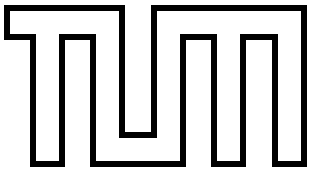


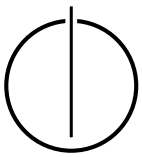
SCHOOL OF COMPUTATION, INFORMATION
AND TECHNOLOGY - INFORMATICS

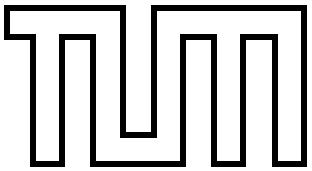
TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

A Comparison of Three-body Algorithms for Molecular Dynamics Simulations

David Martin





SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY - INFORMATICS

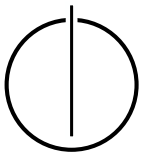
TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

A Comparison of Three-body Algorithms for Molecular Dynamics Simulations

Ein Vergleich von Algorithmen für Dreikörperwechselwirkungen in der Molekulardynamik

Author:	David Martin
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisor:	Samuel Newcome, M.Sc.
Date:	15.11.2022



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

Munich, 15.11.2022 David Martin

Acknowledgements

I would like to thank my family who has supported me in everything I have done over the past years and made it possible for me to study informatics.

I would also like to thank Sam, the advisor of this thesis. He has supported me a lot during the last months and helped me with his valuable feedback to complete this thesis.

Furthermore, I would like to thank the Chair of Scientific Computing and Professor Dr. Hans-Joachim Bungartz for giving me the opportunity to write my thesis in this interesting area.

Abstract

In Molecular Dynamics Simulations, typically the forces between pairs of particles are calculated. However, the inclusion of three-body forces offers the possibility of obtaining more accurate results in certain simulations. When computing the forces between all possible triplets of particles, this is in a complexity class of $\mathcal{O}(n^3)$, so there is interest in developing algorithms to speed this up. To date, however, few algorithms for efficient computation of three-body forces have been developed. This thesis gives a literature review of algorithms already developed and describes how they work. The literature review provides an entry point into the computation of three-body forces and is used to critically analyze the most valuable research directions in this field. The advantages and disadvantages, as well as possible use cases for different algorithms, are discussed. The implementation of three of the most promising of such algorithms for distributed memory environments is presented. All three implementations are investigated on a medium size HPC cluster for their runtime. Furthermore, one of these implementations is examined for its hit-rate and load balance, as well as its accuracy.

Contents

Part I.

Introduction and Background

1. Introduction

Due to the increase in computational power in recent years, it is possible for scientists to study the behavior of molecules and atoms among each other more precisely than before. For this purpose, Molecular Dynamics (MD) Simulations are used to iteratively calculate the forces between molecules and atoms and the resulting movement over multiple time steps. We use the term *particle* in this thesis to denote one of the n molecules or atoms to be simulated.

Since the step sizes are usually chosen very small and each simulation step is computationally intensive, it can take hours, days or months to complete the whole simulation [LZS06c].

Usually in MD Simulations, the forces between each pair of particles are calculated, which is in a complexity class of $\mathcal{O}(n^2)$ for n particles. To speed this up, much research has already been done, and efficient algorithms have been developed.

Scientists have found that, especially in fluid simulations, more accurate results can be obtained if the interactions between each triplet of particles is included in the force calculation [MS99]. Since the computation of all forces between triplets of n particles is in a complexity class of $\mathcal{O}(n^3)$, there is particular interest in developing algorithms to speed this up. In the field of three-body interactions, however, little research has been done to develop such efficient algorithms compared to those for pairwise force calculations.

This thesis focuses on algorithms for three-body interactions that have already been developed. Our goal is to understand how the algorithms work, and to compare their advantages and disadvantages in order to decide which of the research directions we want to pursue in the future. As a part of this work, we implement some of the most promising of these algorithms and study their behavior in practice.

The thesis is structured as follows: In Chapter 2, we give an overview of background knowledge that will be used throughout the thesis. Chapter 3 focuses on algorithms for three-body interactions that have already been developed. We describe how these algorithms work, what advantages and disadvantages they have and how they can be used. In Chapter 4, we present our implementation of three selected algorithms in C++. In Chapter 5, we investigate all three implementations in terms of their runtime. One of the algorithms is examined for its load balance and hit-rate. Furthermore, this algorithm is investigated for its accuracy in comparison to one of the other implementations. In Chapter 6, we provide an outlook on how our implementations can be further improved and how we intend to proceed.

2. Theoretical Background

2.1. Molecular Dynamics Simulation

One step of an MD Simulation typically works as follows: By taking the partial derivatives of potential energy functions, the forces between particles are evaluated. Particles are moved by calculating the new velocities and positions based on the forces using for example a Velocity Verlet Integration [Ver67]. Then forces are reset and the next simulation step begins.

Let Q be the set of all n particles, then for multibody interactions the total force exerted on the particle $i \in Q$ from all other particles $j, k \in Q$ with $i \neq j \neq k$ can be expressed as follows:

$$F_i = -\nabla \left[\phi_1(i) + \sum_j \phi_2(i,j) + \sum_{j,k} \phi_3(i,j,k) + \dots \right] \tag{2.1}$$

Where ∇ is the gradient of the sum of u-body potential functions ϕ_u [KY14]. We use u in this thesis to specify the number of participating particles in an interaction. For pairwise interactions $u = 2$, for three-body interactions $u = 3$.

All particles involved in the simulation are located in a space, which is called *simulation domain* in the following. There are different ways to handle outflows of the domain: One possibility is to exclude particles that leave the domain. Another possibility are periodic boundary conditions, where particles loop to the opposite side of the simulation domain, to simulate an infinite space. A third method is to push particles back into the simulation domain if they reach a boundary by applying a repulsive force to them, to simulate particle movements inside a fixed space [GSBN22].

2.1.1. Short- and Long Range Forces

Potential functions can be classified into short and long range potentials, where short range potentials decay very fast with increasing distance between particles, while long range potentials do not. Typically potentials that decay faster than r^{-D} are short range, the others are long range, where D corresponds to the dimension of the simulation domain, typically $D = 3$, and r corresponds to the distance between particles [Tch20].

In pairwise interactions, for example, the Lennard Jones potential is one of the most common potentials for short range force calculations and the Coulomb electrostatic potential is a common one for long range interactions.

In three-body interactions, the Axilrod-Teller potential is an often used potential for short range calculations and is used, for example, in the simulation of the vapor-liquid transition of noble gases [MS99]. It is defined as follows:

$$\phi_3(i,j,k) = v * \left[\frac{1 + 3 * \cos(\theta_i) * \cos(\theta_j) * \cos(\theta_k)}{(r_{ij} * r_{ik} * r_{jk})^3} \right] \tag{2.2}$$

Where r_{ij} is the distance between particle i and j, θ_i is the angle between $i\vec{j}$ and $i\vec{k}$. The positive coefficient v depends on the molecules being simulated and is based on the ionization energy and the mean atomic polarizability [AT43].

2.1.2. Cutoff Distance

A cutoff distance can be used, so that only the interactions in a certain surrounding area of a particle are calculated, which speeds up the calculation. In contrast to the pairwise interactions in which there is only one obvious possibility, namely the distance between particle i and j, which is checked against the cutoff distance c, in the three-body interactions several variants can be applied, such as:

1. All distances between all particles of a triplet have to be less than c [Mar01].
2. At least a pair out of the three distances between i, j and k have to be less than c [CW00].
3. The sum of all distances from the center of mass to each particle is less than c [L. 13].

We use c in this thesis to denote the cutoff.

2.1.3. Newton's Third Law of Motion

Newton's third law of motion states that a force $\vec{F}_{B \leftarrow A}$ acting from body A on body B also acts in the opposite direction and with the same magnitude from B on A. Accordingly, for a given pair of particles (i, j), in which both particles exert forces on each other, it holds:

$$\vec{F}_{i \leftarrow j} = -\vec{F}_{j \leftarrow i} \tag{2.3}$$

In the context of Molecular Dynamics Simulations, this law can be used to reduce the calculation of the forces between particles. In the case of pairwise interactions, this is trivial and only one of the two forces needs to be calculated for a given pair of paricles (i, j).

In the case of particle triplets, we can also make use of this law according to [Mar01]:

$$\vec{F}_{i \leftarrow jk} + \vec{F}_{j \leftarrow ik} = -\vec{F}_{k \leftarrow ij} \tag{2.4}$$

Where $\vec{F}_{i \leftarrow jk}$ represents the force acting on i from j and k. We can therefore use Newton's third law also for particle triplets, to calculate all the forces on the individual particles within a triplet in one step.

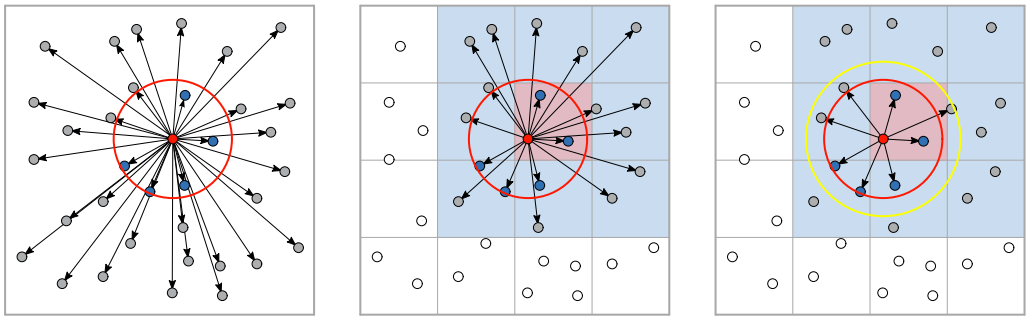
If we use a cutoff, we must ensure that each pairwise distance between the three particles involved is less than the cutoff to ensure that the triplet can be formed from the viewpoint of each particle. This is satisfied by the first and third criterion from 2.1.2 [KY14].

In the remainder of this thesis we will denote the usage of Newton's third law as *newton3*.

2.2. Two Body Algorithms

In this section, we briefly introduce several methods used in pairwise interactions to speed up the $\mathcal{O}(n^2)$ calculation.

2.2.1. Short Range Algorithms



(a) Direct Cutoff (b) Linked Cell (c) Verlet Lists

Figure 2.1.: The red circle represents the cutoff area. In this example, the forces between the red particle and others is calculated. Distances are calculated between the red and all particles with blue or gray filling. Forces are calculated between the red particle and the blue particles. For particles without filling neither distances nor forces are calculated. Source: [GSBN22]

When using a cutoff, different data structures can be used to further reduce computation time.

Direct Cutoff

The naive procedure calculates for one particle i the distance to all other particles in the domain to decide whether the forces for a pair should be evaluated or not. This leads to $\mathcal{O}(n^2)$ distance checks [GSBN22]. See Figure 2.1a for illustration.

Linked Cell

To minimize the number of distance checks, the linked cell method can be used to divide the simulation domain into a regular grid and assigning particles to the corresponding cells, where the edge length of each cell is at least equal to the cutoff radius, which can be seen in Figure 2.1b. In this way, for a given particle, only the distances in neighboring cells need to be computed, which reduces the number of distance calculations for homogeneous particle distributions to $\mathcal{O}(n)$ [GSBN22].

Verlet Lists

Another method to further reduce distance calculations can be implemented using Verlet Lists [Ver67] as shown in Figure 2.1c. The assumption is that particles move only slightly from simulation step t_s to step t_{s+1}. Therefore, a list is created for each particle in the simulation domain, which stores all neighbors within a radius slightly larger than the cutoff,

which is shown as yellow circle in Figure 2.1c. The area between the cutoff in red and the yellow circle is called Verlet Skin, and stores additional particles beyond the cutoff that could move into the cutoff area in the next simulation step. A recalculation of the whole list is only necessary every few simulation steps and can be done using the linked cell method. The number of distance calculations for homogeneous particle distributions is within $\mathcal{O}(n)$ [GSBN22].

2.2.2. Long Range Algorithms

In the field of long range calculations, methods are used that consider distant particle clusters as one large particle in order to obtain an approximate result. In this area, for example, there is the Barnes Hut algorithm [BH86], which uses a tree structure to group the particles in the simulation domain at different granularity levels. During the calculation of the forces, the distance D between a particle and a cluster, as well as the edge length L of the bounding box of the cluster, is used to decide whether the calculation can be performed or the tree structure should be considered more fine-granular. An example can be seen in Figure 2.2.

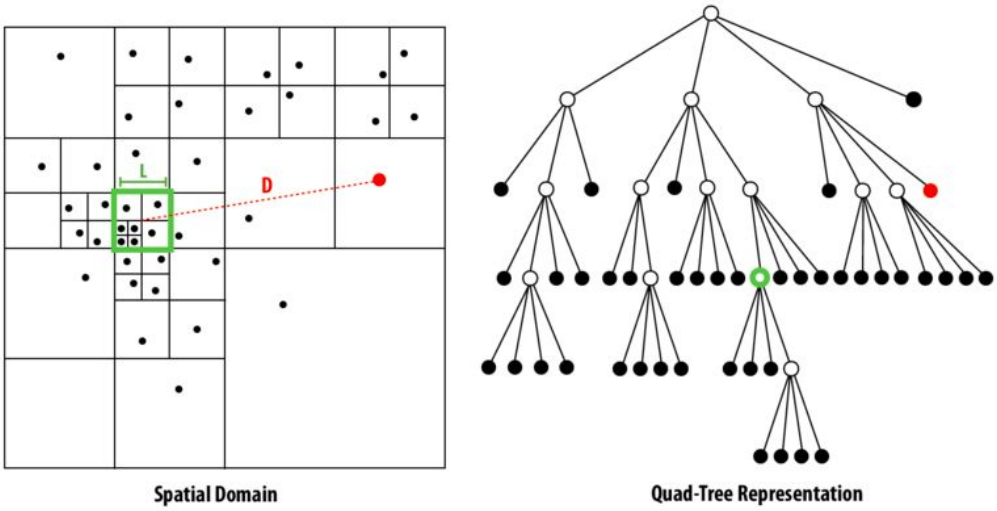


Figure 2.2.: In this example, the interaction between the red particle and the cluster highlighted in green is to be calculated. Based on $\frac{D}{L}$, it is decided whether all particles within the green area can be considered as one large particle, or if the tree structure must be considered at a lower level.
Source: [oCS13]

2.3. Parallelization

Since Molecular Dynamics Simulations are computationally intensive, in addition to the use of the methods mentioned above, techniques are used that parallelize the problem to be computed by breaking it down into pieces and distributing it among several computation units. At the end of the computation, the results of all pieces are combined to obtain the overall result. This can significantly speed up the calculation, but it requires that the respective algorithm supports such parallelization. Since the information of other computing units is often needed during the force calculation, communication between them must be

possible. In the remainder of this thesis, the term *processor* is used to denote a computational unit working on a particular part of the overall problem, and the letter p denotes the number of processors used.

2.3.1. Domain Decomposition Techniques

A domain decomposition partitions the particles within the simulation domain so that the forces can be calculated in parallel by several processors. The decomposition can be either dependent or independent of the position of the particles.

The following three techniques are related to the algorithms of S. Plimpton [Pli95] for pairwise interactions. In the course of this thesis, we present algorithms and implementations for triplet computations that build on these techniques.

Atom Decomposition

In this decomposition, the particles within the simulation domain are distributed evenly among the processors, e.g., based on the ID of the particles. Each processor is responsible for calculating the forces for the particles assigned to it and updating their positions based on the calculated forces. Whether the particles are neighboring in the physical simulation space or not, does not matter in this type of decomposition. Throughout the simulation, the assigned particles remain constant for each processor. Since this method works without relation to the actual position of the particles, it is suitable for force calculations in which all interactions between particles are considered without cutoff distance. We call this type of calculation *direct* in remainder of this thesis. In the course of a simulation step, it is necessary that the information is exchanged between all participating processors, so that each processor can calculate the interactions of its own particles with those of other processors.

Force Decomposition

This decomposition is a generalization of the atom decomposition [SSJ07], where each processor is assigned a part of the particle interactions, called *Force Elements*, to be computed. S. Plimpton [Pli95] uses a $n \times n$ 2D Force Matrix, which can be seen in Figure 2.3, to form all possible combinations of particles in the simulation domain. Each combination of two particles (i, j) in the matrix represents a Force Element to be computed. As can be seen, the matrix contains the elements (i, j) as well as (j, i), which is why it can be used with or without the use of newton3. In the case of using newton3, only the elements in the upper or lower triangular matrix need to be evaluated.

After all Force Elements have been calculated, the matrix can be reduced along an axis to sum all partial forces on the respective particles. The result is a one-dimensional vector containing the summed forces of all particles. Before the next simulation step, and after the particles have been updated, this one-dimensional vector is expanded back to the 2D Force Matrix.

This matrix can either be distributed line by line to processors, which essentially corresponds to the atom decomposition mentioned above. However, it can also be partitioned block-wise, with each processor computing the Force Elements in its assigned block.

In Figure 2.3, a division into 2×2 blocks is shown. Each processor stores the information of the particles in x_α and x_β in local copies, computes the Force Elements from its subcube and stores the results in its local vector f. Thus, communication exchange with all processors is not necessary, since each processor must communicate at most with the processors from its column and row. Analogous to the atom decomposition, the actual position of the particles does not matter, which is why this type of decomposition is also suitable for direct force calculations.

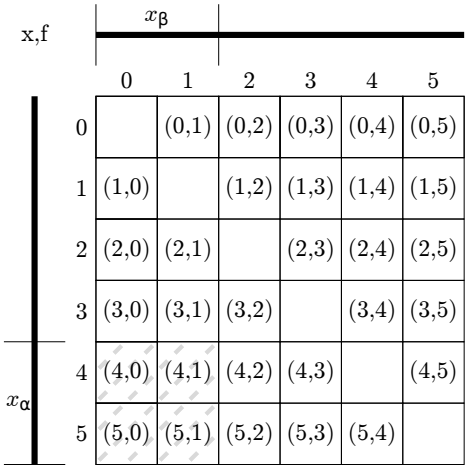


Figure 2.3.: 2D Force Matrix showing an exemplary division into 2×2 blocks.
Adapted from: [Pli95]

Spatial Space Decomposition

In contrast to the two possibilities mentioned above, a *spatial space decomposition* assigns a part of the physical simulation domain to each processor. For example, as in the linked cell method, the domain is divided into a regular grid and each processor is assigned a cell for whose particles it is responsible. Since the positions of the particles change in the course of the simulation, processors must exchange particles if they move beyond the local domain. This type of decomposition is suitable for short range potentials with a cutoff, since communication between processors can be restricted to the (direct) neighborhood.

In the remainder of this thesis, we use the term *regular grid decomposition* to refer to a spatial space decomposition that divides the simulation domain into a regular grid.

2.3.2. Load Balance

By load balance we mean how evenly or unevenly work is distributed among processors. Ideally, each processor has the same amount of work. There are different metrics to measure this. We use the Step Time Variation Ratio (STVR) [LZS06a] throughout this thesis. It is calculated for each processor individually and represents the average calculation time of the forces of each processor for one simulation step in relation to those of all other processors. The average time required by a processor to calculate the forces for one simulation step is denoted by T_i.

$$STVR = \left| \frac{T_i - (\sum_{j=0}^{p-1} T_j/p)}{\sum_{j=0}^{p-1} T_j/p} \right| \tag{2.5}$$

2.3.3. Scalability

In this thesis we distinguish between strong and weak scaling. In strong scaling, we investigate how an algorithm behaves when we apply it to a fixed problem size with different numbers of processors. To determine this, we use the ratio between the serial execution time of an algorithm with one processor T_1 and the parallel execution with p processors T_p. The strong scaling speedup with p processors is calculated as:

$$S_p = \frac{T_1}{T_p} \tag{2.6}$$

Ideally, execution with p processors results in an execution time that is p times faster compared to a single processor. In reality, this is only possible up to a certain point, since the serial part of the software cannot be further parallelized and the communication between the processors additionally dampens the speedup.

In weak scaling, we investigate how an algorithm behaves when we scale the number of processors in proportion to the problem size. This has the consequence that for each number of processors the work per processor remains constant. We measure the weak scale efficiency by dividing the serial computation time by the computation time with p processors, analogous to the speedup as described above. Ideally, an execution with $p \geq 1$ processors always leads to the same execution time as with a single processor. In real world applications, this is not the case, since the additional communication between processors increases the execution time, which dampens the efficiency.

2.3.4. Technical Consideration

To enable communication between the processors, a corresponding infrastructure must be provided. One method is to use high performance computer clusters, such as the SuperMUC-NG[1], in which many individual computational nodes, each with its own main memory, are connected to each other and can thus work on a problem in parallel, with communication possible between the nodes during the calculation. This is also called *distributed memory parallelization*.

In addition to the distributed memory parallelization, there is also the possibility of *shared memory parallelization*, in which, several threads work on a part of the overall problem, while sharing a single main memory. It must be ensured that at a given time only one processor is active at a certain location of the main memory, since race conditions can arise due to the unpredictable execution sequence of threads. However, since we are mostly focusing on distributed memory parallelization in this thesis, this will not be explained further.

[1] `https://doku.lrz.de/display/PUBLIC/SuperMUC-NG`

Part II.

Three Body Algorithms

3. Three Body Algorithms

In this chapter, we discuss various algorithms that have already been developed for three-body interactions in Molecular Dynamics Simulations. We divide them into two categories, direct and approximate algorithms. While direct algorithms compute all interactions between all particles, approximate algorithms use methods to reduce the number of force calculations. This can happen either by using a cutoff, or by considering interactions between groups of particles that are far away from each other as interactions between single particles.

3.1. Direct Approaches

3.1.1. Force Cube

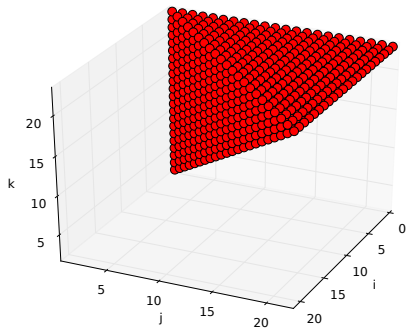


Figure 3.1.: 3D Force Cube only showing triplets with $i < j < k$.
Source: [KY14]

The Force Cube, introduced by Li et al. [LZS06c], is based on the 2D Force Matrix used by S. Plimpton [Pli95] in the force decomposition algorithm. The 2D Force Matrix, described in 2.3.1, is expanded into an $n \times n \times n$ Force Cube. Each index along the three axes of the Force Cube refers to a unique particle and therefore the elements in the Force Cube represent a triplet of particles.

In the following we describe several ways to decompose the Force Cube so that we can distribute the elements to be calculated among processors and then sum the calculated partial forces for each particle.

Note: The algorithms can be implemented either with or without using newton3. Without using newton3, all triplets (i, j, k) with $i \neq j \neq k$ are evaluated. When newton3 is used, the triplets are evaluated only in the case $i < j < k$. The FD-3 and CD-3 methods were originally not introduced using newton3 in the reference. Particle triplets are evaluated only if $i \neq j < k$ holds, to avoid the redundant computation $\vec{F}_{i \leftarrow jk}$ and $\vec{F}_{i \leftarrow kj}$. Since the methods

in subsection 3.1.1, which build on the Force Cube presented here, use newton3, we also refer to the use of newton3 in this subsection so that we can better compare the variants. In this case the Force Cube with all unique triplets can be visualized as in Figure 3.1.

Force Decomposition (FD-3)

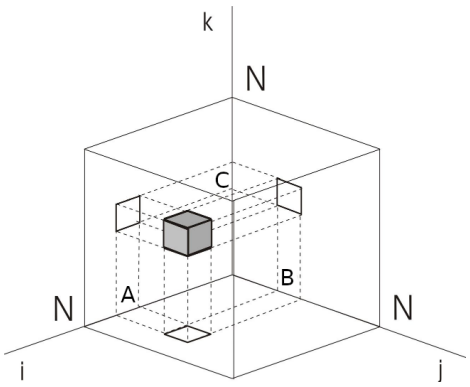


Figure 3.2.: FD-3 method
Source: [LZS06c]

In the 3D Force Decomposition method (FD-3) [LZS06c], the Force Cube is divided into m^3 subcubes. For simplicity, we assume that $m|n$. Each subcube F_{ABC} contains the Force Elements f_{ijk} with $i \in A$, $j \in B$, and $k \in C$ within its local domain, as can be seen in Figure 3.2. Thus, each subcube contains $(n/m)^3$ Force Elements.

If we look at Figure 3.1 with all Force Elements $i < j < k$, we can easily see, that a lot of subcubes are not needed. So in the first step, all these subcubes are pruned out. The remaining subcubes are assigned to processors for calculation.

Within a subcube, there can be still triplets of particles that do not fulfill the $i < j < k$ requirement. To avoid redundant particle triplet calculations within subcubes, only unique triplets are calculated, where $i < j < k$. All other Force Elements are set to zero.

After each processor has calculated the assigned Force Elements, all partial forces are summed. This is done by first summing along one axis of the Force Cube, for example, along the k axis, to obtain the sum of all Force Elements with the same i and k indices. The result is a 2D Force Matrix, which is summed again along another axis, e.g. along the j axis, to obtain the final force vector containing all summed forces for each particle [LZS06c].

Depending on the position of the subcube F_{ABC} within the Force Cube F, different computation loads result. So processors owning subcubes that have the same cube indices, i.e. $A = B = C$, which are all subcubes along the diagonal of the Force Cube have the least computation load, since they only calculate interactions between the same particle subset. Processors that own subcubes with three different cube indices $A \neq B \neq C$ have the most load as they calculate all interactions between three different particle subsets [LZS06c].

Cyclic Decompositions (CD-3, BD-3 and PD-3)

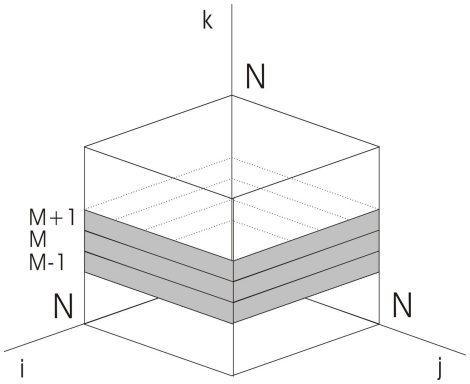


Figure 3.3.: CD-3 method
Source: [LZS06c]

CD-3: The Cyclic Decomposition (CD-3) [LZS06c] divides the Force Cube into slices, as can be seen in Figure 3.3. Slices thus correspond to a 2D Force Matrix and each of these matrices contains Force Elements with one fixed particle k and two variable particles i and j. Each of the n slices is cyclically assigned to processors, for computation, based on their rank. Processors are responsible for calculating the Force Elements in their slices, as well as updating the fixed particles of each of their slices.

Analogous to the FD-3 method, redundancies arise within each slice, since, for example, a slice for a fixed k contains the triplets (k, i, j) and (k, j, i). To avoid this redundant computation, only Force Elements with $i < j < k$ are assigned for computation.

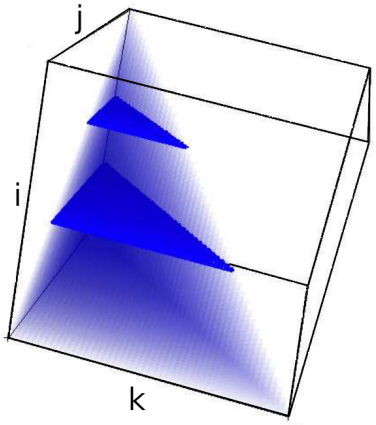


Figure 3.4.: Distribution of work among processors according to the CD-3 method, where only triplets with $i < j < k$ are calculated. The area of the opaque slices gives an impression of how the number of triplets to be calculated differs for the respective slices.
Source: [SSJ07]

The number of elements to be computed for a slice depends on the slice index, which

leads to load imbalance depending on the number of processors. Visually, one can illustrate the assignment of slices to processors by slicing up the tetrahedron in Figure 3.4 along the i-axis.

Depending on the assigned slices, processors need the particle information of other processors. Thus, in slices that contain many Force Elements, more particle information is needed from neighboring processors, or the calculated forces have to be sent back to the owner processors.

BD-3: Based on the Cyclic Decomposition from section 3.1.1, the Balanced Decomposition (BD-3) first distributes slices with odd indices cyclically to the processors, followed by slices with even indices to achieve better load balancing [LZS06b].

PD-3: Also based on the Cyclic Decomposition from section 3.1.1, the Precise Decomposition (PD-3) counts the number of elements to be computed within each slice and then assigns them to processors to ensure an almost perfect load balance [LZS06b].

Force Cube Conclusion

Summarizing the Force Cube methods, we can state that the presented algorithms of Li et al. [LZS06a, LZS06c, LZS06b] enable the parallel computation of three-body interactions by distributing Force Elements to be computed to processors. Depending on the decomposition, processors need the particle information held by other processors. Since the algorithms work without relation to the position of the particles in the physical domain, they are suitable for homogeneous, as well as for heterogeneous particle distributions.

Load Balance: Depending on the chosen type of work distribution on processors (FD-3, CD-3, BD-3, PD-3), load imbalance occurs regardless of the distribution of particles, since there are redundant triplets within the subcubes/slices that are excluded from the force calculation. Thus, the FD-3 algorithm suffers most from load imbalance because, as described above, processors with subcubes $A = B = C$ have the least work, whereas processors with $A \neq B \neq C$ have the most work. The load balance tests of Li et al. with 20 processors using the STVR show that the FD-3 algorithm has almost up to 100 percent relative deviation for some processors, whereas The STVR for all processors with CD-3 is less than 10 percent. All other variants BD-3 and PD-3 show only a slight improvement over the results of the CD-3 algorithm [LZS06c, LZS06b]. A good balancing of the computational work, which is independent of the particle distribution, can be achieved with these algorithms (PD-3), but this has to be calculated explicitly and is not implicitly given by the algorithm.

Weak Scale Efficiency: The algorithms CD-3, BD-3 and PD-3 show an efficiency of about 50 percent at 35 processors compared to the serial execution. The efficiency of FD-3 drops rapidly to below 50 percent and is under 40 percent at 35 processors [LZS06b]. We assume that for the FD-3 algorithm the unfavorable distribution of work is decisive for the rapid drop.

Cutoff Extension: An extension for a cutoff distance is not presented, but would be possible in the same way as implemented by [Pli95] in its 2D Force Matrix algorithms, by creating a neighbor list for each particle before a computation step. Since the presented algorithm is not based on a spatial space decomposition, but Force Elements are assigned independently of the particle positions to processors, each processor would have to check the distances between its own particles and all other particles whether they are within the cutoff. All-to-all communication would therefore be necessary. However, regardless of the chosen cube decomposition, additional load imbalance occurs, as further Force Elements are pruned out, which is likely to result in unacceptable load balancing, especially for the FD-3 algorithm.

Slice- and Volume Symmetric Transformations

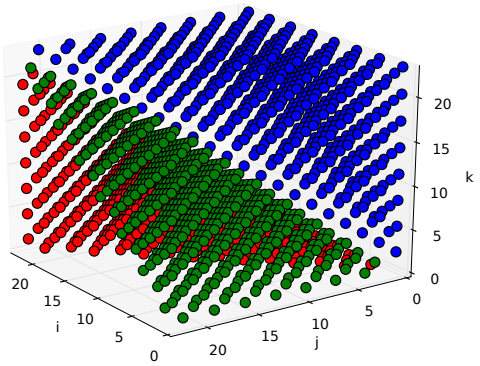


Figure 3.5.: Slice Symmetric Transformation
Source: [KY14]

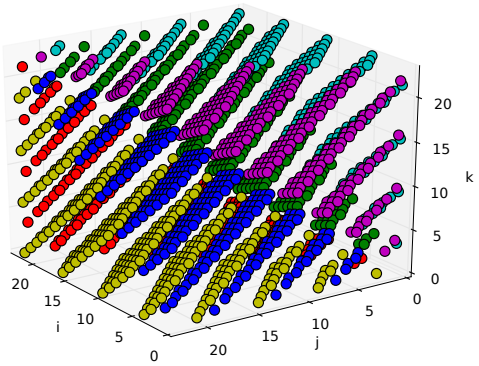


Figure 3.6.: Volume Symmetric Transformation
Source: [KY14]

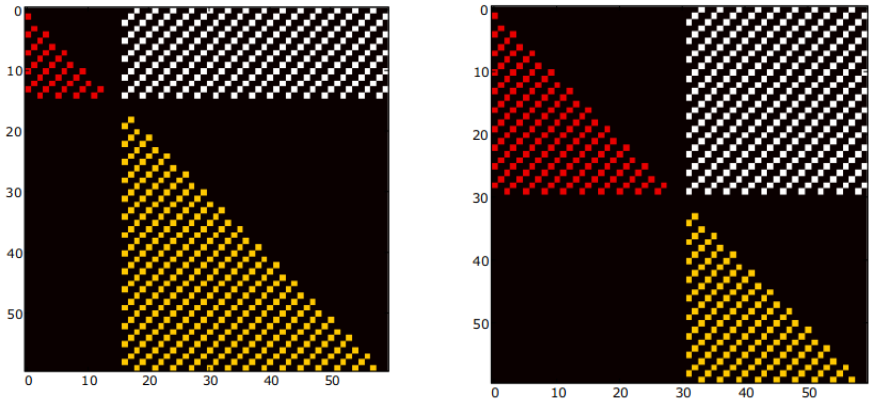
Based on the 3D Force Cube from 3.1.1, Summanth et al. [SSJ07] introduced transformations for the Force Cube such that each processor is assigned the same number of Force Elements without additional computational overhead like the PD-3 method. Two transformations are presented, one allowing slice decomposition, the other force decomposition with subcubes.

Since the Force Cube contains all possible triplet combinations and only unique triplets $i < j < k$ need to be computed in case of using newton3, the number of elements in the Force Cube is reduced, as can be seen in Figure 3.1.

In the case of the Slice Symmetric Transformation, one can imagine that the unique elements from the tetrahedron in Figure 3.1 are distributed in such a way that all slices along the k axis, or to be precise along any axis of the Force Cube, contain the same number of elements to be computed. By equation 3.1 only certain elements in a slice are assigned for computation and the tetrahedron from Figure 3.1 is virtually decomposed into three tetrahedrons, as can be seen in Figure 3.5. The formula works like a 3D checkerboard pattern, assigning a color to each unique element. Analogous to the slice based algorithms in section 3.1.1, slices can now be distributed across processors, where each processor computes

the forces for one fixed particle k and two variable particles i and j, specified by the equation 3.1 [SSJ07]. In Figure 3.7a and 3.7b we can see two slices along of a Force Cube consisting of 60 Particles.

$$F'_{ijk} = \begin{cases} F'_{ijk}, & \text{if } (i > j > k) \text{ and } (i + j + k \equiv 1 \bmod 3) \\ F'_{ijk}, & \text{if } (j > k > i) \text{ and } (i + j + k \equiv 2 \bmod 3) \\ F'_{ijk}, & \text{if } (k > i > j) \text{ and } (i + j + k \equiv 0 \bmod 3) \\ 0, & otherwise. \end{cases} \tag{3.1}$$



(a) (b)

Figure 3.7.: Two slices of the Force Cube using the Slice Symmetric Transformation. The transformation ensures that each slice has the same number of elements to be evaluated. The colors symbolize the three different tetrahedrons from which the Force Elements originate.
Source: [SSJ07]

The Volume Symmetric Transformation works essentially in the same way, but decomposes the tetrahedron from Figure 3.1.1 not into 3, but into 6 single tetrahedrons based on the formula 3.2. The transformed Force Cube is depicted in Figure 3.6. This transformed Force Cube can either be decomposed into slices (atom decomposition), or into subcubes (force decomposition) [SSJ07]. In both cases the Force Elements are evenly distributed.

$$F'_{ijk} = \begin{cases} F'_{ijk}, & \text{if } (i > j > k) \text{ and } (i + j + k \equiv 0 \bmod 6) \\ F'_{ijk}, & \text{if } (i > k > j) \text{ and } (i + j + k \equiv 1 \bmod 6) \\ F'_{ijk}, & \text{if } (j > k > i) \text{ and } (i + j + k \equiv 2 \bmod 6) \\ F'_{ijk}, & \text{if } (j > i > k) \text{ and } (i + j + k \equiv 3 \bmod 6) \\ F'_{ijk}, & \text{if } (k > i > j) \text{ and } (i + j + k \equiv 4 \bmod 6) \\ F'_{ijk}, & \text{if } (k > j > i) \text{ and } (i + j + k \equiv 5 \bmod 6) \\ 0, & otherwise. \end{cases} \tag{3.2}$$

Summary: We can state that in the Slice and Volume Symmetric Transformation, regardless of the order in which slices or volumes are distributed to processors, all processors

receive the same number of Force Elements to be computed, if the number of Force Elements is divisible by p. However, since each slice contains each particle [KY14], all-to-all communication is required so that all processors can update their particles.

Compared to the algorithms of Li et al. [LZS06c, LZS06a, LZS06b], however, load balancing is implicitly given here by the transformation and does not have to be explicitly computed as in the PD-3 algorithm, for example.

Load Balance: Summanth et al. [SSJ07] compare the CD-3 algorithm with their Slice Symmetric Transform for 30 and 50 processors with 8000 particles using the STVR. The results show that the CD-3 method has strong deviations especially for processors with very low, or very high rank, which is due to the assignment of slices to processors described in the section 3.1.1. With the help of their Slice Symmetric Transformation, the STVR can be kept much more uniform and is no longer dependent on the processor rank.

Cutoff Extension: As with the algorithms of Li et al. [LZS06c, LZS06a, LZS06b] there is also the possibility to implement a cutoff, which is mentioned but not implemented. As a result of using a cutoff, the load balancing changes, since in the individual slices or subcubes further computation elements are omitted. Furthermore, each processor would have to test the distances between its own particle against all other particles in the domain, since no spatial domain decomposition is used.

3.1.2. Shifting Algorithms

Unlike the algorithms in 3.1.1 that represent particle triplets as Force Elements and distribute them among processors, these approaches distribute particles evenly among processors within the simulation domain. Each processor is responsible for computing the interactions of its own particles with those of others. Processors are arranged in such a way that in multiple substeps within a simulation step the particle information is exchanged between processors and thus all interactions can be calculated.

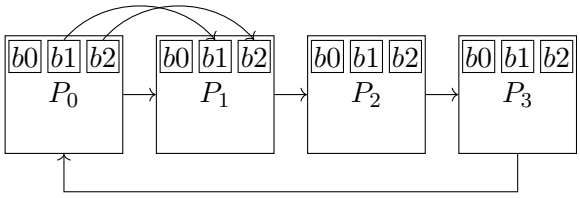
In 2014, P. Koanantakool and K. Yelick [KY14] presented a set of algorithms that allow parallel calculation of forces between particle triplets by exploiting newton3, based on those of [DGK+13] which were developed for particle pairs. In addition to algorithms for computing all triplets in the simulation domain, they also present a cell-based u-body ($u \geq 2$) algorithm with cutoff distance described in section 3.2.4. The following three paragraphs give a brief overview of how their algorithms work.

Let p be the number of processors available for the computation. For simplicity we assume that $p|n$. We arrange the processors in a virtual ring like in Figure 3.8 with left and right neighbors and assign n/p particles to each processor by dividing the set of all particles Q into p distinct subsets $\{Q_0, Q_1, \ldots, Q_{p-1}\}$. Each processor is responsible for updating the particles in its subset Q_r. In this thesis we use r to refer to the rank of a processor. The rank of a processor indicates the virtual position of a processor within all p processors. This can be either a scalar value, as for example in ring communication, but it can also be a vector, indicating for example a Cartesian coordinate.

The idea of these algorithms is that each processor holds three buffers $b0$, $b1$, $b2$ locally in memory. At the beginning of a simulation step, each processor creates three copies of its particles Q_r and stores them in $b0$, $b1$ and $b2$. During a simulation step, the buffers are shifted around the ring of processors so that each processor can form all the particle triplets from its own and other processors' particles. Particle triplets are formed by selecting one particle i from $b0$, one particle j from $b1$, and one particle k from $b2$ with $i < j < k$ to compute the forces between the three participating particles in one step using newton3.

The algorithms alternate between shifting buffers to neighboring processors and calculating the interactions between $b0$, $b1$ and $b2$ within a simulation step. Shifting and calculating takes place in a synchronized manner, so that all processors simultaneously send a buffer to a neighboring processor and receive the particles from another neighbor to then perform the force calculations between the three buffers. After all forces between unique particle triplets have been calculated from all unique particle subset combinations, the particles are sent back to their original owners, who sum the computed forces across all three buffers and then update their particles.

All algorithms presented in this section (Shifting Algorithms) refer to the ones from P. Koanantakool and K. Yelick [KY14].



000	111	222	333
003	110	221	332
002	113	220	331
001	112	223	330
000	111	222	333
030	101	212	323
⋮	⋮	⋮	⋮
010	121	232	303
000	111	222	333

Figure 3.8.: Example with 4 processors, illustrating the shifting scheme of the Naive All Triplets Algorithm

Naive All Triplets Algorithm: In the following we present the naive algorithm, which illustrates the principle but performs redundant shifts.

Initially, each processor computes the forces between all of its own particles. In a nested for loop, the outer loop shifts the buffer $b1$ exactly p times around the ring of processors, the inner loop checks if the particle triplets for one of the 6 possible permutations from the particle subsets in $b0$, $b1$ and $b2$ have already been calculated by this or another processor. If not, all unique triplets are formed from the three buffers and calculated, then buffer $b2$ is shifted. This is done exactly p times as in the outer loop. So, in total, the naive algorithm requires $p^2 + p$ communication steps and must check for each combination of three buffers whether they have already been computed or not.

The shifting scheme is illustrated in Figure 3.8, where the columns under each processor symbolize the indices of the particle subsets currently in $b0$, $b1$ and $b2$ of the respective processor. We call this an *offset-pattern* in the remainder of this thesis. It can be seen that redundant combinations arise that must be filtered out. The implementation for this algorithm is explained in more detail in chapter 4.

All Unique Triplets Algorithm: An improved version of the naive algorithm uses a modified scheme to shift buffers in the ring of processors so that no redundant shifts are performed. This scheme ensures that for a given triplet of three buffers containing the particle subsets (Q_a, Q_b, Q_c), only one unique permutation is formed over the entire simulation step. Therefore, we no longer need to check whether the combination of three buffers has already been calculated.

Just as in the naive algorithm, the new shifting scheme can be implemented with a nested for loop, where the outer loop performs $d = \lfloor p/3 \rfloor$ rounds, the inner loop performs $p - 3 * e$

iterations in round e of the outer loop. At the end of each outer loop iteration, a new buffer out of the three buffers $b0$, $b1$ and $b2$ is chosen in a round-robin manner to be shifted in the next iterations of the inner loop. We call each outer loop iteration a *Phase*.

Algorithm 1 illustrates the procedure, with the two loops mentioned above referring to those in lines 4 and 5. In the first step of Phase 0, the algorithm computes the interactions between its own particles, without shifting a buffer. In the next steps of Phase 0, $b2$ is shifted $s = p$ times in the inner loop and interactions between the three buffers are calculated. During Phase 1 the buffer $b0$ is shifted $s = p - 3$ times. In Phase 2, $b1$ is shifted $s = p - 6$ times. Then $b2$ is picked again, and so on. The algorithm runs until all d Phase have been processed. If p is not a multiple of 3, this scheme ensures that all unique, but not redundant combinations of particle subsets are formed.

Algorithm 1: All Unique Triplets Algorithm

Adapted from: [KY14, Koa17]

Input: U: set of all processor ranks, Q: set of all particles binned into p subsets $\{Q_0 \dots Q_{p-1}\}$ using an atom decomposition

Output: All particles updated in place

```
1  for r ∈ U in parallel do
2      Copy Q_r to b0, b1, b2
3      i ← 2
4      for s ← p to p%3 by −3 do
5          for j ← 0 to s - 1 do
6              if j ≠ 0 or s ≠ p then
7                  shiftRight(b_i)
8              calculateInteractions(b0, b1, b2)
9          i ← (i + 1)%3
10     if 3 | p then
11         shiftRight(b_i)
12         calculateOneThirdOfInteractions(b0, b1, b2, ⌊r/(p/3)⌋)
13     sendBackAndReceive(b0, b1, b2)
14     Q_r ← sumUp(b0, b1, b2)
15     update(Q_r)
```

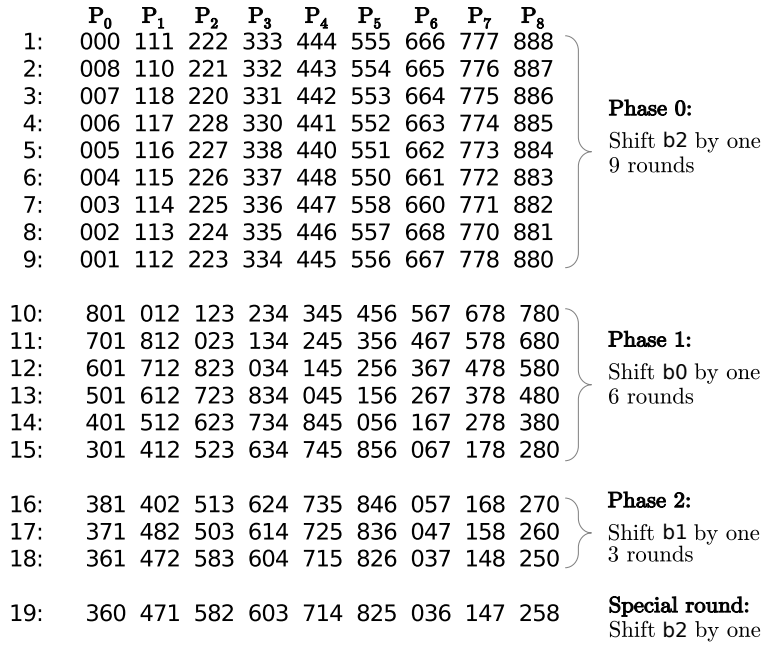


Figure 3.9.: Shifting scheme of the All Unique Tripets Algorithm with 9 processors P_0 to P_8. Each row represents a computation and shift step within a simulation step. The three columns below P_0 to P_8 represent the three buffer owner whose particles are currently located in $b0$, $b1$ and $b2$ respectively.
Source: [Koa17]

A special case arises when the number of processors is divisible by 3. In this case, an additional shift and force calculation is required after the two nested loops to form all unique combinations of particle subsets.

Figure 3.9 illustrates the resulting offset-patterns for the case $p = 9$. Since in this example $3|p$ holds, an additional step is needed to generate all unique combinations of particle subsets. Looking at the last line, we see that processors form permutations of particle subsets in this additional step. For example, P_0 generates the same combination as P_3 and P_6. P_1 generates the same combination as P_4 and P_7, and so on. For an arbitrary number p of processors, the processors P_i, $P_{i+p/3}$, and $P_{i+2p/3}$ each generate a permutation of an offset-pattern.

To avoid redundant computations, each of the three processors involved computes one third of the particle interactions within this combination of particle subsets based on the processor's rank. This means that P_i calculates the $\lfloor \frac{i}{p/3} \rfloor^{th}$ third of the interactions. In this way, load balance is ensured.

The implementation of this algorithm and division of work into thirds for the special case, is explained in more detail in chapter 4.

Embedded All Uniue Triplets Algorithm: Another improvement of the above algorithm skips the first p iterations and starts immediately with $p - 3$ iterations in the inner loop. Within these first $p - 3$ iterations the computations of several buffer combinations are included.

For this purpose, for a particle triplet (i, j, k) it must no longer necessarily hold that $i \in b0$, $j \in b1$ and $k \in b2$. In this case, the algorithm does not copy its own particles into all three buffers at the beginning, but loads directly the particles of neighboring processors into two of its buffers. For example, if its own particles are in $b1$ at the beginning, the particles of the two neighboring processors are loaded into $b0$ and $b2$, respectively. As buffer to be shifted in the first Phase, we choose $b0$. The procedure can be seen in Algorithm 2.

Algorithm 2: Embedded All Unique Triplets Algorithm

Adapted from: [KY14, Koa17]

Input: U: set of all processor ranks, Q: set of all particles binned into p subsets $\{Q_0 \dots Q_{p-1}\}$ using an atom decomposition

Output: All particles updated in place

```
1  for r ∈ U in parallel do
2      Copy Q_r to b1
3      Copy particles from left neighbor to b0
4      Copy particles from right neighbor to b2
5      i ← 0
6      for s ← p − 1 to p%3 by −3 do
7          for j ← 0 to s - 1 do
8              if j ≠ 0 or s ≠ p then
9                  shiftRight(b_i)
10             else
11                 calculateInteractions(b1, b1, b1)
12                 calculateInteractions(b1, b1, b2)
13                 calculateInteractions(b0, b0, b2)
14             if s = p − 3 then
15                 calculateInteractions(b0, b1, b1)
16             calculateInteractions(b0, b1, b2)
17         i ← (i + 1)%3
       // At this point do the same as the All Unique Triplets Algorithm
```

In the very first step, the algorithm performs all interactions between $(b1, b1, b1)$, i.e., the interactions of the own particles, all interactions between $(b1, b1, b2)$ and finally those between $(b0, b0, b2)$. During the first $p - 3$ iteration of the inner loop, the interactions between $(b0, b1, b1)$ are also calculated.

Afterwards, the algorithm continues as the All Unique Triplets Algorithm described above. The algorithm thus saves a constant factor of communication steps.

Communication Avoiding All Uniue Triplets Algorithm: To save further communication, an extension in which processors are combined into groups that work together on a particle subset is presented.

A replication factor f is used to divide the n particles into p/f uniform subsets of size $f * n/p$. The p processors are no longer arranged in a ring, but in a 2D torus with p/f columns and f rows, where a column within the torus represents a group of processors that own the same particle subset.

The algorithm works essentially in the same way as the All Unique Triplets Algorithm for p/f processors. Each column in Figure 3.9 would represent a group of processors. The computation of lines in each column is distributed over processors within the group. Since individual lines have a different computation load, a function `predictPos` determines which lines are assigned to which processor in the group based on the computation costs for the individual lines. For example, a line that computes all interactions between the same particle subset has less computation load than a line that computes the interactions between three different particle subsets.

During a simulation step, the processors exchange particles with processors from the same row of the neighboring groups, which is a horizontal communication as in the above algorithms. After all interactions have been calculated, a reduction is first performed inside the group before the particles are sent back to the owner groups.

We can visualize the save in communication if we consider again Figure 3.9. In the case of the All Unique Triplet algorithm, we use 9 processors. In the case of the Communication Avoiding Algorithm, we could use 36 processors with $f = 4$ to generate the same number of columns and rows of offset-patterns. The additional communication step in the group at the end of each simulation step is negligible compared to the communication that the All Unique Triplet Algorithm would require if we use 36 processors.

Performance: P. Koanantakool and K. Yelick [KY14] focus in their results section on the communication avoiding algorithm, in which processors work together in groups. They show how the algorithm behaves with different values for the replication factor f by running benchmarks on a Blue Gene/Q and a Cray XC30 with varying numbers of processors and particles. For the Cray XC30, a pure MPI implementation was used, in which one process is spawned per core. For the Blue Gene/Q a hybrid MPI/OpenMP implementation was used to fully utilize the 4 hardware threads.

Small scaling benchmarks were performed with 8192 particles distributed on 1024 cores on the Blue Gene/Q, and 6144 particles distributed on 1536 cores on the Cray XC30. On the Blue Gene/Q, replication factors 1 to 32 scaled in powers of two were used. On the Cray XC30 from 1 to 64 scaled in powers of two.

The results show that increasing the replication factor reduces the communication time, but after a certain point the idle time of processors increases. However, with a proper choice of the replication factor, a reduction in communication time of up to 99.98 percent can be achieved [KY14]. With the Blue Gene/Q this can be seen with the factor f=8, with the Cray XC30 with f=16.

It can also be seen that the computation time of the forces with the hybrid implementation for the replication factors $f < 4$ are higher, compared to the other replication factors, where the computation time remains constant. This is explained by the fact that the threads in the hybrid implementation have too little work. In the pure MPI implementation used for the Cray XC30, a constant computation time of the forces can be seen [KY14].

Further large scaling benchmarks with more particles and processors show the essentially same behavior as the small scaling benchmarks.

For all other algorithms no benchmarks were executed.

Summary: An advantage over the Force Cube methods FD-3, CD-3, BD-3 and PD-3 in section 3.1.1, is that they implicitly provide a nearly perfect load balance, which can be realized by a simple atom decomposition. In the case of $p|n$, each processor has exactly the same number of triplets to evaluate. But even if this does not hold, n/p or $n/p+1$ particles are distributed among processors, which does not have a large impact on the load balance.

The Slice- and Volume Symmetric Transformations from section 3.1.1 also provide a more optimal distribution of work, which is implicitly given, but the remaining properties of the Force Cube are preserved, such as the complicated communication scheme. The algorithms of P. Koanantakool and K. Yelick [KY14] offer a more intuitive and always constant communication scheme compared to the Force Cube methods, since particle subsets are always shifted only between the immediate neighbors.

Compared to most algorithms for three-body interactions, which compute the two-body forces separately from the three-body interactions, they can be easily integrated in the algorithms of P. Koanantakool and K.Yelick [KY14]. For example, in the All Unique Triplet Algorithm, two body potentials can be calculated between buffers $b0$ and $b2$ during the first Phase.

Since it is guaranteed that only 3 processors are working on a particle subset at a time, we see also a great opportunity in these algorithms to implement an efficient shared memory implementation with appropriate allocation of locks to threads.

Another advantage over the force cube methods is that this algorithm can be modified to a cell-based algorithm with spatial space decomposition, which allows the use of a cutoff that, unlike the force cube methods, does not necessarily have to compute all the distances between particles in the simulation domain, but only those to particles in neighboring cells. This algorithm is described in more detail in section 3.2.4.

3.2. Approximate Approaches

3.2.1. Multibody Multipole Methods

The Multibody Multipole Methods introduced by Lee et al. [LOG12] are approximate algorithms that consider interactions between particle clusters that are far away from each other as interactions between single particles. They are based on the Barnes-Hut algorithm, with the possibility of computing u-tuples as well. For this description we restrict this again to $u = 3$.

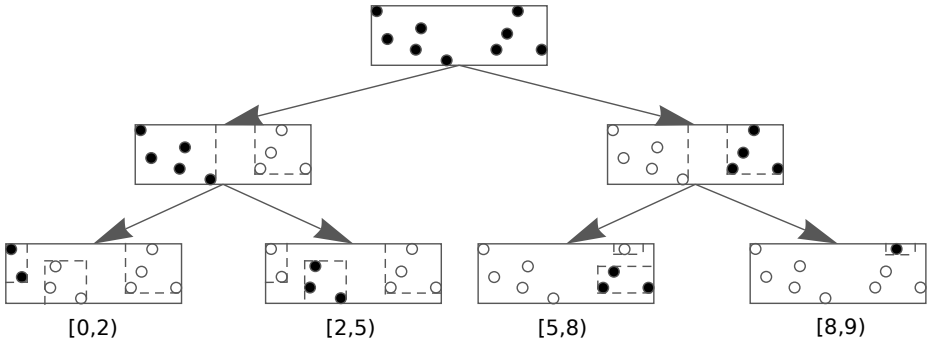


Figure 3.10.: KD-tree used by the Multibody Multipole Methods to group particles at different levels of granularity
Source: [LOG12]

The algorithms roughly follow this scheme: In the first step, a kd-tree [Ben75] is created in a recursive function that groups particles at different granularity levels. See Figure 3.10 for an example. During this recursion also so-called far field moments [LOG12] are computed for the nodes of the tree, as well as their bounding boxes. Far field moments are calculated based on the properties of the particles within each node and are used during the force calculation to treat all particles inside a node as one big particle.

In the force calculation step, the tree structure is traversed to form particle/cluster triplets. This is based on the dual tree approach of [MCG+01], so that instead of a single traversal over the tree structure to form all triplets, three traversals over the same tree structure are performed in a recursive method, which speeds up the formation of the triplets. The recursion aims to terminate as early as possible by using the far field moments for each node from the first step to check whether the result of the force calculation between three nodes is within an error tolerance specified before the simulation [LOG12]. If this is the case, the recursion can be aborted.

Instead of an extension of the Fast Multipole Methods [GR87], Lee et al. propose a simpler method based on a Monte Carlo approximation, which restricts the potentials to monotonically decreasing ones [LOG12]. The reason for this is that they use the minimum and maximum distance between the bounding boxes of the respective nodes as the basis for the approximation.

Performance: The results of Lee et al. [LOG12] were performed on a single core of an AMD Phenom II X6 1100T processor. They use 1000 up to 10000 particles, where the time

for building the kd-tree is in the range of 10^{-4} and $2 * 10^{-4}$ seconds. For the case of 1000 particles this is even less. The time for calculating the interactions is between 0.3 and 10 seconds. The results show that the cost of creating the kd-tree is negligible compared to the computation of the forces.

Three different particle distributions were used for the multibody computation time measurements: Uniformly distributed particles in the 3D unit cube, Uniformly distributed particles in the 3D unit sphere, and an annulus distribution that distributes particles as a ring in a 3D space. In the measurements, the runtime of the algorithm described above was compared with that of a naive algorithm that computes all interactions within the simulation domain [LOG12].

Compared to the naive algorithm, the measurements of the Monte Carlo based approximation for the sphere distribution showed the lowest speedup of about 50 for the two uniform distributions. The experiment with the annulus distribution performed best, achieving a speedup of 1000. For the given values, the algorithms were run with a relative error parameter of $\epsilon = 0.001$ [LOG12].

Summary: The advantage of this algorithm compared to approximate algorithms with cutoff is that it is more suitable for calculating forces between distant particle clusters, and thus for heterogeneous particle distributions. Another advantage is that the error bounds of the force calculation can be determined by the user, so that the accuracy of the result can be adapted to the particular case. A disadvantage is that no parallelization has been presented for this algorithm so far, everything is executed on one core.

3.2.2. Short Range Atom Decomposition

The cutoff algorithm of C.F. Cornwell and L.T. Wille [CW00] is based on an atom decomposition where each processor owns exactly one particle. The algorithm works without newton3 and ensures that each processor stores all particles of other processors locally, which are necessary to calculate all forces on its own particle itself. This eliminates the need to send forces back and sum them up after a simulation step.

The algorithm arranges p processors in a virtual ring. Before a simulation step, all processors create two neighbor lists for its own particle by each processor sending a copy of its own particle once around the ring of processors. In this way, each processor can calculate the distances between its own and all other particles and store a copy of each particle that is within the cutoff distance c in neighbor list NL1. Particles that are between c and $2c$ are stored in neighbor list NL2. An example can be seen in Figure 3.11, where i is the particle owned by a processor r.

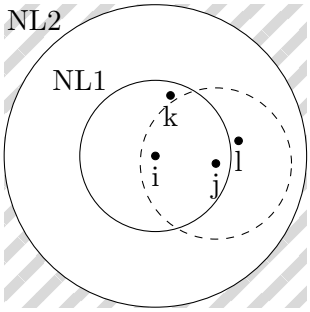


Figure 3.11.: Visualization of the two neighbor lists created by the processor that owns particle i. NL1 contains all particles that fall into the area of the inner circle with a solid line. NL2 contains the particles between the two circles with a solid line.

Cutoff Criterion: The algorithm applies the cutoff distance according to the second criterion in section 2.1.2, where at least a pair out of the three distances has to be less than c.

Pairwise Interactions: For pairwise force calculations, all impacts on i can be calculated by r forming all pairs consisting of i and another particle from NL1. In the example in Figure 3.11, these are the forces $\vec{F}_{i \leftarrow j}$ and $\vec{F}_{i \leftarrow k}$.

Three-body Interactions: For three-body interactions, this is not possible only with NL1. As an example, consider the triplet (i, j, l) with central particle j in Figure 3.11. As can be seen, it is a valid triplet according to the cutoff criterion, since i and l are both in the cutoff region of j, which is symbolized by the dashed circle. This triplet yields also forces on i. However, since processor r has no information about l without NL2, it cannot calculate the force $F_{i \leftarrow jl}$. By using NL2, all particles which are necessary to compute all forces acting on i are stored locally on processor r.

Triplet Creation: The algorithm forms all triplets that can be generated with the own particle i and all other particles j, k with $j \neq k \neq i$ in NL1 and calculates $F_{i \leftarrow jk}$. In addition, it computes all forces $F_{i \leftarrow jk}$ with a central particle j from NL1 and a particle k from NL2 so that the cutoff criterion is satisfied.

Summary: As no use is made of newton3, three partial derivatives must be calculated for a triplet, which are carried out by three different processors. However, this completely eliminates communication after a simulation step to sum up forces, since each processor calculates all impacts on its own particle itself.

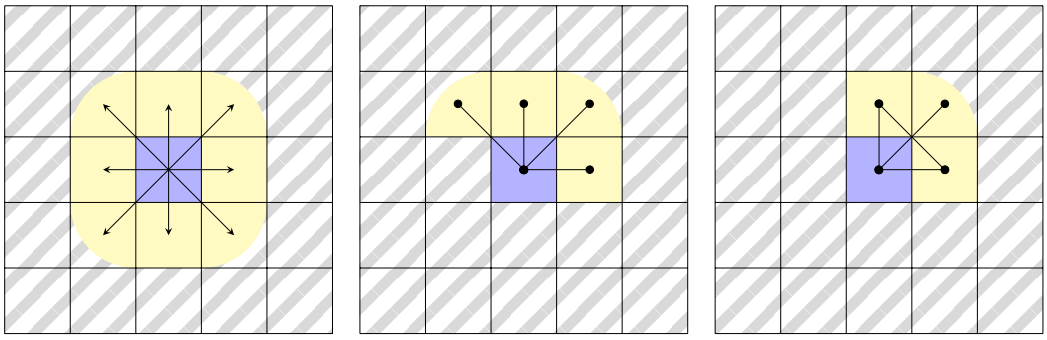
The computational overhead of the second list is within $\mathcal{O}(n)$ [CW00] and can be incorporated into the construction of NL1. All other operations during the simulation step are within $\mathcal{O}(1)$ [CW00].

The results of C.F. Cornwell and L.T. Wille [CW00] show that with increasing number of particles, and thus increasing number of processors, the total wall time remains constant using NL2. In another benchmark, the forces of all three particles involved are calculated by one processor and the results are sent back to the owner processors after a simulation step. In contrast to the use of NL2, a linear increase of the wall time can be seen here, which is caused by the communication. This algorithm can therefore be useful when the communication outweighs the computation of forces.

The idea of this algorithm can be continued by using a regular grid decomposition with linked cells instead of the atom decomposition and let processors own more particles. Neighbor lists can be updated using the linked cells, which accelerates this process. Furthermore, this algorithm can be implemented in existing frameworks that already work with neighbor lists without much additional effort.

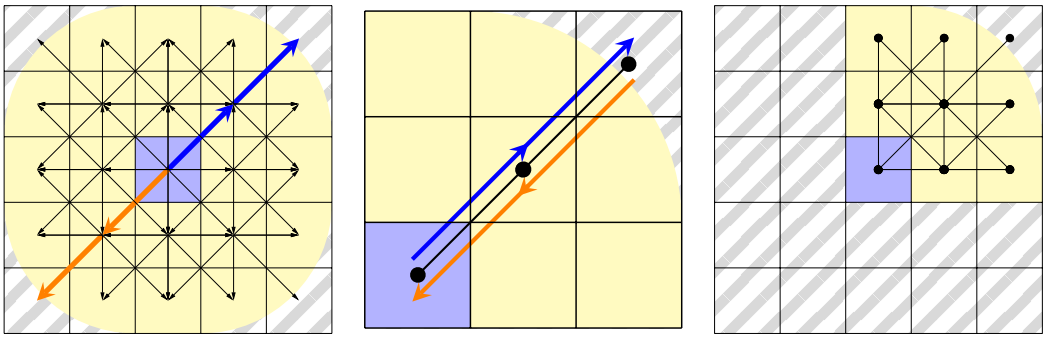
3.2.3. Shift Collapse Algorithm

In 2013, Kunaseth et al. [KKN+13] presented a parallel cell-based algorithm for the computation of u-tuples, where u is arbitrary, using a cutoff and exploiting newton3. The algorithm uses a regular grid decomposition to divide the simulation domain and assigns particles in a cell to one processor. The algorithm is based on the methods shown in Figure 3.12 to reduce the computational cost for pairwise interactions and assumes periodic boundary conditions. In the following, we refer only to the case $u = 3$.



(a) Full Shell (b) Half Shell (c) Eight Shell

Figure 3.12.: Methods for the force calculation of pairwise interactions. In Figure 3.12a, all pairwise interactions in the neighborhood covered by the cutoff radius are calculated without using newton3, which leads to double evaluation of pairs, but we have not to care about race conditions. In Figure 3.12b, newton3 is used and the pairs along the black paths are calculated. The central blue cell and the yellow cells must be protected against race conditions since particles in the blue as well as the yellow cells are updated in this step. The Eight Shell Pattern in 3.12c [BDS07] reduces the area to be protected against race conditions, while still using newton3. The central blue cell and the yellow cells must be protected against race conditions.



(a) (b) (c)

Figure 3.13.: 3.13a shows the Full Shell Pattern consisting of all paths along 3 cells outgoing from the blue one within the yellow cutoff region. The two paths in 3.13b point in opposite directions and particle triplets along these paths generate the same forces, which is why they can be combined into one path. The Eight Shell Pattern in 3.13c is the result after the shift and collapse steps have been applied to the pattern from 3.13a.

For simplicity, we assume that the simulation domain is divided into a regular grid, where the side length of the cells is chosen such that starting from a given cell we can move exactly 2 steps in each direction to neighboring cells within the cutoff distance. In Figure 3.13a this neighborhood is shown for the central blue cell and the area that can be reached within the cutoff radius from particles in this cell is marked in yellow. For a processor r that owns particles of one cell, let the set of all neighboring processors covered by the cutoff radius, including itself, be given as W_r.

Full Shell Pattern: The algorithm iteratively creates all paths $h = (r_0, r_1, r_2)$ starting from a given processor r_0, where $r_1, r_2 \in W_{r_0}$. This creates exactly one path with $r_0 = r_1 = r_2$. The second group contains all paths with $r_0 = r_1 \neq r_2$, where r_2 is a direct neighbor of r_0. For the remaining paths, $r_0 \neq r_1 \neq r_2$ holds, where r_1 is a direct neighbor of r_0 and r_2 is a direct neighbor of r_1. The set of all these paths are called Full Shell Pattern and is depicted in Figure 3.13a.

Shift: As can be seen, numerous possible paths are created starting from the central blue cell, which lead to redundant force calculations, as the same paths just in the opposite direction are created from the viewpoint of other cells. For example, if we look at the orange path in Figure 3.13a, we can easily notice that the same path, only in the opposite direction, is also generated if we would consider the lower left cell as the central one, thus a particle triplet (i, j, k) along the orange path would be formed in reverse order by the lower left cell. Since we use newton3 and calculate all forces on i, j and k in one step, both orders produce the same results. Analogous to the methods for pairwise interactions, paths can be shifted by relaxing the owner-compute rule [KKN+13]. Thus, the orange path can be shifted to the upper quadrant in 3.13a.

Collapse: Figure 3.13b shows the upper right section from the Full Shell pattern 3.13a. The blue and orange paths in this Figure point in the opposite direction and particle triplets

formed from cells along the two paths generate the same forces, in the case of using newton3 [KKN+13]. To avoid this redundancy, paths pointing in the opposite direction are collapsed, which produces the black path that has no specified direction as a result.

Computation Pattern: The set of all unique paths remaining after shifting and collapsing is called a computation pattern in their terminology [KKN+13] and can be computed offline for a given cutoff and decomposition as offset vectors before the simulation.

Using the Computation Pattern: During the simulation, the computation pattern is used for each cell to traverse neighboring cells and form triplets of particles for computing the forces. Triplets are formed by selecting one particle from each cell referenced in the respective path. This can be executed in parallel, so that the computation pattern for multiple cells is executed simultaneously by multiple processors.

Performance: In their benchmarks, Kunaseth et al. [KKN+13] compare 3 different algorithms in a distributed memory environment: FS-MD, SC-MD and Hybrid-MD, where FS-MD uses the Full Shell Pattern, SC-MD the computation pattern from the algorithm presented above and Hybrid-MD a combination of the Full Shell Pattern with Verlet Lists.

For the strong scale benchmarks, $7.7 * 10^5$ uniform distributed particles are used. The result shows an almost perfect scaling behavior with 92.6% efficiency compared to the ideal scaling for the SC-MD algorithm on an Intel Xeon cluster at 768 processors. The other two variants, on the other hand, already show a drop in speed up from about 100 processors and only achieve an efficiency of 24.5% (FS-MD) and 17.1% (Hybrid-MD) at 768 processors.

Furthermore, the runtime for a simulation step of the three variants with 576 processors on the Intel Xeon cluster and different particle densities from 1 to 2500 particles per processor is compared. The SC-MD algorithm has always a significantly lower runtime compared to the FS-MD algorithm, which can be explained by the reduced import volume. Until 2095 particles per core the SC-MD algorithm is faster than the Hybrid-MD. From this point on, however, the Hybrid-MD algorithm performs better, which can be explained as follows: The SC algorithm reduces the import volume, which remains constant as the number of particles within the cells increases, whereas the Hybrid-MD algorithm reduces the number of particle triplets through the neighbor list, which has a beneficial effect on the runtime when the particle distribution is denser [KKN+13].

Summary: Since the algorithm builds upon the methods from Figure 3.12 it can be easily implemented in frameworks that already work with these techniques for two-body potentials. As mentioned before, this computation pattern can be created offline for one or more fixed u, for example $u = 3$, or $u = 2$, and a given cutoff, which allows the integration of different u-body terms of a potential in addition to 3 body interactions. The algorithm can be implemented in both shared and distributed memory environments. In the case of the shared memory implementation, the Eight Shell Pattern reduces the area to be protected against race conditions. Furthermore, by reducing the number of computation paths, the time required to form unique cell combinations is completely eliminated, since such a computation pattern contains only unique paths.

3.2.4. Cutoff Triplet Algorithm

Based on the algorithms from section 3.1.2, the extension of P. Koanantakool and K. Yelick [KY14] mentioned above is presented here, which allows the use of a cutoff radius and is suitable for simulations with periodic boundary conditions. It can be implemented for arbitrary u-body interactions, but in the following we restrict this to $u = 3$. It is a cell-based algorithm that subdivides the physical simulation domain into a regular grid and assigns the particles within a cell to one processor.

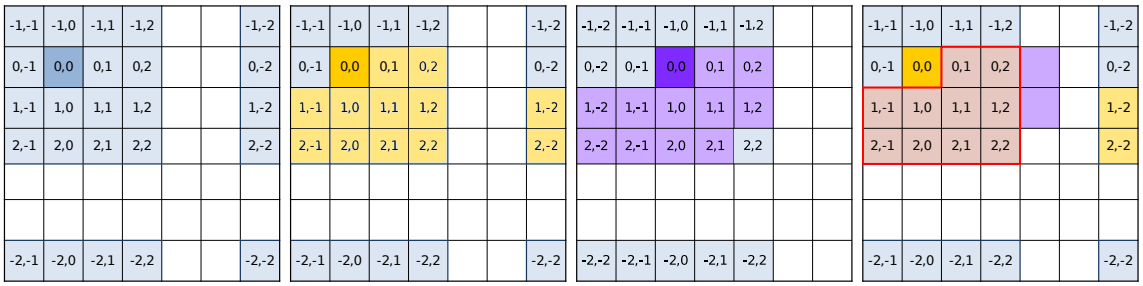


Figure 3.14.: Illustration of the Cutoff Triplet Algorithm using a 2D regular grid decomposition with $dim_x = dim_y = 7$ and a cutoff that spans $q_x = q_y = 2$ cells. Note that the x-axis is here at the vertical, the y-axis at the horizontal. The numbers within the cells represent relative periodic differences starting from a cell, where the first number represents the x coordinate, the second the y coordinate.
Sources: 3.14a: [KY14]; 3.14b, 3.14c, 3.14d adapted from [KY14]

In the following, we consider the 2D domain in Figure 3.14 to describe the algorithm. In a 3D decomposition, the algorithm is based on the same scheme.

Let the number of subdivisions of the simulation domain along each dimension be given by dim_x and dim_y. The number of processors is given by $p = dim_x * dim_y$ and processors are virtually arranged with Cartesian coordinates within the simulation domain.

Let q_x be the number of neighboring cells along the x-dimension covered by the cutoff radius:

$$q_x = \left\lceil \frac{c * dim_x}{d_x} \right\rceil \tag{3.3}$$

Where d_x is the physical domain size along the x-dimension. Analogously, q_y can be calculated.

In order to avoid overlapping of neighboring cells, the following restriction must hold [Koa17]:

$$1 < q_x < \frac{dim_x}{2} \tag{3.4}$$

Analogously, the same must hold for q_y.

Define the set of all processor ranks as:

$$U = \{(x, y) \in \mathbb{Z}^2 \mid 0 \leq x < dim_x \wedge 0 \leq y < dim_y\} \tag{3.5}$$

Let $r \in U$ be the rank of a given processor and the cutoff window w_r be all cells that we need to compute interactions in the given cutoff radius for rank r. In Figure 3.14a, the light blue surrounding represents the cutoff window w_r of the processor $r = (1, 1)$. Note: The coordinates in Figure 3.14a are relative to $r = (1, 1)$, so the dark blue cell with (0,0) actually corresponds to rank (1,1).

If one now computes all interactions between triplets of cells that can be formed in w_r, redundancy arises when using newton3, since other processors would generate a permutation of cells analogously as explained in the **Shift**-paragraph of section 3.2.3.

To avoid this, a processor r considers only ranks in w_r, that have a greater or equal periodic difference relative to its own in lexicographic manner.

The periodic difference of two ranks $a, b \in U$ in x-dimension is defined as:

$$a_x \ominus b_x = \begin{cases} a_x - b_x, & \text{if } |a_x - b_x| \leq \frac{dim_x}{2} \\ sgn(b_x - a_x) * (a_x \circ b_x), & otherwise. \end{cases} \tag{3.6}$$

Where $a_x \circ b_x$ is the periodic distance in x-dimension between a and b:

$$a_x \circ b_x = \big(min(|a_x - b_x|, dim_x - |a_x - b_x|)\big) \tag{3.7}$$

The periodic distance and difference in the y-dimension can be calculated analogously.

A processor rank $r1 \in w_r$ is now only considered if:

$$(r_x \ominus r1_x > 0) \vee (r_x \ominus r1_x = 0 \wedge r_y \ominus r1_y \geq 0) \tag{3.8}$$

In the following, we denote the set of all these ranks as wl_r. In Figure 3.14b, this range is shown in yellow for processor $r = (1, 1)$. Similarly, in Figure 3.14c, the range for processor $r = (1, 2)$ is shown in purple. The coordinates within the cells represent the periodic differences to the neighborhood in w_r.

Algorithm 3: Cutoff Triplet Algorithm

Adapted from: [KY14, Koa17]

Input: U: set of all processor ranks, Q: set of all particles binned into p subsets $\{Q_0 \dots Q_{p-1}\}$ using a regular grid decomposition

Output: All particles updated in place

```
1 for r ∈ U in parallel do
2     for r1 ∈ wl_r do
3         for r2 ∈ wl_r ∩ wl_r1 do
4             Calculate all interactions between particles from r, r1 and r2
5     Update particles of r
```

The algorithm is executed in parallel for each rank $r \in U$ by a processor, as shown in Algorithm 3.

The loop in line 2 iterates in lexicographic order over all ranks $r1 \in wl_r$ and imports their particles. A rank $a \in wl_r$ is lexicographically less than $b \in wl_r$ with respect to r if:

$$((r \ominus a)_x < (r \ominus b)_x) \vee ((r \ominus a)_x = (r \ominus b)_x \wedge (r \ominus a)_y < (r \ominus b)_y) \tag{3.9}$$

The inner loop in line 3 iterates starting from $r1$ in lexicographic order over ranks $r2 \in wl_r \cap wl_{r1}$, imports their particles and calculates all interactions between particles from r, $r1$ and $r2$.

For example, assume that the algorithm is executed from the viewpoint of $r = (1, 1)$ and the loop in line 2 imported the particles from $r1 = (1, 2)$ in this substep. Thus, the inner loop in line 3 would iterate over all ranks $r2$ that are within the red highlighted area in Figure 3.14d.

The given pseudo-algorithm represents the high-level procedure and can be implemented in practice like the direct algorithms from section 3.1.2 with 3 buffers, where $b0$ always contains the particles of the respective processor, $b1$ and $b2$ are shifted between processors during the simulation step. At the beginning, each processor copies its own particles from $b0$ to $b1$ and $b2$. The loop in line 2 of the pseudo-algorithm would shift buffer $b1$, the loop in line 3 would shift $b2$. Since all processors arrange their neighbors in the same lexicographic manner, a uniform scheme can be used to shift buffers. Consider again the processor with rank $r = (1, 1)$ and suppose it imports the particles from processor $(1, 2)$, it would send the current particles to processor $(1, 0)$, which corresponds to rank $(1, 2)$ mirrored by its own rank $(1, 1)$. Details of this implementation in C++ are explained in chapter 4.

Summary: The following properties can be stated for this algorithm: Since it is a cell-based algorithm, the number of assigned particles depends on their distribution within the simulation domain, which is reflected directly in the load balance among processors. However, since only particles from neighboring cells covered by the cutoff radius are imported, the number of distance checks is reduced in contrast to algorithms that distribute particles to processors without relation to their position in the simulation domain. Thus, the computation of distance checks can be significantly reduced compared to the algorithm from section 3.2.2, which uses an atom decomposition. The same applies analogously to the force cube algorithms, which would have to check all distances in the case of a cutoff implementation.

Analogous to the All Unique Triplet Algorithm from section 3.1.2, no redundant shifts are performed, since the lexicographic filtering and ordering ensures that a given combination of particle subsets Q_a, Q_b, Q_c is formed only once during the simulation step. Therefore, each shift is necessary and leads to one computation step.

One advantage over the Shift Collapse algorithm is that this algorithm already provides a scheme for exchanging particle subsets between processors, ensuring that only one processor is working on one of the three copies of a particle subset at any given time. The Shift Collapse

algorithm, on the other hand, computes only the set of all cell combinations, so there is no redundancy when using newton3. However, a scheme for processing the computation pattern is not provided by the Shift Collapse algorithm.

Just as mentioned earlier for the direct algorithms from section 3.1.2, pairwise interactions can be easily integrated and a distributed as well as a shared memory implementation is possible.

3.3. Summary

In this chapter, we presented several algorithms that enable the efficient computation of three-body potentials for Molecular Dynamics Simulations in high performance computing. Most algorithms divide the problem to be computed into parts and distribute them across multiple processors to speed up the computation compared the serial execution. We have presented direct algorithms in section 3.1 that compute all triplet interactions between all particles in the simulation domain and approximate ones in section 3.2 that either use cutoff or treat distant particle clusters as one big particle.

In the group of the direct ones, we have presented algorithms in sections 3.1.1 that work on the basis of the Force Cube and distribute particle triplets to be computed among several processors. Furthermore, we have described a number of shifting algorithms in section 3.1.2, in which particles are distributed among several processors and exchanged between the processors during a simulation step in order to compute all interactions.

We presented an approximate algorithm in section 3.2.1 based on the Barnes Hut algorithm, which is suitable to compute three-body interactions.

Furthermore, we have presented an algorithm in section 3.2.2 that uses two neighbor lists, so that without using newton3 all impacts on a particle i can be calculated on the respective processor that is responsible for particle i.

Last but not least, we presented two cell-based algorithms, where the first algorithm in section 3.2.3 builds upon existing concepts for pairwise interactions and creates a set of unique paths to neighboring cells to avoid redundant cell combinations and thus force calculations. Building on the idea of the first, the second algorithm in section 3.2.4 provides the ability to form all necessary cell combinations without computing such paths by having all processors arrange their neighborhoods in lexicographically the same way. By shifting particle information to neighboring processors, all necessary particle triplets can also be formed in this way. Both algorithms make use of newton3.

Based on this literature review, we conclude that the direct and cutoff shifting algorithms of P. Koanantakool and K. Yelick [KY14] are currently the state-of-the-art algorithms for three-body calculations in MD simulations, since they follow an intuitive scheme and still bring a lot of positive features compared to the other algorithms.

Compared to the Force Cube methods, the direct algorithms of P. Koanantakool and K. Yelick [KY14] implicitly provide an ideal load balance, furthermore the summation of the

particles is easier to realize than in the Force Cube methods, because at the end of each simulation step only three buffers have to be summed up.

Since their cutoff extension is cell-based, fewer distance checks are required compared to the Short Range Atom Decomposition algorithm and the Force Cube algorithms with cutoff.

Compared to the Shift Collapse algorithm, the cutoff extension of P. Koanantakool and K. Yelick [KY14] provides a scheme for processing neighboring cells, which is why we prefer this algorithm.

Furthermore, the algorithms use the newton3 optimization, while still keeping communication at a minimum.

Both the direct and the cutoff algorithms offer the possibility of a communication avoiding implementation, which stores more particles per processor by a replication factor to further save communication.

Last but not least, we can state that the algorithms can be implemented for distributed memory environments, shared memory environments or hybrid.

Part III.

Implementation and Results

4. Implementation

In this chapter we present the implementation of three algorithms described in chapter 3 in C++. We have chosen the algorithms of P. Koanantakool and K. Yelick [KY14] because they present both direct and approximate algorithms with cutoff distance based on the same concept. In this way, we can gain insight into both methods and better compare the results of the direct and cutoff algorithms to see the effects of introducing a cutoff. Furthermore, these algorithms work with the newton3 optimization, while still keeping communication at a minimum, except the Naive All Triplet Algorithm. We have chosen the cutoff algorithm of P. Koanantakool and K. Yelick [KY14] over the Shift Collapse algorithm from [KKN+13] because it already provides a scheme for processing the neighboring cells, with only one processor working on one of the three copies of a particle subset at a time.

Within the presented algorithms we restrict ourselves in this thesis to the Naive All Triplets Algorithm, the All Unique Triplets Algorithm and the Cutoff Triplet Algorithm. The algorithms will be abbreviated as NATA, AUTA, and CTA respectively in the following. All algorithms are implemented in SPMD fashion, where SPMD stands for Single Program Multiple Data. The algorithm is executed on several processors in parallel. Each processor gets exactly one piece of the problem to be computed and executes the algorithm on it. Since particle information must be exchanged between processors in the course of a simulation step, communication between them is required. We use the MPI (Message Passing Interface) standard to enable communication between processors. As a concrete implementation of the MPI standard, we use OpenMPI[1]. In some specific sections of the code we use the linear algebra library Eigen[2].

The execution of a simulation is essentially the same for all three algorithms and can be visualized with the Figure 4.1. All steps marked in yellow are not directly related to the implementation of an algorithm, but are required for its execution. The central node in purple describes a simulation step that is executed with one of the three algorithms. As described in section 3.1.2 and 3.2.4, all three algorithms alternate between exchanging particles and computing interactions between three particle subsets. In the following, a force calculation and communication step is referred to as a substep.

[1] `https://www.open-mpi.org/`
[2] `https://eigen.tuxfamily.org/index.php?title=Main_Page/`

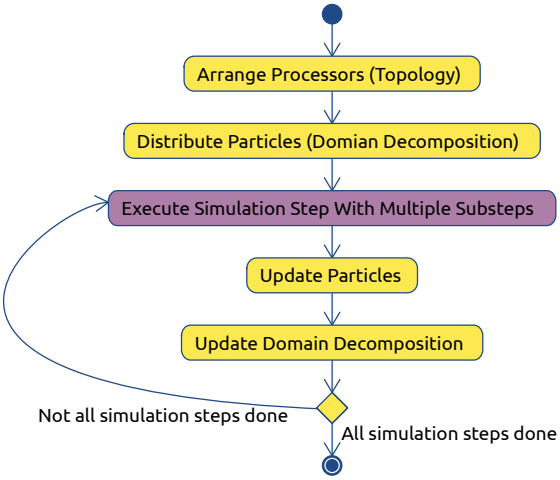


Figure 4.1.: Overview of a simulation flow as we use it in our implementation

4.1. Framework

Before going into the implementation of the three algorithms, we first briefly present the implementation of classes, methods, and structs that are required by all algorithms to perform a simulation step.

Particle: A particle is represented by a C++ struct that stores ID, position, velocity, acceleration, force and mass. An additional boolean value is used to symbolize a dummy particle. The struct implements the `Update` method, which uses a Velocity Verlet Integration to update the particle's position, velocity, and acceleration after a simulation step. The method `GetSqrDistPeriodic` calculates the periodic squared Euclidean distance within the entire simulation domain of the particle to another particle. The periodic distance in one dimension is defined as

$$dist_p(x, y) = min(|x - y|, d - |x - y|) \tag{4.1}$$

, where d is the domain size in one specific dimension. For vectors we apply equation 4.1 component-wise and calculate the squared Euclidean 2-norm from it. We use this later in the CTA algorithm to test against the squared cutoff distance. The method `ResetForce` is called before each simulation step for all particles, so that the respective algorithm can calculate the forces for the upcoming simulation step. The `GetMPIType` method returns a `MPI_Datatype` that describes the size and contents of the struct. To store particles, we use a `std::vector<Particle>`.

```
1  struct Particle {
2      int ID;
3      double pX, pY, pZ;
4      double vX, vY, vZ;
5      double aX, aY, aZ;
6      double fX, fY, fZ;
7      double mass;
8      bool isDummy;
9
10     void Update(double dt, Eigen::Vector3d gForce){
11         // update particle position, velocity and acceleration with a Velocity
               Verlet Integration using the calculated values fX, fY and fZ
12     }
13     void ResetForce(){fX=fY=fZ=0;}
14     double GetSqrDistPeriodic(Particle o, Eigen::Array3d physicalDomainSize){}
15     static MPI_Datatype GetMPIType(){// returns a MPI_Datatype that describes
           all contained Data in this struct}
16 }
```

Listing 4.1: C++ struct that describes a particle

Potential: To calculate the forces between a given particle triplet (i, j, k), we use the Axilrod-Teller potential, as this is a common potential for short-range Molecular Dynamics Simulations. We implement the base class `Potential`, which declares a virtual method `CalculateForces`. The `AxilrodTeller` class implements the `CalculateForces` method using the proposed implementation of G. Marcelli [Mar01]. It computes, for a given particle triplet (i, j, k), all the resulting forces on i, j and k in one step using newton3. We do not go into detail about the implementation, as this would go beyond the topic of this thesis.

Topology: To enable communication between processors, we implement the class `Topology`, which uses the MPI command `MPI_Cart_create`[3] to virtually arrange processors periodically in either one, two, or three dimensions. The command returns a new MPI communicator in which a specific rank (integer value) is assigned to each participating processor. With the help of the method `MPI_Cart_coords`[4], the virtual position of a processor can be determined as a Cartesian coordinate on the basis of the rank. The class `Topology` implements the methods `GetLeftNeighbor(int dim)` and `GetRightNeighbor(int dim)`, which return the respective processor rank of the neighboring processor in a given dimension using the MPI command `MPI_Cart_shift`[5]. Furthermore, the methods `GetWorldSize` and `GetWorldRank` are implemented, which return the total number of processors and the own rank within all processors. For this purpose the MPI commands `MPI_Comm_rank`[6] and `MPI_Comm_size`[7] are used. In the following, the own rank of a processor is denoted by `worldRank`, the number of all processors by `worldSize`.

Domain Decomposition: To distribute particles within the simulation domain among processors before the simulation starts, we use a simple atom decomposition for the NATA and AUTA algorithms, which assigns $\frac{n}{p}$ particles to each processor based on its rank,

[3] https://www.open-mpi.org/doc/v4.1/man3/MPI_Cart_create.3.php
[4] https://www.open-mpi.org/doc/v4.1/man3/MPI_Cart_coords.3.php
[5] https://www.open-mpi.org/doc/v4.1/man3/MPI_Cart_shift.3.php
[6] https://www.open-mpi.org/doc/v4.1/man3/MPI_Comm_rank.3.php
[7] https://www.open-mpi.org/doc/v4.1/man3/MPI_Comm_size.3.php

regardless of particle positions. For the CTA algorithm, we use a regular grid decomposition with periodic boundary conditions, which evenly divides the simulation domain into a grid and assigns the particles of each cell to a processor based on its virtual position assigned by the Topology. In the case of a regular grid decomposition, after each simulation step the method `RegularGrid::Update` is called, which exchanges particles between neighboring processors if they have moved beyond the local domain. We do not go into more detail about this implementation, since it is not directly related to the algorithms, but only serves as a tool for executing a simulation step. It is based on the spatial space decomposition algorithm of S. Plimpton [Pli95].

An overview of the classes is shown in the Figure 4.2. Note: This diagram is used only to visualize the most necessary relationships between classes and their methods. Some associations and auxiliary methods are not shown here.

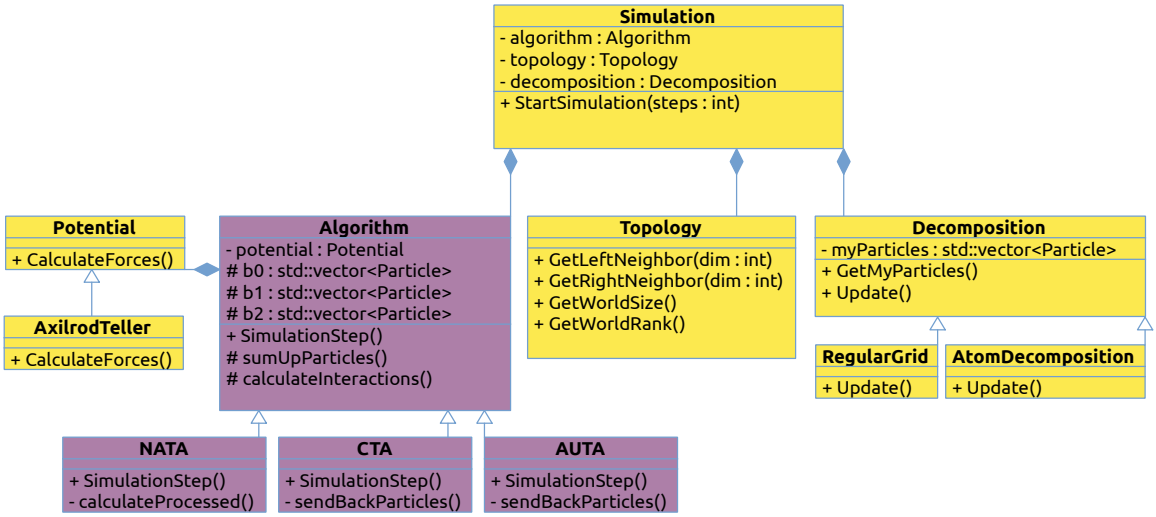


Figure 4.2.: Simplified class diagram that provides a rough structure about our implementation. The classes in purple refer directly to the algorithms whose implementation is presented in this chapter.

4.2. Algorithms

To be able to reuse program code, we use inheritance and implement certain methods shared by all three algorithms in the base class `Algorithm`. This base class implements the methods `calculateInteractions`, which calculates all unique particle interactions with three given buffers. The pseudo-algorithm is shown in Algorithm 4. Furthermore, the base class implements `sumUpParticles`, a method that sums the forces of all particles from three given buffers. A virtual method `SimulationStep` is implemented by the three derived classes (`NATA`, `AUTA` and `CTA`), which perform a simulation step based on the respective algorithm.

Algorithm 4: The algorithm generates all unique triplets between particles from three given buffers $b0$, $b1$ and $b2$. If two or three buffers have the same owner (i.e. the same particle subsets), the index of the respective loop is adjusted so that only different particles are used within a particle triplet to satisfy the requirement ($i < j < k$). The function `checkCutoff` tests in the case of the CTA algorithm whether the distances between the three particles are within the cutoff radius.

Input: Particlesubsets in b0, b1, b2 and owner ranks of b0, b1, b2
Output: calculated forces between all possible particle triplets in place

```
1  for i ← 0 to b1.size() - 1 do
2      if b0[i].isDummy then
3          continue
4      b1LoopIndex ← 0
5      if b1Owner == b0Owner then
6          b1LoopIndex ← i + 1
7      for j ← b1LoopIndex to b1.size() - 1 do
8          if b1[j].isDummy then
9              continue
10         b2LoopIndex ← 0
11         if b2Owner == b1Owner then
12             b2LoopIndex ← j + 1
13         else if b2Owner == b0Owner then
14             b2LoopIndex ← i + 1
15         for k ← b2LoopIndex to b2.size() - 1 do
16             if b2[k].isDummy then
17                 continue
                   // only relevant for the cutoff algorithm
18                 if not checkCutoff(b0[i], b1[j], b2[k]) then
19                     continue
20                 potential.CalculateForces(b0[i], b1[j], b2[k])
```

4.2.1. Naive All Triplets Algorithm

The Naive All Triplets Algorithm can be implemented with a nested `for` loop as described in 3.1.2. The pseudo-algorithm for NATA is shown in Algorithm 5.

Simulation Step: In the function `SimulationStep`, the particles in $b0$ are copied to $b1$ and $b2$ using the vector assignment operator[8]. The outer `for` loop in line 17 has `worldSize` iterations and shifts the particles in buffer $b1$ to the right neighbor in each substep. The inner loop in line 18 first checks whether the particle interactions between the three current particle subsets in $b0$, $b1$ and $b2$ have already been calculated or not, calculates them if necessary and shifts the particles in $b2$ to the right neighbor. Since the two loops shift exactly `worldSize` times around the ring of processors, each processor ends up with its original particles in $b1$ and $b2$, making the `sendBackParticles` step unnecessary. In the final step of a simulation step, the partial forces in $b1$ and $b2$ are added to those in $b0$ and the `Particle::Update` method is called for all particles.

calculateProcessed: Since the naive algorithm generates redundant buffer combinations during a simulation step, each processor stores all particle subset combinations that have already been computed in the list `alreadyProcessed`, which is cleared at the beginning of each simulation step. To check whether a combination of three given particle subsets has already been evaluated, or is computed in this substep by this or another processor, the function `calculateProcessed` in Algorithm 5 computes the owner ranks of the particle subsets located in $b0$, $b1$ and $b2$ of processor i from the perspective of processor i, creates a `Triplet` struct from it and stores it in a C++ `std::vector<Triplet>`. The triplet struct stores three indices and overrides the `==` operator which checks for a given triplet whether it is a permutation of its own stored indices. If this combination has not yet been computed, true is returned only if i matches the own `worldRank`. This prevents that more than one processor computes the same particle triplets, if in a substep multiple processors generate a permutation of the same buffer combination. Only the processor with the lowest rank out of all processors that have generated a permutation computes the interactions. All other processors are idle during this substep.

[8] `https://en.cppreference.com/w/cpp/container/vector/operator%3D`

Algorithm 5: Naive All Triplets Algorithm

Input: decomposition: An atom decomposition that has all particles divided into n/p equal subsets

Output: particles of this processor updated in place

```
1  Function calculateProcessed(step, calculate):
2      for i ← 0 to worldSize - 1 do
3          b1Rank ← mod(i − (step/worldSize), worldSize)
4          b2Rank ← mod(i − step, worldSize)
5          t ← Triplet(i, b1Rank, b2Rank)
6          if not processedContainsPermutation(t) then
7              processed.push_back(t)
8              if i == worldRank then
9                  calculate ← true
10 Function SimulationStep():
11     b0 ← decomposition.getMyParticles()
12     b1 ← copy(b0)
13     b2 ← copy(b0)
14     b1Owner ← b2Owner ← worldRank;
15     alreadyProcessed.clear()
16     step ← 0
17     for i ← 0 to worldSize - 1 do
18         for j ← 0 to worldSize - 1 do
19             calculate ← false
20             calculateProcessed(step, calculate)
21             if calculate then
22                 CalculateInteractions( b0, b1, b2, worldRank, b1Owner,
                    b2Owner)
23             if worldSize > 1 then
24                 b2Owner ← shiftRight(b2, b2Owner)
25                 step ← step + 1
26         if worldSize > 1 then
27             b1Owner ← shiftRight(b1, b1Owner)
28     sumUpParticles()
29     decomposition.setMyParticles(b0)
```

shiftRight: The exchange of data is implemented within the method `shiftRight` using the MPI command `MPI_Sendrecv_replace`[9], which sends the data in a local buffer to one processor, receives data from another processor and stores it in the same buffer in a single step. The implementation is shown in Code Snippet 4.2. As parameters we pass a pointer to the respective buffer, the number of elements in it, the `MPI_Datatype` for the particle struct, the receiver and sender rank, as well as the communicator, which has been created by the topology object at the beginning. To keep track of the original owner of a buffer that is shifted around processors during the simulation step, we pass the owner rank as the `MPI_TAG` parameter for the sending operation and use an `MPI_Status`, so we can return the owner of the received buffer. The MPI instruction `MPI_Sendrecv_replace` saves us work by internally taking care of the sequence of the send and receive operation, thus preventing deadlocks, which we would have to solve manually. An example of how to handle this manually would be to have all processors with even rank send first and all with odd rank receive and vice versa. A restriction of this function is that the number of sent and received elements must be equal, which is why in this algorithm, and also in AUTA algorithm, we use dummy particles in the decomposition at the beginning of the simulation, so that each processor can store the same number of particles even if $p|n$ does not hold. However, at most one dummy particle is used per buffer, so this does not have a large impact on runtime.

```
1 int shiftRight(std::vector<Particle>& buf, int owner)
2 {
3     MPI_Status status;
4     MPI_Sendrecv_replace(buf.data(), buf.size(),
5                          *mpiParticleType, rightNeighbor, owner,
6                          leftNeighbor, MPI_ANY_TAG,
7                          ringTopology->GetComm(), &status);
8     return status.MPI_TAG;
9 }
```

Listing 4.2: Implementation of the buffer exchange using the MPI command `MPI_Sendrecv_replace`, which internally takes care of the send and receive sequence. We use the `MPI_Tag` to send along the original owner of a buffer.

[9] https://www.open-mpi.org/doc/v4.1/man3/MPI_Sendrecv_replace.3.php

4.2.2. All Unique Triplets Algorithm

In this section, we present our implementation of the AUTA algorithm, which, unlike the NATA algorithm, does not perform redundant shifts. As explained in section 3.1.2, this algorithm also uses a double nested `for` loop. However, $b0$ is not fixed in this algorithm, but is also shifted around the ring of processors during a simulation step. At the end of each outer loop iteration, a new buffer is chosen to be shifted in the next Phase. The pseudo-algorithm is shown in Algorithm 6. In our implementation, the function `pickBuffer` returns a pointer to the buffer b_i based on the given value i. The `getOwner` function returns a reference to the respective variable `b0Owner`, `b1Owner` or `b2Owner` based on the parameter i. In the first substep, no shift is performed, only a force calculation step. For each further substep a shift is executed before the force calculation. By this shifting scheme the algorithm generates an offset-pattern as shown for 9 processors in Figure 3.9.

Special Case: To cover the special case if $3|p$ holds, we implement the function `calculateOneThirdOfInteractions` which, based on the rank of a processor, splits the work of the three participating processors P_a, P_b, P_c that have formed a permutation of an offset-pattern. The idea is that each processor first arranges its buffers in the same way so that $b0$ of P_a is equal to $b0$ of P_b and $b0$ of P_c. The same applies to $b1$ and $b2$ of the respective processors. Then, based on the `thirdID`, each processor is assigned a part of the outer loop in Algorithm 4. If the number of particles in $b0$ cannot be divided by 3, processor P_c is assigned the remaining part. We modify Algorithm 4 so that we can pass a starting value and a number of steps for the loop index i.

shiftRight: To shift buffers we use the same procedure as shown in Code Snippet 4.2.

sendBackParticles: Unlike in the NATA algorithm, the processors do not necessarily have their own buffers at the end of a simulation step, which can be seen for example in Figure 3.9. We implement the method `sendBackParticles` which sends the three buffers $b0$, $b1$, $b2$ to their original owners based on the values in $b0Owner$, $b1Owner$ and $b2Owner$. We use the non-blocking MPI commands `MPI_Isend`[10] and `MPI_Irecv`[11] for the data exchange. First, all processors initiate up to three sends (for $b0$, $b1$ and $b2$). Then, we receive the data from other processors. Since in this algorithm all buffers have the same size, as we use dummy particles, we do not need to check for the amount of data to be received. Note: It is important to use non-blocking instructions in this case, otherwise we provoke a deadlock because no processor leaves the sending region until all the data has arrived at the receiver, which would never happen because the receiver also does not leave the sending region if it uses a blocking instruction. The corresponding Code Snippet can be found in the appendix in section A.

[10] `https://www.open-mpi.org/doc/v4.1/man3/MPI_Isend.3.php`
[11] `https://www.open-mpi.org/doc/v4.1/man3/MPI_Irecv.3.php`

Algorithm 6: All Unique Triplets Algorithm

Input: decomposition: An atom decomposition that has all particles divided into n/p equal subsets

Output: particles of this processor updated in place

```
1  Function calculateOneThirdOfInteractions(thirdID):
2      bufList ← [(b0, b0Owner), (b1, b1Owner), (b2, b2Owner)]
3      sortAscendingByOwner(bufList)
4      start ← thirdID * getBuf(bufList[0]).size()/3
5      steps ← getBuf(bufList[0]).size()/3
6      if thirdID == 2 then
7          steps ← getBuf(bufList[0]).size() − 2 * steps
8      CalculateInteractions(
9        getBuf(bufList[0]), getBuf(bufList[1]), getBuf(bufList[2]),
10       getOwner(bufList[0]), getOwner(bufList[1]), getOwner(bufList[2]),
11       start, steps
12     )
13 Function SimulationStep():
14     b0 ← decomposition.getMyParticles()
15     b1 ← copy(b0)
16     b2 ← copy(b0)
17     b0Owner ← b1Owner ← b2Owner ← worldRank
18     i ← 2
19     b_i ← pickBuffer(i)
20     for s ← worldSize to mod(worldSize, 3) by −3 do
21         for j ← 0 to s - 1 do
22             if j > 0 or s ≠ worldSize then
23                 getOwner(i) ←shiftRight(b_i, getOwner(i))
24             CalculateInteractions( b0, b1, b2, b0Owner, b1Owner, b2Owner)
25         i = mod((i + 1), 3)
26         b_i ← pickBuffer(i)
27     if mod(worldSize, 3) == 0 then
28         getOwner(i) ←shiftRight(b_i, getOwner(i))
29         thirdID ← worldRank/(worldSize/3)
30         calculateOneThirdOfInteractions(thirdID)
31     if worldSize > 0 then
32         sendBackParticles;
33     sendBackParticles()
34     sumUpParticles()
35     decomposition.setMyParticles(b0)
```

4.2.3. Cutoff Triplet Algorithm

In this section, we present our implementation of the Cutoff Triplet Algorithm (CTA) from section 3.2.4. The implementation can be used with a 1D, 2D, as well as 3D regular grid decomposition. In the following, we restrict ourselves to the implementation for a 2D decomposition, since this variant is a good way to illustrate the algorithm. The variants for the 1D and 3D decompositions are implemented analogously and are therefore not discussed in detail. Each processor has a virtual position within the simulation domain, which can be represented as a Cartesian coordinate and was assigned to the processor at the beginning by the Topology. Using the MPI command `MPI_Cart_coords` we can request these coordinates for a specific processor. In the following, these Cartesian coordinates are called the *rank* of a processor. In Figure 4.3a such ranks are exemplarily shown for 49 processors in a 7×7 decomposition.

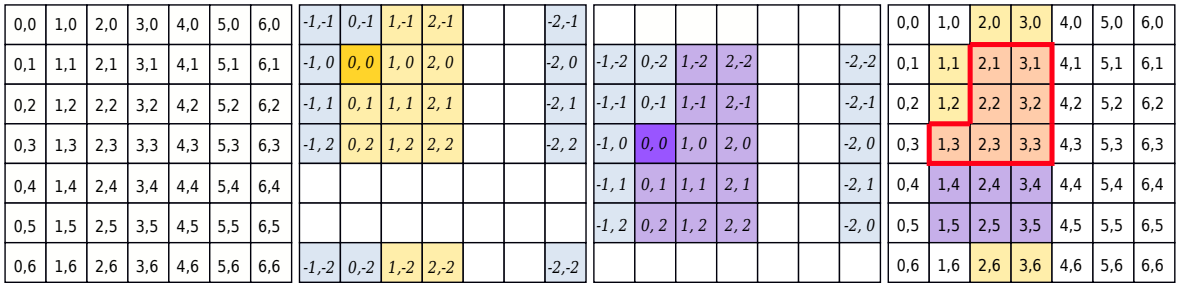


Figure 4.3.: Illustration of the CTA Algorithm with a 2D 7×7 regular grid decomposition. The cutoff distance covers 2 neighboring cells in this example. 4.3a shows the absolute coordinates (ranks) of the processors in the domain. 4.3b shows in blue the cutoff window $w_{(1,1)}$ of processor $(1, 1)$, as well as the periodic differences to other processors relative to $(1, 1)$ in italics. In yellow all neighboring processors are shown whose periodic difference is lexicographically greater than or equal to $(1, 1)$. We denote this as $wl_{(1,1)}$. 4.3c shows this analogously to 4.3b for the processor with rank $(1, 3)$. 4.3d shows the intersected lexicographical cutoff window $wl_{(1,1)} \cap wl_{(1,3)}$. Note: In this Figure, unlike the one in the theory section for the CTA algorithm, the x and y coordinates are on the horizontal and vertical axes, respectively.
adapted from: [KY14]

The entire pseudo-algorithm of our implementation is shown in Algorithm 7. We briefly explain how a simulation step with this algorithm works in the function `SimulationStep`, then we go into more detail about the individual auxiliary functions.

In the following let r be the processor from whose point of view we consider the execution of the algorithm and wl_r the cutoff window of r, which contains only the ranks that have a greater or equal relative periodic difference to r. The periodic difference of ranks in one dimension is defined as in equation 3.6. For details see theory section 3.2.4.

Our implementation works with a double nested `for` loop, where the outer loop in line 13 iterates in lexicographic order over ranks r_2 starting from r to the end of wl_r and imports their particles into $b1$. The inner loop in line 14 calculates the forces between $b0$, $b1$, $b2$ and

iterates in lexicographic order over ranks r_3 starting from r_2 to the end of wl_r and imports their particles into $b2$ if $r_3 \in wl_r \cap wl_{r_2}$. See also Figure 4.3 for illustration.

To determine the number of loop iterations, the length of wl_r is calculated at the beginning of the simulation. This can be seen in line 1 of Algorithm 7.

Since the inner loop in line 13 always iterates starting from rank r_2 over ranks $r_3 \in wl_r \cap wl_{r_2}$, the particles of r_2 are not only imported into $b1$ at the end of each outer loop iteration, but also into $b2$. See lines 23 to 24.

Algorithm 7: Cutoff Triplet Algorithm

Input: nCbX, nCbY: The Number of neighboring cells inside the cutoff window along x and y respectively, c : The cutoff distance, decomposition: A 2D regular grid decomposition that has all particles binned into the respective cells

Output: particles of this processor updated in place

```
1  numSteps ← 1 + nCbY + (nCbY * 2 + 1) * nCbX
2  offsetVecB1 ← offsetVecB2 ← Vec(0)
3  cartRank ← cartRankFromIntRank(worldRank)
4  Function shiftLeft(buf, owner, src, dst, offsetVector):
5      handleOffsetVector(src, dst, offsetVector)
6      srcInt ← intRankFromCartRank(src)
7      dstInt ← intRankFromCartRank(dst)
8      return mpiShift(buf, owner, srcInt, dstInt)
9  Function SimulationStep():
10     b0 ← decomposition.getMyParticles()
11     b1 ← b2 ← copy(b0)
12     b1Owner ← b2Owner ← worldRank
13     for i2 ← 0 to numSteps - 1 do
14         for i3 ← i2 to numSteps - 1 do
15             CalculateInteractions( b0, b1, b2, worldRank, b1Owner, b2Owner, c)
16         if i3 < numSteps − 1 then
17             (src, i3) ← getSrcInner(i2, i3, cartRank)
18             dst ← dstFromSrc(src, cartRank)
19             b2Owner ← shiftLeft(b2, b2Owner, src, dst, offsetVecB2)
20     if i2 < numSteps − 1 then
21         srcOuter ← srcInner ← getSrcOuter(i2, cartRank)
22         dstOuter ← dstInner ← dstFromSrc(srcOuter, cartRank)
23         b1Owner ← shiftLeft(b1, b1Owner, srcOuter, dstOuter, offsetVecB1)
24         b2Owner ← shiftLeft(b2, b2Owner, srcInner, dstInner, offsetVecB2)
25     sendBackParticles()
26     sumUpParticles()
27     decomposition.setMyParticles(b0)
```

getSrcOuter (loop over wl_r): The function `getSrcOuter` in Algorithm 8 calculates, based on a loop index $i2$, the relative coordinate of a processor with respect to r in wl_r after $i2$ steps. These relative coordinates are added to the rank of the processor r and wrapped around the virtual grid of processors using a modulo operation, with the vector *dim* as divisor, which stores the number of processors in each dimension. This way we get the rank of the processor that owned the particle subset at the beginning of the simulation that we want to import. The variable `nCbY` is the number of adjacent cells in y-direction covered by the cutoff radius.

Algorithm 8: getSrcOuter

```
1 Function getSrcOuter(i2, rank):
2     cutoffLenY ← 2 * nCbY + 1
3     y ← mod((i2 + nCbY),cutoffLenY) - nCbY
4     x ← (i2 + nCbY) / cutoffLenY
5     return mod(vec(x,y) + rank, dim)
```

getSrcInner (Intersected Cutoff Window): In the inner loop in line 14 of Algorithm 7, that shifts $b2$, we use the function `getSrcInner` to get all the processor ranks that are in $wl_r \cap wl_{r_2}$. The idea is to determine the processor rank in wl_r after $i3 + 1$ steps starting from r, and at the same time calculating the rank in wl_{r_2} when we do $i3 + 1 - i2$ steps starting from r_2. Since wl_r and wl_{r_2} are different, which can be seen in Figures 4.3b and 4.3c for example, the result of both views is not necessarily in the cutoff window $wl_r \cap wl_{r_2}$. Since all processors arrange their neighbors in the same way, we can increment $i3$ within this function and compute ranks until both views match again. The parameter $i3$ is passed as a pointer in our implementation, so the incremented value for $i3$ is used as index in further iterations of the loop. Thus, starting from rank r_2, we take a maximum of $numSteps - i2$ steps, which is from r_2 to the end of wl_r, but skip the ranks that are not in $wl_r \cap wl_{r_2}$. The pseudo-algorithm is shown in Algorithm 9.

Algorithm 9: getSrcInner

```
1 Function getSrcInner(i2, i3, rank):
2     b1InitOwner ← getSrcOuter(i2, rank)
3     do
4         src ← getSrcOuter(i3+1, rank)
5         srcTmp ← getSrcOuter(i3+1-i2, b1InitOwner)
6     while src ≠ srcTmp and i3 < numSteps and i3 = i3 + 1
7     return (src, i3)
```

Offset Vector: Since the buffers are shifted between processors in the course of the simulation step, they are no longer at their original position, which we determined using the functions mentioned above. To determine the current position of the buffer, we use the offset vectors `offsetVecB1` and `offsetVecB2` for the buffers $b1$ and $b2$, on which we accumulate the difference of the new and the last rank at each shift.

Calculate Destination from Source: Since all processors work according to the same shifting scheme, the destination rank can be determined based on the source rank by mirroring the source rank to its own rank. This is calculated by the function `dstFromSrc`.

Offset Vector Adjustment: Before the function `shiftLeft` executes the communication step, the function `handleOffsetVector` first adds/subtracts the offset vector `offsetVecB1` or `offsetVecB2`, depending on whether b1 or b2 is shifted, to the source/destination rank. This function also adjusts the respective offset vector accordingly for the next substep by adding the difference of the positions to the offset vector based on the previous value of a loop iteration and the current value. Afterwards the actual shift is executed by the function `mpiShift`. The MPI instruction `MPI_Cart_rank`[12] is used to map the Cartesian coordinates back to integer ranks.

mpiShift: The MPI shift is implemented using the non-blocking send and receive commands `MPI_Isend` and `MPI_Irecv`. In contrast to the two direct algorithms, the number of particles to be sent and received can differ here, since this algorithm works with a regular grid decomposition and does not use dummy particles. The command `MPI_Isend` initiates the send process and returns afterwards. This is necessary so that each processor first sends information about the data to be sent to the receiver and then waits for those of the sender. As tag-parameter we use analogous to the shift procedure for the direct algorithms the owner of the respective buffer, so we can track the original owner of a particle subset. With the blocking command `MPI_Probe`[13] we wait for the data input of the source rank. By `MPI_Get_count`[14] we can request the number of elements sent and prepare the receiver buffer. We use the function `std::vector::resize`, which allocates the required memory for the buffer. Then we can use `MPI_Irecv` to receive the data. We use the source rank as sender and `MPI_ANY_TAG`, since the received buffer can have any original owner. The actual tag (owner) of the received message is stored in the status and returned at the end of the function. It is important to wait for the completion of both operations afterwards, so we only continue when all send and receive operations are complete. The corresponding Code Snippet can be found in the appendix in section A.

checkCutoff: In Algorithm 4, for each possible particle triplet (i, j, k) that we can form, we check whether the distances between (i, j), (i, k), and (j, k) are within the cutoff. Only if all distances are less than or equal to the cutoff, the forces for this triplet are calculated.

sendBackParticles: To send particles back to their owners after a simulation step, we use the same procedure as in Code Snippet 1, except here we only send back $b1$ and $b2$, since $b0$ is fixed.

[12] `https://www.open-mpi.org/doc/v4.1/man3/MPI_Cart_rank.3.php`
[13] `https://www.open-mpi.org/doc/v4.1/man3/MPI_Probe.3.php`
[14] `https://www.open-mpi.org/doc/v4.1/man3/MPI_Get_count.3.php`

1D and 3D Decompositions: The implementations of the 1D and 3D decompositions follow essentially the same scheme. For 1D decomposition, the value for `numSteps` is calculated as follows:

$$numSteps = 1 + nCbX \tag{4.2}$$

So, starting from a one-dimensional rank, we shift only to the left. In the case of a 3D decomposition, `numSteps` can be calculated this way:

$$\begin{aligned} numSteps = 1 &+ nCbZ + nCbY * (nCbZ * 2 + 1) \\ &+ nCbX * ((nCbY * 2 + 1) * (nCbZ * 2 + 1)) \end{aligned} \tag{4.3}$$

The pseudo-algorithms for the method `getSrcOuter` of the 1D and 3D variants are shown in Algorithm 10.

Algorithm 10: getSrcOuter for 1D and 3D decompositions

```
1 Function getSrcOuterThreeD(i2, rank):
2     cutoffLenY ← 2 * nCbY + 1
3     cutoffLenZ ← 2 * nCbZ + 1
4     z ← mod((i + nCbZ), cutoffLenZ) − nCbZ
5     y ← mod((((i + nCbZ) / cutoffLenZ) + nCbY), cutoffLenY) − nCbY
6     x ← (i + ((cutoffLenZ * cutoffLenY)/2))/(cutoffLenZ * cutoffLenY)
7     return mod(vec(x,y,z) + rank, dim)
8 Function getSrcOuterOneD(i2, rank):
9     return mod(i2 + rank, dim)
```

4.3. Correctness

To ensure that the algorithms work correctly, we implement unit tests using GoogleTest[15]. We use an MPI listener[16] for GoogleTest to allow the output of multiple processors involved. By using MPI commands (`MPI_Gatherv`, `MPI_Allreduce`), the results of all involved processors are collected and checked after the test. The following tests are performed:

- All unique buffer combinations are processed
- All unique particle triplets are calculated
- Test the operator == of triplet struct to check for permutations
- further constructor tests for particle struct

For the test that checks for all unique buffer combinations, all unique triplets of processor ranks for a given number p are created for the direct algorithms in a 3-nested loop. For the cutoff algorithm, we implemented the mathematical definitions from the reference [KY14],

[15] `https://github.com/google/googletest`
[16] `https://github.com/LLNL/gtest-mpi-listener`

which, for a given number of processors p and a cutoff distance c, creates the set of all triplets from processor-ranks that a given processor must traverse in its cutoff window according to the lexicographic ordering. Similarly, for each processor, the triplets from processor-ranks within the intersected cutoff windows are also computed. This set is then compared to the actual computed buffer interactions. Our tests show that exactly the identical set of buffer combinations is computed for both the direct and cutoff algorithms.

The number of all unique particle triplets is tested in this way only for the direct algorithms, by again using a 3-nested loop to create all particle triplets that are possible with n particles. Subsequently, all actually computed particle triplets from all processors are collected and compared. Again, our tests show that exactly the same particle triplets are computed.

To verify that the cutoff algorithm computes all required particle interactions, the results were compared to those of the direct algorithm for an appropriately large value for c, such that all interactions are computed. To ensure that the cutoff algorithm produces the same results with a different number of processors at constant input, the calculated forces are compared with one processor up to 125 processors. All results show identical forces, which leads us to conclude that the cutoff algorithm works correctly.

Overall, we conclude that the implemented algorithms work correctly and compute all unique interactions.

5. Results

In this chapter we analyze the algorithms implemented in section 4. We investigate the shifting scheme of the two direct algorithms in section 5.1, the scalability of all three algorithms in section 5.2, the load balance of the CTA slgorithm with different particle distributions in section 5.3, the hit-rate of the cutoff algorithm for different regular grid decomposition strategies in section 5.4, as well as the deviation of the calculated forces between the CTA algorithm and the AUTA in section 5.5.

All our measurements were performed on the Linux cluster segment CoolMUC-2[1] of the Leibniz Supercomputing Center (LRZ)[2]. It is a medium size HPC cluster with 812 nodes each with 28 Haswell-based 64-bit cores. An FDR14 Infiniband interconnect is used between the nodes. For the communication between processors we use the OpenMPI implementation version 4.1.2. All programs were compiled with gcc 9.4.0 and optimization level O3.

For the timing measurements of an entire siumlation step, we use Google Benchmark[3]. To enable measurement across all processors involved, we use a custom method that collects the times of all processors after each benchmark using a `MPI_Allreduce`[4] operation that chooses the maximum time, since this represents the total runtime of the program.

For single function measurements, we use a self-implemented time measurement based on the chrono library[5] with the `std::chrono::system_clock` that measures time in selected methods. The results are stored using RapidJson[6]. Similar to the procedure we use for time measurements with Google Benchmark, all participating processors send their time measurements to one specific processor, which creates the output file based on these values. For each function to be measured, the maximum time among all processors is chosen within a substep. Afterwards, these maximum times of all substeps are accumulated, which provides the total time that has been spent within the respective function in one simulation step.

For our measurements, we use different particle distributions, which are briefly presented here:

- Uniform: Random distributed particles using the C++ `std::uniform_real_distribution`[7]
- Grid: Evenly distributed particles with equal spacing in each dimension.

[1] `https://doku.lrz.de/display/PUBLIC/CoolMUC-2`
[2] `https://www.lrz.de/`
[3] `https://github.com/google/benchmark`
[4] `https://www.open-mpi.org/doc/v3.0/man3/MPI_Allreduce.3.php`
[5] `https://en.cppreference.com/w/cpp/chrono`
[6] `https://rapidjson.org/`
[7] `https://en.cppreference.com/w/cpp/numeric/random/uniform_real_distribution`

- Closest Packed: Hexagonally arranged particles, to provide the most compact homogeneous distribution.
- Gauss: Random distributed Particles using the C++ `std::normal_distribution`[8] with a mean and standard deviation
- Clustered Gauss: Uniformly distributed gaussian particle clouds

In all our investigations we use the Axilrod-Teller potential to calculate the three-body interactions. We have set the coefficient for the Axilrod-Teller potential to $v = 1$ for the sake of simplicity.

5.1. Shifting Scheme of the Direct Algorithms

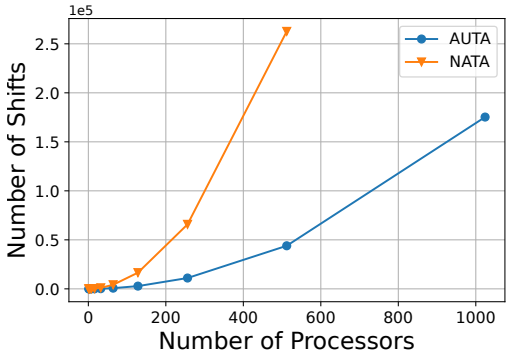


Figure 5.1.: Comparison between number of shifts of the NATA and AUTA algorithms

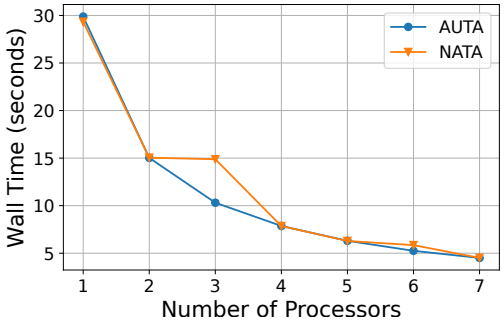


Figure 5.2.: Load imbalance of the NATA algorithm when the number of processors is divisible by 3. The AUTA algorithm distributes the load evenly

In this section, we compare the shifting scheme of the NATA and AUTA algorithms. As mentioned in the theory section 3.1.2 and Algorithm 5, the naive algorithm performs redundant shifts. The increase in shifts over the all unique triplet algorithm is shown in Figure 5.1. The AUTA algorithm only performs shifts that lead to a force calculation step, so the number of shifts is always equal to the number of calculation steps minus one.

In Figure 5.2 we see how the behavior of both algorithms affects the runtime of a simulation step if processors form a permutation of an offset-pattern in a substep, which happens whenever the number of processors is divisible by 3, as mentioned in theory part 3.1.2. In the implementation of the naive algorithm, in such a case only the processor with the lowest rank that has generated a permutation computes the particle interactions, all other processors are idle. For details, see function 1 in the implementation part. This is directly reflected in the runtime, as can be seen for 3 and 6 processors. The AUTA algorithm, on the other hand, distributes the work evenly among the participating processors in such a case.

[8] `https://en.cppreference.com/w/cpp/numeric/random/normal_distribution`

5.2. Scalability

5.2.1. Direct Algorithms

In the following, we analyze the scaling behavior of the two direct algorithms. In both experiments, a uniform particle distribution is used. For the time measurements of the direct algorithms, the function `calculateInteractions` includes both the time needed to form the triplets and the time to calculate the forces. The time of one simulation step is measured, which includes the force calculation of all $\binom{n}{3}$ interactions of the particles and the updating of their positions, velocities and accelerations.

Strong scaling

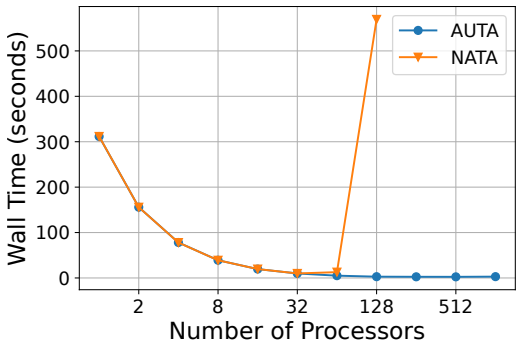


Figure 5.3.: Total strong scaling wall time for NATA and AUTA.

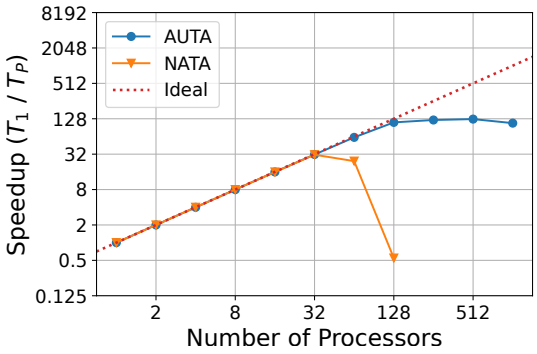


Figure 5.4.: Strong scaling speedup for NATA and AUTA.

Figure 5.3 and 5.4 shows the results of the strong scaling experiment of the two direct algorithms. 2048 particles were used and the number of processors is scaled from 1 to 1024 in powers of two.

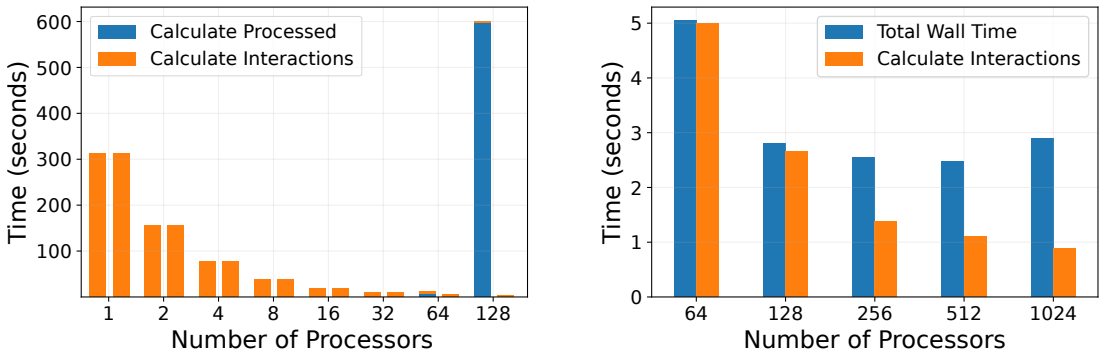
In Figure 5.4, the speedup S is shown for a specific number of processors. As can be seen, both algorithms scale almost perfectly up to 32 processors, which is due to the fact that the communication between the processors is not yet a major issue here. Since the calculation of the forces between particles is the most time-consuming part, and this work is distributed evenly, a good scaling behavior can be achieved.

Above 64 processors we see a significant drop in the naive algorithm. We did not run benchmarks for $p > 128$ for the naive algorithm as they did not terminate within a period of eight hours.

For the AUTA algorithm, we see a drop in speedup starting at 128 processors. We are interested first why the execution time of the NATA algorithm increases so strongly starting from a certain point. A first consideration makes the high number of shifts, which also include many redundant shifts, of this algorithm responsible for the speedup decrease. As we can see in Figure 5.1 the number of shifts increases significantly compared to the AUTA algorithm. However, it can also be seen that the AUTA algorithm does more shifts for

$p \geq 512$ than the NATA algorithm does for $p = 128$. Thus, our first consideration cannot be the only reason. In our single measurements in Figure 5.5a, we can see a significant increase in the `calculateProcessed` function of the NATA algorithm, which keeps track of already processed buffer combinations. We can thus state that this function is responsible for the drop in speedup and the increase in execution time of the naive algorithm.

The AUTA algorithm does not have this function, because the shifting scheme ensures that a buffer combination is generated only once during a simulation step.



(a) (b)

Figure 5.5.: Figure 5.5a shows the sum of the maximum times among all processors of a substep for the functions `calculateProcessed` and `calculateInteractions`, which reflects the total time spent in a simulation step in the respective functions. The left bars refer to the NATA algorithm, the right bars to the AUTA algorithm. Only the NATA algorithm has the `calculateProcessed` function, which consumes almost no time up to 32 processors, but then causes an unacceptable runtime. The times for the calculation of the triplets are almost identical. Figure 5.5b shows the deviation of the total wall time and the calculation of the particle triplets for the AUTA Algorithm from 64 to 1024 processors, which we trace back to the increasing communication and idle time among processors.

In Figure 5.5b we have compared the wall time of a simulation step and the time spent in the function `calculateInteractions` starting from a processor number of 64 to analyze the drop in speedup for the AUTA algorithm. Since we could not obtain meaningful results for the measurements of the communication time with a large number of communication steps by our chosen method for the time measurement of individual functions, we compare the total wall time with the calculation time of the particle interactions. Since the other measurable times, such as summation and updating of particles, which ranged from $2.88 * 10^{-6}$ to $2.61 * 10^{-5}$ seconds, are negligible, we assume that the difference between the wall time and the computation time of particle interactions is largely due to communication and idle time.

We conclude that the decrease of the speedup is mostly caused by the increasing number of communication steps between the processors. On the other hand, the decrease in speedup is also caused by the fact that the number of particles per processor continues to decrease and at a certain point speedup is no longer possible. For the calculation of forces, we see

that from 64 to 128 processors almost still an ideal speedup is possible, from then on this is not achieved any longer. A good scaling behavior is possible for this experiment up to $p = 128$ processors. Above 128 processors, the advantage of more resources decreases. And from 512 processors, poor scaling behavior can be seen. We can thus state that for our benchmark with 2048 particles, a value for $p > 512$ is not reasonable, and a processor should ideally have at least 16 particles, so that an optimal speedup can be achieved with our implementation.

Note: We have performed further tests with 8192 particles for the AUTA algorithm, that showed a good scaling behavior up to 512 processors, which is due to the fact that each processor has at least 16 particles and thus a good scaling behavior can be achieved. The results can be seen in the appendix section B.

Weak scaling

For the weak scaling benchmarks, the number of processors is scaled in proportion to the number of particle interactions. For one processor, 2048 particles are used, which results in $\binom{2048}{3}$ interactions. To keep the number of interactions per processor constant, we use the following equation, which we solve for n:

$$\binom{n}{3} = \binom{2048}{3} * p \tag{5.1}$$

In this way, we ensure that each processor has $\binom{2048}{3}$ interactions to compute.

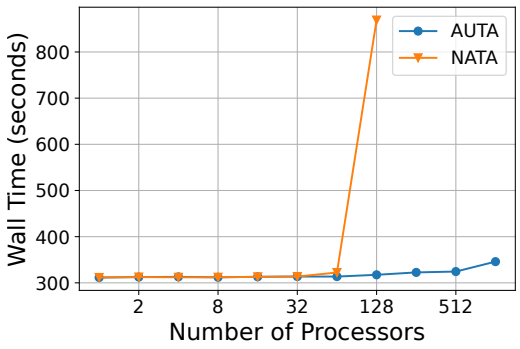


Figure 5.6.: Total weak scaling wall time for NATA and AUTA

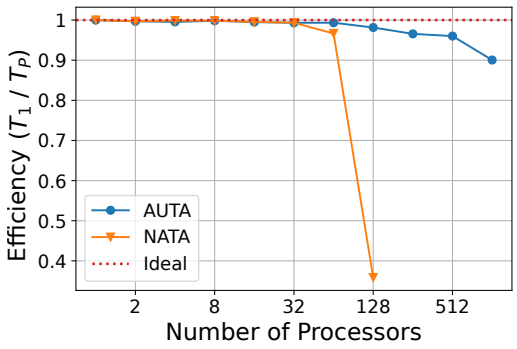


Figure 5.7.: Weak scaling efficiency for NATA and AUTA

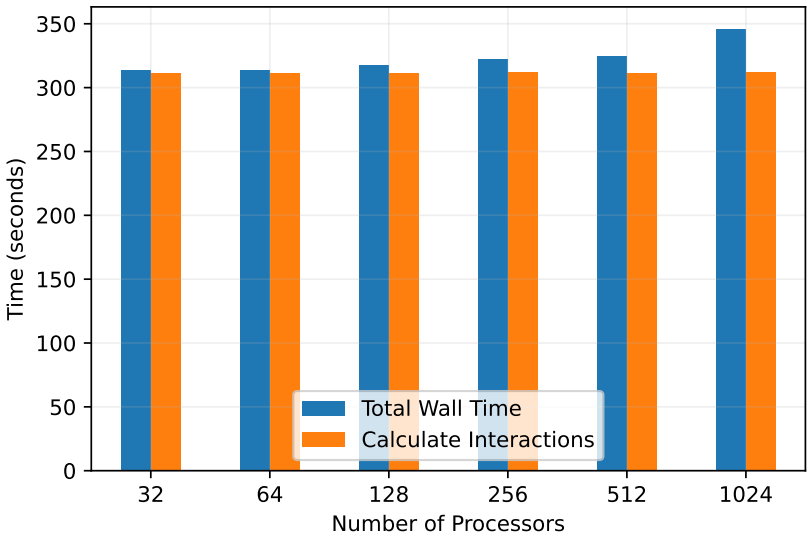


Figure 5.8.: The deviation of the total wall time and the calculation of the particle triplets for the AUTA Algorithm from 32 to 1024 processors. We attribute the growing gap to the increasing communication overhead and idle time of the processors.

The results of the weak scaling experiment in Figure 5.6 and 5.7 show the same phenomenon for the NATA algorithm as described above. The method `calculateProcessed` causes an increase in the wall time, which leads to a drop in efficiency. For the AUTA algorithm the effect of the increasing communication is a bit more noticeable here. A drop in efficiency can already be seen beginning with 64 processors and with 1024 processors AUTA needs about 111 percent of the time needed for 1 processor.

In Figure 5.8 we see the comparison between the total wall time and computation time of the particle triplets for the AUTA algorithm from 32 to 1024 processors. As with the strong scaling benchmarks, other individual measurements were taken in addition to the `calculateInteractions` function, but compared to the evaluation of the triplets, they were so marginal that they are not shown in the Figure. We conclude that most of the difference between the wall time and the computation time of the particle triplets can be attributed to the increasing communication or idle time of processors. As can be seen, the computation time of the particle triplets remains constant for different numbers of processors, as expected.

In summary, we can state that the nearly ideal distribution of work mentioned in the theory section 3.1.2 is reflected in practice in the almost constant computation time in Figure 5.8. As the number of processors increases, the number of communication steps also increases, and with it the time required for communication. We can also state that for this algorithm the number of processors should be in proportion to the number of particles, because in this case the decrease of the speedup can only be attributed to the communication, but not to the calculation of the interactions, as in the strong scaling experiment.

5.2.2. Cutoff Algorithm

For the CTA Algorithm, the results of the weak scaling measurements are discussed in the following. For the following experiment, the domain size is scaled directly in proportion to the number of processors along the x-axis. It is equal to $(5 * p) \times 10 \times 10$. In the same way, the number of particles within this domain is scaled. It is equal to $2048 * p$, using a uniform distribution. This ensures that the density of particles remains constant and each processor has approximately the same amount of work. The domain decomposition strategy chosen is the naive one, which splits the simulation domain along the x-axis into uniform slices and assigns particles within them to individual processors. For the cutoff distance, $c = 2.4$ is used. Starting from one processor, the number of processors is increased in powers of two up to 1024. One simulation step each is executed, which contains the calculation of all forces, update of the particle positions and update of the domain decomposition.

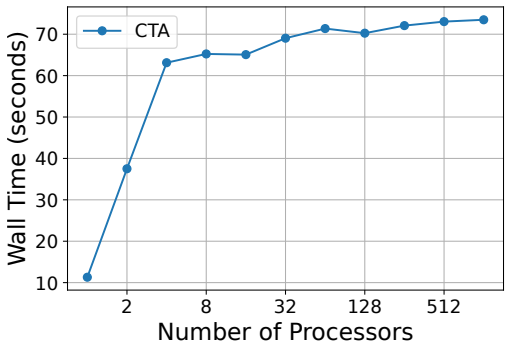


Figure 5.9.: Weak scaling wall time for the CTA algorithm.

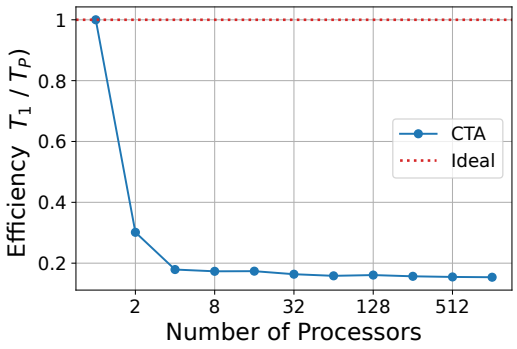


Figure 5.10.: Weak scaling efficiency for CTA algorithm.

In Figures 5.9 and 5.10 we can observe a significant increase in wall time and a drop in efficiency between the transition from one to two and from two to four processors. As the number of processors increases further, a slight increase in wall time or drop in efficiency can be seen. To examine this result in more detail, we perform more fine-grained time measurements with the same input data and look at the times that occur within individual methods. In contrast to the single measurements for the direct algorithms, the measurement for the function `calculateInteractions` is separated here into the time needed to form particle triplets, which in the cutoff algorithm also includes the distance calculations to check whether the forces for a triplet should be calculated, and into the time needed to calculate the forces.

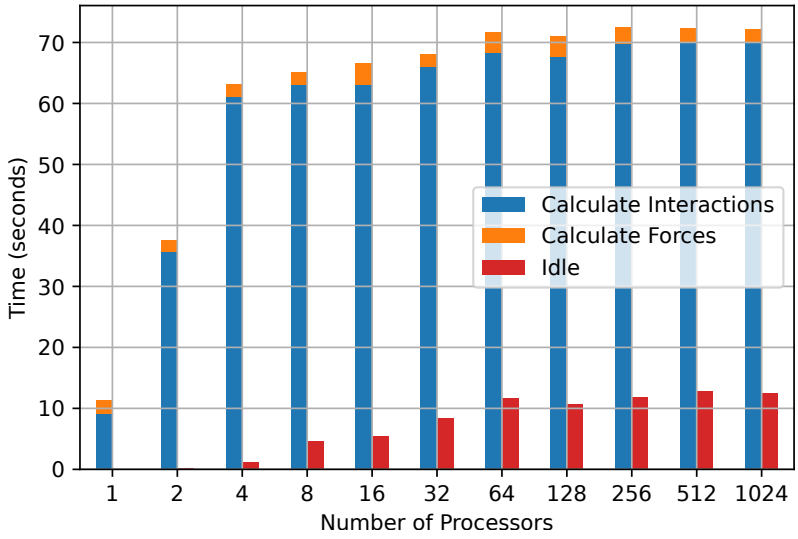


Figure 5.11.: Time breakdown for CTA algorithm. The increasing time in `calculateInteractions` is due to the increasing number of distance checks that need to be performed.

In Figure 5.11 we see the individual times of the methods `calculateInteractions` and `CalculateForces`. The function `calculateInteractions` is responsible for forming unique particle triplets from the three buffers $b0$, $b1$ and $b2$ and calculating distances within the triplet to decide if a particle triplet should be computed or not. Using our implementation of the Axilrod-Teller potential, the method `CalculateForces` calculates the forces for a given particle triplet (i, j, k).

We can see a significant increase in the `CalculateInteractions` function from one to two and from two to four processors as well. This can be explained as follows: In the case of one processor, only the distances between its own particles are evaluated. This corresponds to offset-pattern 000 for the three buffers and the distances between all the $\binom{2048}{3}$ particle triplets are calculated. When moving to two processors, the distances between the own particles and those of the respective neighbor must also be computed. The processors calculate the following offset-patterns: 000 and 001. Since each processor has about 2048 particles, the distances between $\binom{2048}{3} + \binom{2048}{2} * 2048$ particle triplets have to be evaluated, which are interactions between own particles and interactions between two own particles and one from the neighboring processor. Analogously, the same happens in the transition from 2 to 4 processors. Each processor computes the offset-patterns 000, 001 and 011, which is a work of $\binom{2048}{3} + (\binom{2048}{2} * 2048) * 2$. For higher numbers of processors only the interactions between these three offset-patterns must be computed, since only communication with the direct neighbor is necessary for the chosen cutoff. Note: The case $p = 2$ is an edge case, in which the computation is only performed for the offset-patterns 000 and 001, since otherwise redundant computations would result.

A second bar shows the increasing idle time within all processors. The idle time corresponds

to the time that a processor must wait after a calculation step until all other processors are ready for the communication step. We have intentionally shown these measurements as a second bar because the idle time is also included within the times of `CalculateInteractions` and `CalculateForces`. An increase can be seen as the number of processors increases, but flattens out. This increase is also reflected in the measurements of `CalculateInteractions` and `CalculateForces`. The rise can be explained as follows: With only one processor, no communication is needed. With two processors, exactly one communication step is required (shift of $b2$). With 4 processors and more, 2 communication steps are always required, namely the shift of $b2$ and then of $b1$. Since the communication takes place synchronously, the idle time increases when several processors are involved.

Note: As described for the direct algorithms, our method of measuring the time for individual methods made it difficult to determine the time for MPI instructions, which led to large deviations when the number of communication steps is high. The measurements of the idle time here are more exact, since only 2 communication steps are measured, however, they are also not completely exact, why they should give only an impression, why the wall time with increasing number of processors slightly increases.

In summary, we can say: The significant increase in wall time in Figure 5.9 or decrease in efficiency in Figure 5.10 is due to the increasing number of distances to be evaluated between buffers. The further slight increase is due to idle time during communication between processors. As expected, the computation time of the forces remains relatively constant.

Note: further individual measurements of other methods such as `sendBackParticles`, `sumUpParticles` and `mpiShift` were performed, but the values were so small that they could not be seen in Figure 5.11 and are therefore not shown. The sum of the total times spent within a simulation step in these methods ranged from $1.11 * 10^{-4}$ to $1.32 * 10^{-2}$ seconds.

5.3. Load Balance of the Cutoff Algorithm

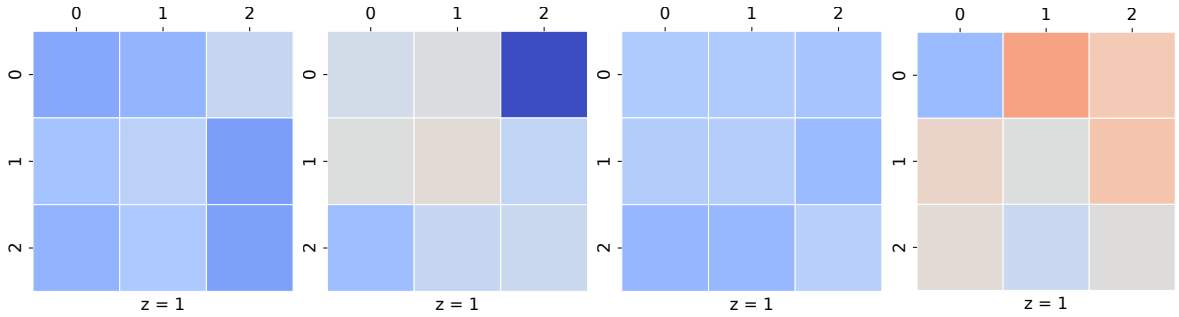
In the following, we analyze the load balance of the CTA algorithm for different particle distributions.

For this purpose, we use the Step Time Variation Ratio, which is explained in more detail in the theoretical background 2.3.2. Depending on the distribution of particles within the physical simulation domain, different computation loads arise for each processor because the allocation of particles to processors depends on the positions of particles in the physical simulation domain. The following particle distributions are used within a $10 \times 10 \times 10$ domain.

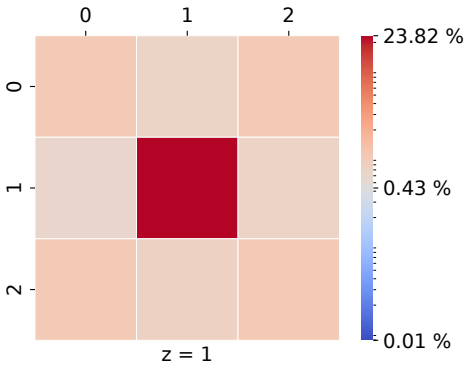
- 8000 uniformly distributed particles.
- 8000 particles with a fixed spacing of 0.526 in each dimension resulting in 20 Particles in each dimension.
- 8118 closest packed particles with a spacing of 0.575 in each dimension.
- 8000 particles divided into 25 uniformly distributed Gaussian particle clouds with 320 particles each, a mean of $\mu = 0$ and a variance of $\sigma^2 = 0.2$ in each dimension.

- A Gaussian particle cloud consisting of 8000 particles with a mean of $\mu = 5$ and a variance of $\sigma^2 = 1.5$ in each dimension centered in the domain.

Note: For the closest packed particles, we could not generate exactly 8000 particles because our particle generator uses a single spacing value for all dimensions, as well as a box size that limits the number of particles in each dimension. The 8118 particles was the closest possible number to 8000 that we could generate.



(a) Uniform (b) Grid (c) Closest Packed (d) Clustered Gauss



(e) Gauss

Figure 5.12.: Step Time Variation Ratio (STVR) for different particle distributions using the CTA Algorithm. The cutoff radius is constant at $c = 2.0$ and 27 processors each are used, arranged in a $3 \times 3 \times 3$ regular grid decomposition. The middle slice with respect to the z-axis of the $3 \times 3 \times 3$ decomposition is shown for each particle distribution. On the horizontal and vertical axis are respectively the x- and y-positions of the processors within the $3 \times 3 \times 3$ decomposition. The minimum and maximum value of the STVR is the global minimum and maximum over all 5 results, so that the different distributions can be compared. The color values are displayed using a logarithmic scale.

	Number of particles			Number of interactions		
	min	max	median	min	max	median
uniform	248	332	294	680913	1161060	851974
grid	252	343	294	474984	1851794	897205
closest packed	270	336	294	767936	1272499	976122
clustered gauss	0	883	327	0	47777633	5903286
gauss	2	4356	42	41	1068030394	151873

Table 5.1.: The number of particles assigned to processors for different particle distributions and the resulting number of calculated interactions

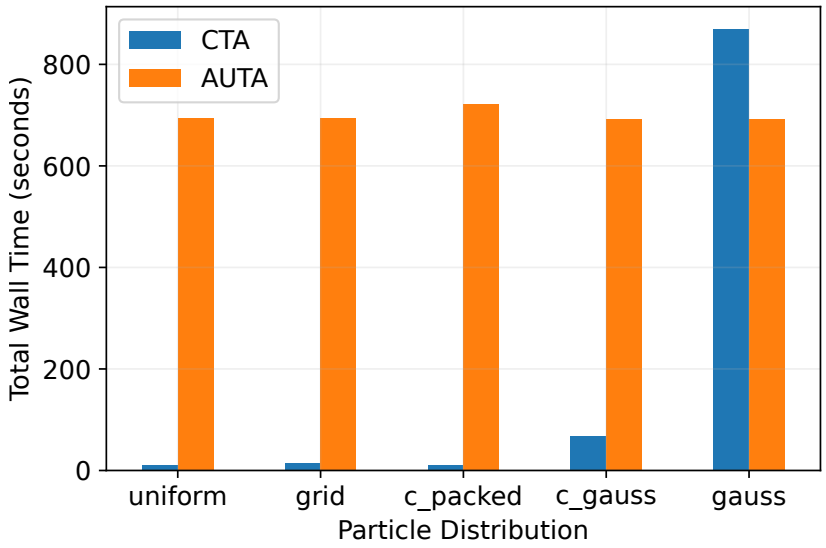


Figure 5.13.: Comparison between the total wall time of the CTA and AUTA algorithms for different particle distributions. We use the labels c_packed and c_gauss for closest packed and clustered gauss respectively.

The results from Figure 5.12 are as we expected, since the assignment of particles to processors is related to their position in the physical simulation domain. Thus, for the uniform distribution in Figure 5.12a and the closest packed distribution in Figure 5.12c, we see a smooth distribution of load across processors, while a larger imbalance is observed for distributions where the number of assigned particles varies more.

The number of minimum and maximum particle counts and the resulting force calculations, as well as the median across all 27 processors, can be seen in Table 5.1.

We have compared the computation times of an entire simulation step with all five particle distributions and the use of the CTA algorithm with those of the AUTA algorithm, when using the same distributions. Figure 5.13 shows the results in seconds respectively. Note:

Since 8118 particles were used for the closest packed particle distribution, the calculation time here for the AUTA algorithm deviates somewhat more compared to the other distributions.

It can be seen that the computation times for the given cutoff of $c = 2.0$ are significantly lower than those of the direct algorithm for all distributions except the Gaussian distribution, due to the smaller number of particle triplets to be evaluated.

For the Gaussian distribution, we see that the computation time of the direct algorithm is below that of the cutoff algorithm. This is because for the CTA algorithm, more than 50 percent of the particles, 4356 to be exact, are assigned to the processor in the middle. In comparison, 296(+1) particles are assigned to each processor of the direct algorithm, which results in an almost perfect load distribution.

We can state in conclusion that execution time can be significantly reduced with the CTA algorithm compared to the direct computation of all interactions. There are certain edge cases where direct computation can be advantageous. For example, this may be the case for highly heterogeneous particle distributions when regular grid decomposition is used, as we do, which divides the domain into uniform cells. Or when the number of particles is very small, so that computing the distance checks takes more time than computing the forces. However, by adapted decomposition strategies, such as a dynamic cell size for heterogeneous particle distributions, the load balance could also be optimized for the CTA algorithm, so that a faster runtime would presumably be possible compared to the direct calculation.

5.4. Hitrate of the Cutoff Algorithm

Since we have implemented the cutoff algorithm with the possibility of working with a 1D, 2D or 3D regular grid decomposition, we are interested in how the different decomposition strategies affect the hit-rate and the runtime of a simulation step.

We compare 2 different regular grid decomposition strategies. The first divides the physical simulation domain evenly into slices along the x-axis and assigns them to processors. This decomposition is called naive in the following and corresponds to a 1D decomposition.

The second strategy creates either a 1D, 2D, or 3D regular grid decomposition based on a given number of processors. Our assumption is that a decomposition in which the individual cells of the 3 dimensional physical domain take the shape of a cube will provide a better hit-rate and hence runtime. This is because the cutoff radius can be represented by a sphere and the bounding box of a sphere corresponds to a cube. This decomposition is called dynamic in the following and works according to this scheme:

1. Decompose the number of processors into prime factors
2. Form all decompositions into all possible partitions from the prime factors
3. Keep the decompositions that have the prime factors divided into either 1, 2 or 3 partitions
4. Calculate the products of the factors within the partitions of these decompositions. Each of these decompositions represents a way in which the domain can be partitioned into either 1, 2, or 3 dimensions with a given number of processors. However, keep only those that have at least 3 cells in each dimension, which is necessary to satisfy the cutoff restriction of the algorithm. An exception is the 1D decomposition, in which also only one or 2 cells may occur.
5. Choose from these partitions the one that is best balanced. So the shape is closest to a cube. This is done by calculating the best line of fit for the products within the decompositions. The decomposition with the lowest slope is chosen, preferably a 3D, then a 2D and only as a last possibility a 1D decomposition.

As can be seen, for a prime number no other decomposition can be found than the naive 1D decomposition. With $p = 64$, for example, a decomposition can be found that corresponds to a cube: $(4, 4, 4)$ with the balancing value 0.

We define the hit-rate as:

$$h = \frac{interact_a}{interact_p} \tag{5.2}$$

Where $interact_a$ corresponds to the actually calculated particle triplets in a simulation step, $interact_p$ to all possible particle triplets that can be calculated within the imported neighbor cells. The final result is the average value of all hit-rates from the individual processors.

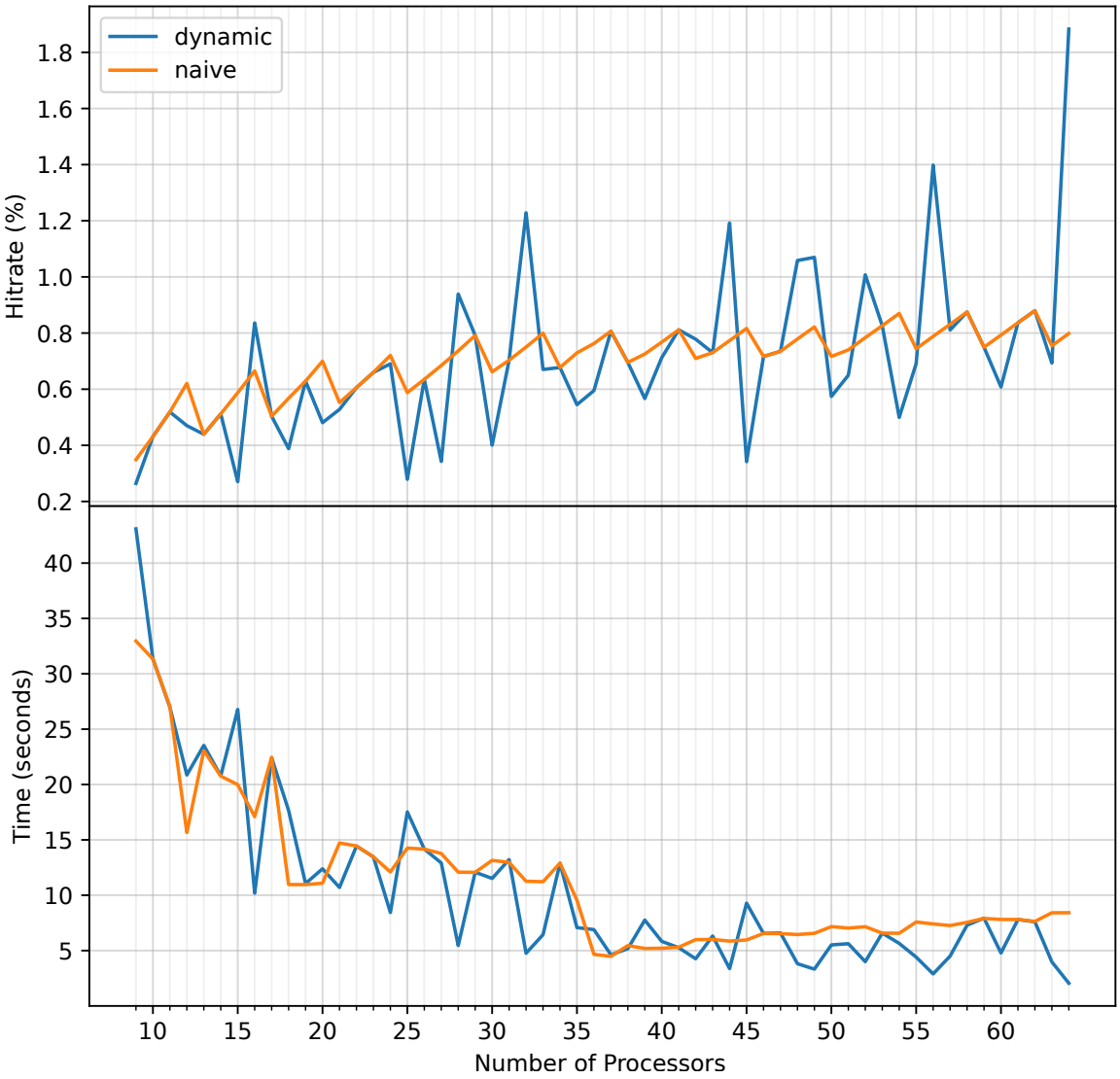


Figure 5.14.: Hitrate and total wall times of one simulation step using the naive and dynamic decomposition strategies from 9 to 64 processors. We compare the two variants only above a number of 9 processors, since for $p < 9$ no decomposition other than the naive one is possible. For this experiment, we use 8118 closest packed particles within a $10 \times 10 \times 10$ domain. The reason for this choice is that we want the hit-rate to depend on the decomposition strategy rather than the distribution of particles and since the best balancing was achieved in our load balance test with closest packed particles, we decided to use this. As cutoff distance we use $c = 2.4$

As we can see in Figure 5.14 the hit-rate is in most cases also reflected in the execution time, especially with few processors strong differences are noticeable. An improved hit-rate can be achieved for the dynamic decomposition in some cases. For example, with 32 or 64 processors, the execution time can be reduced compared to the naive strategy. For 32

processors, $(8, 4)$ decomposition was used, and for 64 processors, $(4, 4, 4)$ decomposition was used.

However, some naive decompositions also yield a higher hit-rate than dynamic decompositions, as can be seen, for example, at $p = 15$, $p = 25$, and $p = 45$. This can be explained by the fact that, despite the more cubic division of the domain in the dynamic decomposition, a larger volume is created overall for each processor in which distances have to be checked.

In some cases, however, a lower execution time can be seen despite a lower hit-rate, such as with $p = 60$. After evaluating the data of individual processors, it turned out that this is due to the load balance. Thus, the number of particles allocated or triplets evaluated for the naive strategy differs greatly for 60 processors, since a slice has a width of only $1/6$. In this case, dynamic decomposition is advantageous, since the work is distributed more evenly.

In conclusion, we can state that our assumption regarding the dynamic decomposition, although true in many cases, cannot be used in general and still has a certain naivety, since many factors, such as particle distribution, cutoff distance or number of processors, are decisive for the hit rate.

5.5. Accuracy Comparison

In the following, we investigate the deviation between the results of the CTA algorithm and those of the AUTA algorithm, using the direct algorithm as a basis. The idea is to get an overview of how much the calculated forces with different cutoff distances deviate from those of the direct method and how this affects the calculation time of a simulation step.

We use the mean squared error to calculate the average deviation of the calculated forces with the CTA algorithm compared to those of the AUTA algorithm. The mean squared error is defined as follows:

$$MSE = \frac{1}{n} \sum_{i=1}^{n} (F_i - \hat{F}_i)^2 \tag{5.3}$$

Where $\hat{F}_i$ is the force vector of the i^{th} particle out of n particles calculated by AUTA algorithm, analogously F_i is the force vector of the i^{th} particle calculated by CTA algorithm.

For this test, 1024 uniformly distributed particles in a $10 \times 10 \times 10$ domain are used. Both algorithms were executed with 1 and 16 processors. The decomposition strategy chosen is the naive method, which splits the domain along the x-axis to grant the same type of decomposition for each number of processors. Of course, the deviation also depends on the particle distribution. However, investigations with different distributions were not performed in this work.

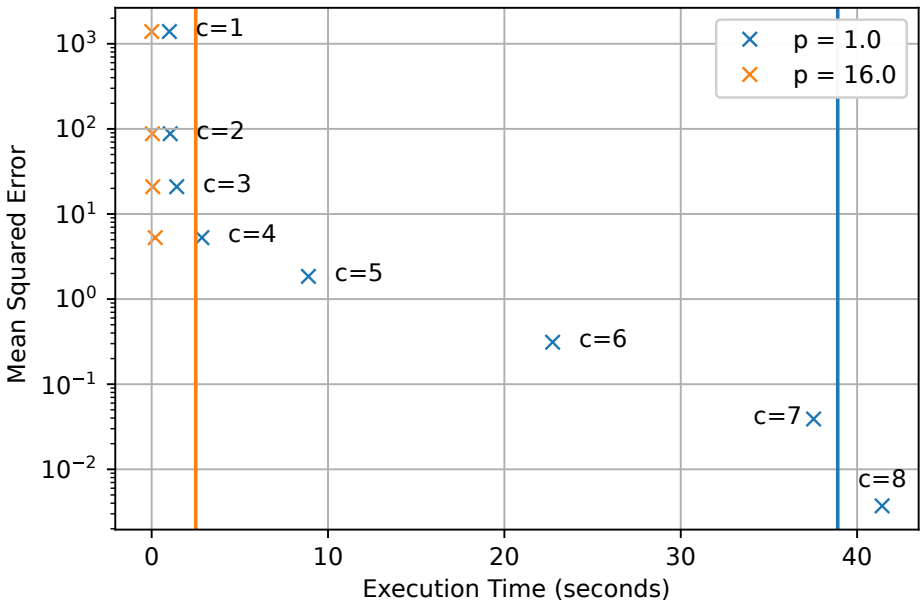


Figure 5.15.: The mean squared error of the CTA algorithm relative to the AUTA algorithm is shown on the y-axis. The x-axis shows the execution time for one simulation step. The vertical lines represent the execution time of the AUTA algorithm with a respective number of processors. The individual points in the Figure represent a combination of a cutoff and a number of processors executed with the CTA algorithm. The values with $p = 1$ and $p = 16$ are identical for $c = 1$ to $c = 4$. For $c > 4$ we can only calculate results with one processor, otherwise the cutoff restriction, as mentioned in section 3.2.4, would be violated.

In Figure 5.15, we can see that the computation time for the AUTA algorithm with one processor is faster than the cutoff algorithm with one processor and $c = 8$. This is because the CTA algorithm has to check additionally for all possible triplets whether they should be evaluated or not. In this extreme case it may make sense to use the AUTA algorithm instead of the cutoff algorithm.

For smaller cutoff distances we see an increasing deviation in accuracy from the calculations of the AUTA algorithm. It can be stated that relatively small deviations occur in changes of the cutoff range between $c = 5$ and $c = 8$, in which the volume of the cutoff-sphere differs strongly. In the transitions between small cutoff radii, such as $c = 1$ and $c = 2$, in which the volume of the cutoff-sphere differs less, strong deviations can be observed. So we can state that for larger radii the results are more stable.

Since the cutoff radius represents a sphere in 3D space, only a relatively small increase in execution time is seen for the transitions between smaller radii, since the volume of the sphere changes less. For the transitions at larger radii, a higher increase can be seen.

Another property that can be observed in this Figure is that the cutoff algorithm cannot calculate all possible cutoff distances for a given number of processors. This is due to the property of the algorithm that restricts the number of neighboring cells. See theory section 3.2.4 for details.

Part IV.

Future Work and Conclusion

6. Future Work

Since in this work we first got an overview of algorithms for the computation of three-body interactions, as well as provided basic implementations, we want to focus now in more detail on improving our implementations. We are especially interested in improvements to the Cutoff Triplet Algorithm, since the computation time for most particle distributions can be reduced from $\mathcal{O}(n^3)$ to $\mathcal{O}(n)$.

As we can see from the results in chapter 5, a big bottleneck for the Cutoff Triplet Algorithm at the moment is the distance computation as many particles in the cells fall beyond the cutoff. To reduce this, we want to integrate Verlet Lists. In pairwise calculations, it was shown that in theory the hit-rate can be significantly increased by using Verlet Lists compared to Linked Cells. For example, with a skin radius of $s = c * 1.2$, the hit-rate can be increased from 16 percent for Linked Cells to almost 58 percent [GST+19]. As can be seen in the investigation of the hit-rate in section 5.4, despite a 3D regular grid decomposition, in which the cell size was chosen only slightly larger than the cutoff for 64 processors, the hit-rate is still significantly lower compared to the values mentioned above for pairwise calculations, therefore special attention should be paid to the hit-rate in three-body interactions. We see currently a great opportunity in the use of Velet Lists to reduce the number of redundant distance checks.

At the moment, our shifting scheme for the CTA algorithm works based on an offset vector. Another possibility is to shift only to direct neighbors and no longer necessarily keep the lexicographic order, but still process all necessary buffers of processors within the yellow marked area in Figure 4.3b. Such a scheme might look like this for the example in Figure 4.3b: $(1,1)$, $(1,2)$, $(1,3)$, $(2,3)$, $(2,2)$, $(2,1)$, $(2,0)$, $(2,6)$, $(3,6)$, etc. Where the given ranks refer to the absolute positions in Figure 4.3a. Such a scheme was not implemented in the context of this work, since a correct working of the algorithms was in the foreground and accordingly a simpler but more intuitive scheme on basis of the lexicographic order was used. However, building on the current implementation, such an optimized scheme can be implemented. One advantage is that this makes many auxiliary methods that we use in the current implementation obsolete and thus the program code can be made clearer.

Since the virtual position of the processors is fixed over the course of the simulation, another possibility is to calculate all offset-patterns and their processing order only once at the beginning of the simulation. This would keep the more intuitive lexicographic scheme, but the calculation of the source and destination ranks would no longer be required in each step, which could have a positive effect on the runtime.

Finally, the Cutoff Triplet Algorithm can be implemented for shared memory environments. Since only three computation units (e.g. threads) are working on a particle subset at a time, we currently see a great opportunity to develop an efficient implementation using a suitable scheme for assigning locks to threads.

7. Summary

The goal of this thesis was to provide an overview of existing algorithms for computing three-body interactions in Molecular Dynamics Simulations, to understand how they work, and to determine their properties. We presented already existing algorithms for direct computation of all interactions in the simulation domain, as well as approximate ones. We concluded that the algorithms of P. Koanantakool and K. Yelick [KY14], are the most promising for us, since they implicitly bring some positive properties compared to the other algorithms. As part of this work, three algorithms of P. Koanantakool and K. Yelick [KY14] were implemented and analyzed in more detail.

We have implemented and studied two direct algorithms and a cutoff algorithm introduced by P. Koanantakool and K. Yelick [KY14]. The algorithms were implemented for distributed memory environments. To ensure that our implementation forms all unique particle triplets and calculates their forces, unit tests were implemented. Based on the results of the tests, we were able to verify that the algorithms work correctly. In order to be able to execute a simulation step with test input, a rudimentary test framework was implemented.

For the direct algorithms, it turned out that the Naive All Triplets Algorithm no longer works efficiently with many participating processors. Nevertheless, we consider the implementation as an important foundation, since we were able to better understand the principle of these algorithms and could implement alternative algorithms based on this.

For the All Unique Triplets algorithm, we observed that a good scaling behavior can be achieved with a suitable ratio of particle number and processors.

For the Cutoff Triplet Algorithm it turned out that in our implementation the distance calculation is a big bottleneck, since many triplets in the cells are beyond the cutoff value.

It was observed that the cutoff triplet algorithm has a significantly better runtime compared to the direct methods for most particle distributions, but has a high load imbalance for heterogeneous distributions, such as a Gaussian particle cloud.

Our investigation of the Cutoff Triplet Algorithm further showed that an appropriate decomposition strategy can help to reduce the computation time, but this should be chosen depending on the simulation and is related to many factors, such as the particle distribution, or the chosen cutoff.

We also analyzed the accuracy of the Cutoff Triplet Algorithm by comparing the results against the All Unique Triplets Algorithm using the Axilrod-Teller potential in order to

get a first impression for future work on how much the results deviate when using different cutoff distances.

Finally, we have provided a brief outlook on future research directions in this field, with a particular focus on the methods of P. Koanantakool and K. Yelick [KY14].

Part V.

Appendix

A. Code Listings

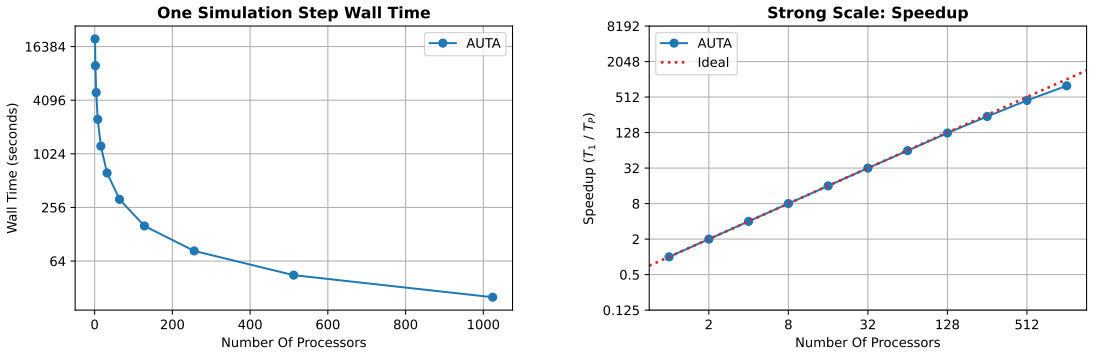
```
1  void sendBackParticles()
2  {
3      MPI_Request requestSend0, requestSend1, requestSend2;
4      MPI_Request requestRecv0, requestRecv1, requestRecv2;
5
6      bool b0Sent = false, b1Sent = false, b2Sent = false;
7
8      if (b0Owner != worldRank) {
9          MPI_Isend(b0.data(), b0.size(), *mpiParticleType, b0Owner, 0,
10                     ringTopology->GetComm(), &requestSend0);
11         b0Sent = true;
12     }
13     // do the same for b1 and b2
14
15     // all buffers have the same size
16     int numRecv = b0.size();
17
18     if (b0Owner != worldRank) {
19         b0Tmp.resize(numRecv);
20
21         MPI_Irecv(b0Tmp.data(), numRecv, *mpiParticleType, MPI_ANY_SOURCE, 0,
               ringTopology->GetComm(),
22                     &requestRecv0);
23     }
24     // do the same for b1 and b2
25
26     if (b0Sent) {
27         MPI_Wait(&requestSend0, MPI_STATUS_IGNORE);
28         MPI_Wait(&requestRecv0, MPI_STATUS_IGNORE);
29     }
30     // do the same for b1 and b2
31
32     if (b0Sent) {
33         b0 = b0Tmp;
34         b0Tmp.clear();
35     }
36     // do the same for b1 and b2
37
38     b0Owner = worldRank;
39     // do the same for b1 and b2
40 }
```

Listing 1: Procedure that we use to send particles in a buffer to their original owner after a simulation step. We use non-blocking communication to avoid deadlocks. A temporary buffer is used so that we can send and receive simultaneously without overwriting data.

```cpp
int mpiShift(std::vector<Particle>& buf, int owner, int src, int dst)
{
    tmpRecv.clear();

    int numRecv;
    MPI_Status status;
    MPI_Request requestSend, requestRecv;

    MPI_Isend(buf.data(), buf.size(), *mpiParticleType, dst, owner,
        cartTopology->GetComm(), &requestSend);

    MPI_Probe(src, MPI_ANY_TAG, cartTopology->GetComm(), &status);
    MPI_Get_count(&status, *simulation->GetMPIParticleType(), &numRecv);

    tmpRecv.resize(numRecv);

    MPI_Irecv(tmpRecv.data(), numRecv, *mpiParticleType, src, status.MPI_TAG,
        cartTopology->GetComm(),
                &requestRecv);

    MPI_Wait(&requestSend, MPI_STATUS_IGNORE);
    MPI_Wait(&requestRecv, MPI_STATUS_IGNORE);

    buf = tmpRecv;

    return status.MPI_TAG;
}
```

Listing 2: Function to exchange buffers for the CTA algorithm. We use non-blocking send and receive commands to avoid deadlocks. With the MPI commands `MPI_Probe` and `MPI_Get_count` we can request the amount of data to receive from the sender.

B. Benchmarks



(a) (b)

Figure .1.: Strong scale benchmark of the AUTA Algorithm with 8192 uniform particles from 1 to 1024 processors. We can see an almost perfect speedup, since each processor has at least 16 particles up to 512 processors.

List of Figures

List of Tables

List of Algorithms

Bibliography

[AT43] BM Axilrod and Ei Teller. Interaction of the van der waals type between three atoms. *The Journal of Chemical Physics*, 11(6):299–300, 1943.

[BDS07] Kevin J Bowers, Ron O Dror, and David E Shaw. Zonal methods for the parallel execution of range-limited n-body simulations. *Journal of Computational Physics*, 221(1):303–329, 2007.

[Ben75] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[BH86] J. H. Barnes and Piet Hut. A hierarchical o(n log n) force-calculation algorithm. *Nature*, 324:446–449, 1986.

[CW00] C.F. Cornwell and L.T. Wille. Parallel molecular dynamics simulations for short-ranged many-body potentials. *Computer Physics Communications*, 128(1-2):477–491, June 2000.

[DGK+13] Michael Driscoll, Evangelos Georganas, Penporn Koanantakool, Edgar Solomonik, and Katherine Yelick. A Communication-Optimal N-Body Algorithm for Direct Interactions. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 1075–1084, Cambridge, MA, USA, May 2013. IEEE.

[GR87] Leslie Greengard and Vladimir Rokhlin. A fast algorithm for particle simulations. *Journal of computational physics*, 73(2):325–348, 1987.

[GSBN22] Fabio Alexander Gratl, Steffen Seckler, Hans-Joachim Bungartz, and Philipp Neumann. N ways to simulate short-range particle systems: Automated algorithm selection with the node-level library AutoPas. *Computer Physics Communications*, 273:108262, April 2022.

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

[KKN+13] Manaschai Kunaseth, Rajiv K. Kalia, Aiichiro Nakano, Ken-ichi Nomura, and Priya Vashishta. A scalable parallel algorithm for dynamic range-limited n -tuple computation in many-body molecular dynamics simulation. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12, Denver Colorado, November 2013. ACM.

[Koa17] Penporn Koanantakool. *Communication Avoidance for Algorithms with Sparse All-to-all Interactions*. PhD thesis, University of California, Berkeley, 2017.

[KY14] Penporn Koanantakool and Katherine Yelick. A Computation- and Communication-Optimal Parallel Direct 3-Body Algorithm. In *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 363–374, New Orleans, LA, USA, November 2014. IEEE.

[L. 13] L. D. O'Suilleabhain. Three body approximation to the condensed phase of water. Master's thesis, University of California, Berkeley, Berkeley, CA, 2013.

[LOG12] Dongryeol Lee, Arkadas Ozakin, and Alexander G Gray. Multibody multipole methods. *Journal of Computational Physics*, 231(20):6827–6845, 2012.

[LZS06a] Jianhui Li, Zhongwu Zhou, and Richard. J. Sadus. A Cyclic Force Decomposition Algorithm for Parallelising Three-Body Interactions in Molecular Dynamics Simulations. In *First International Multi-Symposiums on Computer and Computational Sciences (IMSCCS'06)*, pages 338–343, Hangzhou, Zhejiang, China, June 2006. IEEE.

[LZS06b] Jianhui Li, Zhongwu Zhou, and Richard J. Sadus. Modified force decomposition algorithms for calculating three-body interactions via molecular dynamics. *Computer Physics Communications*, 175(11):683–691, 2006.

[LZS06c] Jianhui Li, Zhongwu Zhou, and Richard J Sadus. Parallelization algorithms for three-body interactions in molecular dynamics simulation. In *International Symposium on Parallel and Distributed Processing and Applications*, pages 374–382. Springer, 2006.

[Mar01] Gianluca Marcelli. *The role of three-body interactions on the equilibrium and non-equilibrium properties of fluids from molecular simulation*. PhD thesis, University of Kent (United Kingdom), 2001.

[MCG+01] Andrew W Moore, Andy J Connolly, Chris Genovese, Alex Gray, Larry Grone, Nick Kanidoris II, Robert C Nichol, Jeff Schneider, Alex S Szalay, Istvan Szapudi, et al. Fast algorithms and efficient statistics: N-point correlation functions. In *Mining the Sky*, pages 71–82. Springer, 2001.

[MS99] Gianluca Marcelli and Richard J. Sadus. Molecular simulation of the phase behavior of noble gases using accurate two-body and three-body intermolecular potentials. *The Journal of Chemical Physics*, 111(4):1533–1540, July 1999.

[oCS13] CMU School of Computer Science. The Barnes-Hut Algorithm. `http://15418.courses.cs.cmu.edu/spring2013/article/18`, 2013. accessed November 1, 2022.

[Pli95] Steve Plimpton. Fast parallel algorithms for short-range molecular dynamics. *Journal of computational physics*, 117(1):1–19, 1995.

[SSJ07] J. V. Sumanth, David R. Swanson, and Hong Jiang. A symmetric transformation for 3-body potential molecular dynamics using force-decomposition in a heterogeneous distributed environment. In *Proceedings of the 21st annual international conference on Supercomputing - ICS '07*, page 105, Seattle, Washington, 2007. ACM Press.

[Tch20] Nikola Plamenov Tchipev. *Algorithmic and Implementational Optimizations of Molecular Dynamics Simulations for Process Engineering*. PhD thesis, Technische Universität München, 2020.

[Ver67] Loup Verlet. Computer "experiments" on classical fluids. i. thermodynamical properties of lennard-jones molecules. *Physical review*, 159(1):98, 1967.