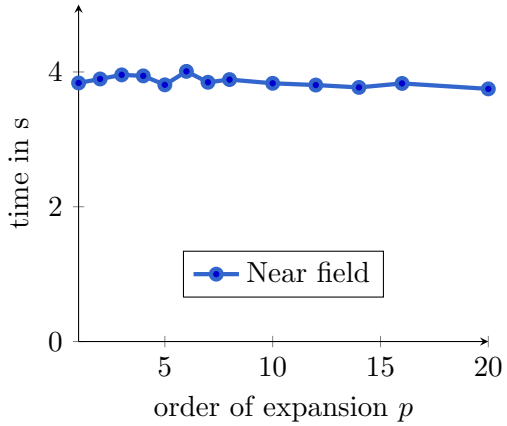



Figure 4.3.: Near field run-time depending on the order of expansion for 4000 particles in a non-adaptive octree with depth 3.

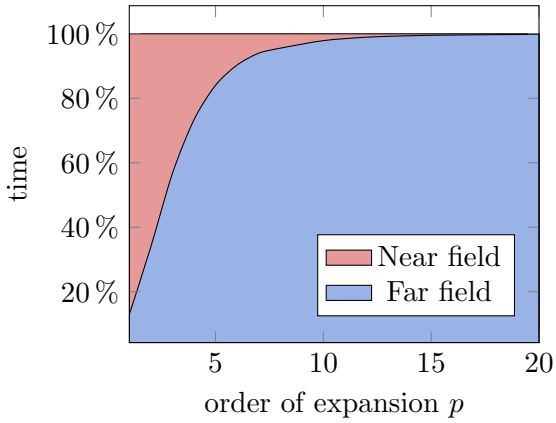



Figure 4.4.: Distribution of run-times depending on the order of expansion for 4000 particles in a non-adaptive octree with depth 3.

Since the near field run-time does not depend on the order of expansion, whereas the far field run-time increases with higher orders of expansion, it is obvious that at some point the near field run-time becomes so small compared to the far field run-time, that it is no longer relevant. This can be seen in Figure 4.4. For $n = 4000$, $d = 3$, and an order of expansion of 14 or more the far field calculation takes up more than 99% of the total run-time. Obviously for larger numbers of particles the near field becomes more expensive, so that balanced run-times can be achieved even for higher orders of expansion.

The far field calculation consists of five operators, as explained in Subsection 2.3.3. However they do not take the same amount of time to finish. In fact, most of the execution time is spent on the Multipole-to-Local operator. In Figure 4.5 this distribution across the operators can be seen. The Multipole-to-Local operator is not displayed in the graphic, otherwise it would be too big compared to the other operators, making them hard to see. Since the remaining time is spent on the Multipole-to-Local operator, it is easy to see, that over 97% of the run-time is spent on the Multipole-to-Local operator, except for very small orders of expansion. Therefore this operator is critical for good performance.

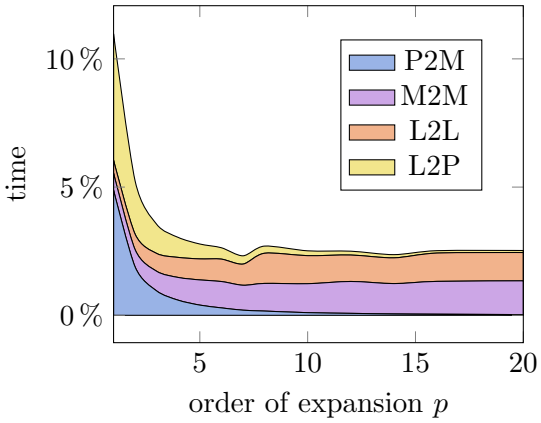



Figure 4.5.: Distribution of run-times across the different operators depending on the order of expansion for 4000 particles in a non-adaptive octree with depth 3. The rest of the run-time is spent in the Multipole-to-Local operator.

### 4.1.2. Tree Depth

Compared to the order of expansion, the depth of the tree does not have a straight forward effect on the performance. A low depth of the tree means less far field interactions, whereas a high depth of the tree results in more far field interactions. Every interaction that is not covered by the far field must be done in the near field. Both too many far field interactions and too many near field interactions lead to bad performance.

This can easily be seen by giving extreme examples. A non-adaptive octree will be assumed, because it is easier to construct examples with it. On the one hand, if there are 100000 particles in the domain, but the octree only has a depth of two, there are a total of $8^2 = 64$ leaf cells. An inner cell has $3 \cdot 3 \cdot 3$ near field cells, so for these cells $27/64 \approx 42.2\%$ of the interactions will be covered by near field calculations, which are in $O(n^2)$. On the other hand if there are only 1000 particles in the domain, it is obvious that the naive $O(n^2)$ implementation would still be quite fast. However, the operators in the far field interactions of the Fast Multipole Method mostly depend on the depth of the tree, so a high tree depth would make the far field interactions take a lot of time.

#### Non-Adpative Octree

The first tests were done using a non-adaptive octree, so all leaves are on the same level. This creates a fairly predictable far field phase, because its run-time mostly depends on the number of cell interactions in the operators, which obviously depends on the depth of the tree. In four tests different numbers of particles and orders of expansion were used. The exact parameters for every test case can be seen in Table 4.1.

A depth of one or two does not make much sense in the context of the Fast Multipole Method, because there will not be any far field interactions. However the tests have still been performed with such small tree depths, to show what happens in that case. The results are displayed in Figure 4.6 and Figure 4.7 for 4000 and 40000 particles respectively.

The far field has a relatively simple development in that its run-time increases, as the tree gets deeper. With more cells in the tree, the operators need to be applied to more cells.

| Test Case | Number of particles (n) | Order of expansion (p) |
|---|---|---|
| 1 | 4000 | 4 |
| 2 | 4000 | 8 |
| 3 | 40000 | 4 |
| 4 | 40000 | 8 |

Table 4.1.: Overview of the four test cases.

Since this is an octree, increasing the depth by one results in around eight times as many cells in the tree. So even if the depth is only increased by one, the far field calculation takes much longer.

The near field is more complicated. The general idea is that with a higher tree depth more interactions are covered by the far field, so they are no longer calculated in the near field, making the near field faster. However this is not completely true in this implementation, as the near field calculation with tree depth $d = 5$ takes actually longer than with $d = 4$.

In order to get a better understanding of what is happening, the results will be explained for every depth $d$ of the octree .

**d=0** There is only a single cell in the octree, so all interactions are calculated directly in $O(n^2)$. The quadratic complexity results in a very long run-time for large numbers of particles. This is basically the same as performing a Direct Sum algorithm on the entire domain. The overhead of the Particle-to-Multipole and Local-to-Particle operators is so small compared to the near field run-time, that the algorithm is only slightly slower than a pure Direct Sum algorithm.

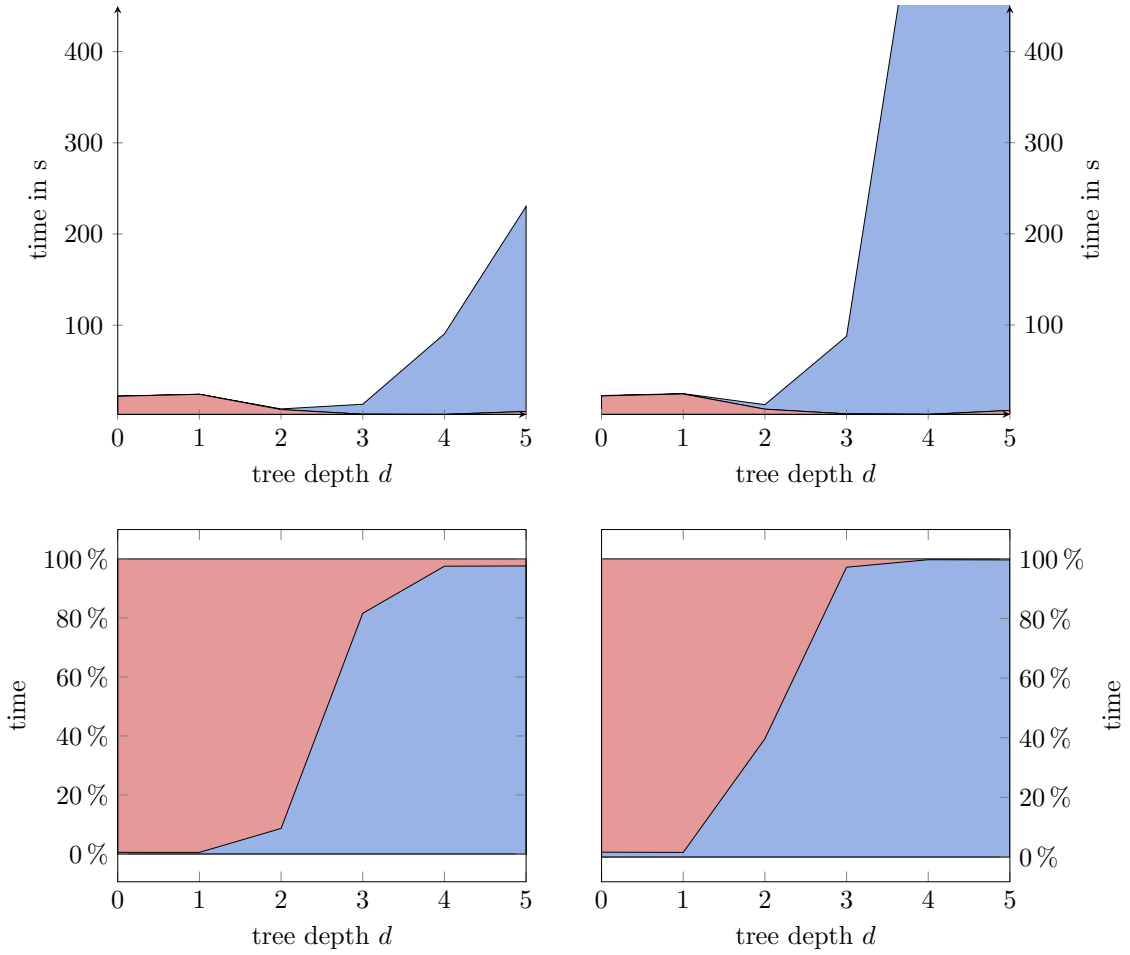
**d=1** There are eight cells on the leaf level of the octree. However they are all neighbors, such that still all calculations are done in $O(n^2)$. The additional cells add some overhead and result in less efficient particle access, resulting in worse performance than for $d = 0$.

**d=2** This is the first time far field interactions are done, so the performance improves. Especially for a large number of particles, the run-time is noticeably shorter.

**d=3 and d=4** The near field keeps getting faster, but the far field becomes a lot slower. If the number of particles is very high, the performance gain in the near field outweighs the costs of the far field.

**d=5** The near field run-time increases compared to $d = 4$. There are now $8^5 = 32768$ leaf cells, such that iterating over them adds a lot of overhead, which results in worse performance. Additionally the leaf cells are very small and accessing particles in regions smaller than AutoPas container cells is inefficient, as explained in Subsection 2.4.2.

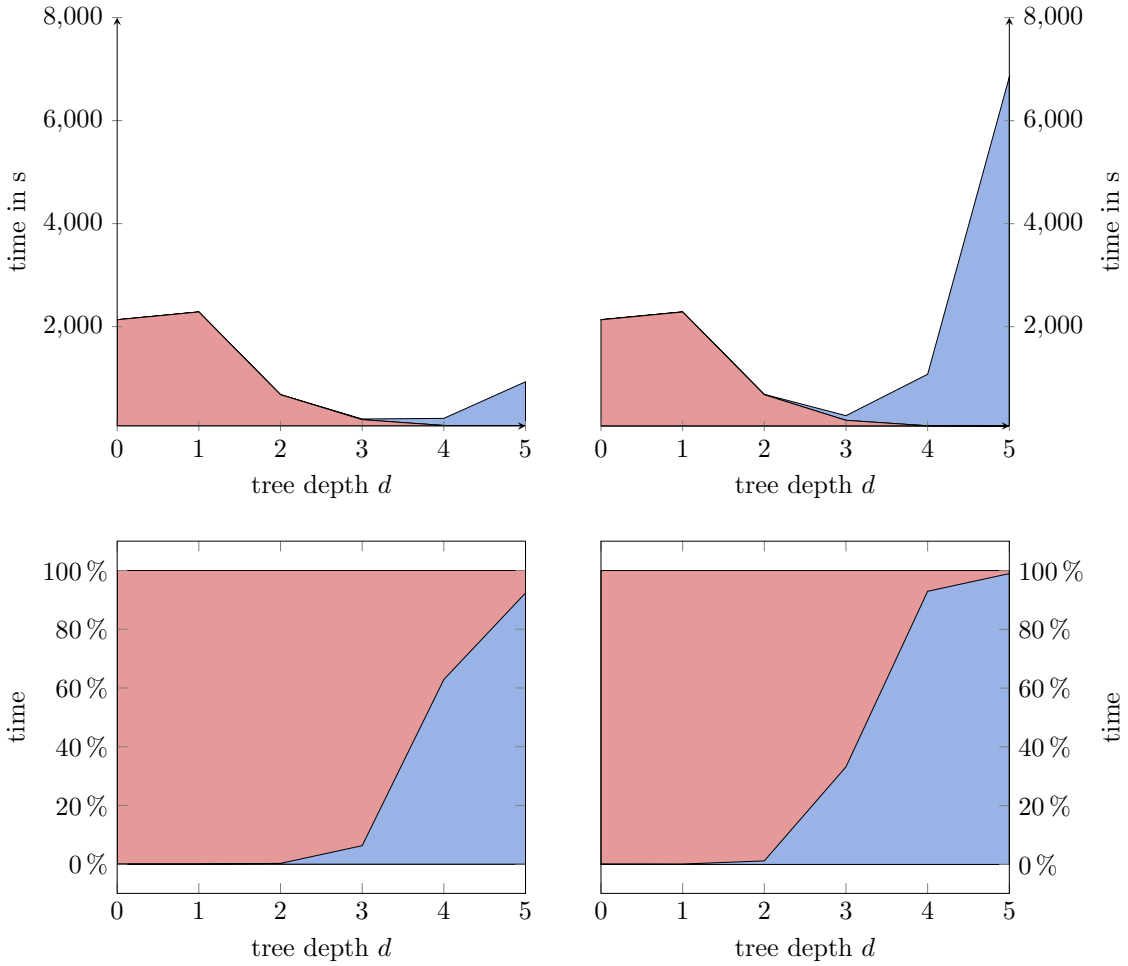
In Subsection 4.1.1 it was shown, that the majority of the far field execution time is spent on the Multipole-to-Local operator. The test was performed using an octree with a depth of three. Figure 4.8 and Figure 4.9 show the distribution of the far field run-time across the operators for different tree depths. It is apparent, that for a tree depth of three or more the Multipole-to-Local operators takes up at least 90% of the far field run-time.




(a) $n = 4000$, $p = 4$

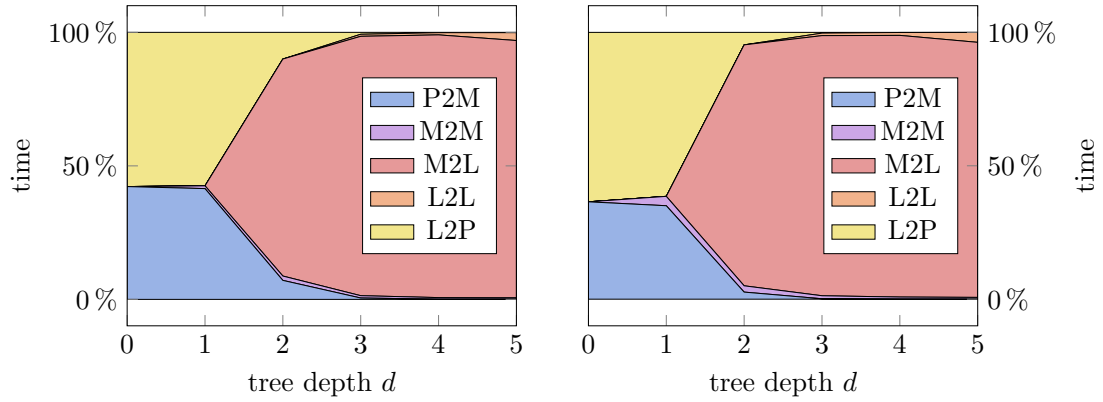
(b) $n = 4000$, $p = 8$

Figure 4.6.: Absolute and relative far field (blue) and near field (red) run-times for 4000 particles in an octree with different depths using an order of expansion of 4 (left) and 8 (right).



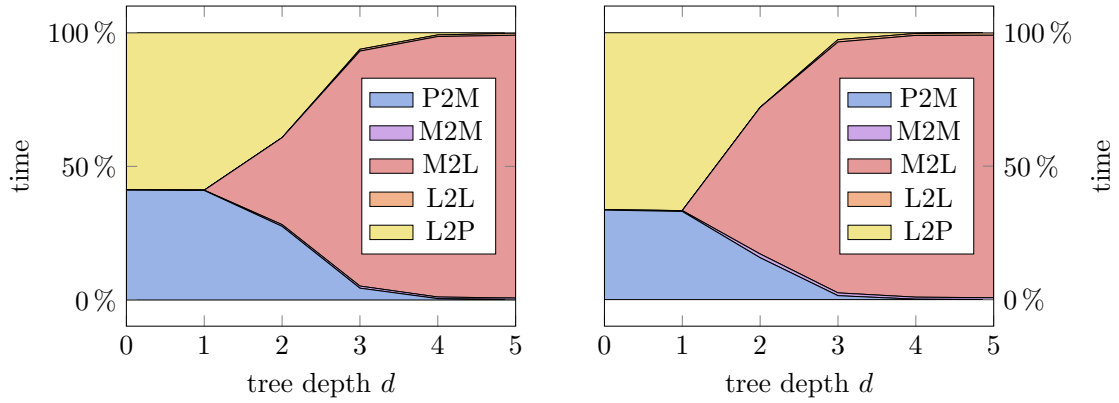

(a) $n = 40000$, $p = 4$ (b) $n = 40000$, $p = 8$

Figure 4.7.: Absolute and relative far field (blue) and near field (red) run-times for 40000 particles in an octree with different depths using an order of expansion of 4 (left) and 8 (right).




(a) $n = 4000$, $p = 4$ (b) $n = 4000$, $p = 8$

Figure 4.8.: Distribution of run-times across the different operators depending on the tree depth for 4000 particles. The order of expansion is 4 (left) and 8 (right).




(a) $n = 40000$, $p = 4$ (b) $n = 40000$, $p = 8$

Figure 4.9.: Distribution of run-times across the different operators depending on the tree depth for 40000 particles. The order of expansion is 4 (left) and 8 (right).

Considering that the fast-multipole-method is best used for large-scale simulations with a high number of particles, it can be assumed that the number of particles and also the tree depth will be high. Together with a reasonably big order of expansion for at least decent accuracy, the Multipole-to-Local operator will generally make up 95% or more of the far field run-time.

A direct consequence is that optimizations to the other operators can only have limited effect, as even if their run-times would be reduced to zero, the total far field run-time would only be reduced by 5%. This means, optimizations should focus on the Multipole-to-Local operator and the near field.

### Adaptive Octree

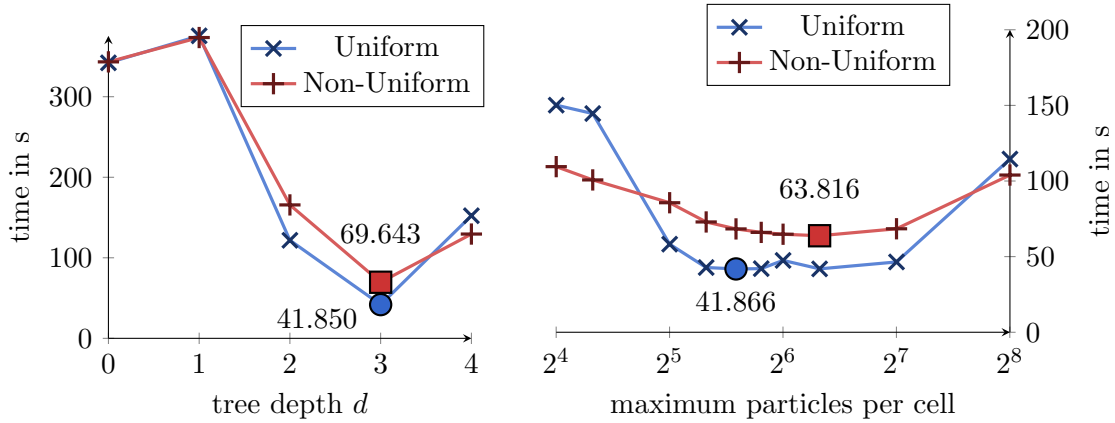
So far, only uniformly distributed particles have been used. The big advantage of using an adaptive octree is the better performance for highly non-uniform distributions of particles. In the non-adaptive octree, some leaf cells may have much more particles than others, resulting in long near field calculations for these cells.

For uniform particle distributions, the adaptive octree turns into an octree with fixed depth for a large number of particles. Due to the law of big numbers, a uniform distribution of many particles will lead to roughly the same number of particles in every cell, so the adaptive tree generation will create a tree very similar to the octree with fixed depth. The adaptive tree generation algorithm also takes some time, since for every cell the number of particles in it are counted to decide, whether the cell needs to be divided further. Because of that, for a uniform distribution of particles, the non-adaptive octree will be slightly faster.

Since the non-adaptive and adaptive octree use completely different parameters, it is not easy to compare the two approaches. As already mentioned, for a uniform distribution the adaptive algorithm will create a tree that is very close to the tree generated by the non-adaptive method. For example, if there are $n = 16000$ particles and the non-adaptive octree has depth $d = 3$, then this leads to 512 leaf cells and $16000/512 = 31.25$ average particles per leaf cell. Using a value slightly above that, for instance 40, for the maximum number of particles per cell will cause the adaptive algorithm to create a tree very similar to the non-adaptive algorithm. As a result, the run-times in this case are very similar and only the more expensive tree generation for the adaptive algorithm makes it slower.

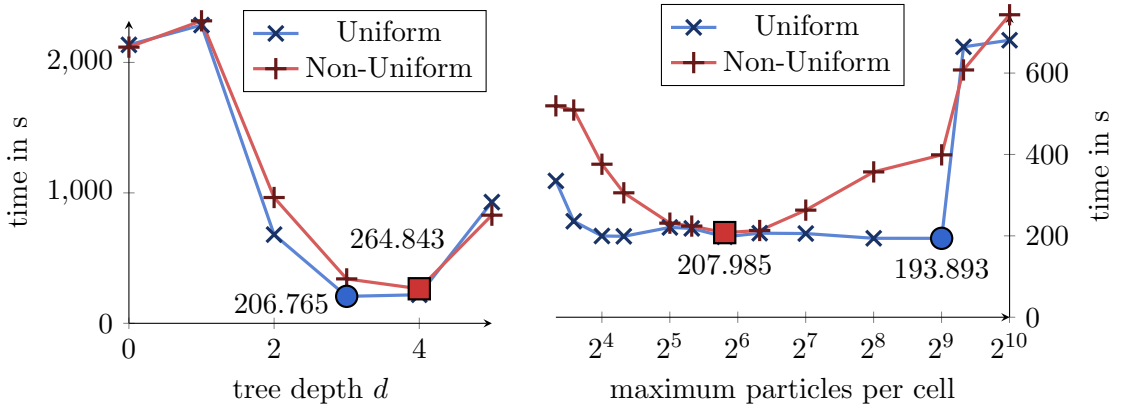
In order to see, how the two approaches perform, two tests have been made with $n = 16000$ and $n = 40000$. For the non-uniform particle distribution, a scenario with two particle clusters was used. Each cluster was generated using a Gaussian distribution. The particle density in the center of these clusters is very high, such that near field calculations with big cells lead to bad performance. The adaptive algorithm will have more and smaller cells in these areas, reducing the number of particles used in each $O(n^2)$ Direct Sum calculation. Figure 4.10 shows the comparison between the non-adaptive octree and the adaptive octree for the uniform and the non-uniform particle distribution of 16000 particles. The Fast Multipole Method can become very slow, if the tree size does not fit to the number of particles. This can be seen for $d = 0$ and $d = 1$ in Figure 4.10 (a). Because of that, only the best performance will be considered, when comparing the adaptive octree with the non-adaptive one. The shortest run-times for both the uniform and non-uniform distribution are marked in Figure 4.10. For the uniform distribution, the best run-times are almost the same, as expected. On the other hand, for the non-uniform distribution the adaptive octree is around 8% faster, showing that the adaptive octree handles non-uniform particle distributions better than the non-adaptive octree.

Increasing the number of particles results in a more expensive near field calculation. The non-uniform particle distribution slows down the near field calculation for the non-adaptive octree. So it is to be expected, that for more particles the advantage of the adaptive octree is even bigger. This can be seen in Figure 4.11. Here, the number of particles was increased to 40000. Again, the best run-times for the uniform distribution are very similar for both octress. However, the non-uniform distribution is around 21% faster for the adaptive octree. So for non-uniform particle distributions and a high number of particles, the adaptive octree is noticeably faster than the non-adaptive octree.




(a) Non-Adaptive Octree (b) Adaptive Octree

Figure 4.10.: Comaprison between non-adaptive and adaptive octree for uniform and non-uniform particle distribution of 16000 particles. The fastest run-times for both distributions are indicated by the circle and square.




(a) Non-Adaptive Octree (b) Adaptive Octree

Figure 4.11.: Comaprison between non-adaptive and adaptive octree for uniform and non-uniform particle distribution of 40000 particles. The fastest run-times for both distributions are indicated by the circle and square.

## 4.2. Accuracy

Since all near field calculations are done using the Direct Sum algorithm, their results are exact. So in this section only the far field is relevant. The error bounds as outlined in [Gre87] only depend on the order of expansion.
In order to verify, that these error bounds also hold for this implementation, the results of this implementation are compared to the exact results.

In [Gre87] and [BG], error bounds are given for every cluster to cluster interaction in the multipole to local operator depending on the distances of the clusters, their size, and the sum of charges in them. Since many of these cluster to cluster interactions happen with different distances in every interaction, it is very hard to make use of the exact error bounds. However, the error bounds can be written as:

$$aQ \cdot \left(\frac{1}{c}\right)^{p+1} \tag{4.1}$$

Here, $a$ and $c$ depend on the distance between the clusters and $Q$ is the sum of charges in one of the clusters. Instead of using $Q$, relative errors are given, which are independent from the sum of charges. The factor $a$ will be ignored, because it does not affect how quickly the error diminishes as the order of expansion is increased. For $c$ the minimum value that $c$ can reach will be used to get worst case error bounds.

The variable $c$ is defined as $d > (c+1)r$, with $d$ being the distance of the two cells and $r$ the maximum radius of the surrounding spheres [BG]. With three cells in one line, the outer cells are the closest two cells, that can have far field interactions in the Fast Multipole Method. In this case the distance between the centers of the cells is $2\sqrt{2}$ times the maximum radius of the surrounding spheres, so that $2\sqrt{2}-1$ is the minimum value $c$ can attain.

Then the error depending on the order of expansion $p$ is in $O((\frac{1}{c})^{p+1})$, with $c = 2\sqrt{2}-1$. This can also be seen in Figure 4.12, since for large enough $p$ the maximum error is below $1/100 \cdot (\frac{1}{c})^{p+1} \in O((\frac{1}{c})^{p+1})$.

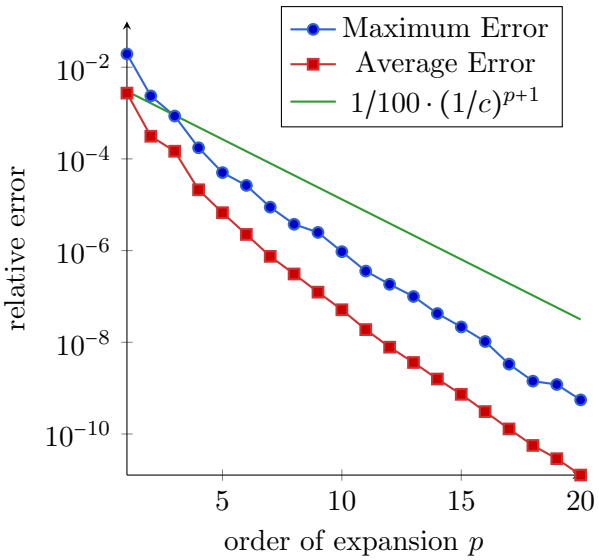



Figure 4.12.: Maximum and average relative error depending on the order of expansion compared to the theoretical error bound of $O((1/c)^{p+1})$.

# Part III.

# AutoPas Integration

# 5. Implementation

In the previous part the Fast Multipole Method was built on top of AutoPas, only using AutoPas containers to store and access the particles. Here, the Fast Multipole Method is directly integrated into AutoPas, allowing it to take advantage of the inner structure of the AutoPas containers. This implementation uses the same operators as the octree implementation in Part II and therefore the optimizations from Section 3.2 are also used here.

## 5.1. Tree Generation

This implementation also uses a tree for the Fast Multipole Method, however in this case it is a binary tree. This allows the use of arbitrary cuboids for the domain. As this implementation uses a binary tree instead of an octree it will also be referred to as binary tree implementation. The tree generation depends on the type of the AutoPas container. In this section it will be explained how the structure of some of the AutoPas containers can be used to create a binary tree for the Fast Multipole Method.

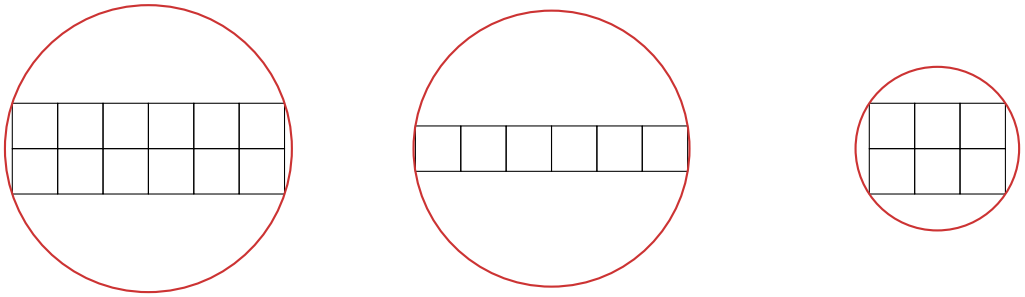
### 5.1.1. Direct Sum

The Direct Sum container (Section 2.4.1) uses a domain that is not divided any further. Therefore this container would also have a binary tree with only a single cell. Because of that, it does not make sense to apply the Fast Multipole Method to it.

### 5.1.2. Linked Cells

The Linked Cells container (Section 2.4.1) uses a regular grid to divide the domain into equally big cells. The cutoff radius is not used directly for the Fast Multipole Method, as the Fast Multipole Method is used to calculate long-range forces. However, the Linked Cells container is a good container for the Fast Multipole Method, as it has a regular grid of cells, which can be used to build a tree structure on top of it. Generally, the cells of the container are not cubes and the number of cells in every direction are not always powers of two, so the octree implementation from Part II would not work.

The binary tree from the Fast Multipole Method adds new cells on top of the Linked Cells grid, so it is important to distinguish between cells from the AutoPas container and cells from the binary tree. The cells from the AutoPas container form a regular grid and all of them have the same size. On the other hand, the size of the cells of the binary tree is not the same for all cells, because child cells obviously need to be smaller than their parent cell. In order to take advantage of the structure of the container, it is important that tree cells do not split the cells of the container. This means, a cell of the AutoPas container is either



(a) Parent Cell ($r \approx 3.16$) (b) Child Cell ($r \approx 3.04$) (c) Child Cell ($r \approx 1.80$)

Figure 5.1.: The parent cell (a) and its two possible child cell types with surrounding spheres. The surrounding sphere in (c) is much smaller than in (b), because the cell side lengths in (c) are more similar.

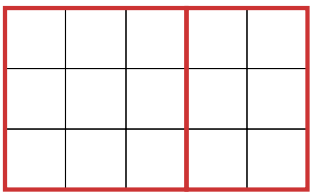


Figure 5.2.: If the longest side of a binary tree cell has an odd number of AutoPas cells, the child cells will have different sizes.

fully included in a cell of the binary tree or not at all. As explained in Subsection 2.4.2, this will lead to efficient particle access.

The binary tree is generated in a similar way as the octree explained in Subsection 2.3.1. A parent cell is divided recursively into two children. Since there are only two children, the cell is only split along one side. In order to get good performance, it is important to handle as many interactions in the far field as possible. The minimum distance between two cells to be able to interact depends on the maximum radius of the spheres around the cells, as explained in Section 2.3. A cube can be contained in a sphere that is only slightly larger than the cube itself. As opposed to this, cuboids with highly different side lengths require much larger spheres, such that the cells can only interact with other cells that are far away. Because of that, the algorithm always splits a cell close to the middle of its longest side to keep the sphere around it as small as possible and allow interactions with as many other cells as possible. Figure 5.1 shows how much smaller the surrounding sphere can be, if the cell is divided along its longest side.

It is not always possible to divide the parent cell exactly at the middle of the longest side, since the longest side may have an odd number of AutoPas cells. In that case one child cell will have a side that is one AutoPas cell longer than the other child cell. This can be seen in Figure 5.2.

Using this method recursively, the entire domain can be divided into smaller binary tree cells, until all leaves of the tree contain only a single AutoPas cell. The result of this process applied to a $2 \times 5$ domain can be seen in Figure 5.3.

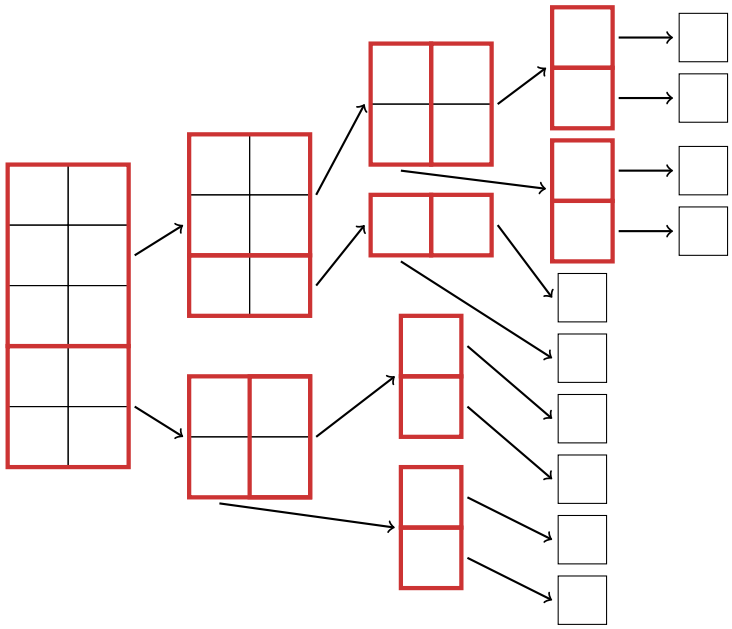


Figure 5.3.: Subdivision of a $2 \times 5$ domain into single cells.

### 5.1.3. Verlet Lists

The verlet list container (Section 2.4.1) has a regular grid like the Linked Cells container, so the tree is generated in the same way as described in Subsection 5.1.2.

## 5.2. Near Field List and Interaction List

Since the binary tree is more general than the octree and can have arbitrary cell sizes, there is no easy way to find the near field and interaction lists. Instead, only the condition for being well separated from Section 2.3 is used. Algorithm 6 describes how the near field and interaction lists are built for the binary tree.

At the start of the algorithm both the near field and interaction list are empty for every cell in the binary tree. The most important part when setting up the near field and interaction lists is that every area in the tree interacts with every other area exactly once, so that every particle interaction is done once.

The function `init_lists` handles all interactions for the area of *thisCell* with the area of *interactingCell*, so after calling this function the area covered by *thisCell* interacts with the area covered by *interactingCell* exactly once. This means, calling this function with both parameters being the root cell, sets up all near field and interaction lists.

At first the function checks, whether the two cells can directly interact in the far field. This is done by checking, whether the distance between the cells is large enough and this depends on the maximum radius of the surrounding spheres. If the cells can interact, *interactingCell* is added to the interaction list of *thisCell* and the function terminates, since the interaction between these two cells is now done and their children must not interact with each other, so that no interactions are calculated multiple times.

If such an interaction is not possible, because the cells are too close to each other, interactions must be done with their children. Replacing `init_lists(x,y)` with two calls to `init_lists(x.child1,y)` and `init_lists(x.child1,y)` still results in correct near field and interaction lists, because `init_lists(x,y)` handles all interactions between the area of cell $x$ and the area of cell $y$ and the area of $x$ is the same as the combined area of $x.child1$ and $x.child2$. The same is true, if the parameters are swapped. Here, the cell with the largest side length is chosen to be replaced by its children, so that both cells are always of similar size. This way the surrounding spheres are of similar size as well, such that many far field interactions can be done early on, reducing the overall number of interactions. If the cell with the largest side length is a leaf, it must be a cell consisting of only a single AutoPas cell, so all side lengths are one. Obviously, the other cell cannot have a side length of more than one, so it is also a single AutoPas cell. In that case the algorithm cannot continue with the children of the larger cell and *interactingCell* is added to the near field list of *thisCell* instead.

The function `init_lists` is called with both parameters being the root, so that all near field and interaction lists of the tree are set up correctly.

**Algorithm 6:** Generation of the near field and interaction list.

```
1  Function get_distance (thisCell, interactingCell):
2      ...
3  Function is_leaf (thisCell):
4      ...
5  Function check_interact (thisCell, interactingCell):
       // Checks whether the cells are far enough away to be able to
          interact.
6      maxRadius ← max{thisCell.sphereRadius, interactingCell.sphereRadius}
7      distance ← get_distance (thisCell, interactingCell)
8      return distance > 3 · maxRadius
9  Function get_largest_length(thisCell):
       // Returns the length of the largest side (x, y or z).
10     return max{thisCell.xLength, thisCell.yLength, thisCell.zLength}
11 Function init_lists(thisCell, interactingCell):
       // If the cells can interact with each other, add the other cell
          to the interaction list.
12     if check_interact (thisCell, interactingCell) then
13         thisCell.interactionList ← thisCell.interactionList ∪ interactingCell
14     else
           // Traverse the tree by going to the children of the bigger
              cell.
           // If the bigger cell does not have children, add the other
              cell to the near field list.
15         if get_largest_length (thisCell) >get_largest_length (interactingCell)
            then
16             if not is_leaf (thisCell) then
17                 init_lists (thisCell.child[0], interactingCell)
18                 init_lists (thisCell.child[1], interactingCell)
19             else
20                 thisCell.nearFieldList ← thisCell.nearFieldList ∪ interactingCell
21         else
22             if not is_leaf (interactingCell) then
23                 init_lists (thisCell, interactingCell.child[0])
24                 init_lists (thisCell, interactingCell.child[1])
25             else
26                 thisCell.nearFieldList ← thisCell.nearFieldList ∪ interactingCell
```

# 6. Results

In this chapter the performance and accuracy of this implementation are analyzed.

## 6.1. Performance

Since the operators are the same for both implementations, similar performance and accuracy as in Chapter 4 are to be expected. The main difference lies in the near field and interaction lists.

### 6.1.1. Order of Expansion

Because the operators are the same as in the octree implementation and the order of expansion is only used in the operators, the influence of the order of expansion on the run-time of the algorithm is the same as explained in Subsection 4.1.1. So here the run-time is again in $O(p^4)$, where $p$ is the order of expansion. The plot in Figure 6.1 shows the same result as Figure 4.1. Overall, the performance depending on the order of expansion behaves the same as for the octree, which was already discussed in detail in Subsection 4.1.1.

### 6.1.2. Tree Size

In Section 4.1.2, different tree depths were used to control far field and near field run-times. For the binary tree not all leaf cells are on the same level (see Figure 5.3), so two binary trees could have the same depth, but a different number of AutoPas cells. Because of this the side lengths of the domain in terms of AutoPas cells are used instead.

This implementation allows arbitrary cuboid domains, but for these tests a cubic domain was used, so that it can be compared better to the octree implementation. For the same reason, the parameters are also unchanged from Table 4.1.

In the first two tests, the number of particles was set to $n = 4000$ and for the order of expansion $p = 4$ and $p = 8$ were used. The run-times for these two tests are shown in Figure 6.2. With only 4000 particles, the Fast Multipole Method is only slightly faster than the Direct Sum algorithm for $p = 4$ and overall slower for $p = 8$. Additionally, for $p = 4$ the Fast Multipole Method has low accuracy, so that the Direct Sum calculation is better in this scenario. This behavior is very similar to the octree results from Section 4.1.2.

Similar to the octree implementation, the big advantage of the Fast Multipole Method can be seen, if the number of particles is increased. The run-times for $n = 40000$ particles are displayed in Figure 6.3. Here, the run-times for a side length of 1, which essentially is the same as the Direct Sum algorithm are very high and the run-time decreases, if the number of cells is increased. For $p = 4$, the optimal performance is achieved with a side length of 14 AutoPas cells, whereas for $p = 8$, the run-time is the shortest for a side length of 10.

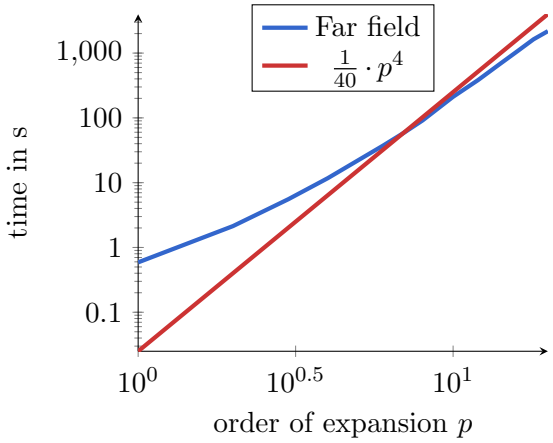
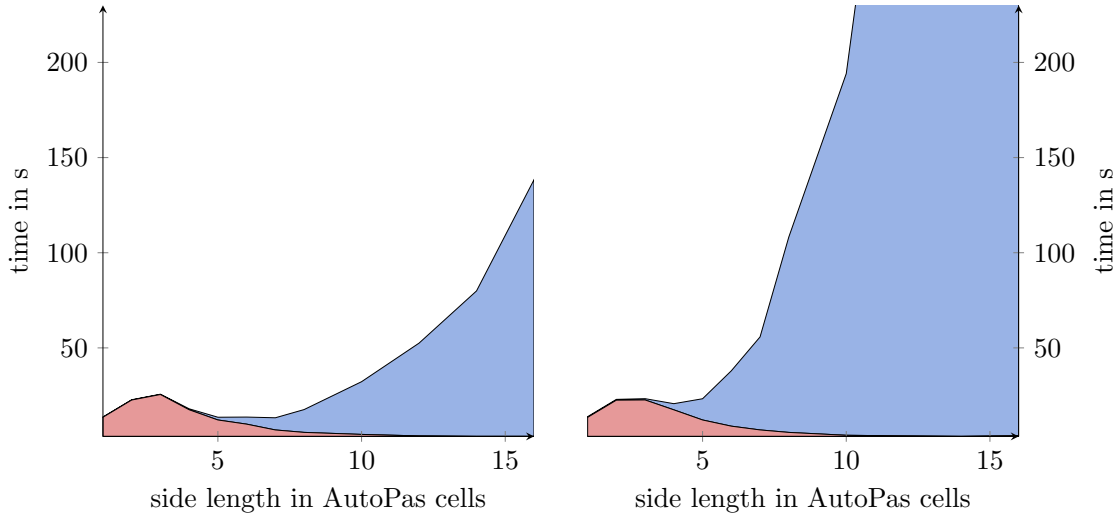
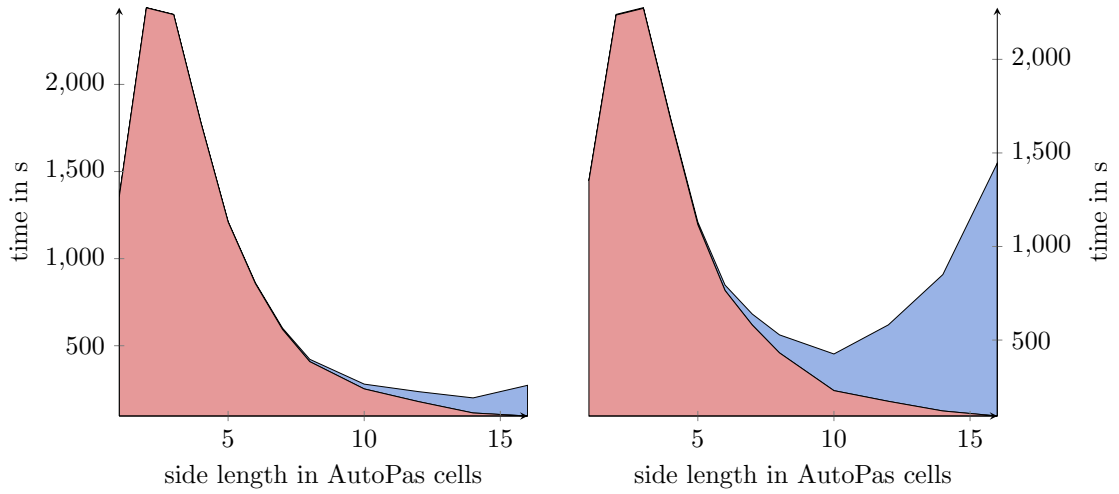



Figure 6.1.: Far field run-time depending on the order of expansion for 2000 particles in a binary tree built on a domain with $8 \times 8 \times 8$ AutoPas cells..




(a) $n = 4000$, $p = 4$ (b) $n = 4000$, $p = 8$

Figure 6.2.: Absolute far field (blue) and near field (red) run-times for 4000 particles in a binary tree with different numbers of AutoPas cells using an order of expansion of 4 (left) and 8 (right).




(a) $n = 40000$, $p = 4$ (b) $n = 40000$, $p = 8$

Figure 6.3.: Absolute far field (blue) and near field (red) run-times for 40000 particles in a binary tree with different numbers of AutoPas cells using an order of expansion of 4 (left) and 8 (right).

The performance of the binary tree performs very similar to the octree, which was explained in 4.1.2. For a side length of one, there is only a single AutoPas cell, so all interactions are calculated in the near field. The overhead of the Particle-to-Multipole and Local-to-Particle operator is insignificant compared to the $O(n^2)$ near field calculation, such that for a side length of one the performance is the same as for a Direct Sum algorithm. If the side length is increased to two or three, the performance gets worse, as additional overhead is added by performing the near field calculation over several cells. For a side length of four or more, the near field gets faster, as increasingly many interactions are covered by the far field, while the far field gets slower. If there are enough particles, the Fast Multipole Method reaches its best performance for a side length of four or more. After increasing the side length further, the far field will become too expensive at some point, that it outweighs the better near field performance.

## 6.2. Accuracy

The accuracy of the binary tree and the octree are very similar. The operators are the same, so the same error bounds as explained in Section 4.2 apply for the binary tree as well. The only difference is that $c$ has a different minimum value for the binary tree. The definition $d > (c+1)r$ is the same as before, with $r$ being the maximum radius of the surrounding spheres and $d$ the distance between the centers of the spheres. However since the minimum interaction distance was strictly defined as $3r$, the minimum value $c$ can attain is 2. This gives overall slightly more accurate results than before, which can be seen in Figure 6.4. The maximum relative error is below $1/1000 \cdot (\frac{1}{c})^{p+1} \in O((\frac{1}{c})^{p+1})$ for large enough values of $p$, so the relative error is also in $O((\frac{1}{c})^{p+1})$.

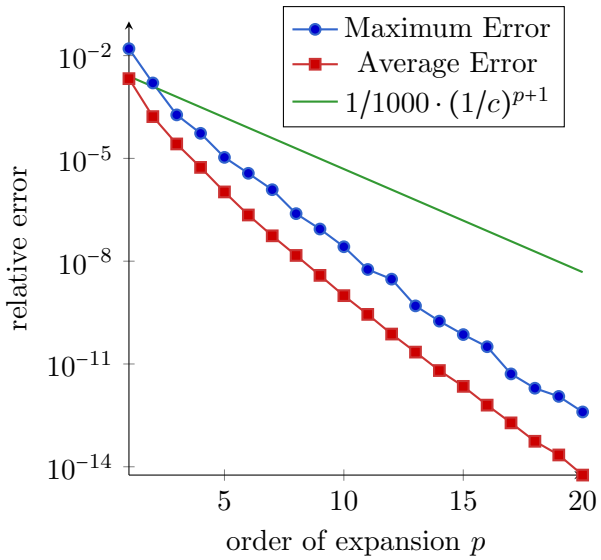



Figure 6.4.: Maximum and average relative error depending on the order of expansion compared to the theoretical error bound of $O((1/c)^{p+1})$.

# Part IV.

# Conclusion

# 7. Comparison

In this thesis two implementations of the Fast Multipole Method were presented. Here, the advantages of each approach are discussed. The performance of both methods was explained thoroughly in Section 4.1 and Section 6.1 and it is the most important aspect of this comparison. The Fast-Multipole-Method is an approximation algorithm for a $O(n^2)$ problem that achieves $O(n)$ complexity and therefore fast run-times are the main advantage of the Fast-Multipole-Method.

Only considering the algorithm of the Fast-Multipole-Method itself, the octree implementation is faster. The near field and interaction lists are a lot less complex and smaller, resulting in overall less interactions.

However, the binary tree has a few important advantages that make it better in most situations. The binary tree allows arbitrary cuboid domains, so it can be used more flexibly. It also enables more fine tuned tree sizes compared to the octree, since the octree always has a power of eight as number of leaves, whereas the number of leaves for the binary tree is the product of the three side lengths in terms of AutoPas cells. Since cubic domains can be handled by both the octree and the binary tree, the comparison is based on cubic domains. Table 7.1 shows possible numbers of leaves for both tree types in a cubic domain.

Achieving good performance depends a lot on having a good balance between far field and near field run-times. For the octree, the far field run-time increases massively, if the tree depth is increased by one, because the number of leaves is multiplied by eight. As a result, the optimal performance for the octree will often times be worse than for the binary tree. This can be seen, if the tests with $n = 40000$ and $p = 4$ are compared. The plots of these tests are shown in Figure 4.7 and Figure 6.3. For the binary tree, the best performance is achieved for a side length of 14, which results in $14^3 = 2744$ AutoPas cells. For the octree, no tree depth with around 2744 leaves exists. For a depth of $d = 3$, there are only $8^3 = 512$ leaves and for $d = 4$, the number of leaves is too high with $8^4 = 4096$. This can be observed in the run-times, as shown in Table 7.2. The binary tree performs slightly better, because it can use tree sizes that fit the number of particles better.

Another advantage of the binary tree is the fact, that the cell borders of the tree cells overlap with the cell borders of the AutoPas cells, resulting in efficient particle access, as explained in Subsection 2.4.2. For the octree, the performance depends on the size of the leaves compared to the size of the AutoPas cells and how well they align. If the cell size of the AutoPas container can be chosen freely and the desired depth of the octree is known, the

| Number of Leaves | 1 | 8 | 27 | 64 | 125 | 216 | 343 | 512 |
|---|---|---|---|---|---|---|---|---|
| Octree Depth | 0 | 1 | - | 2 | - | - | - | 3 |
| Binary Tree Side Length | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

Table 7.1.: Overview of the number of leaves in the binary tree compared to the octree.

| Number of Leaves | 512 | 1000 | 1728 | 2744 | 4096 |
|---|---|---|---|---|---|
| Octree Depth | 3 | - | - | - | 4 |
| Octree Run-Time | 207 | - | - | - | 219 |
| Binary Tree Side Length | 8 | 10 | 12 | 14 | 16 |
| Binary Tree Run-Time | 422 | 280 | 236 | 200 | 273 |

Table 7.2.: Performance comparison between the octree and the binary tree implementation for $n = 40000$ particles and order of expansion $p = 4$.

cell size can be chosen, such that the leaves of the octree overlap with cells of the AutoPas container. In that case, the octree will perform better than the binary tree, since it combines the advantages of both approaches. Obviously, this does not work for an adaptive octree, because then the depth of the leaves is not known.

Overall the octree has the better best-case performance. Using an adaptive octree, it also performs better for non-uniform particle distributions. The binary tree on the other hand is more flexible and yields fast run-times more reliably, since particle access is always fast.

# 8. Summary

In this thesis two implementations of the Fast Multipole Method for AutoPas were presented. The Fast Multipole Method was implemented for the Coulomb potential and it could accurately approximate the results with good performance. Especially for larger numbers of particles the Fast Multipole Method performed much better than the Direct Sum approach.

The number of leaves in the tree structure is a deciding factor for good performance. On the one hand, if there are too many leaves, the far field operators are so expensive, that it outweighs the better near field performance. On the other hand, if the are too few leaves, the near field calculations are still very expensive and the algorithm is not much faster than a Direct Sum implementation. Here, the binary tree implementation has the benefit, that the number of leaves can be controlled better, so that near field and far field run-times are more balanced. More research is required to find out which number of leaves should be chosen for a certain number of particles and order of expansion to get optimal performance.

# 9. Outlook

It was shown, that the Fast Multipole Method can be used in AutoPas to efficiently calculate long-range forces. So far, only the Coulomb potential was implemented, but operators for other potentials could also be added. It is also possible to develop a kernel-independent Fast Multipole Method[YBZ04], so that AutoPas can compute long-range forces for arbitrary potentials.

Furthermore the near field calculation could be integrated better into AutoPas to take advantage of its auto-tuning and overall highly optimized algorithms. However, here it is important to prevent AutoPas from doing interactions between particles, that have already been done by the far field from the Fast Multipole Method.

In order to further improve performance, a parallel implementation of the Fast Multipole Method can be made using libraries such as OpenMP[1] and MPI[2].

[1] `https://www.openmp.org`
[2] `https://www.mpi-forum.org`

# Part V.

# Appendix

# List of Figures

# List of Tables

# Bibliography


[BG] Rick Beatson and Leslie Greengard. A short course on fast multipole methods.

[BH86] Josh Barnes and Piet Hut. A hierarchical O(N log N) force-calculation algorithm. *Nature*, 324(6096):446–449, Dec 1986.

[DLMF] *NIST Digital Library of Mathematical Functions*. http://dlmf.nist.gov/, Release 1.0.25 of 2019-12-15. F. W. J. Olver, A. B. Olde Daalhuis, D. W. Lozier, B. I. Schneider, R. F. Boisvert, C. W. Clark, B. R. Miller, B. V. Saunders, H. S. Cohl, and M. A. McClain, eds.

[EHB+13] Wolfgang Eckhardt, Alexander Heinecke, Reinhold Bader, Matthias Brehm, Nicolay Hammer, Herbert Huber, Hans-Georg Kleinhenz, Jadran Vrabec, Hans Hasse, Martin Horsch, Martin Bernreuther, Colin W. Glass, Christoph Niethammer, Arndt Bode, and Hans-Joachim Bungartz. 591 tflops multi-trillion particles simulation on supermuc. In Julian Martin Kunkel, Thomas Ludwig, and Hans Werner Meuer, editors, *Supercomputing*, pages 1–12, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[GK08] TIMOTHY C. GERMANN and KAI KADAU. Trillion-atom molecular dynamics becomes a reality. *International Journal of Modern Physics C*, 19(09):1315–1319, 2008.

[Gra17] Fabio Gratl. Task based parallelization of the fast multipole method implementation of ls1-mardyn via quicksched. Master's thesis, Institut für Informatik 5, Technische Universität München, Garching, November 2017.

[Gre87] Leslie Greengard. The rapid evaluation of potential fields in particle systems. Research Report 533, Department of Computer Science, Yale University, april 1987.

[GST+] Fabio Gratl, Steffen Seckler, Nikola Tchipev, Hans-Joachim Bungartz, and Philipp Neumann. Autopas: Auto-tuning for particle simulations.

[KP] J. Kurzak and B. M. Pettitt. Fast multipole methods for particle dynamics.

[KP90] Martin Karplus and Gregory A. Petsko. Molecular dynamics simulations in biology. *Nature*, 347(6294):631–639, 1990.

[Spu97] Rainer Spurzem. Astrophysical n-body simulations: algorithms and challenges, 1997.

[Tam79] Igor E. Tamm. *Fundamentals of the Theory of Electricity*. Mir Publishers, 1979.

[vGB90] Wilfred F. van Gunsteren and Herman J. C. Berendsen. Computer simulation of molecular dynamics: Methodology, applications, and perspectives in chemistry. *Angewandte Chemie International Edition in English*, 29(9):992–1023, 1990.

[YBZ04] Lexing Ying, George Biros, and Denis Zorin. A kernel-independent adaptive fast multipole algorithm in two and three dimensions. *Journal of Computational Physics*, 196(2):591 – 626, 2004.