FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Optimization and Evaluation of the Linked-Cell Algorithm

Christian Menges

FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Optimization and Evaluation of the Linked-Cell Algorithm

Optimierung und Evaluierung des Linked Cell Algorithmus

Author:	Christian Menges
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisor:	Fabio Alexander Gratl, M.Sc.
Date:	August 16, 2019

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

Munich, August 16, 2019 Christian Menges

Acknowledgements

Thank you to the Chair of Scientific Computing in Computer Science (SCCS)[1] of TUM's informatics department and Professor Hans-Joachim Bungartz for offering me the opportunity to write my Bachelor's thesis.

I cannot thank my advisor Fabio Gratl enough for his deep introduction to molecular dynamics, for his patient guidance and constant constructive feedback. Apart from my advisor, I would like to thank Steffen Seckler for the fruitful discussions and support, especially for dealing with the build system and LGTM.

[1]`https://www5.in.tum.de/wiki/index.php/Home`

Abstract

The Linked-Cell algorithm is used as an underlying data structure in Molecular dynamics simulations to store and organize particles. It allows us to quickly find neighbors of a particle within a given radius without checking all stored particles, which is necessary to effectively simulate short-range interactions. In this thesis, four different approaches for faster neighbor search and improved SIMD performance are presented and evaluated. First of all, the variation of the cell size is discussed, continuing with two schemes to combine multiple cells to maximize their performance using Single Instruction, Multiple Data operations. Further, the reduction of search space for possible neighbors using sorted cells is discussed. The last chapter deals with an outlook on adaptive approaches which allows a combination of different cell sizes and sorted cells. All these techniques show a performance improvement depending on the characteristics of the simulated experiment, especially on the density of particles and their distribution inside the domain.

Zusammenfassung

Der Linked-Cell Algorithmus beschreibt eine grundlegende Datenstruktur zum Speichern und Organisieren von Partikeln in Molekulardynamik Simulationen. Er ermöglicht insbesondere das schnelle Ermitteln von benachbarten Partikeln innerhalb eines festgelegten Radiusses, was für die effektive Simulation von Potentialen mit kurzer Reichweite unabdingbar ist. In dieser Thesis werden vier Möglichkeiten der Verbesserung der Nachbarschaftssuche und der beschleunigten Auswertung von SIMD Operationen präsentiert. Beginnend mit der Variation der Zellengröße, werden zwei Algorithmen zum Kombinieren mehrerer Zellen gezeigt, wodurch sich die Performanz von SIMD Operationen deutlich erhöht. Anschließend wird die Suchraumminimierung mit Hilfe der Sortierung von Partikeln innerhalb der Zellen diskutiert. Abschließend wird ein Ausblick auf adaptive Varianten des Linked-Cell Algorithmus gegeben, wobei eine Kombinierung von verschiedenen Zellgrößen sowie der Sortierung von Partikeln ermöglicht wird. Alle präsentierten Optimierungen zeigen eine Verbesserung der Performanz in Abhängigkeit von den Eigenschaften des simulierten Experiments. Insbesondere ist ein starker Einfluss der Partikeldichte und der Verteilung der Partikel im Simulationsgebiet zu verzeichnen.

Contents

Part I.

Introduction and Background

1. Introduction

Molecular dynamics (MD) simulations allow researchers from different fields to explore the interactions of up to trillions of particles on a nanometer level. The applications reach from molecular biology [Tid97] over thermodynamics to chemistry and are categorized in two major groups. First of all, Biological experiments, like protein unfolding, often involve only a small number of highly complex molecules. Moreover, there are engineering applications, such as thermodynamics and chemistry, which usually involve simple molecules, but experiments consist of many more particles. Recent developments in hard- and software allow running simulations that are sufficiently large to be compared with actual measurements in real experiments [TSH+19].

The simulated objects interact with each other by creating contracting or repelling forces. This phenomenon can be observed on a macroscopic (planets) and microscopic (molecules) level and is called N-body problem. Since each object interacts with all other objects, the solution has a quadratic complexity $\mathcal{O}(N^2)$, which doesn't scale to trillions of particles. Quadratic complexity can be avoided by exploiting the characteristics of the active potential fields. Many important potentials, like the Lennard-Jones potential, are so-called short-range potentials. In contrast to long-range potentials, like gravity, short-range potentials show a fast decay in magnitude, which allows interactions beyond a chosen cut-off radius to be omitted without losing much precision. Besides the actual computation of the potentials, the search for particles within the cut-off radius is the most expensive task in simulations. To avoid a full search, which again would have quadratic complexity, space partitioning data structures are used to reduce search space.

These data structures provide the highest potential for improvements since they are independent of the calculated physical potential. Furthermore, the search for close neighbors can consume more than 50% of the computation time[1] of an MD-simulation, hence already small improvements show a significant time reduction. In the following thesis, we introduce different techniques to optimize the neighbor search and improve SIMD performance using the Linked-Cell (LC) algorithm.

LCs, also called Cell-Lists, and Verlet-Lists (VL) are used to find neighboring particles quickly. Both are space portioning data structures where LC applies a partitioning based on the domain and VL based on the position of the particles. LC divides the whole domain into a Cartesian grid of adjacent blocks/cells which contain the particles for their represented region. The search for neighboring particles is then reduced to the cell of the particle and the surrounding cells. If particles leave the space of its cell, an update of the LC container is necessary, which rearranges all particles into the correct cell.

[1]slide deck from Univ.-Prof. Dr. Michael Bader: `https://www5.in.tum.de/lehre/vorlesungen/sci_compII/ss12/moldyn_03.pdf`

VLs store a list of neighboring particles within a ”skin” radius[2] for each particle [Ver67]. If a particle leaves or enters the sphere defined by the ”skin” radius, the VL needs to be updated. The size of the skin radius determines the time after which an update is necessary. The ”skin” radius has the cut-off radius as its lower bound, but to reduce the number of updates the radius is usually set to a larger value. This allows the particles to move slightly while keeping the data structure valid. VL are fast in finding neighbors but the construction and update procedure is quite expensive because it requires a full search. This can be avoided by coupling LC and VL. The LC container is only used to update the VL, while VL are used for computation.

Since LCs are used directly and as a basis for VLs, their performance is crucial for simulations.

[2]The definition of ”skin” radius is not consistent. Here, it is used as a total radius around a particle in which neighbors are added to the VL. In AutoPas, the ”skin” is an additional term, which must be combined with the cut-off radius to retrieve the total radius.

2. Theoretical Background

2.1. Intermolecular Potential

Particles exert different potentials on each other, which results in forces and movement. These potentials can be categorized in long and short-range potentials. Long-range potentials, such as gravity, do even have a significant impact when the objects are far away from each other. Short-range potentials are characterized by a fast decay of magnitude. Therefore, it is possible to omit the effect of this potentials if the objects exceed a certain cut-off distance to each other. This distance is called cut-off radius r_c. Also, potentials used for simulations are only an approximation of the real behavior since the correct formulas are too complex to be computed efficiently.

One of the most important potentials for Molecular dynamics (MD) is the Lennard-Jones (LJ) 12-6 potential [Rap04]. The potential between two particles i and j with position x_i and x_j is given by:

$$u_{LJ}(r) = 4\epsilon\left(\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right), \quad \textit{with } r = ||x_i - x_j|| \tag{2.1}$$

The LJ potential combines Pauli repulsion and van der Waals forces. σ describes the distance where neither repulsion nor contraction are present, $u_{LJ}(\sigma) = 0$. ϵ is the depth of the potential well. Figure 2.1 shows that this potential quickly goes to zero. Since the evaluation of potentials is a computationally expensive task (high exponents), it is reasonable to avoid as many evaluations as possible. The amount of evaluated potentials is regulated by the cut-off radius.

Furthermore, the potentials are often adapted for easier computation. This explains the chosen exponents in the LJ potential. The higher exponent is exactly twice the smaller

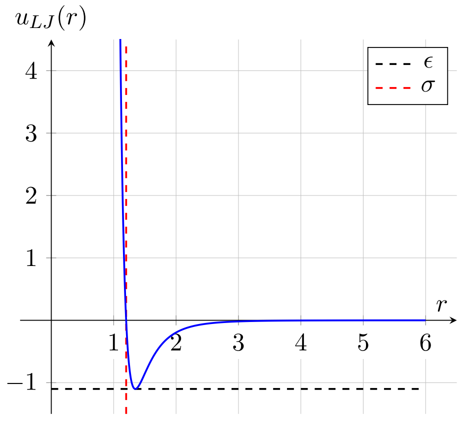


Figure 2.1.: Lennard-Jones-Potential for $\epsilon = 1.1$, $\sigma = 1.2$

exponent, resulting in only one square operation.

Potentials manifest themselves into forces which lead to movement. The force which is caused by a potential $u(r)$ is calculated according to Equation 2.2.

$$f(r) = -\nabla u(r) \tag{2.2}$$

Simulations often directly use the force equations because the velocity of a particle depends on the active force.

2.2. Newton 3 Optimization

Newton's third law of motion states that "When one body exerts a force on a second body, the second body simultaneously exerts a force equal in magnitude and opposite in direction on the first body"[New87]. This law can be used to cut the number of force calculations in half. By calculating the force between two particles i and j, the calculated force f is added to f_i and subtracted from f_j. The resulting complexity still is in $\mathcal{O}(N^2)$, but only half of the time is needed. When this optimization is applied, it is necessary to keep track of which interactions are already evaluated. This limits possible traversal schemes, which are going to be discussed in Section 2.5.

2.3. Data representation: SoA and AoS

Simulation data consists of objects (Particles, Planets, etc), which are defined by their attribute values. These attribute values can be stored in two different ways: inside a Structure of Arrays (SoA) or an Array of Structures (AoS). AoS is an object-oriented approach which groups different attributes belonging to a single object in one structure. The data of multiple objects is then stored in a single continuous array. In contrast, SoA groups attributes according to their purpose. For example, all positions on the x-axis are stored in one array and the positions on the y-axis in another array. The data of a particular object can be retrieved by collecting all attributes out of the arrays using the same index.

Both data layouts have advantages and disadvantages. AoS allows simple insertion and deletion of objects since all values are stored together. If the total size of an object's attributes is smaller than the size of a cache line (constructive interference size) it is possible to retrieve the information of one object by a single memory access. SoA offers the same effect for attributes of the same purpose. This is useful to fill Single Instruction, Multiple Data (SIMD) registers and apply operations to multiple values at the same time.

In the context of Molecular simulations, it is not possible to clearly identify the best suited data layout since object based as well as attribute based access is needed.

2.4. Linked-Cell Algorithm

LCs is a space partitioning algorithm which allows efficient neighbor search. The simulation domain is divided into a Cartesian grid of adjacent cells. Each cell stores all particles within the region represented by itself. The 3D index of the cells can be transformed into a 1D-index, which allows storage of all cells in a continuous 1D-array. Most of the time, the cell size is chosen to match the cut-off radius. Sometimes, this is not possible since the domain size is not divisible by the cut-off radius. In this case, the next greater integer divisor of the domain size is used. As the cell size is equal or bigger than the cut-off radius, all particles which are potential interaction partners must be located in the cell of the particle or in the surrounding $3^{dimension} - 1$ cells, which share a face, edge or vertex. The surrounding cells can easily be found due to the regular partitioning of the Cartesian grid. The number of particles inside a cell is bounded, thus the maximum number of possible interaction partners is fixed. This reduces the complexity from $\mathcal{O}(N^2)$ for a full search to $\mathcal{O}(N)$. If the cell size exactly matches the cut-off radius, the probability that a particle within the search space is a neighboring particle is shown in Equation 2.3.

$$HR = \frac{\frac{4}{3}\pi r^3}{(3 \cdot r)^3} = \frac{4}{3 \cdot 27}\pi \approx 0.155 = 15,5\% \tag{2.3}$$

This probability is called hit rate (HR) and the corresponding complementary probability is an important indicator for unnecessary distance calculations during the simulation.

When particles leave their cell, it is necessary to perform an update and rearrange all particles to the correct cells. The update process has linear complexity $\mathcal{O}(N)$, since the procedure of checking and updating the cell applied to each particle has constant complexity $\mathcal{O}(1)$.

There are different options for particles which move outside of the boundary of the domain [GKZ07, p. 37-38]:

Outflow The leaving particles are deleted. This wouldn't happen in real experiments because real particles cannot get lost.

Reflecting The particles bounce back at the domain boundary. This can be implemented by adding a "ghost" particle which is created when the particle is near to the boundary. Due to repulsive forces the particle is redirected and stays inside the domain. The ghost particle has the same distance to the boundary as the approaching particle and is located mirror-inverted. It is important that the "ghost" particle only affects the approaching particle it was created for [GKZ07, p. 69].

Periodic If a particle moves out of the domain, it moves into the domain on the opposite side of the domain. This simulates an infinite domain size.

Specialized Besides the previously mentioned boundary conditions, there are specialized variants to mimic different kinds of physical effects like a heated reflecting boundary.

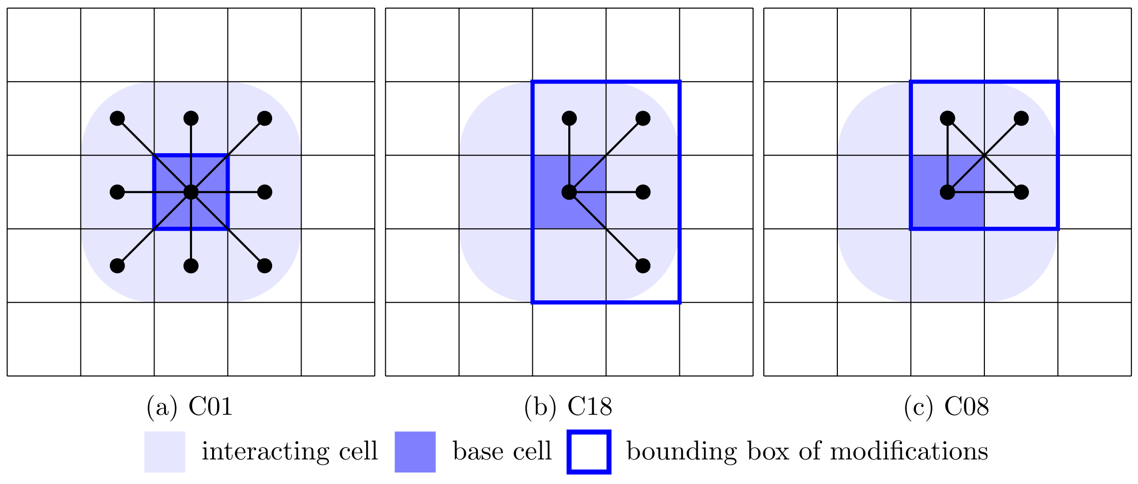


Figure 2.2.: Traversal base steps: During a domain traversal, base steps are applied to each cell. The base cell represents the current cell. All interactions are defined relative to the base cell. To avoid data races, it is necessary to define read-only and modified cells. Modified cells are represented by a rectangular bounding box. The rectangular shape allows a simple domain partitioning.

2.5. Traversals

We now want to consider different domain traversal schemes to find all possible interaction partners. A traversal scheme consists of a domain partitioning and a base step which is applied to each cell. Domain partitioning is important for parallelization since the elements can be assigned to multiple threads. We will discuss partitioning based on domain coloring and slicing. Each cell must interact with all surrounding cells but the ordering of interactions can vary. If a cell i is adjacent with a cell j, it is not important whether the particle interactions between those cells take place when i or j is the currently worked on cell. A base step is an interaction pattern which represents one specific ordering, thus it determines if the interactions are calculated in i or j.

2.5.1. Base steps

If all interactions from a base cell and its surrounding cells should be evaluated, a star-like pattern is the simplest approach (Figure 2.2a). This base step is called C01 and all interactions take place within a single base step. The same interactions are as well computed for all other base steps but are distributed among different base cells. C01 is completely symmetric, making it impossible to use Newton 3 (N3) optimization. Otherwise interactions would be evaluated multiple times. Without N3 optimization, changes are only applied to the base cell while other cells are only read. Therefore, multiple threads can work on cells in arbitrary order without interfering.

To enable N3 optimization, the symmetry must be removed. This happens in the so-called C18 traversal (Figure 2.2b). The number of cell interaction is halved, but these interactions apply to both interacting cells. Therefore, all interactions from C01 still happen, but are spread over multiple base steps. For example, the interaction of the base cell with the

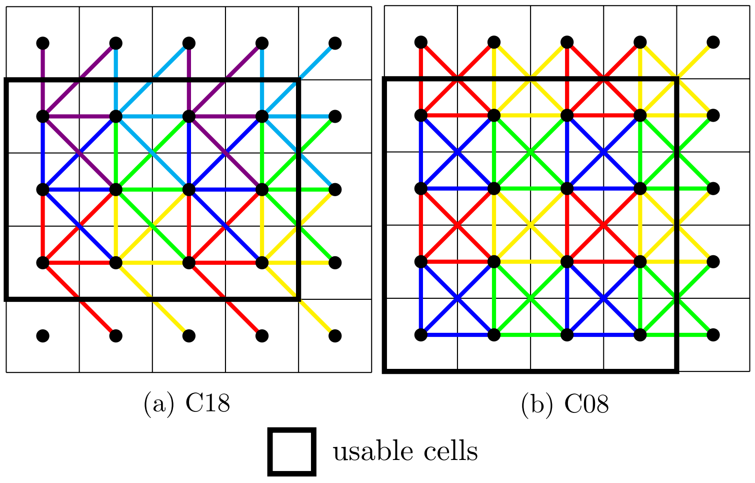


Figure 2.3.: Domain coloring: Each color is executed separately to prevent interference and data races. The C18 base step operates on a 2×3 grid, requiring 6 colors. One C08 base step only affects cells on a 2×2 grid, causing the usage of 4 colors

bottom left cell in C01 now takes place if the bottom left cell is the base cell. Since all interactions of cells affect both cells, the only difference between enabled and disabled N3 optimization is the duplication of calculations.

C08 (Figure 2.2c) is a variation of C18, trying to condense all interactions inside the smallest possible area (3D version shown in Figure 5.3). This allows a higher degree of parallelism and increases the cache-efficiency since less cells are touched in a single base step. The only difference between C08 and C18 is the interaction to the bottom right in C18 which is now shifted one cell to the top. As a result, not all interactions involve the base cell.

All base steps access other cells. If the base cell is directly located at the boundary of the LC container, not all requested cells are present. To avoid requests of non-existing cells, the interactions which require these cells can be removed for the cells at the boundary, introducing much more complexity. Another solution is to add an additional halo cell layer around the domain and only apply the base step to the cells inside the domain boundary. Now, all requested cells are present. For C18 and C08, it is not necessary to add halo layers to all direction as seen in Figure 2.3 (usable cells).

2.5.2. Domain coloring

Parallelization is crucial for MD simulations to compute all interactions in a reasonable amount of time and is applied at different levels: data level, node level and multi-node level. Data level parallelism is discussed in more detail in Chapter 6. Node level and multi-node level function according to a similar principle: The domain is divided into multiple chunks and each chunk is evaluated from a different thread/node. Parallelism is not going to be discussed on the multi-node level in more detail but rather concentrate on node level parallelism based on threads.

Except for the C01 traversal base step, all base steps modify additional cells. Modification is a mutual exclusive action, thus interference of different threads must be strictly avoided.

Interference doesn't take place if two threads are not working on overlapping regions at the same time. A region is given by the cells modified by a base step. To keep it simple, we only consider cubic (in 2D: rectangular) regions of modifications. This causes a region to have the size of the minimal enclosing bounding box shown in Figure 2.2.

Finding non-overlapping schedules is a graph coloring problem. The interactions between cells resemble a graph, which is colored with the base step pattern (Figure 2.3).

For C01, no interference can take place because no surrounding cells are modified. Hence, only one color is needed and threads can work on each base cell completely independent from each other.

C18 modifies surrounding cells. The bounding box of modified cells has a size of 2×3 (2D) or $2 \times 3 \times 3$ (3D). If all threads work on base cells which are further apart to each other than the size of the bounding box of modified cells, no data race is possible. We use the minimal interference free distance and a barrier based synchronization mechanism. For the configuration shown in Figure 2.2, 6 colors are needed, which results in the domain coloring shown in Figure 2.3a. The corresponding 3D traversal requires 18 colors. Each color is shifted so that in the end each cell was executed as base cell.

C08 requires only $2^{dimension}$ colors. This makes it more efficient than the C18 traversal, since less iterations and barriers are necessary. Furthermore, smaller regions show better load balancing since areas with more particles are distributed between more threads.

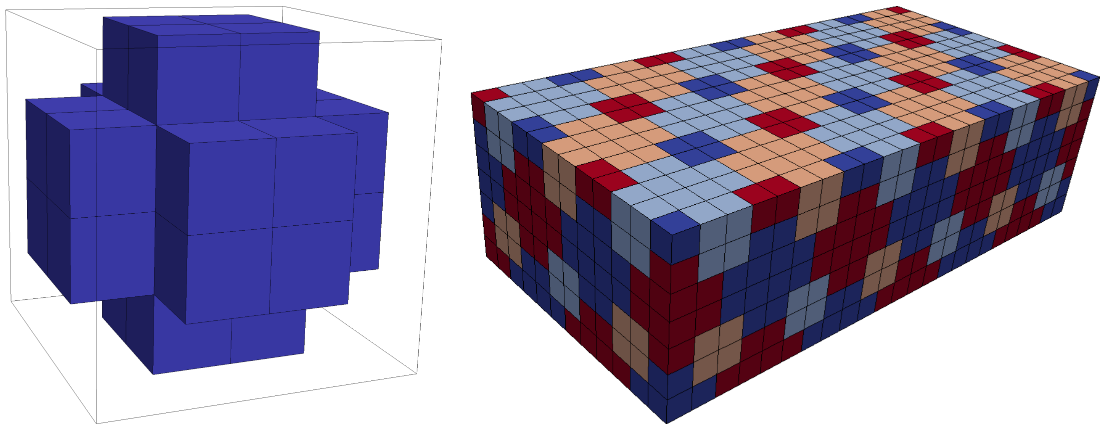

(a) Base step (b) 3D partitioning

Figure 2.4.: C04 Traversal: The C04 base step describes a cell formation in which each cell is evaluated using the C08 base step. The 3D partitioning results in 4 colors. Graphics by Nikolai Tchipev

A special case of domain coloring is the so-called C04 traversal, introduced by Nikolai Tchipev. It uses the C08 base step, but only subdivides the domain into four colors [TSH+19]. This is only possible because there is an intermediate step. At first, the domain is divided into regions using the cross-like shape shown in Figure 2.4a, resulting in a partitioning shown on the right. Now, all threads work on a different region (crosses) of the same color and apply the C08 base step to each cell within the base step region. The low number of colors reduces the time consumption of barrier synchronization between each color.

2.5.3. Slice based

Besides coloring, the domain can be partitioned into slices. Each slice is processed by a single thread. All cells inside a slice are evaluated continuously which provides better locality, thus a better performance inside the slices. This partitioning is independent from the used base step, making all base steps applicable inside the slices. Since the interactions between particles are also present between the slices it is necessary to introduce mechanisms like locking to avoid data races (not needed for C01). When cell layers (slice of thickness 1), which are close (less than cut-off) to another slice are evaluated, the slice itself must be locked. The same applies for layers which are accessed in the base step. As soon as no more cell of the locked slice is needed, the lock is released. Locking is an expensive operation, but the probability of waiting for a lock is small. The major issue of sliced partitioning is load balancing, which is difficult or even impossible. If the objects are not homogeneously distributed inside the domain, then some slices contain more objects than others. This causes a longer execution time for these slices, while other threads are already finished and idle. More slices provide better load balancing but also increase the probability of waiting for a lock. It is also possible to maintain statistics on how particles are distributed inside the domain and create slices accordingly.

3. Description of Tools

3.1. AutoPas

All optimizations discussed were implemented and tested inside the AutoPas library [GST$^+$19]. AutoPas is part of the TalPas[1] (Task-based Load Balancing and Auto-tuning in Particle Simulations) project and aims at providing a self-optimizing framework for N-body simulations. Algorithmic autotuning is an important part of many computer science applications, but it is new in the context of molecular dynamics. Autotuning solves two problems:

1. A lot of simulations change their behavior during the simulation. For example: Simulations of spinodal decomposition start with relatively fast-moving objects. Therefore it is necessary to choose all simulation parameters according to this behavior. The speed of the objects influences the time step size, which must be small to avoid missing interactions with fast-moving objects. During the simulation, the object's speed decreases and clusters of objects are created. Due to the clustering, some cells contain a lot of objects while others contain only a few. The starting parameters are not suitable anymore: time step size could be increased and the cell size adapted appropriately. If the simulation parameters are static, this is not possible.
2. Researchers using MD usually are not experienced in computer science. However, in order to run simulations in an acceptable amount of time, a lot of knowledge and experience in this discipline is needed to implement the parallel algorithms and choose the fastest configuration. Nevertheless, even for experts, it is not always evident which configuration will provide the best performance. AutoPas tries to bridge this gap of knowledge by providing an easy to use interface and finding the optimal configuration by itself.

Autotuning is done by using statistics and measurements of different configurations which have a high likelihood to deliver the best performance.

The autotuner can apply several different containers, like Direct Sum (DS), LC, VL and different variations of those containers which are integrated into the library. Along with the containers, different domain traversal schemes are implemented (for a full list, see Figure A.2)

The whole library is designed for parallel execution. On the data-level, vectorization is used to achieve higher throughput. The traversals are parallelized using thread based parallelism with OpenMP[2].

AutoPas is developed to deliver the best node-level performance and includes no functionality to run on multiple nodes. To run larger simulations, AutoPas was integrated into ls1

[1]`https://wr.informatik.uni-hamburg.de/research/projects/talpas/start`
[2]`https://www.openmp.org/`

	CoolMUC-2	**SuperMUC-NG**
CPU	Intel Haswell Xeon E5-2697 v3	Intel Skylake Xeon Platinum 8174
base frequency [GHz]	2.6	3.1
SIMD (AVX)	AVX2	AVX2, AVX-512
Level 1 cache size [KB]	14 x 32 instr. 14 x 32 data	24 x 32 instr. 24 x 32 data
Level 2 cache size [KB]	14 x 256	24 x 1024
Level 3 cache size [MB]	35	33
Thermal Design Power (TDP) [W]	145	240
Cores per Node	28	48
RAM per Node [GB]	64	96 (Thin nodes) 768 (Fat nodes)

Table 3.1.: Technical specifications of benchmark platforms

mardyn[3] [NBB+14]. This makes it possible to run simulations on multiple nodes by using ls1 mardyn for MPI communication between nodes and an instance of AutoPas on each note. The multiple instances of AutoPas allow an adaptive autotuning based on the regions which are processed by the notes.

AutoPas is open source software[4] and licensed under BSD 2-Clause.

3.2. Computation Platforms

All benchmarks were run on infrastructure provided by the Leibnitz Supercomputing Center[5] (LRZ). In the following sections we will briefly introduce the used hardware and their characteristics.

3.2.1. CoolMUC-2

CoolMUC-2 [6] is a Linux (SUSE Linux Enterprise Server 11) cluster attached with Intel Haswell Xeon E5-2697 v3 [7] processors. This processor supports AVX2 which allows 256-bit wide vector operations. In addition, the huge Level 3 cache relative to the number of cores (2.5 MB per core) is notable. All benchmark programs on this system were compiled with GCC 8.3.

[3]`http://www.ls1-mardyn.de/home.html`
[4]`https://github.com/AutoPas/AutoPas`
[5]`https://www.lrz.de/`
[6]`https://www.lrz.de/services/compute/linux-cluster/overview/`
[7]`https://ark.intel.com/content/www/us/en/ark/products/81059/intel-xeon-processor-e5-2697-v3-35m-cache-2-60-ghz.html`

3.2.2. SuperMUC-NG

SuperMUC-NG is the main super computer of the LRZ and takes the ninth place on the top 500 list[8]. The system features 311,040 cores and scores 19.476 PFlop/s on LINPACK benchmark[9]. This system is of major interest for SIMD operations. The CPU, Intel Skylake Xeon Platinum 8174[10], supports AVX-512 which allows to process eight 64-bit wide floating point numbers at a time.[11]

To run more energy-efficient the system uses a direct warm water cooling. In addition, the processors are usually run in 205 W mode instead of 240 W [12]. This makes the characteristics of the processor similar to Xeon Platinum 8168[13].

[8]https://www.top500.org/lists/2019/06/

[9]https://doku.lrz.de/display/PUBLIC/Hardware+of+SuperMUC-NG

[10]https://ark.intel.com/content/www/us/en/ark/products/136874/intel-xeon-platinum-8174-processor-33m-cache-3-10-ghz.html

[11]https://doku.lrz.de/display/PUBLIC/Details+of+Compute+Nodes

[12]Information retrieved from internal correspondence with LRZ

[13]https://ark.intel.com/content/www/us/en/ark/products/120504/intel-xeon-platinum-8168-processor-33m-cache-2-70-ghz.html

4. Related Work

The work of this thesis is based on numerous research topics.

William Mattson and Betsy M. Rice investigated the impact of different cell sizes in [MR99]. We can confirm their findings for AoS, but not for SoA.

Petro Gonnet proposed to sort particles along the line which is drawn by connecting the centers of two interacting cells [Gon07]. After projection onto this line, the search for interaction partners for a fixed particle can stop if the projected positions are further apart than the cut-off radius. The achieved speed up was approximately 28 % which can be verified for dense domains by our results shown in Chapter 7. This topic was further investigated by [WG11]. For their implementation, sorting was always more beneficial than the number of particles in a cell exceeds ≈ 19. In contrast, for our implementation already ≈ 8 particles are enough to justify the use of sorting.

FDPS (Framework Developing Parallel Particle Simulation Codes) is similar to AutoPas / ls1 mardyn and aims to provide a flexible and easy to use interface for the fast development of particle simulations. It supports many different short and long-range potentials and comes with a lot of different configuration options. This library doesn't support autotuning. One of the implemented containers is an adaptive KD-tree which divides the domain according to the distribution of particles [ITH$^+$16].

Part II.

Optimizations

5. Varying Cell Sizes

In the following sections we will discuss the influence of the chosen cell size in comparison to the standard cell size which is equivalent to the cut-off radius. To keep the varied cell size independent from a particular cut-off radius, it is defined as a factor, which will be called cell size factor (CSF). Therefore, the absolute cell size is given by CSF · cut-off. The CSF determines the maximum number of neighboring cells in each direction which needs to be evaluated to consider all interactions. This number is called *overlap*. The C01 base step in Figure 2.2 illustrates this. In this figure the CSF equals 1.0. Thus, it is necessary to take one cell on the left and right as well as one cell on the top and bottom into account. Consequently, the overlap equals 1. The mathematical expression of the overlap value is given in Equation 5.1.

$$overlap = \left\lceil \frac{1}{CSF} \right\rceil \tag{5.1}$$

5.1. Decreasing Cell Sizes

5.1.1. Intuition

Decreasing the cell size allows a better discretization of the interaction sphere (IS). Figure 5.2 illustrates the change in the discretization for different CSF while the cut-off radius is constantly 1.0. If the CSF equals one, the IS must be approximated by a square. For a CSF of $\frac{1}{2}$ the IS still resembles a square but the search space got already reduced since the base cell got smaller. If the CSF is divided again, the interactions sphere starts to form a circle. The cells at the edges can be excluded since they are completely outside of the IS.

Better approximations of the IS decrease the search space for neighboring particles and therefore increase the HR. Figure 5.1 shows the HR dependent on the applied CSF for a 3D IS. The discontinuities for values smaller than 1.0 are caused by the discretization and mark exactly the values where additional cells can be excluded or must be included. The major jumps at $\frac{1}{2}$ and 1.0 are values which divide the cut-off radius without remainder. The HR drops immediately for a smaller value because additional cells are needed to cover all possible interactions. The additional cells are almost completely outside the IS, increasing the search space. We want to illustrate this effect for a CSF of 1.0. This CSF allows all interactions to take place within a $3 \times 3 \times 3$ cell block. If the factor is infinitesimally smaller than 1.0 it is no longer possible to use a $3 \times 3 \times 3$ block. Now, a $5 \times 5 \times 5$ cell block without the cells at the edges is required to cover all interactions. In total, 54 additional cells are needed.

The smaller discontinuities e.g. at $\approx$0.707 (Equation 5.2) show the previously discussed case of whole cells being excluded because they are completely outside the IS. Mathematically, the distance between the closest corners of the base cell and the interacting cell must be greater or equal to the cut-off radius.

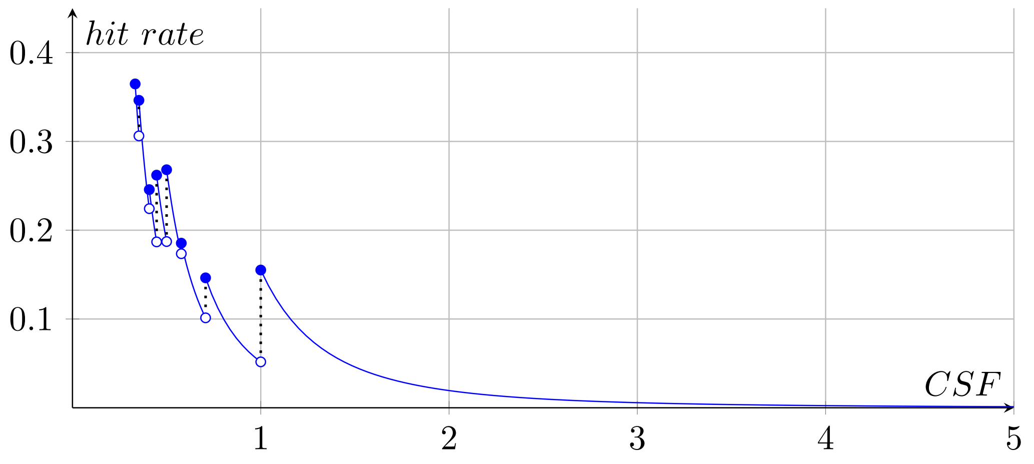


Figure 5.1.: Hit Rate for CSF between $\frac{1}{3}$ and 5

For ≈0.707 (Equation 5.2) this distance is equivalent to the 2D diagonal:

$$r_c \leq \sqrt{2 \cdot (CSF \cdot r_c)^2} \Leftrightarrow CSF \geq \sqrt{\frac{1}{2}} \Leftrightarrow CSF \geq 0.707 \tag{5.2}$$

With this information we can define a formula for the HR in 2D. The HR is equivalent to a packing problem and is closely related to the problem of how many lattice points are contained in a n-dimensional sphere. For a normal circle with radius r the number of lattice points $N(r)$ inside is defined by the Gauss's Circle Problem[1]

$$N(r) = \underbrace{1}_{1} + \underbrace{4 \cdot \lfloor r \rfloor}_{2} + \underbrace{4 \cdot \sum_{i=1}^{\lfloor r \rfloor} \left\lfloor \sqrt{r^2 - i^2} \right\rfloor}_{3} \tag{5.3}$$

The formula divides the circle into different regions and computes the number of points inside for each of them. Starting from a point in the middle (1), all points on the axis are added (2). The sum reflects a single area at the corners. Since these areas are of equivalent size, the sum is multiplied by 4 (3).

In our case, the search space is given by a square whose edges are rounded with radius r. This shape is almost equivalent to a circle, except that additional points were added around the axis. By adding these points to Equation 5.3 we receive the number of cells within a 2D IS:

$$N'(r) = 1 + 8 \cdot \lceil r \rceil + 4 \cdot \lceil r - 1 \rceil + 4 \cdot \sum_{i=1}^{\lfloor r \rfloor} \left\lfloor \sqrt{r^2 - i^2} \right\rfloor \tag{5.4}$$

Note, that the floor function in (2) is replaced by a ceiling function. The original formula requires a point to be inside of the circle. Nonetheless, since cells must be already included if only a part of them is within the IS, the ceiling function is necessary.

[1] Weisstein, Eric W. "Gauss's Circle Problem." From MathWorld–A Wolfram Web Resource. `http://mathworld.wolfram.com/GausssCircleProblem.html`

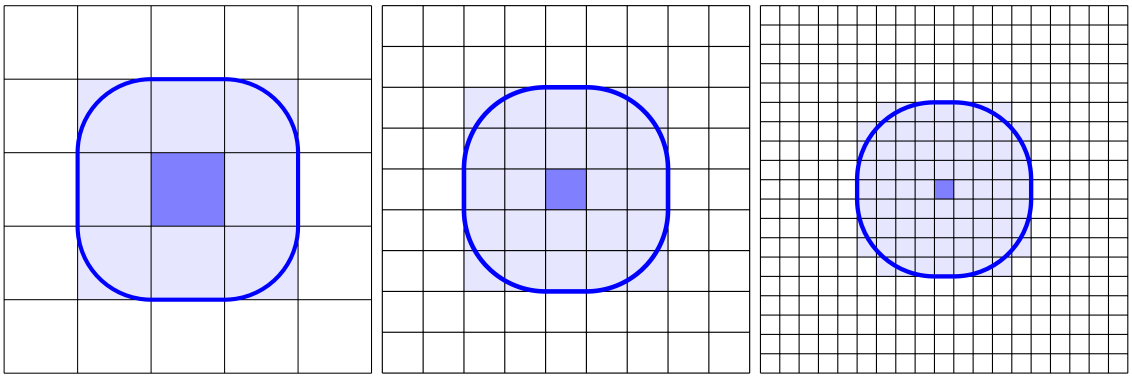

(a) CSF = 1, overlap = 1 (b) CSF = $\frac{1}{2}$, overlap = 2 (c) CSF = $\frac{1}{4}$, overlap = 4

Figure 5.2.: Discretization of the interaction sphere

The 2D-HR is than given by

$$HR(CSF) = \frac{\pi \cdot r_c^2}{N'(1/CSF) \cdot (CSF \cdot r_c)^2} = \frac{\pi}{N'(1/CSF) \cdot CSF^2} \tag{5.5}$$

Equation 5.5 shows that the optimal HR of one is not reachable with a discrete approximation since a circle could only be packed with infinitesimally small squares. Therefore, the CSF must go towards zero and it holds that

$$\lim_{CSF \to 0} HR(CSF) = 1 \tag{5.6}$$

At the same time the number of cells $N'(1/CSF)$ strives towards infinity. However, for CSF $\gg 0$ the number of cells grows rapidly. The caused overhead soon eats up the benefit of higher HRs. For a CSF of 0.5 it is necessary to take 125 cells into account, these are almost 5 times more cells than for a CSF of 1.0. For a CSF of $\frac{1}{3}$ over 300 cells are required.

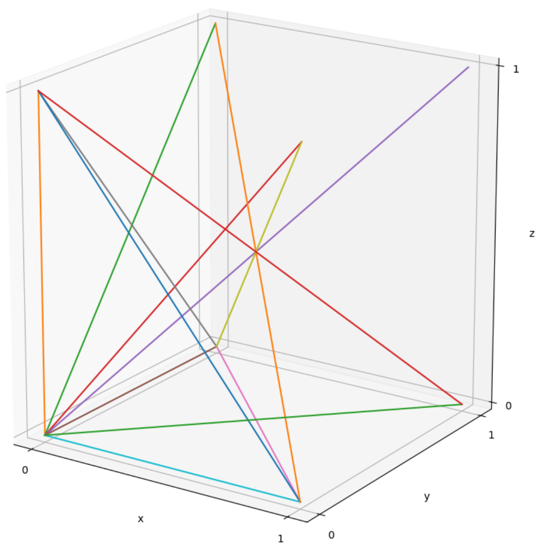
5.1.2. Implementation

A decrease in CSF causes an increase of the overlap value. All base steps shown in Figure 2.2 assume an overlap value of one and are no longer sufficient to cover all cells within the cut-off radius. In case of the C01 and C18 base step, the generalization to greater overlap values is trivial. Only the search space for valid cells needs to be increased to match the overlap value. Finding interaction pairs for the C08 traversal is more complex. The simplest variant is to use the C18 base step and shift all interactions into a cell-block of size $(overlap + 1)^3$. This includes all necessary interactions for C08. However, there are multiple possible shifts and for optimizations discussed in Chapter 6 it is important that the interactions are structured. Therefore, we propose a new algorithm which can be seen in algorithm 1. The position/index of a cell is expressed relative to the current base cell, which makes it possible to calculate all offsets once and only add them to the current base cell position/index.

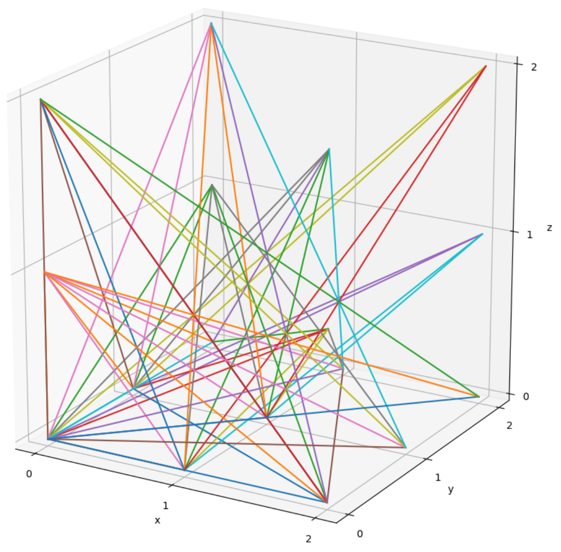
Lines 1-4 generate a 3D Cartesian grid of cells in the size of $overlap + 1$ in each direction. This grid will be used to get the position/index of cells in the next step. Afterwards, the algorithm iterates again over all positions, but now connects cells to form cell pairs.

Algorithm 1: Cell pair algorithm C08

```
Input: overlap
Output: cellPairs
// generation of 3d coord representing single cells
1  for $x \leftarrow 0$ to $overlap + 1$ do
2      for $y \leftarrow 0$ to $overlap + 1$ do
3          for $z \leftarrow 0$ to $overlap + 1$ do
4              cells.append([x,y,z])
// generation of interactions between cells (3d coord)
5  for $x \leftarrow 0$ to $overlap + 1$ do
6      for $y \leftarrow 0$ to $overlap + 1$ do
           // store index of current cell on base plate
7          index $\leftarrow$ cells[x * (overlap + 1)$^2$ + y * (overlap + 1)]
8          for $z \leftarrow 0$ to $overlap + 1$ do
               // origin (front left)
9              cellPairs.append([ cells[z], index ])
               // back left
10             if if y != overlap and z != 0 then
11                 cellPairs.append([ cells[(overlap + 1)$^2$ - (overlap + 1) + z], index ])
               // front right
12             if x != overlap and (y != 0 or z!= 0) then
13                 cellPairs.append([ cells[overlap * (overlap + 1)$^2$ + z], index ])
               // back right
14             if y != overlap and x != overlap and z!= 0 then
15                 cellPairs.append([ cells[(overlap + 1)$^3$ - (overlap + 1) + z], index ])
```



(a) overlap = 1



(b) overlap = 2

Figure 5.3.: Pairwise interactions of cells according to the C08 base step

Pairs are formed by looking at a cell on the x-y plane (z = 0) and connecting this cell with cells which are located at the vertical edges (front left(origin), front right, back left and back right) of the created grid. It is not possible to always connect a point on the x-y plane to all four vertical edges since this would cause multiple evaluations of the same cell pair when the base cell changes. The results for overlap 1 and 2 can be seen in Figure 5.3

To keep it simple, the provided pseudo-code doesn't contain a distance check to skip cell pairs, whose distance is greater than the cut-off radius.

5.2. Increasing Cell Sizes

Increasing cell size is a contra-intuitive approach to increase performance in special circumstances. As seen in Section 5.1, the variation of cell sizes must balance the size of the search space and management overhead introduced by additional cells. The management overhead decreases for larger CSF, making it faster if this overhead is a significant part of the computation. This is the case if the domain is sparse. Depending on the density, the CSF can be arbitrarily increased. The edge case that the cell size equals the domain size is called DS, since each particle directly interacts with all other particles. Another advantage of bigger cells is that SoA operations perform better for a higher number of particles (reasoning in Chapter 6).

Since the interactions cannot be further reduced, the traversal scheme stays the same for all CSF ≥ 1.0 except for DS. DS consists of only one cell, requiring no traversal scheme. Figure 5.1 shows the inverse cubic decay of the HR formulated in Equation 5.7 for values of the cell size factor greater or equal 1.0.

$$HR(CSF) = \frac{\frac{4}{3}\pi}{3^3 \cdot CSF^3}, \quad CSF \geq 1.0 \tag{5.7}$$

5.3. Evaluation

We tested the impact of varying cell sizes on CoolMUC-2. To avoid distortions, all benchmarks were run with a single thread. The benchmark for C04 only contains results for CSF ≥ 1.0. Due to the special domain coloring of C04, it is not possible to implement smaller cell sizes since this might cause data races.

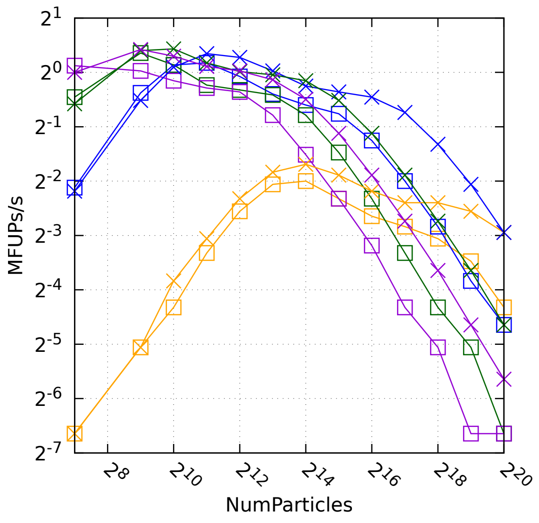
Figure 5.4 shows the results for C01, C04, C08, and C18. All traversals show a similar behavior. If the domain is sparsely populated, increasing the cell size yields better results. This is plausible, since in sparse domains the management overhead becomes more apparent and explains the poor results for decreased cell sizes. If the domain is more dense, smaller cell sizes get more beneficial. The management overhead in this case is hidden by the avoided unnecessary distance calculations. In real world examples, the density usually is not extremely high, which justifies to not lower the CSF under $\frac{1}{3}$.

Both data layouts are mostly equally affected by the changing cell sizes. SoA performs worse than AoS when the domain is sparse and always surpasses the performance of AoS if the domain gets denser.

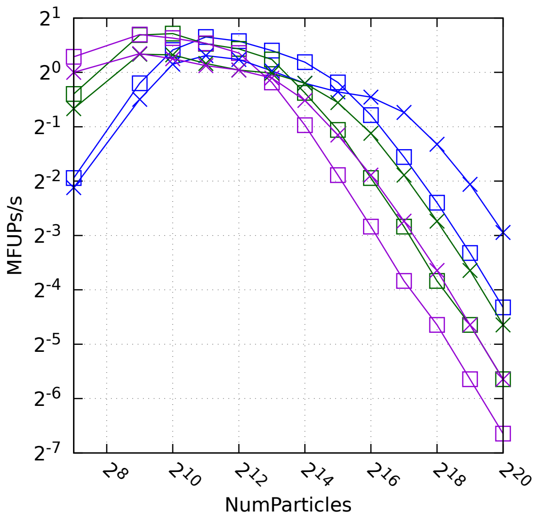
For AoS and high densities, the reached speed-up for CSF $= 0.5$ is comparable with the findings of William Mattson and Betsy M. Rice [MR99]. This is not the case for SoA. SoA requires even higher densities to be more efficient for CSF $= 0.5$ than for CSF $= 1.0$ which limits the scope of application for SoA in combination with small CSF.

All benchmarks assume a sufficiently homogeneous distribution of particles. If the distribution is extremely heterogeneous, it is difficult to find an optimal CSF since the optimal CSF varies for different regions inside the domain. Here, adaptive approaches should be considered.

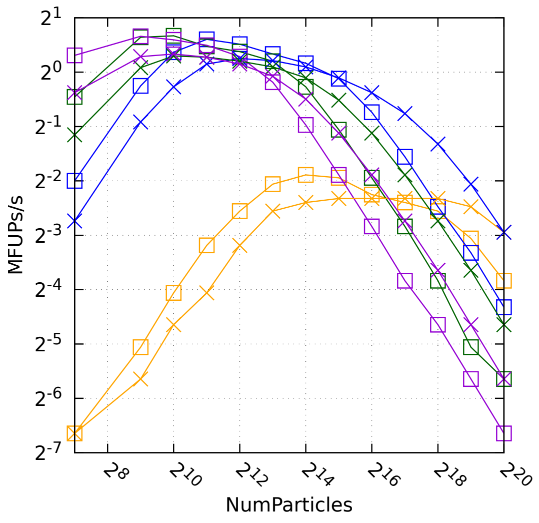
The results show that the peak performance can be optimized depending on the density of the domain. This can be exploited by tuning mechanisms to pick an optimal CSF. Implementations for auto tuning should be aware of the discontinuities shown in Figure 5.1.



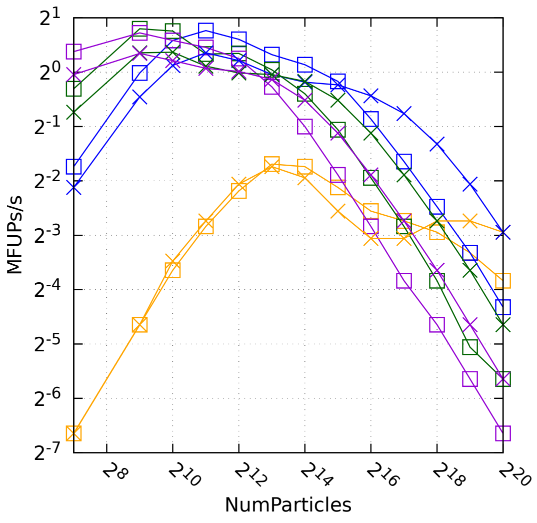
(a) C01



(b) C04



(c) C08



(d) C18

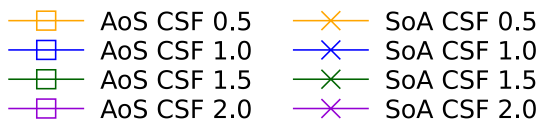


Figure 5.4.: Performance impact of different CSF in a simulation with uniformly distributed particles in a domain of size $20 \times 20 \times 20$ and cut-off radius 1.0. All traversals show that smaller CSF are beneficial for high densities whereas higher CSF provide significant performance improvements for sparse domains.
All benchmarks are executed without Newton 3 optimization.
(benchmark system: CoolMUC-2)

6. Combining SoA Buffers

6.1. Principle

In AutoPas, the calculation of functors using data layout SoA is done in three steps:

1. **Loading:** The data attributes needed for calculation are transformed into the SoA structure. This is usually done for each cell, so that each cell has its own SoA buffer.
2. **Evaluation:** The functor is applied to the previously build SoA buffers.
3. **Extraction:** The data is transformed back into the normal storage layout.

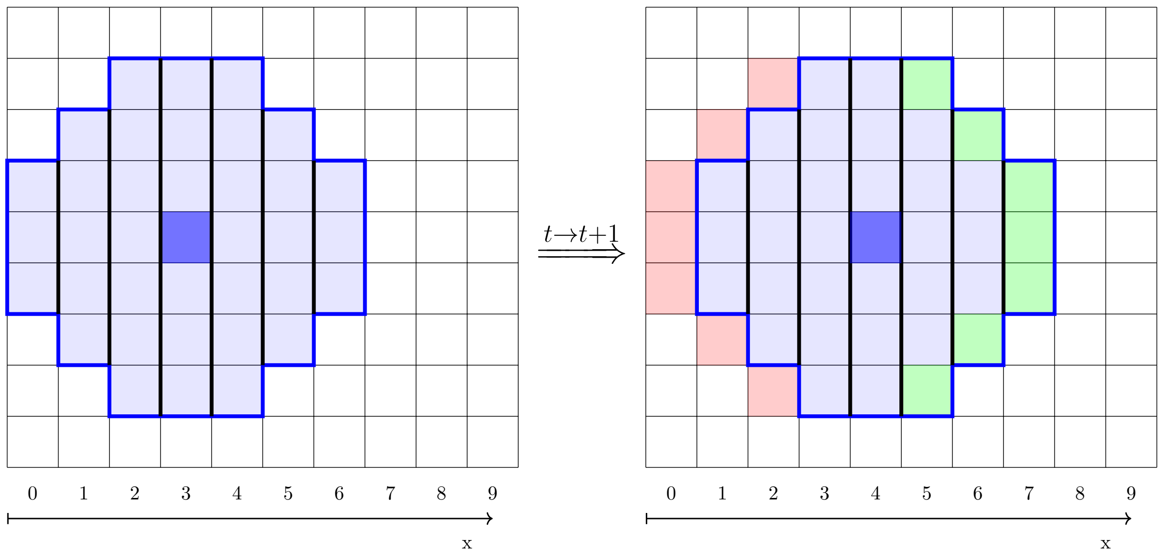
The extensive copy operations during the loading and extraction phase require excellent performance of the evaluation phase. The evaluation phase consists of the actual traversal and the execution of the functor. The best performance can be achieved when the number of successive SIMD operations during the functor calculation is as high as possible. Since SIMD operations consume more energy and therefore produce more heat than scalar operations, the CPU internal clock frequency calculator lowers the frequency automatically. This effect varies for different SIMD operations and gets increasingly worse for bigger vector registers. The Intel manual states that Xeon Skylake processors run about 11-13% slower for AVX2 and even 26-28% for AVX-512[1]. This slowdown is compensated by the saved clock cycles, but since the frequency adaption is delayed, it causes scalar operations which directly follow SIMD operations to be run ineffectively.

The number of successive SIMD operations correlates with the number of particles in the SoA buffers. Regarding the cell-wise creation of SoA buffers, the performance increases if more particles are stored inside one cell. To increase the number of particles inside a single cell the cell sizes could be increased. However, this causes a lot of unnecessary distance calculations, as shown in Chapter 5. Another approach is to form SoA buffers which represents a combination of SoA buffers from multiple surrounding cells just before the evaluation of the functor. The major problem of this approach is the huge number of copies. To keep the number of copies as low as possible, it is important which neighboring cells form a new buffer and how this buffer evolves during the traversal. We will discuss this in detail for the C01 and C08 base step.

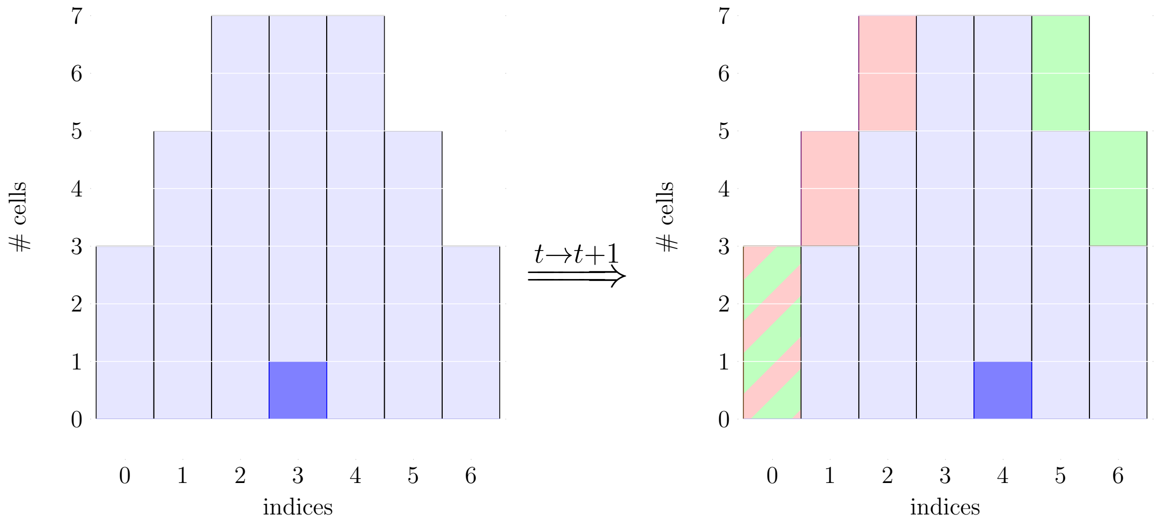
6.2. C01

The simple symmetric structure of C01 is optimal for combined SoA buffers. As C01 does not allow N3 optimization, it is possible to create combined SoA buffers which are only used to read the data of the interacting cells.

[1] `https://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/xeon-scalable-spec-update.pdf`



(a) Interactions



(b) Circular Buffer

⬜ interacting cell 🟦 base cell 🟩 insertions 🟥 deletions ┃ ┃ slices

Figure 6.1.: Evolution $t \rightarrow t+1$ of interactions (top) and circular buffer (bottom) for C01 with combined SoA buffers

Since the values in the combined buffers are never modified, it isn't necessary to write information back to the cells other than the base cell. The normal C01 traversals allows an arbitrary evaluation order of cells inside the domain. Although this would still be possible with combined SoA buffers, we only discuss the case of successive evaluation of cells along one axis. Otherwise it would be necessary to store much more data which negatively influences the performance. The algorithm works with each axis. However, if the domain has not the form of a cube and the particles are homogeneously distributed, it is reasonable to use the longest axis, since this reduces the total number initialization operations. For further description, we use the x-axis. With the previously mentioned evaluation order we have that each evaluated cell is either a neighbor of the previously evaluated cell or it is the first evaluated cell on the chosen axis.

The sphere of interacting cells is divided into slices along the y-axis, as seen in Figure 6.1a. Each slice represents a combined SoA buffer. We assume that the creation of combined SoA buffers starts at the first evaluated cell of an axis. This assumption is only possible because arbitrary execution orders are not allowed. Otherwise, the execution could start somewhere on the axis. Combined SoA buffers are based on reuse of existing data, but if the starting point is non-deterministic it is unknown whether there is already data to be reused or if the buffers must be initialized. Since there is no previously evaluated cell on the beginning of the axis, the whole buffer is initially filled by copying the data from the cells. The combined SoA buffers are stored in a circular/ring buffer, shown in Figure 6.1b. To keep track of the position of the first slice inside the circular buffer, an additional variable for the start index is necessary which is initialized to zero. Inside the combined SoA buffer, the particles are ordered according their insertion. Note, that the base cell (dark blue) always represents the first cell in the slice buffer.

Now, all interactions can be calculated by iterating over all slices and compute the interactions with the current base cell. Since information in the buffers is not persistent, it is important to use the SoA buffer of the base cell and not the copy inside of the combined buffer. If the current base cell interacts with the slice which contains a copy of the base cell, it is necessary to exclude the copy from the calculations. Otherwise, particles would interact with themselves. It is not possible to remove the copy from the buffer slice, since the current base cell is an interacting cell of the next base cell. Therefore, the SoA data structure provides functionalities to define a custom view on the underlying data structure. Since the copy of the current base cell is the first cell in the buffer slice, it is possible to set the start of the view to the first particle after the base cell.

In the next step, the sphere of interactions moves one cell further. Figure 6.1 shows that most of the new and old interaction cells are the same. To reduce the number of copies, we'll keep the combined SoA buffers from the previous step and only apply an update to them. Here, the symmetry of interactions can be exploited. Each combined SoA buffer is a LIFO (Last In, First Out) buffer, meaning that they are demolished in the reverse order of their construction. This effect can be seen as well in the circular buffer. The former leftmost slice goes completely out of scope and can be deleted from the circular buffer. The position of this slice inside the circular buffer is represented by the start index inside the buffer. At the same time, a new SoA buffer is created to represent the rightmost slice, which has just entered the IS. This slice is written on the same index inside the SoA buffer as the former leftmost slice. Since the first slice has moved inside the circular buffer, the start index is incremented. Due to the circular characteristics, an additional modulo operation is applied

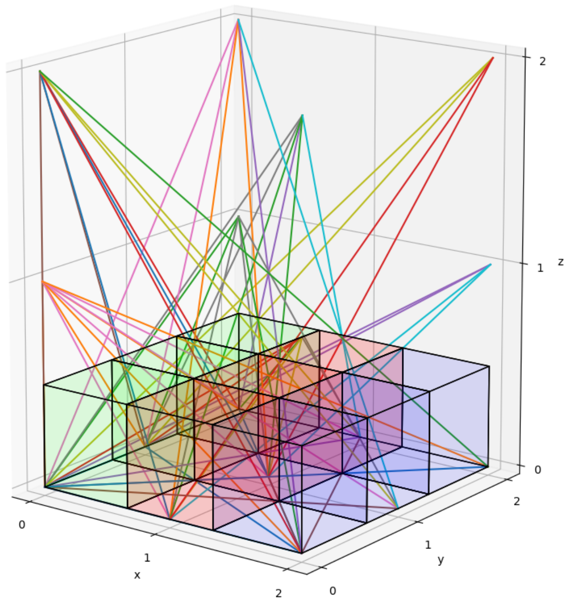


Figure 6.2.: The graphic shows the position and orientation of stripes in the base plate of the C08 base step (overlap = 2).

to the start index to jump back to the first slice in the ring buffer if the end is reached. At this point, the buffer is fully updated and all interactions can be evaluated. This procedure is repeated until the end of the axis is reached.

Since the offsets of the interacting cells relative to the base cell do not change during the traversal, they can be computed beforehand. All offsets are stored in a 2D-array where the first dimension represents the individual slices of the IS and the second dimension represents the cell offsets inside the slice. The cell offsets are sorted to resemble the order of growth/destruction of the combined SoA buffers. This is important since the initialized buffer must show the same behavior as a buffer which was updated multiple times.

The described algorithm has no dependencies outside of the current evaluated cell and can be extended to use multiple threads. Each thread calculates its own line / row of cells and therefore requires its own circular buffer and the corresponding start index inside. Both values are often accessed and modified, making it important to avoid false sharing.

6.3. C04SoA

The combination of SoA buffers is also possible for asymmetric base steps, like C08. As for the C01 traversal, it is again reasonable to use successive evaluation of cells along one axis. This reduces the number of colors to 4, hence the name C04SoA traversal.

The traversal scheme of C08 shown in Figure 5.3 shows that the base plate (z = 0) of the base step is most suited for combination. We are going to apply the same principle as for the C01 base step and cut the base plate into slices which look like stripes, see Figure 6.2. In contrast to C01, these stripes have the same size. This simplifies the update process, since it is only necessary to delete the expired stripe and add the new one. Independent of

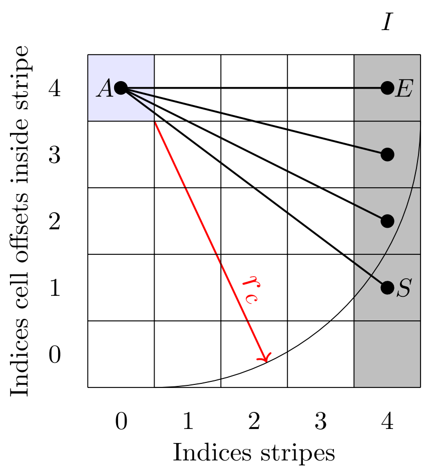


Figure 6.3.: Representation of the base offsets 2D array. Each cell contains an offset relative to a base cell. In the example the cell A interacts with stripe I. Due to the cut-off radius, only the cells between S and E are needed.

the usage of N3 optimization, the values inside the combined SoA buffers must be written back into the cells. This adds an additional copy.

All offsets inside the base plate are stored in a two-dimensional array. The first dimension represents the individual stripes and the second dimension represents the cell offsets inside the stripe. This data structure is used for the update process which is analogous to C01.

In case of the C01 traversal with combined SoA buffers, a base cell always interacts with the whole combined slice (except for the base cell). This is not always possible for C04SoA since the combined stripes are not approximating the IS. To still use only the particles from cells within the cut-off radius, it is necessary to define intervals of cells inside the stripes. Again, a two-dimensional array, *offsets*, is used. The first dimension represents the stripes used in base offset and the second dimension represents pairs of a cell offset and an interval of indices in a stripe of the base offsets data structure. The correct stripe index can be found by using the same index in the base offsets data structure as in *offsets*.

We want to illustrate this with the example shown in Figure 6.3. Consider a cell with relative offset A. This cell interacts with multiple stripes, but for a moment we just pick one of these stripes. This stripe is represented by the index I (here 4) in base offsets. Inside this stripe, A interacts with multiple cells. Due to the form of the IS, these cells must occur in a successive order. The first interaction cell in the stripe has the index S (start, here 1) and the last the index E (end, here 4). Then, a pair of the form $[A, [S, E]]$ is added to the stripe with index I in *offsets*. Note that the same index is used as in the base offset data structure. This procedure is repeated for all other stripes, with those A interacts.

At the beginning of a new line, all stripes need to be initialized. At the same time, we maintain another data structure *OffsetsInBuffers* which stores the index of the first particle of each cell inside the stripes. The last value of each array is the number of particles in the corresponding stripe. This data structure is also created as two-dimensional array. Next, the computation of interactions starts. Therefore, the traversal iterates over the *offsets* data structure and over each array of pairs. In general, the cell represented by the first element in a pair is the first interaction cell. Further, an appropriate view must be set on the current stripe, to select only the particles from cells within the cut-off radius. The view start and end can be retrieved by using the indices S and E of the interval and look up the corresponding positions inside the stripe in the *OffsetsInBuffers* data structure.

There are two special cases for which the first element in a pair represents a cell which is stored in the combined buffers.

The first special case is given for offset zero. This cell represents the current base cell and interacts with the cells inside of their own buffer stripe as well as with ranges of other stripes.

The second special case is given by the last cell of the stripe which contains the base cell (first stripe of base offset). If this cell is given as offset, all possible ranges must be located in other stripes since the interactions with the current stripe are already covered by the first special case.

Both special cases are resolved by creating a view on the first stripe which represents exactly the given cell and letting the particle interact with the particle given by the interval.

6.4. Evaluation

To compare the impact of different sizes of vector registers, we tested combined SoA buffers on CoolMUC2 and on SuperMUC-NG (single threaded, no N3 optimization).

The benchmark reveals unexpected results. The combination of SoA buffers is especially useful when the domain is dense. Sparse domains cause a significant slowdown. This effect arises from the previously mentioned frequency regulation. Even with combined SoA buffers, the number of successive SIMD operations is relatively small when the domain is sparse and the management overhead for combination dominates.

Although the vector registers of SuperMUC-NG are twice as large, the results do not differ significantly in comparison to CoolMUC-2. AVX-512 requires a lot of data to be more efficient and it is likely that there is still not enough data to leverage the performance of AVX-512. Furthermore, simulations are usually memory bound which limits the achievable performance.

In case of multiple threads, it is possible that the workload balancing is not as optimal as for uncombined SoA buffers since combined SoA buffers require to compute a whole line with a single thread. This is only relevant if the particles are aligned along those lines, but for real world examples this is not the case.

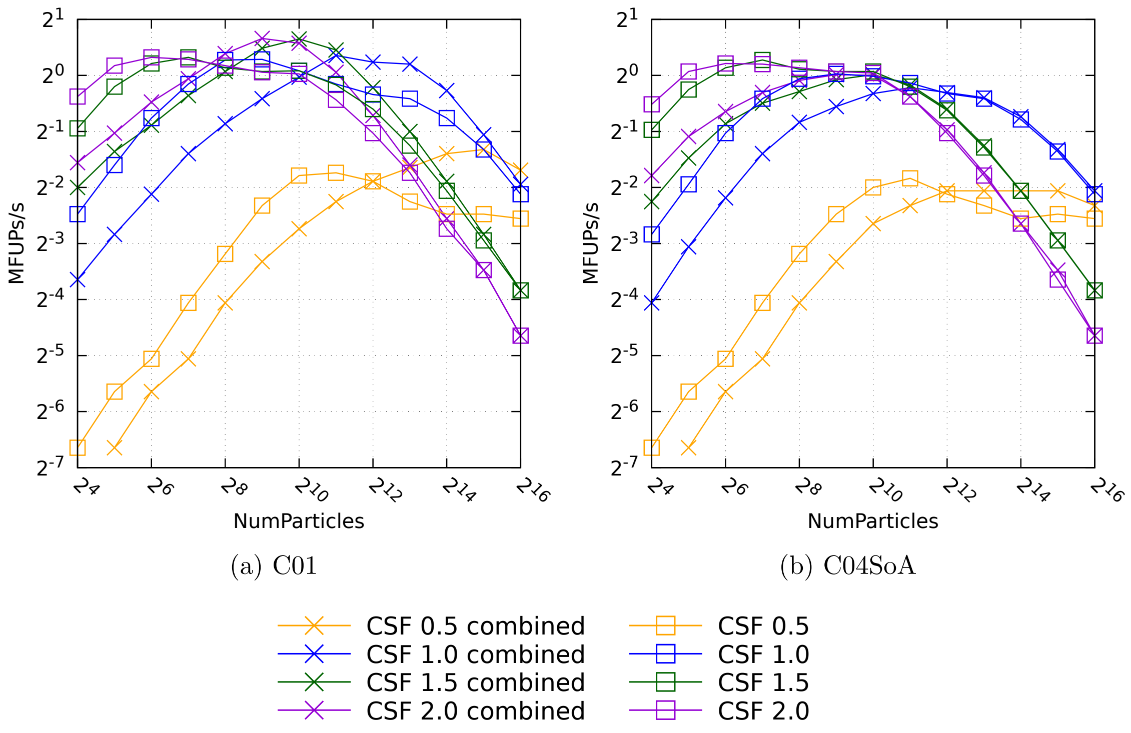


Figure 6.4.: Benchmark of C01 and C04SoA with combined SoA buffers. For all CSF the effect of relatively large overhead for small numbers of particles can be seen. For higher numbers, the overhead is vanished by the gained speed up during calculation.
(benchmark parameters: CoolMUC-2, domain size = $10 \times 10 \times 10$, cut-off = 1.0, no N3 optimization, uniform distribution of particles)

7. Sorting

In Chapter 5 we have seen that the HR can be optimized by varying the cell size. Small cell sizes lead to higher HRs since the search space can be reduced. However, if a cell is partly within the cut-off radius, all particles inside the cell need to be checked. We now want to reduce the cell internal search space using sorting.

7.1. Implementation

Gonnet proposes to sort particles inside of the cells to reduce the number of unnecessary distance calculations [Gon07]. In case of cell pairs, the particles are projected onto the line which connects both cell centers. This projection is done using the dot product of the particle position x_i and the normalized vector r (see Figure 7.1).

$$p_i = x_i \cdot r, \quad with \ ||r|| = 1.0 \tag{7.1}$$

To avoid recurring calculations of r, it is calculated and stored with the cell offsets.

For larger CSF values, it is reasonable to sort single cells, too. The particles are sorted along the projected positions p_i. Before the distance of two particles is calculated, it is checked whether the difference of projected positions is smaller than the cut-off radius. If the difference of projected positions is larger than the cut-off radius, the real distance must be larger as well and the distance calculation can be avoided.

Sorting can be realized in two different ways. First, it is possible to directly sort the particles inside their cells. On the one hand, the sorting procedure takes longer because a lot of data needs to be moved. On the other hand, evaluation during the traversal is slightly faster because successively evaluated particles are located close to each other which is advantageous for prefetching and cache efficiency. The second approach is a sorted view.

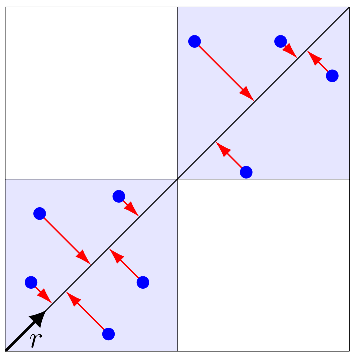


Figure 7.1.: Projection of particle positions onto the line connecting cell centers. The line is mathematically described by the normalized vector r.

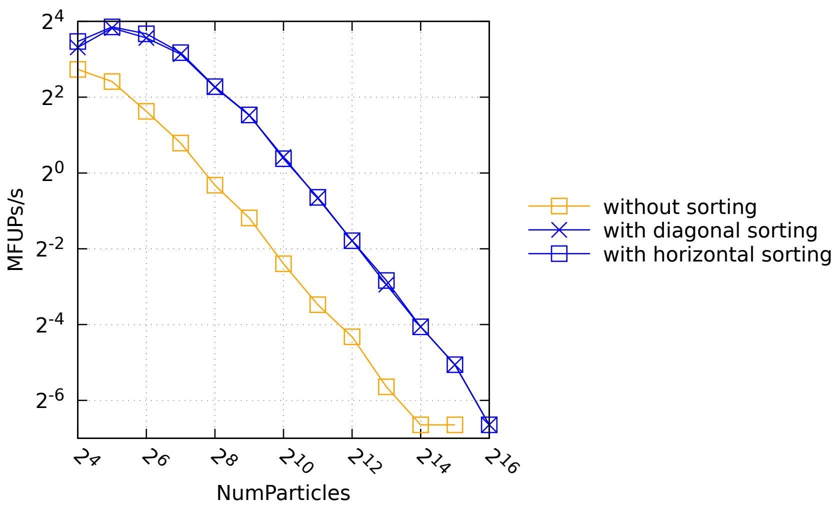


Figure 7.2.: Benchmark Direct Sum with AoS and no N3 optimization. The sorted version is on average ≈ 7 times compared to the unsorted version. It is also clear that it makes no difference if the cell is sorted horizontally or diagonally. (benchmark parameters: CoolMUC-2, domain size $= 10 \times 10 \times 10$, cut-off $= 1.0$, no N3 optimization, uniform distribution of particles)

A sorted view maintains a sorted data structure which contains pairs of projected positions (1D) and pointers to particles. This data structure can be sorted quickly, but provides a potentially slow pointer indirection. We have chosen the second approach since it does not modify the underlying particle data structure. The first approach would require slow locking operations since multiple threads cannot work with the same cell simultaneously.

The efficiency of sorted cells depends on the number of particles inside. For small numbers, the sorting overhead subsumes the benefit. Due to heterogeneous distributions, it is reasonable to decide for each cell/cell-pair whether sorting is likely to be beneficial or not.

7.2. Evaluation

The performance of sorting correlates with the cell size. Therefore, the biggest performance gain can be expected for DS. Figure 7.2 shows that sorting DS is always beneficial. The average speedup is 700%. In theory, the speed up should correlate with the reduction of the search space. DS without sorting search in the whole domain of size $X \times Y \times Z$.

The sorted version sorts the particles along the x-axis which result in a search space of size $(2 \cdot r_c) \times Y \times Z$. In the measurements, the domain has a size of $10 \times 10 \times 10$ and a cut-off radius equal to 1.0. Thus, the sorted version should be $\frac{10 \cdot 10 \cdot 10}{(2 \cdot 1.0) \cdot 10 \cdot 10} = 5 = 500\%$ times faster. In reality the speed-up is even faster.

In case of cubic cells, Figure 7.2 clearly shows that it makes no difference whether the cells are sorted horizontally or along the diagonal. If the cells have a different shape, the chosen axis becomes essential. It is reasonable to choose the axis of the longest cell length.

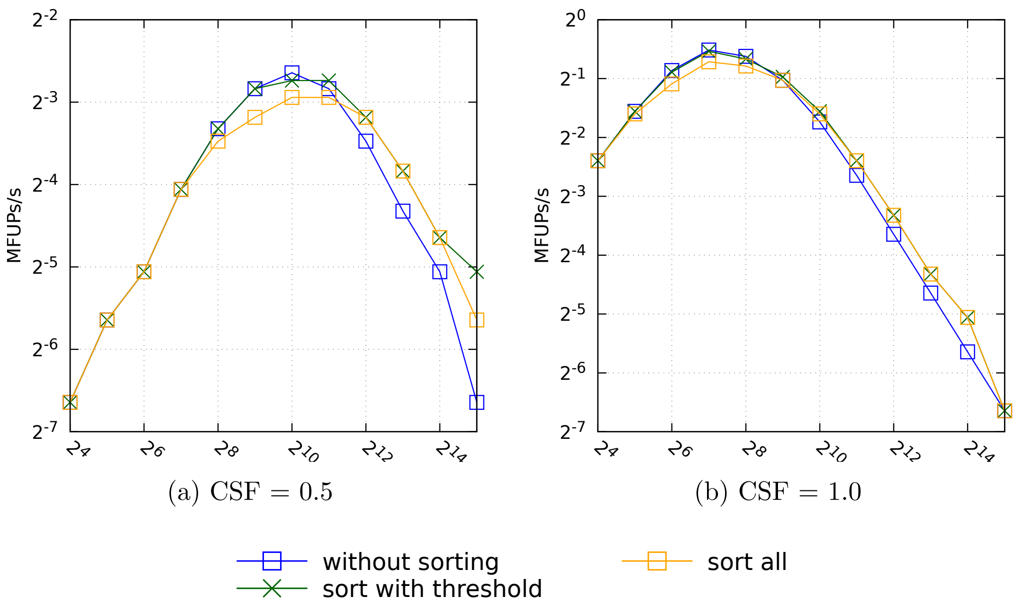


Figure 7.3.: Benchmark C08 for a heterogeneous domain: For a small number of particles there is no notable difference between sorting and normal LC. For more particles, no sorting achieves a higher performance, since the sorting overhead is large in comparison. For more than 2^9 particles, sorting always achieves the best results. The version with a threshold follows the optimal strategy or surpasses them. (benchmark parameters: CoolMUC-2, domain size = $10 \times 10 \times 10$, cut-off = 1.0, no N3 optimization, Gaussian distribution of particles with standard derivation 1.0)

This differentiates the particle position to the highest degree and allows the majority particles to be excluded from the distance check.

We now want to look at the classical LC algorithm. Sorting is a cell internal optimization, making the relative performance improvement independent from the traversal base step.

The additional overhead introduced by sorting is only beneficial if enough particles are present. If the domains are homogeneously filled with particles, it is possible to decide globally whether sorting should be used. Unfortunately, real experiments often show heterogeneous behavior. This is accommodated by enabling sorting for each cell or cell-pair depending on the number of particles inside. Here, it is crucial to find an optimal threshold to start sorting. We simulated an heterogeneous domain by distributing the particles according to a Gaussian distribution with a standard deviation of one. Figure 7.3 shows the results for the C08 base step without sorting, sorting all cells with more than 8 particles and for sorting all cells. For both CSF values, the version with threshold follows the curve with the highest performance or even outperforms them.

The results match the results achieved by Gonnet [Gon07], but only if enough particles are present.

All in all, DS should always use sorting and LC provide the best performance with threshold based sorting.

8. Adaptive Linked-Cell algorithm

In Chapter 5 we have seen that the optimal CSF depends on the density of the domain. This is problematic if the domain is not homogeneous. To achieve best performances for these scenarios as well, the domain can be partitioned into cells of different sizes where each cell approximately contains the optimal number of particles.

8.1. Data structure

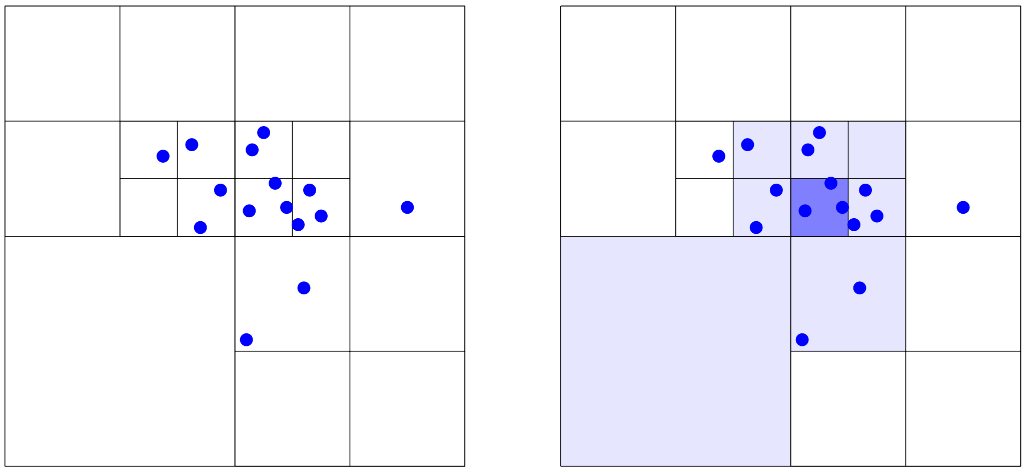
Many different data structures exist to adaptively partition a domain. Common data structures are octrees and k-d trees. Octrees use a fractal like partitioning. The domain is initially divided into a Cartesian grid of 8 blocks of equal size. If further granularity is needed, each of these blocks can be again recursively divided into 8 blocks. The 2D version of an octree is called quadtree. The resulting cell structure for an quadtree is shown in Figure 8.1a. Since it is difficult to use octrees, with cell sizes smaller as the cut-off radius it is reasonable to limit the refinement. This has the advantage that all cells which would exist for max. refinement could be created at program start. All cells which are not used, because the refinement procedure hasn't used them, are empty. Having all cells of the octree in one continuous array allows the reuse of many methods of the normal LC algorithm.

8.2. Traversal

The structure of the octree makes it difficult to implement parallel traversals, especially traversals using N3 optimization. A fixed pattern as shown for C18 in Figure 2.2 is not directly applicable since it might happen that the granularity of the neighboring cells is not fine enough. Therefore, the C01 base step is most suited. In contrast to the C01 base step described in previous chapters, it is not possible to use fixed cell offsets to find neighboring cells. Each cell must explicitly store its neighbors. These lists must be updated when the partitioning of the octree changes. One way to find all neighbors is the algorithm[1] described by David Geier which is based on [Sam89]. This algorithm starts at the current cell and searches first for neighbors of greater or equal size. If a neighbor cell shows a higher granularity, the found cells must be further refined. This is done in a second phase, where only the cells which are adjacent to the current cell are selected. The result of the neighbor search can be seen in Figure 8.1b. It is possible to store the address of the neighbor node or to store the index position inside the cell array which represents the neighbor cell.

During the traversal, each cell is chosen once as base cell and performs the interactions with their neighbors. This can be optimized by using sorted cells, discussed in Chapter 7. The normalized vectors, which indicate the relative positions of the neighbor cell to the current cell, can be stored along with the neighbor node/index.

[1] `https://geidav.wordpress.com/2017/12/02/advanced-octrees-4-finding-neighbor-nodes/`


(a) Domain partitioning

(b) Neighborhood (light blue) around base cell (dark blue)

Figure 8.1.: Structure and neighborhood relation of a quadtree

In contrast to thread based parallelism used for the normal LC, task based parallelism is better suited for octree traversals. Tasks are recursively created based on the tree branches. The creation of tasks is stopped if the number of particles in the children of the current node is smaller than a threshold value. This allows a good load balancing. The creation of tasks can be done using OpenMP taskloops in combination with an if-clause which limits the number of created tasks depending on the node height. The scheduler of the task environment decides which thread works on which tasks. It would be difficult to use threads directly because of different workloads for different tree branches.

8.3. Evaluation

Adaptive LC should only be used if the domain is heterogeneous. Otherwise, the additional complexity for tree construction and neighbor search does not pay of. For heterogeneous domains it is difficult to predict in advance how the octree will perform. Here, it depends on the number and size of formed particle clusters. When clusters can be split into multiple cells, while other areas which contain only a few particles are untouched, high speed-ups are achievable (10 % and more).

The major disadvantage of using octrees as data structure is their fixed layout. The number of cells at each edge must be a power of 2. In the worst case, the minimum cell size is almost twice as big as the cut-off radius. In theory, this issue can be resolved by allowing CSF smaller 1.0. Nonetheless, this adds a lot of complexity to the neighbor search.

Another disadvantage is the symmetric partitioning with each refinement. If a lot of particles are located in only one octant, it is necessary to create 8 cells instead of only two.

Octree based LC should only be used in conjunction with autotuning to switch between normal LC and adaptive LC, when the characteristics of the simulation change.

Part III.

Conclusion

8.4. Summary

Four variations of the LC algorithm were shown and evaluated: Variation of the cell size, combination of SoA buffers, sorting of cells and an adaptive approach.

The benefit of varying the cell sizes depends on the density of the domain. Sparsely populated domains profit from larger cells, while dense domains gain more performance for smaller cell sizes.

If cell based SoA buffers are used, then a lot of different small SoA buffers need to be evaluated during the traversal. The combination of these buffers is reasonable for dense domains to achieve a higher number of successive SIMD operations. For low densities the overhead of creating the additional buffers subsumes the gained speedup during the calculation.

We have seen that sorting the cells is extremely beneficial for DS and should be always applied. For LC, the performance gain depends on the number of particles inside the cells. To accommodate heterogeneous domains, the decision if sorting is useful or not is taken for each cell individually.

For heterogeneous domains, adaptive approaches should be considered. We have seen that chosen data structure and the characteristics of the simulated experiment are critical for performance.

8.5. Future Work

This thesis only discussed cubic cells, but other cell shapes like tetrahedrons are also possible. The cell shape determines the size of the search space and the number of neighbor cells, but also the storage complexity and the neighbor search. The impact of different cell shapes should be evaluated.

Many different adaptive space partitioning data structures exist. The discussed limitations of octrees can be avoided by using a more flexible data structure. Therefore, especially those data structures which allow a flexible partitioning without requiring special characteristics of the simulated space should be implemented and compared.

Different studies show that the memory layout and thread-based partitioning can be optimized using spacing filling curves. [WG11] already mentioned an improvement using this memory layout. Space-filling curves can be used to optimize the normal LC algorithm, but also the adaptive version.

Optimal workload balance between threads is difficult and often threads idle while waiting for another thread. This problem arises because the execution of a thread cannot start with the next time step as long as the whole domain is unfinished. This limitation disappears when threads are aware of the areas which are not finished. If these areas are known, threads can already start to compute the next time step for the finished areas. At the moment, it is unclear whether this in-time parallelism boosts the performance or provides too much overhead.

Part IV.

Appendix

A. AutoPas Software Architecture

To clarify the circumstances of implementation we will briefly discuss the basic software architecture of AutoPas.

AutoPas is designed to be as generic as possible. There are four basic object types which are directly relevant for simulations:

1. Particles
2. Containers
3. Traversals
4. Functors

Particles represent the simulated objects. All particle implementations must be a subclass of `ParticleBase`, which contains basic attributes (position, velocity, force, mass, id) necessary for storage and organization inside the library. The attributes' floating point and integer type is adaptable with template arguments, which allows variation of the consumed memory per particle and the floating point precision.
Containers provide the necessary data structures for storing the simulated objects. Currently, different variations of VLs and LCs as well as DS are implemented. Containers distinguish between the simulation domain and a halo which represents the area outside of the domain. The halo is used to temporarily store particles which leave the simulation space. Library users can access those particles to implement boundary conditions. In addition, containers offer region iterators, which allows iteration over particles within a specified region. An AutoPas instance always contains only one container object.
Traversals apply functors to all pairs of objects within a given cut-off radius. Figure A.2 shows all currently available traversals. Some of them are limited to only a specific container, while others can operate on multiple containers. All traversals inherit a common interface from `TraversalInterface`. The subclass `CellPairTraversal` implements basic functionalities for all traversals which operate on cells. Therefore, all traversals for LC and VL which use LC for construction are subclasses of `CellPairTraversal`.
Functors represent the actual applied calculation, e.g. LJ potential. Currently, only short-range interactions are supported.

New functors must inherit from the class `Functor`. `Functor` provides some common functionalities as well as a common interface. If SoA is used as data layout, the storage data layout (currently, only AoS supported) must be transformed into SoA. `Functor` provides implementations of `SoALoader` and `SoAExtractor` which performs the transformation and reverse transformation. To avoid a full copy of all attributes during transformation, the functor implementations must provide information about which values are needed to perform a functor calculation and which values are calculated. Since the SoA data structure is based on tuples, this information must be provided at compile time, because tuple element access can only be realized with template arguments. C++17 allows no direct static polymorphism

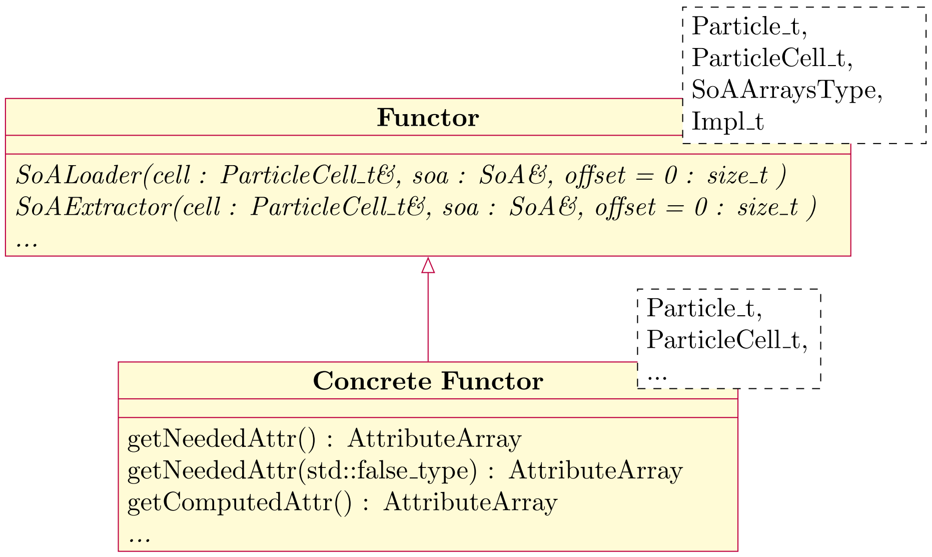

Particle_t,
ParticleCell_t,
SoAArraysType,
Impl_t
Functor
SoALoader(cell : ParticleCell_t&, soa : SoA&, offset = 0 : size_t)
SoAExtractor(cell : ParticleCell_t&, soa : SoA&, offset = 0 : size_t)
...
Particle_t,
ParticleCell_t,
...
Concrete Functor
getNeededAttr() : AttributeArray
getNeededAttr(std::false_type) : AttributeArray
getComputedAttr() : AttributeArray
...

Figure A.1.: Class diagram functor

(virtual constexpr functions[1] are supported with C++20). To be still able to access the information defined in a subclass of `Functor` the curiously recurring template pattern (CRTP) is applied. This idiom integrates the type of the subclass into the type of the super class. `Functor` calls the constexpr methods `getNeededAttr` and `getComputedAttr` to retrieve an array of attributes which must be copied. There are two different `getNeededAttr` methods. The version without argument is used to retrieve all needed attributes when N3 optimization is used whereas the version with argument `std::false_type` provides the necessary attributes for no N3. Usually, the attributes needed for N3 are the union of the attributes needed without N3 and the computed attributes.

The user of the library has to implement at least the functors and the simulated objects. Afterwards, an instance of the `AutoPas` class is created to interact with the library. Direct access of internal data types is not recommended.

[1] `https://en.cppreference.com/w/cpp/language/constexpr`

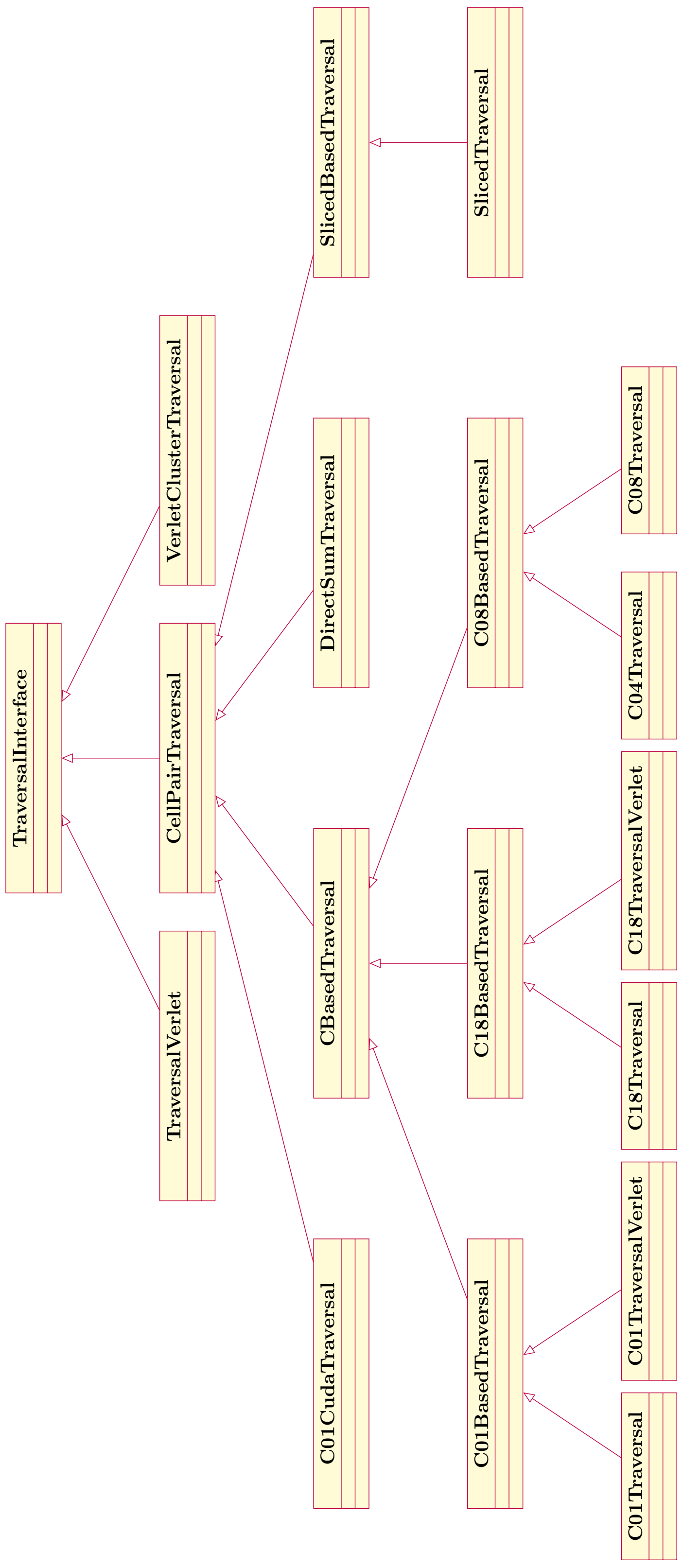


Figure A.2.: Class diagram traversals

B. Bug reports filled

B.1. GCC

- Bug 90855 - OpenMP: collapse clause rejects template argument as parameter[1]

B.2. fmtlib/fmt

- #1184: lgtm alerts[2]

[1]`https://gcc.gnu.org/bugzilla/show_bug.cgi?id=90855`
[2]`https://github.com/fmtlib/fmt/issues/1184`

Abbreviations

AoS Array of Structures

CSF cell size factor

DS Direct Sum

HR hit rate

IS interaction sphere

LC Linked-Cell

LJ Lennard-Jones

MD Molecular dynamics

N3 Newton 3

SIMD Single Instruction, Multiple Data

SoA Structure of Arrays

VL Verlet-Lists

List of Figures

List of Tables

Bibliography

[GKZ07] Michael Griebel, Stephan Knapek, and Gerhard Zumbusch. Numerical simulation in molecular dynamics, vol. 5 of texts in computational science and engineering, 2007.

[Gon07] Pedro Gonnet. A simple algorithm to accelerate the computation of non-bonded interactions in cell-based molecular dynamics simulations. *Journal of Computational Chemistry*, 28(2):570–573, 2007.

[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 (IPDPS)*, 2019.

[ITH$^+$16] Masaki Iwasawa, Ataru Tanikawa, Natsuki Hosono, Keigo Nitadori, Takayuki Muranushi, and Junichiro Makino. Implementation and performance of FDPS: a framework for developing parallel particle simulation codes. *Publications of the Astronomical Society of Japan*, 68(4), 06 2016.

[MR99] William Mattson and Betsy M. Rice. Near-neighbor calculations using a modified cell-linked list method. *Computer Physics Communications*, 119(2):135 – 148, 1999.

[NBB$^+$14] Christoph Niethammer, Stefan Becker, Martin Bernreuther, Martin Buchholz, Wolfgang Eckhardt, Alexander Heinecke, Stephan Werth, Hans-Joachim Bungartz, Colin W. Glass, Hans Hasse, Jadran Vrabec, and Martin Horsch. ls1 mardyn: The massively parallel molecular dynamics code for large systems. *Journal of Chemical Theory and Computation*, 10(10):4455–4464, 2014. PMID: 26588142.

[New87] I. Newton. *Philosophiae naturalis principia mathematica.* J. Societatis Regiae ac Typis J. Streater, 1687.

[Rap04] D.C. Rapaport. *The Art of Molecular Dynamics Simulation.* Cambridge University Press, 2004.

[Sam89] Hanan Samet. Neighbor finding in images represented by octrees. *Computer Vision, Graphics, and Image Processing*, 46(3):367 – 386, 1989.

[Tid97] Bruce Tidor. Molecular dynamics simulations. *Current Biology*, 7(9):R525 – R527, 1997.

[TSH$^+$19] Nikola Tchipev, Steffen Seckler, Matthias Heinen, Jadran Vrabec, Fabio Gratl, Martin Horsch, Martin Bernreuther, Colin W Glass, Christoph Niethammer, Nicolay Hammer, Bernd Krischok, Michael Resch, Dieter Kranzlmüller, Hans Hasse, Hans-Joachim Bungartz, and Philipp Neumann. Twetris: Twenty trillion-atom simulation. *The International Journal of High Performance Computing Applications*, Jan 2019.

[Ver67] Loup Verlet. Computer "experiments" on classical fluids. i. thermodynamical properties of lennard-jones molecules. *Phys. Rev.*, 159:98–103, Jul 1967.

[WG11] Ulrich Welling and Guido Germano. Efficiency of linked cell algorithms. *Computer Physics Communications*, 182(3):611–615, 2011.