DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Implementing the Linked Cell Algorithm in AutoPas using References

Marco Papula

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Implementing the Linked Cell Algorithm in AutoPas using References

Implementierung des Linked Cell Algorithmus in AutoPas mit Hilfe von Referenzen

Author:	Marco Papula
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisor:	M.Sc. Fabio Alexander Gratl, M.Sc. Steffen Seckler
Submission Date:	15.09.2020

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

Munich, 15.09.2020 Marco Papula

Abstract

AutoPas is an open-source library implemented in C++ that provides a variety of Molecular Dynamics (MD) algorithms optimized for most modern hardware. The library supports auto-tuning, meaning the library will periodically evaluate the whole domain and if another algorithm is expected to increase performance, switch containers accordingly. This process currently involves copying every particle in the domain into a new container, cancelling out some if not all of the performance gained from switching containers. This thesis describes a proof of concept implementation of the Linked Cells container, whose cells operate on references with the actual particles being stored in another structure. The thesis demonstrates the correctness of the implementation and gives a rough estimate of its performance.

Kurzfassung

AutoPas ist eine in C++ implementierte Open-Source-Bibliothek, die eine Vielzahl von MD-Algorithmen bereitstellt, die für die modernste Hardware optimiert sind. Die Bibliothek unterstützt Auto-Tuning, d.h. die Bibliothek wertet periodisch die gesamte Domäne aus und wechselt Container entsprechend, wenn ein anderer Algorithmus die Leistung steigert. Dieser Prozess umfasst derzeit das Kopieren jedes Partikels in der Domäne in einen neuen Container, wodurch ein Teil, wenn nicht sogar die gesamte durch den Containerwechsel gewonnene Leistung zunichte gemacht wird. Diese Arbeit beschreibt eine Proof-of-Concept-Implementierung des Linked Cells Containers, dessen Zellen auf Referenzen arbeiten, wobei die eigentlichen Partikel in einer anderen Struktur gespeichert werden. Die These demonstriert weiters die Korrektheit der Implementierung und gibt eine grobe Abschätzung ihrer Leistung.

Contents

1 Introduction

1.1 Motivation

Molecular Dynamics (MD) simulations, giving insight into the behaviour of particles and molecules in atomic detail at very fine-grained time steps, has consistently been gaining more and more attention over the years as shown in Figure 1.1. Responsible for this tremendous rise in popularity are significant improvements in performance, accuracy and accessibility in both the algorithms themselves as well as hardware and software combined with the explosion of experimental structural data. [1]

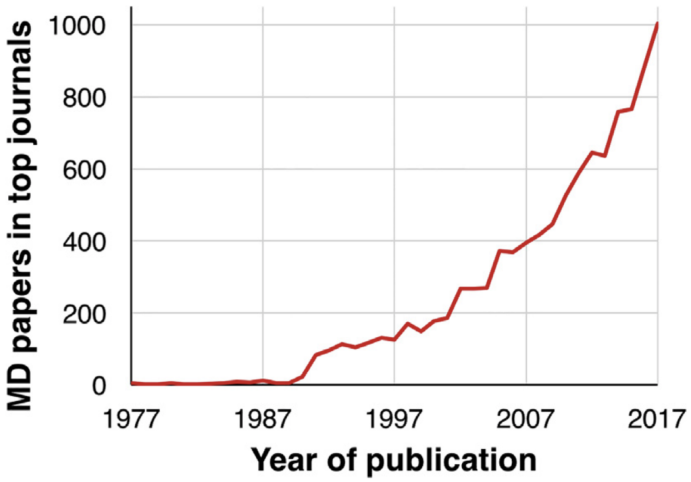


Figure 1.1: The number of publications per year that include the term molecular dynamics in either the title, abstract, or keywords for the top 250 journals selected by impact factor taken from [1].

MD simulations influence experimental work in several ways. For example, in ligand and protein design, there is a vast number of candidates that need to be evaluated. MD simulations are used as a faster and more cost-effective although less accurate filter, leaving only candidates, that have shown promise in the simulation, to be tested in an actual physical experiment. [1]

More commonly, MD simulations are used to gain a deeper understanding of the physical system, e.g. a biomolecule or a new drug. While it's often not possible to give the same insight using experiments, they can be used to verify a Hypothesis, generated from the simulation, by verifying certain aspects or predictions. [1]

1.2 Goal

The goal is to provide an alternative implementation of the linked cells, in literature also referred to as linked cell list, algorithm operating only on references but giving a similar performance, in order to gain a speedup when switching from one AutoPas container to another during auto-tuning. Currently, when the AutoPas library wants to switch from one algorithm to another, every particle has to be copied over into the new data structure the new algorithm operates on, resulting in a lot of overhead. If all algorithms provided an implementation operating only on references instead of storing the actual particles, this step would be reduced to simply rearranging the references. Sorting the particles inside the common structure would give additional performance gains. This thesis is supposed to act as a proof of concept, that similar performance can be reached with this approach. The same concept of operating on references and storing the actual particles in another data structure could then also be applied to other algorithms, greatly increasing the performance of the auto-tuning feature provided by the AutoPas library.

2 Theoretical Background

2.1 Computer Memory

2.1.1 Structure

The speed of executing Central Processing Unit (CPU) operations has always exceeded the bandwidth of the memory subsystem it is connected to. This leads to the CPU being idle while waiting for data from the memory to be fetched, thus wasting precious CPU cycles. To minimize this idle time, memory is constructed in layers as shown in Figure 2.1. The top layers are closer to or even part of the CPU and have lower latency and smaller capacity but come at a much greater manufacturing cost per bit than the lower ones. CPU performance is growing at a much faster rate than memory speed, indicating that leveraging cache memory as much as possible will only grow in importance for High Performance Computing (HPC). [2] [3]

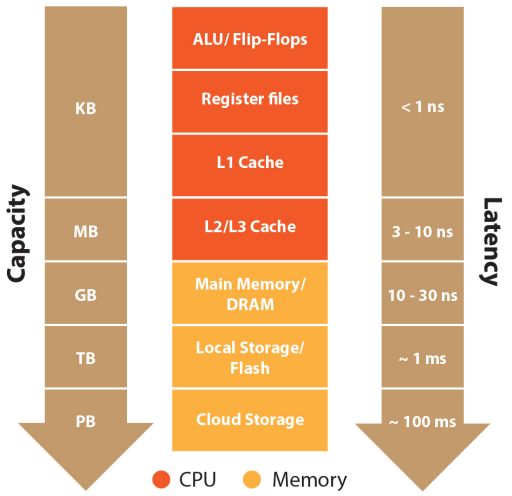


Figure 2.1: Shows the layered nature of computer memory based on [2]. As memory size increases, so does latency [2].

2.1.2 Cache prefetching

Cache prefetching refers to the process of fetching either data or instructions into cache memory before they are needed. For this thesis, data prefetching is of significantly more

importance, so prefetching will henceforth be used synonymously with data prefetching. [4] When assessing the runtime of an algorithm, the big O notation, in literature also often referred to as the Landau Notation, is being used. In the big O notation, memory access always assumes Random-access memory (RAM) access with a cost of one unit of time, with the unit of time being defined as the time it takes the CPU to perform one operation. This makes the comparison of algorithms much simpler because implementation details and hardware-specific optimizations are not taken into account. In reality, memory access latency is hugely dependent on which kind of memory is being accessed, as shown in Figure 2.1. [5]

Taking the example of naive matrix multiplication from [6], the runtime complexity under the assumption that RAM access takes one unit of time, is $O(n^3)$. In a real-world setting, cache misses and address translation difficulties slow the computation down significantly [6]. A plot of time versus problem size from [6] puts the real runtime complexity of the naive matrix multiplication example close to $O(n^5)$.

This shows the tremendous impact, that cache prefetching can have on computational cost. While there are a vast number of prefetching strategies optimized for different hardware, the common goal of those is to leverage regularities in the memory access patterns [6].

2.2 Molecular Dynamics

MD simulations are methods for analyzing the physical interactions of atoms and molecules over a certain period of time. This is achieved by calculating the force exerted on each particle by every other particle in the system. By then utilizing Newton's law of motion to calculate the new position of all particles, the result will represent the state of the system at the next time step. Repeating this process several times will result in a 3D animation of the system in atomic detail, which can be viewed as predicting the trajectories of every particle as a function of time. In a simulation, it is possible to control almost every aspect of the system by setting the initial configuration. A feat much more difficult if not downright impossible to achieve in an actual experiment. [1]

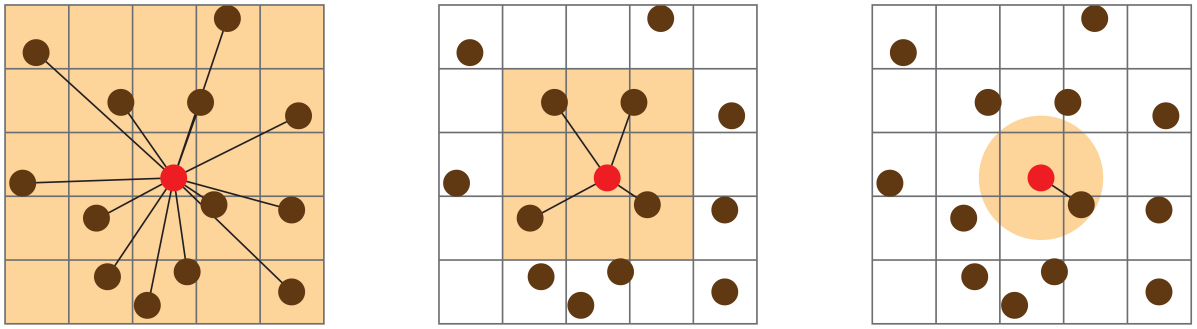
Molecular systems usually contain a vast number of particles, making it close to impossible to determine the properties of the system analytically. MD simulation avoids this problem by using numerical methods for the calculations of pairwise interactions. Long simulations with lots of particles are therefore numerically ill-conditioned, which leads to them generating cumulative errors in numerical integration. Those can not be eliminated entirely, but they can be minimized with proper selection of algorithms and parameters of the model. [7] [8]

2.3 Linked Cells Algorithm

2.3.1 The Algorithm

Computing all pairwise interactions between particles results in a runtime complexity of $O(n^2)$ [5]. The inverse-squared law states, that a specific physical quantity is inversely proportional to the square of the distance. Therefore the force calculation for particles with

increasing distance between them converges to 0 at a fast rate. The linked cells, or also linked cell lists, algorithm uses this property and simply neglects the influence of particles beyond a certain predetermined distance to another particle referred to as the cutoff radius and simply treats the result of those calculations to be zero. The domain is separated into cells of fixed size at least as big as the cutoff radius as shown in Figure 2.2. The cells have to be at least the size of the cutoff radius because otherwise more than merely the neighbouring cells would have to be taken into consideration. Due to the cells having a constant size, and assuming that a reasonable constant was chosen, there is only a limited number of particles that can be contained within each cell. This allows the linked cells algorithm to run in O(n).



(a) The full domain with all particles interacting with each other.

(b) Only the interactions between the particles inside the neighbouring cells and the red one in the center.

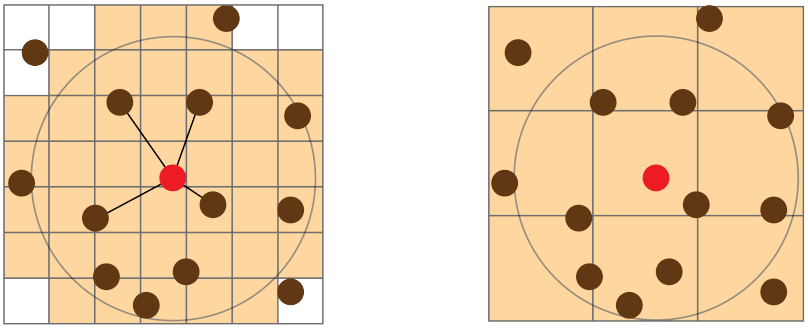
(c) Only interactions of particles inside each others cutoff radius.

Figure 2.2: (a) shows the full domain with the black lines indicating that the calculation for those particles with each other has to be done. (b) shows that by only calculating the interactions between particles inside the cell and it's direct neighbours, the number of computations drops rapidly with (c) showing only the particles within the cutoff radius. The area coloured in brown shows the part of the domain that has to be searched. Everything is relative to the particle marked in red.

2.3.2 Performance

Despite the algorithm scaling linearly with the number of particles, the constant factor neglected in the Landau Notation, still leaves room for improvements. Furthermore, the subsection on Cache prefetching states, that the linear runtime of the algorithm is dependent on the efficient implementation of the algorithm. It is therefore greatly impacted by memory latency, given that MD simulations are expected to have a very large number of particles that have to be stored. Particles belonging to the same or a neighbouring cell will be accessed together. By storing those particles next to each other in memory, ideally within the same cache line, the number of cache misses can be reduced dramatically [6].

Comparing the coloured area in 2.2b and 2.2c, it becomes apparent that a large portion of the neighbouring cells is still neglected because it resides outside the cutoff radius. Several



(a) Displaying the linked cells algorithm with a cell size less than the cutoff radius.

(b) Displaying the linked cells algorithm with a cutoff radius for the particle highlighted in red that does not extend beyond the neighbouring cells.

Figure 2.3: (a) Shows that a larger number of cells have to be searched, but the overall area that is being looked at for this particle is smaller than in the traditional setting with a cell size larger than the cutoff radius shown in (b). The grid cells represent the cells of the algorithm and the grey circle the cutoff radius.

variations of the algorithm have been proposed to target this and other shortcomings [9]. For example, some variations of the algorithm use a cell size smaller than the cutoff radius to try and increase the efficiency of the algorithm by attempting to minimize the area within cells but beyond the cutoff radius as shown in Figure 2.3. This comes at the cost of an increased number of cells that have to be searched, beyond simply the neighbouring ones. [9]

3 Technical Background

3.1 Templates in C++

3.1.1 Introduction

Templates in C++ are a very powerful language feature intended to increase code reusability. Templates exist as either a class or a function template which refers to the scope in which the template parameters can be used in. Templates give the ability to write generic programs. C++ generics have an actual code generator engine running behind them, extending their capabilities far beyond those of generics in other programming languages. For example, Java generics don't accept primitive types, only their wrapped versions such as java.lang.Integer, while C++ templates do. [10] [11]

Templates are especially useful when implementing data structures like linked lists or sorting algorithms with the example of bubble sort shown Figure 3.1 or the linked cell list structure which the linked cells algorithm operates on. The behaviour of the structure is independent of the type the structure will be used with. [10]

```
4
5  template <class T>
6  void bubbleSort(T a[], int n) {
7      for (int i = 0; i < n - 1; i++)
8          for (int j = n - 1; i < j; j--)
9              if (a[j] < a[j - 1])
10                 swap(a[j], a[j - 1]);
11 }
```

Figure 3.1: The bubble sort algorithm implemented in C++ using templates.

3.1.2 Compile-time resolution

Templates are code snippets, that the compiler then instantiates at compile-time via code generation, thus giving no runtime overhead, and since they are a language feature they come with the full support of type checking and scope. The resolution creates a new class or function based on it's templated version for every type the template is instantiated with, both explicitly by specifying the type and implicitly by the type being inferred.

3.1.3 Drawbacks of Templates

Excessive use of templates can result in creating unnecessarily large executables, due to the need to create a new instantiation of the templated class or function for every type the template was specified with. This also results in a significant increase in compile-time. Furthermore, a lot of compilers have historically poor support for templates, resulting in confusing and unhelpful error messages and reduced portability. Given that the resolution of templates happens at compile-time, static analyzers in integrated development environments (IDEs) have very limited options to assist the developer. This and the poor compiler support result in templates being difficult to debug. [10]

3.2 AutoPas

3.2.1 Introduction

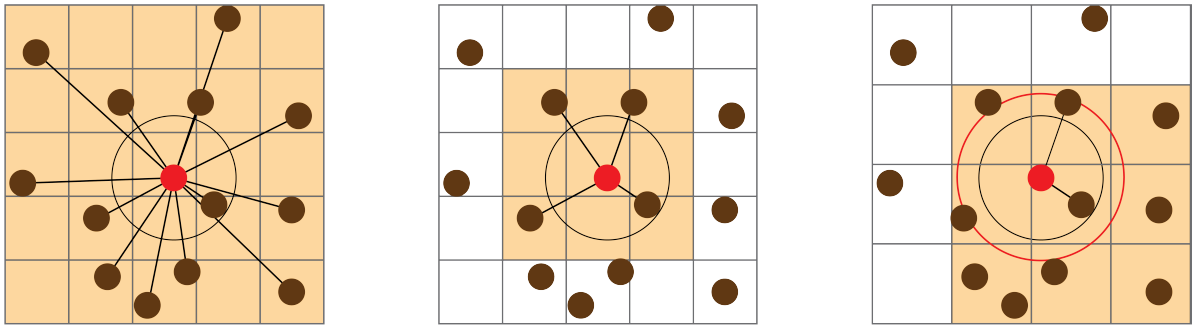
AutoPas [12] is an open-source node-level performance library written in C++ [13]. Given the highly custom nature of Most computer software written for MD simulations, has a highly custom nature, often being only applicable to a very limited set of problems optimized for certain hardware. The same software applied to a different set of problems or new hardware, offering new features, efficiency suffers greatly or a lot of performance optimization options are underutilized. This results in low code reusability and therefore significant overhead for researchers. [13]

AutoPas was built for short-range particle interactions, where particles that are sufficiently far apart from each other can be neglected. The library further uses auto-tuning in order to select the optimal container and algorithm for a specific scenario at runtime to increase performance. [13]

Newton's third law of motion states, that when one body exerts a force onto another second body, the second body simultaneously exerts a force equal in magnitude and opposite in direction on the first body [14]. For the force calculation in MD simulations this gives the following equation: $F_{ij} = -F_{ji}$ where F_{ij} is the force of an object i exerted onto another object j. By always applying F_{ij} to j and $-F_{ji}$ to i, only half of the force calculations have to be performed. This optimization is integrated into the AutoPas library and can be utilized by setting the Newton3 field to true.

3.2.2 AutoPas Containers

To store the particles for iterating over them, AutoPas uses so-called containers. They are classes that implement different algorithms for the pairwise interactions between particles. The idea is to provide iterators over particle pairs, hiding the implementation details of the algorithm. The following section will explain the direct sum, linked cells and the verlet lists containers, Figure 3.2 showing the algorithms implemented. For more details on the other containers or the AutoPas library in general, refer to [13].



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

Figure 3.2: The brown area shows the cells considered by the corresponding algorithm. The black circle shows the cutoff radius and the red one the verlet radius. Comparing (b) and (c), it's also shown that the cells for Verlet Lists are larger than the ones for the Linked Cells. Everything is relative to the particle marked in red.

Direct Sum

Direct Sum is implementing the naive approach of storing all particles in one continuous list and simply calculating the pairwise interactions between every particle pair within the cutoff radius. The algorithm iterates over every possible particle pair, only performing any kind of calculation if the pairwise distance is below the cutoff radius, giving a runtime of $O(n^2)$ shown in 3.2a.

Linked Cells

The Linked Cells container implements the algorithm of the same name shown in 3.2b. The theory behind the algorithm has largely already been covered, with the exception of the choice of cell size. The AutoPas implementation uses a cell size equal to or larger than the cutoff radius. All cells are of the same size, so the cell size is simply determined by a vector containing the domain size divided by the largest natural number, so that the result is larger than the cutoff radius, in every dimension. For a domain size of 6.2 x 5.4 x 2.3 and a cutoff radius of 1.1 the resulting cell size is 1.2 x 1.35 x 1.15.

Verlet Lists

The Linked Cells algorithm searches a cuboid formed by the neighbouring cells which is significantly larger than the sphere formed by the cutoff radius. This results in at least 84% of the area considered, given by the size of the cuboid, which is larger than or equal to r^3, divided by the sphere of the cutoff radius, given by $\frac{3}{4}\pi r$, contains particles that can be neglected.

Verlet Lists, unlike Linked Cells, have every particle store a list of neighbouring particles within a verlet radius, considering only those particles within the sphere formed by said verlet

radius. As particles move, they might fall outside this radius as well as new particles entering, resulting in the need to update those lists every so often. In order for those calculations not to cancel out the performance increased gained by not searching the whole domain, the frequency of those rebuilds has to be kept as small as possible, while still occurring often enough to keep the result accurate. This forces the verlet radius to be larger than the cutoff radius, but still being small enough for its sphere to be smaller in volume than the linked cells cuboid.

To keep the verlet radius as small as possible and make rebuilds more efficient, verlet lists can be used together with the linked cells. This is done by making the cell size at least as large as the verlet radius, reducing the rebuild overhead by only needing to search the neighbouring cells like in the Linked Cells algorithm and still not having to search the larger cuboid. This is shown in 3.2c.

4 Implementation

4.1 Template Refactoring

4.1.1 With Particle Cell Template

Previously to this thesis, the Particle Cell template was treated as a component specified by a user using the AutoPas library. In Figure 4.1, all the major components, which previously contained the ParticleCell template, are coloured in orange. Most containers could operate on all cell types, for which AutoPas only provided the Full Particle Cell as an example. This allowed the user to create their own cell types.

With the introduction of the Reference Linked Cell, the previous assumption, that every container could operate on every type of cell, does no longer hold. The Full Particle Cell expected to receive actual particles, while the new Reference Particle Cell expects them in the form of a pointer and does not take care of storing the particle. This resulted in the need for the overall Particle Cell template structure to be changed.

4.1.2 Without Particle Cell Template

The nature of the Reference Cell, introduced in the next chapter, makes the removal of the template at a high level necessary. The template was removed from all components above the actual container with the small exception of the abstract Functor class.

Figure 4.1 shows two methods below the class highlighted in green that received the template instead of the whole class. The functor only ever needs the cell template for those two function calls with child classes only needing it to pass it to the parent class. Templating the functions instead of the whole class allows for the removal of the template from the class and does not change the behaviour, since the functions receive a cell as a parameter and can infer the template type based on that.

For the AutoPas class, the cell template specification will be handled by the containers themselves. Classes above the Container Selector only ever use the template to pass it to their templated child classes, therefore its removal does not affect the behaviour. The containers themselves will after this thesis, make the decision, which cells they operate on, taking care of the problem, that some cells only operate on references while others need the actual particle.

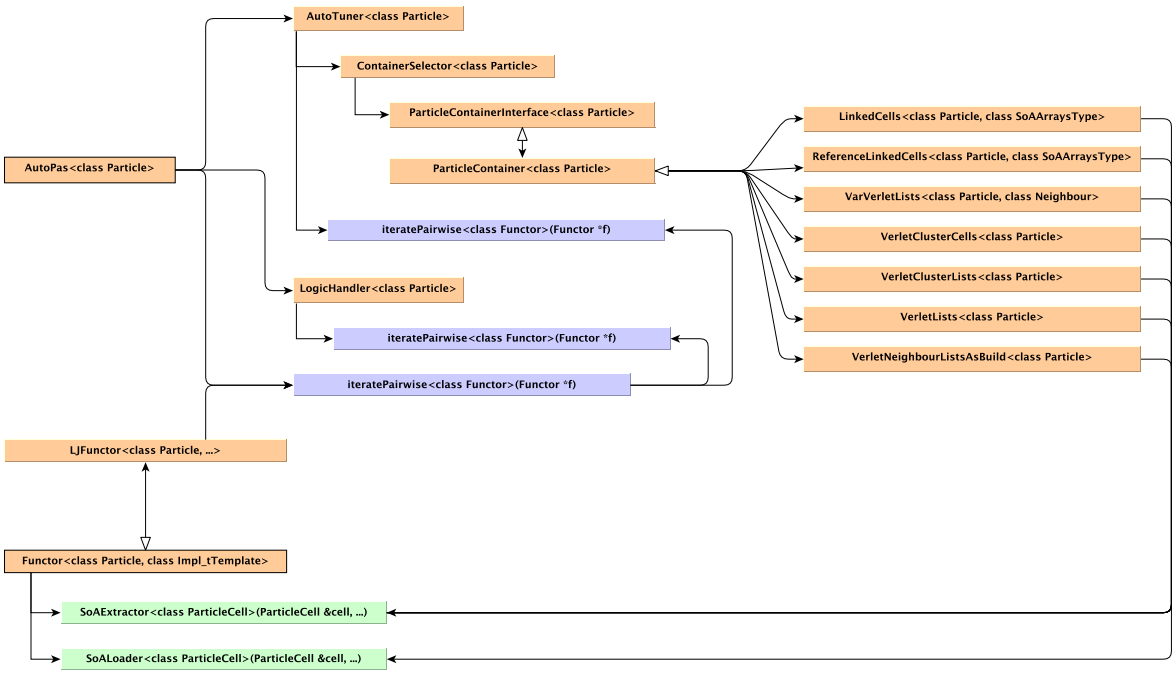


Figure 4.1: From Classes and functions coloured in orange the Particle Cell template was removed. The two functions marked in green got the Particle Cell template added to them and can infer the type based on the cell argument given as a parameter. The iterate Pairwise functions marked in blue are not affected by the template refactoring but show how the Functor template gets added to the library. The white arrows indicate inheritance similar to UML diagrams and the black arrows indicate control flow. The diagram shows a simplified view of the whole AutoPas library, displaying the main parts of the code, where the Particle Cell template was previously present.

4.2 Reference Linked Cell

4.2.1 Particle Vector

To implement the linked cells algorithm by only providing the cells with a reference to the particle, a new data structure is needed to store the actual particles. This data structure will be referred to as the Particle Vector. It has to store all particles and keep track of the validity of the references to those.

In order to leverage cache prefetching, an `std::vector` was chosen to hold the particles internally inside the Particle Vector, because it stores them in memory continuously, thus leveraging cache prefetching, while still being of variable length, because the size can't effectively be specified at compile time. This is represented by the private _particleListImp field in Figure 4.2. A `std::vector` has a certain amount of memory allocated and if an insert operation exceeds this capacity, the structure performs a resize operation, after which the validity of the references is no longer certain. Similarly, once a particle is deleted, particles with a higher index than the deleted one will update their location in memory and get moved, invalidating their references. The Particle Vector provides a method for checking, whether there are particles in the container, which have not yet been added to a cell, as well as a method keeping track of both delete and resize operations in order to specify, whether previously valid particles having become invalidated and the cells have to be cleared and all particles added to their cells again. The container provides two separate methods for this, to prevent rebuilding all cells for every new particle added, even if old references remain intact. It also provides a method to tell the Particle Vector, that the current status quo of references is valid. This is needed because the Particle Vector has no interactions with the actual cells and therefore also does not know when they are rebuilt with the new references. The interactions with the other components as well as the methods and fields discussed above are shown in Figure 4.2.

The Particle Vector also provides an iterator starting at the first dirty particle, which refers to the first particle whose references has been invalidated or that has not yet been added to a cell. This allows for the container to check if there are dirty particles in the Particle Vector, if the cells need to rebuild and then regardless of the cells being rebuilt, iterate over all particles that need to be added to cells.

The Particle Vector only functions as a lookup table for the cells, indicated by the arrow with the dashed line in Figure 4.2. This leads to particles no longer being able to simply be deleted from the cell, but they provide a way to be marked for deletion. The Particle Vector provides a method to remove all those particles marked for deletion. This will almost certainly invalidate some references, which leads to the cells having to be rebuild. This is done in order to prevent the `std::vector` from continuously growing in size, but does not affect the calculations in any way. To prevent duplicate halo particles, the Particle Vector also provides a method to delete all halo particles.

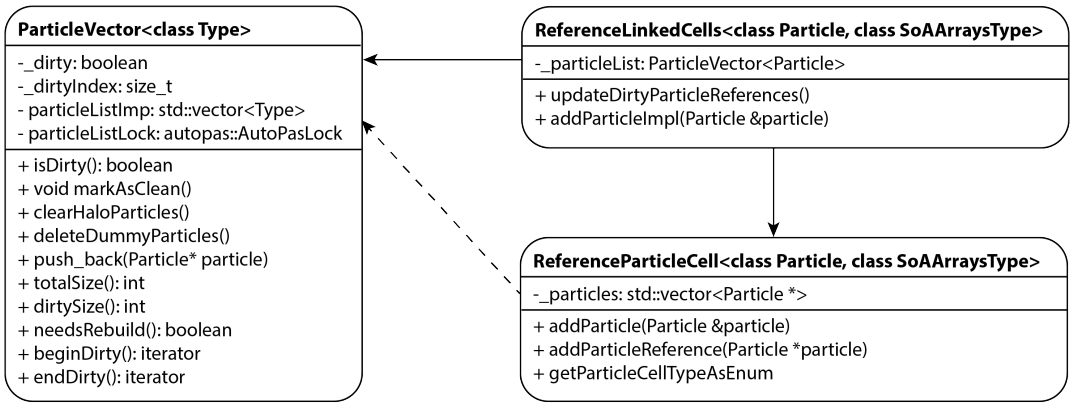


Figure 4.2: A simplified view of the Reference Linked Cells container. The Reference Linked Cells class performs operations on both the Reference Particle Cell and the Particle Vector, indicated by the arrows with solid lines. The only interaction between the Reference Particle Cell and the Particle Vector are the reference lookups performed by the cell, indicated by the arrow with the dashed line. Not all fields and methods are shown for the Reference Linked Cells and the Reference Particle Cell. Inheritance is also omitted.

4.2.2 Reference Cell

The implementation also requires a new type of cell, which operates only on references. The Reference Cell stores only references internally, with the actual particle being held in the Particle Vector, and also uses a `std::vector` for it, for the same reasons the Full Particle Cell already provided by the library does, which is mainly cache prefetching. In order to conform to the interface of the AutoPas library, it resolves the references it stores before returning them.

For the Reference Cell, deletion of a particle can no longer be performed by simply removing the particle from the cell. Doing so would result in only the reference being removed, while the Particle Vector would still keep the actual particle stored. Once a rebuild is triggered, this particle would then be added back into the cell instead of being completely removed. For this reason, the Particle class also had to be updated. A field, indicating whether the particle can be deleted or not, was added. The delete operation inside the Reference Cell is updated to mark the particle for deletion, before removing the reference from the cell.

The Reference Particle Cell also inherits the addParticle method from the ParticleCell class. This method is not needed by the Reference Particle Cell, so it throws an exception if called and a second method operating on references is added as shown in Figure 4.2. The remaining functionality is analogous to the Full Particle Cell with slight adaptations to the Reference Cell operating on references.

4.2.3 Reference Linked Cell Container

To implement the Linked Cells algorithm using references a new container similar to the existing Linked Cells container is being added. The Reference Linked Cells container will share most of its implementation code with the existing Linked Cells container.

The most apparent change is the fact, that the Reference Linked Cells container will operate on the Reference Particle Cell instead of the Full Particle Cell, utilizing the Particle Vector to store the actual particles and only pass the references to the cell. This functionality is adapted for halo particles.

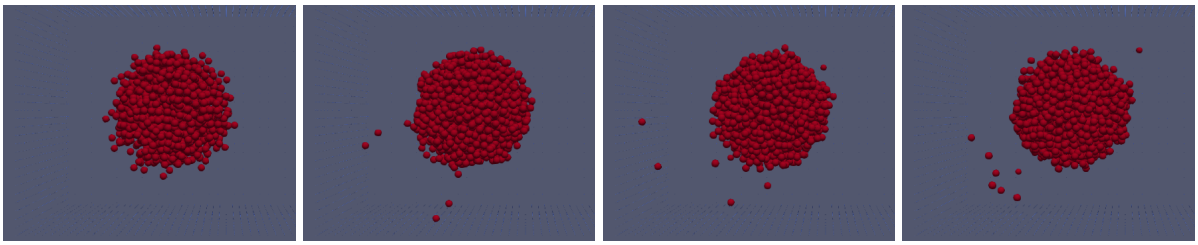
The container also has to keep track of the dirty flag of the Particle Vector and add dirty particles to the cells, rebuilding those whenever older references become invalidated, shown by the solid lines with arrows in Figure 4.2.

5 Results

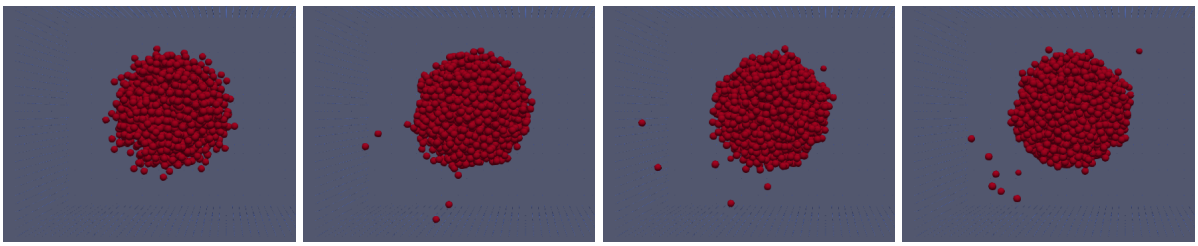
5.1 Correctness

The implementation of the Linked Cells algorithm using references should display the same behaviour as the already existing Linked Cells implementation of the AutoPas library.

This assumption is supported by the excessive unit testing suite of the AutoPas library. Figure 5.1 also shows an exemplary simulation at various time steps. It is clearly visible, that both simulations output the exact same snapshot at every timestep.



(a) LC: 5000 iterations (b) LC: 11000 iterations (c) LC: 17000 iterations (d) LC: 23000 iterations



(e) RLC: 5000 iterations (f) RLC: 11000 iterations (g) RLC: 17000 iterations (h) RLC: 23000 iterations

Figure 5.1: The falling drop configuration provided by md-flexible example of the AutoPas library for the Linked Cells container displayed in the first row in (a), (b), (c) and (d) and the Reference Linked cells container in the second row in (e), (f), (g) and (h). The simulation for both containers is shown at 5000, 11000, 17000 and 23000 iterations. Every image in the first row is absolutely identical with the corresponding one in the second row, leading to the conclusion that both containers perform the exact same calculation.

5.2 Performance

The falling drop configuration provided by the md-flexible example of the library was executed for both the Linked Cells and the Reference Linked Cells container without OpenMP on a single thread. Both simulations where run several times on the CoolMuc2 [15], with the hardware specifications shown in Table 5.1.

The Linked Cells container was able to perform an average of 23700 iterations in 2 hours where the Reference Linked Cells container managed to only complete an average of 23500.

Table 5.1: Hardware Specifications of the CoolMuc2

Specification	Value
Number of nodes	812
Cores per node	28
Hyperthreads per core	2
Core nominal frequency	64 GB (Bandwidth 120 GB/s - STREAM)
Memory (DDR4) per node	2.6 GHz
Bandwidth to interconnect per node	13,64 GB/s (1 Link)
Bisection bandwidth of interconnect (per island)	3.5 TB/s
Latency of interconnect	2.3 s
Peak performance of system	1400 TFlop/s

The above provides the hardware specification by the entire CoolMuc2 Linux cluster. Performance measurements where performed without using OpenMP, therefor only using a single thread, on 2 nodes.

Those performance measurements, due to their limited nature, can only give a rough estimate of the Reference Linked Cells containers performance. Since the Linked Cells container has optimizations in place to specifically utilize multi-threading, it is very likely to expect worse results for simulations with OpenMP active. The estimate does, however, show, that, with proper optimization strategies implemented for the Reference Linked Cells container, it is very likely, that the performance difference will become negligible.

6 Summary

This thesis consists of two parts. The first part is the removal of the Particle Cell template. This is necessary for the Reference Particle Cell container to be able to specify its own Particle Cell template, which, unlike the Full Particle Cell, needs to operate on references instead of actual particles. The removal of the Particle Cell template decreases the complexity of the codebase. It also simplifies the libraries interface, which does no longer require the user to specify a Particle Cell template.

The second part of the thesis implements the Reference Linked Cells container. It is also demonstrated the correctness of the implementation, both by running the AutoPas unit test suite as well as comparing snapshots of the simulation performed by the Reference Linked Cells container against the Linked Cells container. The thesis also gives a rough performance estimate for the new container, which seems promising but requires both further evaluation and optimization for the Reference Linked Cells container to rival the Linked Cells containers performance.

7 Outlook

More, extensive performance evaluation is needed to properly benchmark the Reference Linked Cells implementation against the one already provided. It is to be expected, that the Reference Linked Cells container will, in its current form, perform worse, due to too many cell rebuilds being performed as well as caching not being leveraged to it's fullest extend.

7.1 Performance Optimization

7.1.1 Reduce the number of Cell rebuilds

To decrease the number of cell rebuilds, the following suggestions could be investigated:

Rebuild only before iteration

The particle checks the Particle Vectors dirty flag for every particle added, owned and otherwise, and a rebuild is performed every time the Particle Vectors internal `std::vector` performs a resize operation. This leads to cells being rebuilt several times instead of only once before the iteration loop gets started. Triggering cell rebuilds only inside the rebuildNeighborLists does not pass the AutoPas unit test suite. This means that either tests have to be adapted or a new function with the desired properties has to be created.

Halo Particle Vector

Both halo particles as well as owned particles are stored inside the same container. All halo particles are removed after every iteration, leading to possibly a large number of owned references also being invalidated. This could be prevented by creating a separate Particle Vector inside the Reference Linked Cells container that only stores halo particles. As a result, clearing all cells of halo particles and adding them back in again for the next iteration would not invalidate any owned particle references, of which there is expected to be a drastically larger amount.

7.1.2 Sorting the Particle Vector

The Reference Linked Cells does, unlike the Linked Cells container, not periodically re-sort its particles. This leads to particles belonging to the same cell possible drifting further apart in memory over time, missing out on a lot of caching benefits with increasing simulation length. The performance improvements gained from sorting particles are further explained in [16]. The Particle Vector could then also contain several dummy particles evenly distributed

throughout the internal `std::vector`, to prevent said vector to perform a resize operation. This of course comes at the cost of more memory being needed by default.

Adapting the particle class to also contain a boolean field, indicating whether the particle reference has become invalid during the last re-sorting, alongside an address field, keeping track of where the particle that previously occupied this slot was moved to. By only allowing for the swapping of two particles, this does not create much overhead and could drastically increase performance, because the container could be sorted at will without any cell rebuilds being required.

7.2 Apply to different containers

After the Reference Linked Cells container has been improved performance-wise, the same concept can be applied to the other containers as well rather easily, since most of them also use linked cells internally. During auto-tuning, a feature integrated into the AutoPas library, one container is being translated into another, which means copying every particle into a different location inside the new container that is chosen by AutoPas as the most efficient option. If all containers were using the same Particle Vector to store the actual particles, this translation step would be reduced to simply rearranging all the references. This is expected to vastly increase performance, as copying full particles involves a lot more overhead than moving references.

List of Figures

List of Tables

Bibliography

[1] S. A. Hollingsworth and R. O. Dror. "Molecular dynamics simulation for all". In: *Neuron* 99.6 (2018), pp. 1129–1143.

[2] B. Jovanović, R. M. Brum, and L. Torres. "MTJ-based hybrid storage cells for "normally-off and instant-on" computing". In: *Facta universitatis-series: Electronics and Energetics* 28.3 (2015), pp. 465–476.

[3] A. S. Tanenbaum and H. Bos. *Modern operating systems*. Pearson, 2015.

[4] A. J. Smith. "Cache memories". In: *ACM Computing Surveys (CSUR)* 14.3 (1982), pp. 473–530.

[5] I. Chivers and J. Sleightholme. "An introduction to Algorithms and the Big O Notation". In: *Introduction to Programming with Fortran*. Springer, 2015, pp. 359–364.

[6] B. Alpern, L. Carter, E. Feig, and T. Selker. "The uniform memory hierarchy model of computation". In: *Algorithmica* 12.2-3 (1994), p. 72.

[7] A. Raval, S. Piana, M. P. Eastwood, R. O. Dror, and D. E. Shaw. "Refinement of protein structure homology models via long, all-atom molecular dynamics simulations". In: *Proteins: Structure, Function, and Bioinformatics* 80.8 (2012), pp. 2071–2079.

[8] L. Greengard. "The numerical solution of the n-body problem". In: *Computers in physics* 4.2 (1990), pp. 142–152.

[9] U. Welling and G. Germano. "Efficiency of linked cell algorithms". In: *Computer Physics Communications* 182.3 (2011), pp. 611–615.

[10] D. Vandevoorde and N. M. Josuttis. *C++ Templates: The Complete Guide, Portable Documents*. Addison-Wesley Professional, 2002.

[11] J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley, D. Smith, and G. Bierman. *The Java Language Specification: Java SE 14 Edition*. Oracle America, Inc., 2020, pp. 216–219.

[12] F. A. Gratl and S. Seckler. *AutoPas*. URL: `https://github.com/AutoPas/AutoPas`.

[13] F. A. Gratl, S. Seckler, N. Tchipev, H.-J. Bungartz, and P. Neumann. "AutoPas: Auto-Tuning for Particle Simulations". en. In: *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. Rio de Janeiro: IEEE, May 2019. ISBN: 9781728135106. DOI: `10.1109/ipdpsw.2019.00125`. URL: `https://ieeexplore.ieee.org/document/8778280`.

[14] I. Newton. *The Principia: mathematical principles of natural philosophy*. Univ of California Press, 1999.

[15] Leibniz-Rechenzentrum. *CoolMUC-2*. URL: `https://doku.lrz.de/display/PUBLIC/CoolMUC-2`.

[16] R. Fair, X. Guo, and T. Cui. "Particle sorting for the projection based particle method". In: *Engineering Analysis with Boundary Elements* 109 (2019), pp. 199–208.