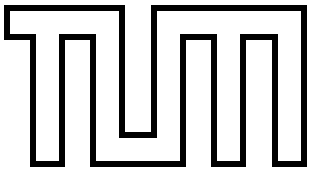


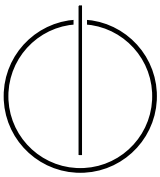
FAKULTÄT FÜR INFORMATIK

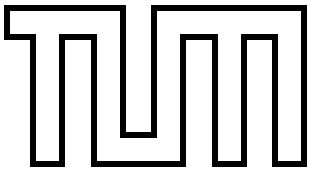
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Integrating Kokkos into AutoPas for hardware agnostic particle simulations

Ludwig Gärtner





FAKULTÄT FÜR INFORMATIK

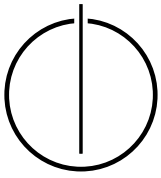
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Integrating Kokkos into AutoPas for hardware agnostic particle simulations

Integration von Kokkos in AutoPas für hardware-agnostische Partikelsimulationen

Author:	Ludwig Gärtner
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisor:	Fabio Alexander Gratl, M.Sc.
Date:	14.02.2022



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

Munich, 14.02.2022 Ludwig Gärtner

Acknowledgements

I want to thank the Chair of Scientific Computing in Computer Science and Professor Hans-Joachim Bungartz for the opportunity to write this thesis and the support I received with organizational matters.

Thank you Fabio, the advisor of this thesis, for your support throughout this entire thesis. Your help, pretty much at any time of the day, and our many inspiring discussions contributed greatly to its outcome.

Also thank you to Charlotte, for your emotional support during stressful times.

I gratefully acknowledge the Leibniz Supercomputing Centre for funding this project by providing computing time and support on its Linux-Cluster.

Abstract

AutoPas is an auto tuner for N-body simulations. Kokkos is a library that offers hardware independency through abstractions. The goal of this thesis was, an integration of Kokkos into AutoPas to improve performance portability.

This integration for now is limited to the KokkosLinkedCells container, which uses a central particle storage and an index based metadatastructure to link particles to cells in the simulation domain.

However, the implementation for this thesis was not finished to a point where it could be run on systems other than conventional multi core systems. Instead, the performance of the new KokkosLinkedCells container is compared to that of the regular LinkedCells container to determine the overhead that is introduced by this additional layer of abstraction. Overall, both containers perform very similarly in scenarios where work is evenly distributed among all threads, but the absence of algorithmic load balancing or dynamic scheduling harms the performance of the KokkosLinkedCells container in scenarios where particles are focused in one part of the simulation domain.

Zusammenfassung

AutoPas ist ein Autotuner für N-body Simulationen. Kokkos ist eine Bibliothek, die hardwareunabhängige effizente Parallelisierung ermöglicht, indem sie Hardwarespezifisches hinter einer uniformen Schnittstelle verbirgt. Das Ziel dieser Masterarbeit war es, Kokkos in AutoPas zu integrieren, um dessen hardwareunabhängige Performance zu verbessern.

Diese Integration wurde in einem ersten Schritt nur für den LinkedCells Container entwickelt, welcher einen zentralen Partikelspeicher und eine indexbasierte Metadatenstruktur verwendet, um die Simulationsdomäne in Zellen zu unterteilen.

Allerdings wurde die Implementierung für diese Arbeit nur insoweit fertig gestellt, dass sie nur auf einem konventionellen Mutlicore System ausgeführt werden kann. Deshalb wurde der KokkosLinkedCells Container direkt mit dem LinkedCells Container verglichen, um den Einfluss der zusätzlichen Abstraktion festzustellen, die Kokkos mit sich bringt. Für Szenarien, in denen die Partikel gleichmäßig über alle Zellen verteilt sind, ist die Performance beide Container annähernd gleich. In Szenarien, in denen die Partikel auf einen Bereich der Simulationsdomäne konzentriert sind, ist die Leistungsfähigkeit des KokkosLinkedCells Containers eingeschränkt, da dessen Implementierung bisher noch kein dynamisches Scheduling beinhaltet.

Contents

Part I.

Introduction and Background

1. Introduction

Performance portability is an important trait of modern software. As systems vary, software cannot be tailored to execute perfectly on one particular combination of hardware and software components. This is also applies to high performance computing applications like Molecular Dynamics simulations. The amount of horizontal or vertical scaling, the availability and speed of memory, the simulation scenarios etc.; they all affect the runtimes of solvers and solving approaches for such simulations.

AutoPas solves this problem as an auto tuner for N-body simulations. It optimizes a combination of different building blocks to optimally fit the current scenario. [GSBN21] This is done by testing and measuring the configuration combinations and creating a performance profile in the first steps of the simulation. Afterwards, the simulation runs with the best configuration, and after a predefined amount of time the performance portfolio is updated. This process is repeated until the termination condition is fulfilled.

The next step in using all the given resources of a computing environment is to also offload work onto possible hardware accelerators. As some of those, especially GPGPUs, are architecturally very different from CPU based computation environments, the simulation code base has to be flexible enough to perform well on both.

Kokkos offers exactly that flexibility by providing abstractions to fit the current scenario. [TLGA+22]

This flexibility is particularly important for storing data and parallel code execution. Kokkos offers exactly that by providing abstract data structures and operations to dispatch parallel code execution which internally adapt to fit the current scenario. [TLGA+22]

This thesis was motivated by increasing the performance portability of AutoPas by executing its existing algorithms on GPGPUs. Using Kokkos for this brings the advantage of supporting not just GPGPUs but all of the platforms that are supported by Kokkos. However, as this requires a lot of low level changes, this thesis is just laying the groundwork for a complete transition in the future.

2. Theoretical Background

For the theoretical background this chapter includes a small introduction to molecular dynamics simulations and how certain algorithms are applied to reduce their computational complexity. These algorithms were designed for CPU based systems, and do not work efficiently on GPGPU based systems without modifications. So, afterwards the difference between CPU and GPGPU-accelerated architectures is outlined as well as the difficulties that arise when trying to write code that runs efficiently on both.

2.1. Molecular Dynamics Simulation

Molecular dynamics simulations calculate the forces that particles within a simulation domain exercise on each other. Therefore, forces between all particle pairs have to be calculated and then applied, leading to a computational complexity of $O(n^2)$ with n being the number of particles.

In reality, multiple forces have to be regarded when calculating the interaction of particles. In 1924, John Lennard-Jones and John Edwards proposed the Lennard-Jones potential as a mathematical model to simplify the calculation of the potential energy that arises from these different forces. The model is nowadays often used in molecular dynamics simulations. [GKZ07]

2.1.1. Lennard-Jones Potential

The Lennard-Jones potential is described as

$$U_{l,j}(P_1, P_2) = 4\epsilon \left((\frac{\sigma}{R})^{12} - (\frac{\sigma}{R})^{6} \right) \tag{2.1}$$

and is visualized in Figure 2.1 [LJ31].

For very close ranges particles exercise a strong repulsion on each other. The repulsion falls off very quickly and is overpowered by a longer ranged attractive force. However, with increasing particle distance, these attractions converges to zero.

The range at which the forces are considered negligible is also described as cutoff in the following work. By neglecting all particle pairs with a greater distance than the cutoff, the calculation can be split into two parts:

- The distance calculation.
- The potential calculation if and only if the distance $R \leq \textit{cutoff}$.

In this way, the distance is still calculated for every particle pair, but the more expensive Lennard-Jones potential is reduced to the amount of close neighbors of each particle.

The cutoff distance is variable rather than a fixed distance. An increased cutoff leads to a more accurate simulation but also an increased computational workload.

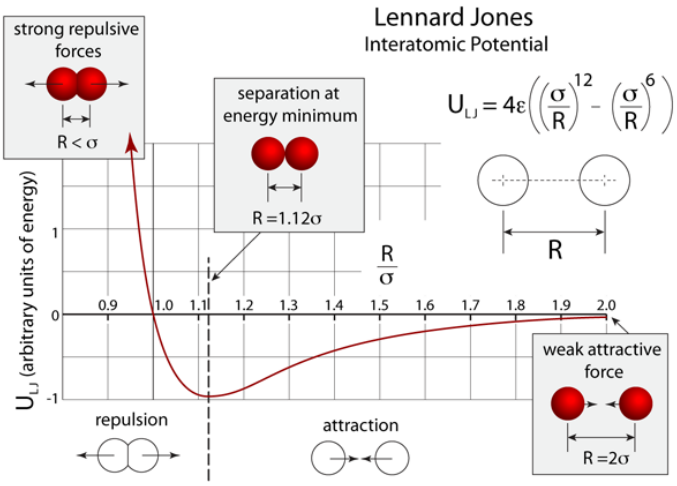


Figure 2.1.: The Lennard-Jones potential visualized based on particle distance. Image taken from `http://atomsinmotion.com/book/chapter5/md`

2.1.2. Linked Cells Algorithm

The Linked Cells Algorithm takes further advantage of the cutoff. It uses a datastructure to store the particles in such a way that without calculating the distance it can be derived if particles are further apart from each other than the cutoff. This is done by dividing the simulation domain into box shaped cells with a sidelength equal to the cutoff distance. After sorting the particles in their respective cells, it can be guaranteed that particles are not within the cutoff distance if they are not in the same or neighboring cells (including diagonal ones) without needing a distance calculation.

For example, in the middle of Figure 2.2 the two dimensional simulation domain is divided into four by four cells. When trying to find the particles within the cutoff of the red particle only the red cell and its direct neighbors (light blue) have to be searched.

When choosing the cutoff proportional to the amount of particles, the complexity of the distance calculation process is reduced to $O(n)$, because particles in one cell are only checked against particles in a constant number of cells. [Gra17]

However, there is still a large overhead of unnecessary distance evaluations: Placing a sphere with radius equal to the cutoff inside a block of three by three by three cells, each with a sidelength equal to the cutoff reveals that the former covers only about

$$\frac{\Pi\frac{4}{3}c^3}{(3c)^3} = \frac{4\Pi}{81} = 15.5\% \tag{2.2}$$

of the latter's volume. This means that on average only 15.5% of distance calculations are made for particles that within cutoff distance. For the remaining particle pairs this distance calculation is ideally avoided.

Splitting the simulation domain into cells also helps with task based parallelism, either with shared or distributed memory systems:

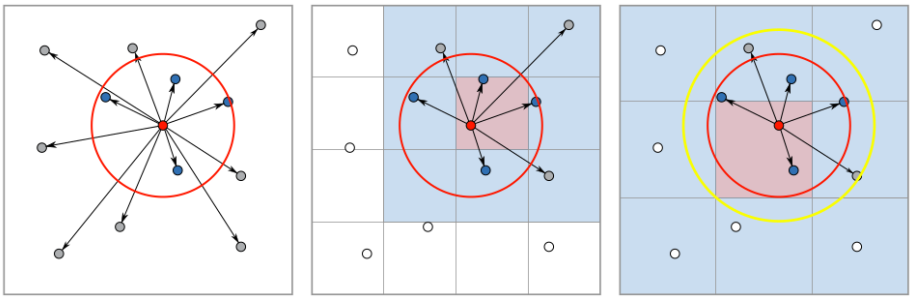


Figure 2.2.: This figure shows different algorithms that reduce the computational workload of md-simulation: only distance calculations on the left, Linked Cells in the middle, and cell based Verlet Lists on the right. Taken from [GSBN21]

- In a shared memory system, multiple tasks can safely access the same cell structure, as long as each task only updates its designated cell.
- In a distributed memory system, the neighboring cells of the local subdomain are exchanged as so called halo cells rather than exchanging the entire simulation domain.

Another optimization can be implemented as long as the chosen potential is symmetrical. Newton's third law of motion (or *newton3*) states that two interacting bodies apply the same magnitudal force to each other but in opposing directions.

$$F_{A,B} = -F_{B,A} \tag{2.3}$$

If this symmetry is given, the force calculations only have to be done once for each particle pair and then can be applied to both interacting particles.

If this principle is applied in parallel race condition problems arise, because tasks not only update their own data but also data of other tasks. This has to be avoided, by for example traversing the cells in such a way that parallel tasks never access each other's data.

2.1.3. Colored Traversals

Here, the idea is to assign each cell in the linked cells algorithm a color, so that all cells with the same color can be safely processed at the same time.

When the task for one cell only updates that cell's data, so without newton3, a coloring with one color (*c01*) suffices.

With newton3, each cell only has to perform the force calculations within itself and between certain cellpairs in its vicinity.

A one dimensional example can be seen in Figure 2.3. Every $task_x$ calculates the forces for particle pairs within $cell_x$, as well as where one particle is in $cell_x$ and the other is in $cell_{x+1}$. This means that the forces between particles from $cell_{x-1}$ and $cell_x$ are already taken care of by $task_{x-1}$. Each task is updating the particles of two cells, so a two colored traversal would make this threadsafe if the cells with even indices are handled in parallel first, followed by the cells with odd indices.

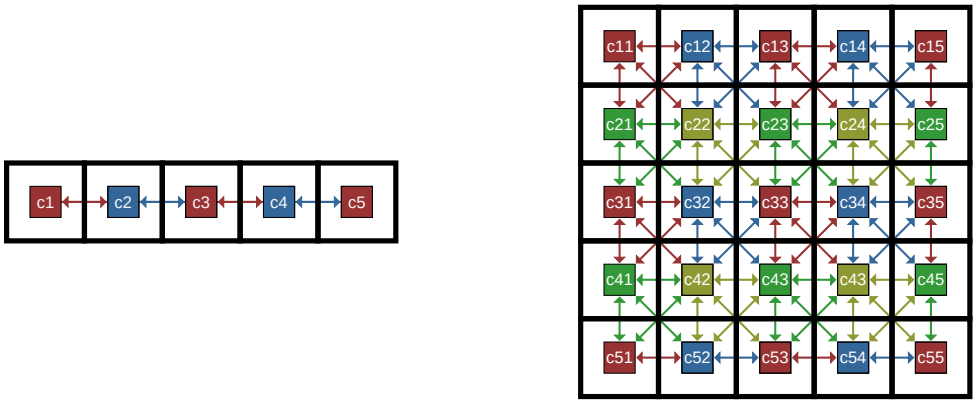


Figure 2.3.: The visualization for a one dimensional c02 coloring on the left, and a two dimensional c04 coloring on the right.

The right side of Figure 2.3 shows a four colored traversal for two dimensional simulation domains. In that example, a task handles forces within its cell as well as all forces between the cells in the two by two square that expands to the lower right.

This pattern continues for three dimensional examples. Each task updates a cell and all forces between cells in the two by two by two cube that expands in one direction. Therefore, when using newton3 a safe traversal is using $2^3 = 8$ colors (*c08*) with equally colored cells exactly two cells apart in each direction.

2.2. GPGPU Accelerated Architectures

This section covers the differences of GPGPU accelerated systems compared to conventional hpc architectures, and how these differences affect the way the aforementioned algorithms for md-simulations are implemented.

Memory Spaces

When dealing with GPGPUs the first difference is that they come with their own memory and execution space, termed here as *device memory space* and *device execution space*. [Gho12] Their respective counterparts are the *host memory space* and *host execution space*. Each execution space can generally only access its own memory space. As a result, the programmer has to manually manage the memory transfer between the two memory spaces, as some functionalities are only available on one of the execution spaces. For example, sending/receiving halo cells via MPI is typically restricted to *host memory*, so for each simulation step the sent and received buffers have to be copied over.

Newer NVIDIA architectures, namely kepler class (SM30) or newer, support unified memory access. Unified memory is a virtually shared memory that can be accessed by host and device processes, This means the cuda system migrates the accessed memory pages to the memory of the accessing execution space.

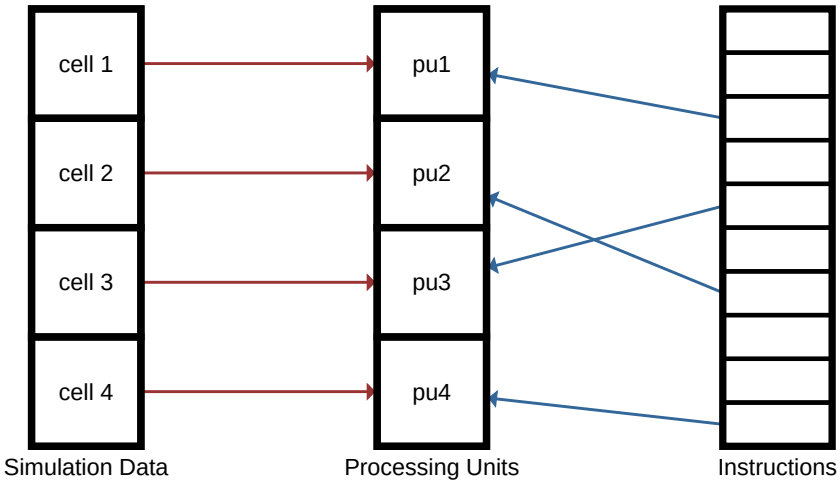


Figure 2.4.: A visual example on MIMD parallelism. Each processing unit gets particle data from different cells, and instruction depending only on their own process through the instruction set.

As this application does not require a lot of memory transactions between host and device memory and it restricts the application to certain NVIDIA GPGPUs (pascal or newer), a unified memory was not considered in the design process for this thesis' implementation. [nvi]

Parallel Execution

While multi core CPUs typically implement a task based multiple instruction multiple data (MIMD) style of parallelism, GPGPUs are better compared to vector or single instruction multiple data (SIMD) style parallelism. The former spawns a couple of tasks which run asynchonously and independently, while the latter executes the same instructions on different data. As a practical example, the MIMD approach lets the programmer parallelize the force calculations of different cells, while the calculations within the cell are done sequentially. An example state of this is visualized in Figure 2.4. Every processing unit holds some data from one cell and performs some instruction on it.

In contrast to that, the SIMD approach calculates the pairwise forces of a set of particles and another set of particles in parallel, but processes the cells one after another. Figure 2.5 shows the processing units getting consecutive data from the same cell and the same instruction from the instruction set. After the entire instruction set is performed on all particles, the next set of particles is processed.

This difference becomes relevant for the programmer when looking at how the two architectures access their own memory. In Multi Core systems, each core has its own cached memory which fetches a consecutive block of data from the shared memory and copies it into a private cache. Changing this private buffer invalidates that data in the shared memory as well as other private buffers that happen to have fetched the same data. So for this architecture, it is most efficient to separate the data each core is going to use to avoid this

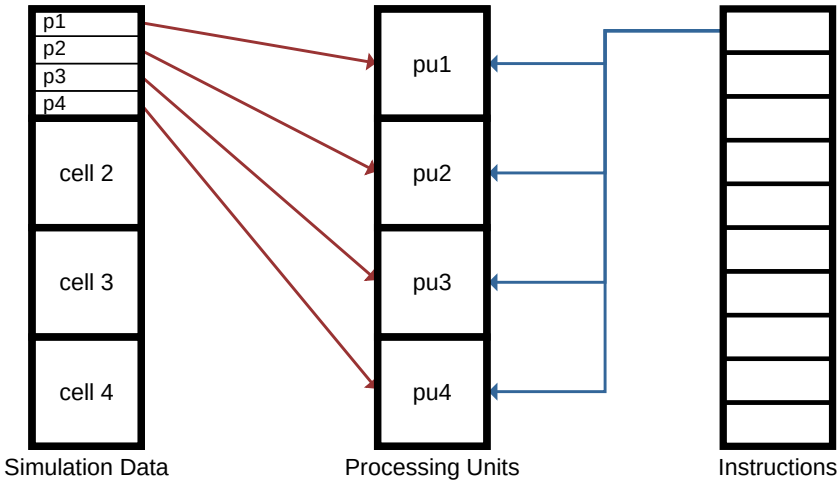


Figure 2.5.: A visual example on SIMD parallelism. All processing units ideally get consecutive data and perform the same instruction on it.

false sharing.

GPGPUs also fetch data in consecutive blocks, and they work best if consecutive threads on the GPGPU work with consecutive data. So, if the threads one to 32 work on the data points one to 32 in an array respectively, only a single memory access is necessary. Whereas, when the data is spread out every data point has to be fetched independently. Consequently, for this system to work best the data that is accessed in parallel needs to be close together. [Gho12]

This difference in optimal memory management is the main problem when trying to write code that is efficient for both environtments.

3. Technical Background

This chapter focuses on explaining the state of AutoPas before this work started. It aims to give an understanding of which parts need to be changed when integrating the Kokkos library as well as the functionalities of Kokkos itself.

3.1. AutoPas

AutoPas is an auto tuner for molecular dynamics simulations. Its target group are scientists that know what they want to calculate, but lack the expert knowledge in high performance parallelization. This means, AutoPas takes a user defined particle and particle properties as well as a functor for the force calculations two particles exercise on each other. A functor, in this context, is an executable object that calculates the force two given particles exercise on each other. Lastly the user has to define an initial scenario for the simulation. [GSBN21]

To start a simulation, an AutoPas object is built as a combination of several components. First a *particle container* is chosen. This container then defines the *particle cell* type and the subset of *shared memory traversals* available for it. Then there are some container independent options: whether to use *newton3* optimization, and the choice between the datalayouts *AoS* and *SoA*. [GSBN21]

The following subsections explain the tasks of each component and describe one example, respectively.

3.1.1. AutoPas Containers

A container in AutoPas describes the datastructure that is used to derive particle locations. One example for this is based on the Linked Cells algorithm, see Subsection 2.1.2, named the *LinkedCellsContainer* in AutoPas.

The first task of a container is to build and maintain the data structure for particle storage. For the *LinkedCellsContainer* this means creating an array of particle cells, which divide the simulation domain. When adding particles, the container makes sure they are stored in the correct cell and after the particles' positions are updated, this check has to be repeated for every particle in case they moved between subdomains.

In theory, this container update has to be done after every simulation step. However, the cells can be chosen slightly larger than the cutoff distance (Subsection 2.1.2). Consequently, a particle that travels a short distance into a neighboring cell is still correctly influenced by all particles within its cutoff distance. This brings the option of updating containers less frequently and thereby reduces the computational workload. But increasing the size of the cells also decreases the potential hit-rate of particles within the particles' cell plus neigh-

bor cells, and is a tradeoff that needs to be considered when creating the container. [GSBN21]

The second task of a container is to perform actions on all or a subset of particles. This can include position updates of all non-halo particles, writing the state of all particles in a certain region to a file, or deleting particles. To perform this task, the container creates an iterator for all affected particles, which is then returned the user. [GSBN21]

The final task of a container is to add all of its cells to a compatible traversal so that the pairwise particle forces can be computed. Also, the container has its own set of implemented traversals, as they depend heavily on the underlying datastructure [GSBN21]. The implementation details of the traversals is found in Subsection 3.1.3 or the AutoPas github repository [1].

Other implemented containers in AutoPas include:

- Direct Sum: the simplest container with only one cell for all particles.
- Referenced Linked Cells: a variant of the Linked Cells container with a central particle storage that only keeps particle references in its cells.
- Verlet Lists: Shown on the right of Figure 2.2, this container uses an underlying linked cells algorithm to build reference lists for every particle, pointing to each particle that is within cutoff distance (shown in red). Again an increased buffer distance can be used to rebuild this structure less frequently (shown in yellow).

For all implemented containers please refer to the official AutoPas paper [GSBN21].

3.1.2. Particle Cells

The particle cell has one core purpose: tying particles to their subdomain of the simulation. For that it uses a dynamically sized vector which allows the adding and removing of particles at any time.

There are two types of particle cells available: the *FullParticleCell* and the *ReferencedParticleCell*. Each container can only use one particle cell, with the *ReferencedParticleCell* being only used by the *ReferencedLinkedCells* container, and the *FullParticleCell* being used by every other cell based container in AutoPas. The only difference is, that the *ReferencedParticleCell* stores only references to particles which are actually stored in a central particle buffer, whereas the *FullParticleCell* stores the particles itself.

Apart from creating iterators that are used by the container, they contain no logic.

3.1.3. Traversals

The next component of the AutoPas object is the traversal. A traversal describes a way of iterating over particle cells in parallel, so that each cell can compute its internal pairwise forces as well as the pairwise forces between its own particles and particles of neighboring cells. As described in Subsection 2.1.3, parallelizing over cells becomes non-trivial when using newton3 optimizations.

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

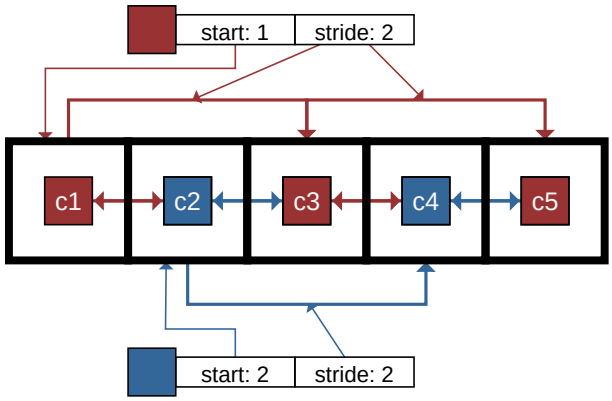


Figure 3.1.: A visualization of the start and stride vectors for a one dimensional c02 coloring. The start vector indicates the first element of this iteration. The stride vector defines the relative index of the next element in each iteration step.

When using cell based containers colored traversals are one solution method. In such a traversal, each color defines start and stride vectors. Figure 3.1 visualizes the start and stride vectors for a one dimensional co2 traversal. The start vector tells on which index this color's iteration starts, and the stride indicates how many cells are skipped per iteration step. This way, multiple traversals can be defined by only changing the amount of colors and their respective start and stride vectors:

- A simple c01 traversal is defined by only one color with $start = [0, 0, 0]$ and $stride = [1, 1, 1]$. With only one color, all cells are processed in parallel, making this traversal not newton3 safe (see Subsection 2.1.3). The advantage of this traversal is the parallel efficiency. No cores idle while waiting for others to finish their tasks.
- The more advanced c08 traversal defines its start vectors as all cells of a $2 \cdot 2 \cdot 2$ cube, and a stride of $[2, 2, 2]$. This leaves eight sequentially executed parallel sections where each cell is safe to update its neighbors (see Subsection 2.1.3). With its even spread of equally colored cells, the load balancing tends to be very good for this traversal. However, there are still eight fences after every color is processed which could harm the parallel efficiency. Additionally, only one eighth of the simulation domain can be processed in parallel. This could further harm the parallel efficiency for small simulation domains with few cells or computing platforms with a lot of parallel processing power. [GSBN21]

Figure 3.2 shows the general code for a colored traversal, which iterates over all cells with the same color in parallel, waits for all cells to finish processing, and moves on to the next color.

For a full list of the traversals and detailed explanations please refer to the official AutoPas paper [GSBN21] or the AutoPas github repository [2].

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

```
for (c in colors)
    parallel_for (index[3] in #cellsPerDimension / c.stride)
        x = index_x + c.start_x
        y = index_y + c.start_y
        z = index_z + c.start_z
        process_cell(x + c.strice_x,
                     y + c.stride_y,
                     z + c.strice_z)
    fence()
```

Figure 3.2.: This is the pseudo code for how colored traversals iterate over cells. Each color defines a start and stride vector. The colors are processed sequentially, but all cells that are indexed by the same color (ie. $cellindex = start + i * stride$ for some i) are processed in parallel.

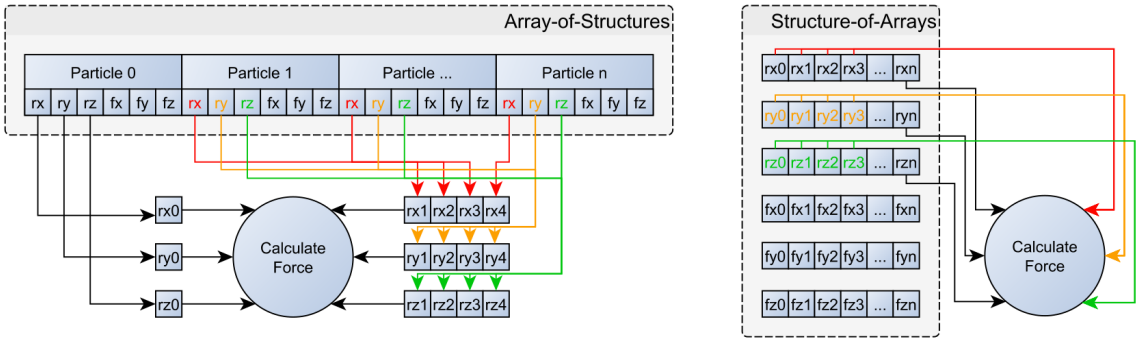


Figure 3.3.: The available datalayouts of AutoPas. The Aos data layout on the left stores the attributes of particles as structs and therefore closely together in memory. The SoA data layout on the right separates the attributes of each particle and gathers them in an array. Taken from [GSBN21]

3.1.4. AoS vs SoA

The two available datalayouts for AutoPas refer to the layout the particles are stored in when handed to the traversals.

For the AoS (Array of Structs) layout, the particles are stored as structs, one after the other. This is visualized in the left part of Figure 3.3. The AoS layout makes fetching and moving entire particles very easy as only consecutive memory is worked on. AoS is the superior datalayout for purely task based MIMD parallelism, also because cache invalidations are more easily avoided.

For the SoA (Struct of Arrays) layout, the particle attributes are stored as arrays. Each particle is distributed over the different attribute arrays, and one particle x is reassembled with the values from each attribute array at index x. This is visualized on the right part of Figure 3.3. This datalayout is superior when working on vector based SIMD parallelised systems, especially on GPGPUs with coalesced memory access described in Section 2.2.

It should be noted that the particle storage in the cells always is in the AoS layout. The

cells transform the particles into an SoA layout on demand and have to do so each time a traversal is initiated and conversely when a traversal is finalized.

3.2. Kokkos

As already mentioned in Section 2.2, writing code that should run on several HPC platforms is difficult. Kokkos as a library solves exactly that problem by "[providing] key abstractions for both the compute and memory hierarchy of modern hardware." [TLGA+22]

This section focuses on the parts of Kokkos used for the integration in AutoPas: the Kokkos::View datastructure and the parallel dispatches. For more information please refer to the Kokkos paper [TLGA+22] or their github repository [3].

3.2.1. Kokkos::View

The Kokkos::View, or just view, is a fixed size array datastructure. When creating a view, the programmer defines the memory space in which the data will be stored. In case Kokkos is compiled for device acceleration, this memory space defaults to the device memory space (eg. GPU memory), otherwise it defaults to host memory space. Note that the metadata of a view is always accessible from both, host execution space and device execution space. Views have an automatic reference counting, meaning they allocate data on creation, and deallocate the data automatically when all references to that data have gone out of scope.

subviews can be created from parts of a view. A subview is a regular view that does not point to its own data, but to the data of the view it has been created from. This means the Subview has full read and write power on the data it is pointing to, but cannot modify the meta information of the original view (eg. resizing).

A view in the host memory space is accessed regularly in the host execution space, but to add or modify data in a device memory space view, the programmer has to do one of two things:

- dispatch a device sided function that accesses and changes the data.
- deep copy an entire view from host to device memory or deep copy single data points into temporary Subviews of size one.

The first restrictions occur when looking at the data a view can store. Kokkos::Views can only hold datatypes T that follow a couple of rules; mostly around their constructors and destructors. T must have a default constructor and a default destructor, and both must not allocate or deallocate memory and have to be threadsafe. This is because Kokkos internally initializes every datapoint in a newly created view in parallel using the default constructor. [TLGA+22] Additionally, creating views inside views is technically possible, but it is highly discouraged in the Kokkos documentation [4]. The main reason for this is, again, that on construction of a view Kokkos allocates the memory and initializes every datapoint of said view by executing the default constructor in parallel. When this datapoint contains another view, the allocation of data in the default constructor violates one of the

[3] `https://github.com/kokkos/kokkos`
[4] `https://github.com/kokkos/kokkos/wiki/View`

restrictions of datatypes that can be stored in a Kokkos::View.

Lastly, views have between one and eight dimensions. The amount of dimensions has to be defined during the declaration at compile time, but the size of each dimension can be set on instantiation. When declaring multi dimensional views, Kokkos offers an additional layout option. This layout determines the mapping of logical indices to the physical memory offsets. The two predefined layouts are *LayoutLeft* and *LayoutRight*. With *LayoutLeft* the leftmost indexed datapoints of a view are closest together in the physical memory, and vice versa.

Section 2.2 describes the issue of oposite memory layouts being optimal for MIMD and SIMD parallelization. Kokkos fixes this by defaulting device memory views to *LayoutLeft* and host memory views to *LayoutRight*. Consequently, the array can be logically addressed equally when programming for both systems, but the underlying memory layout changes based on the currently used architecture. [TLGA+22]

3.2.2. Kokkos parallel operations

Kokkos offers thre parallel operations, however, for the following implementation only two are relevant:

- a parallel for loop called *parallel_for*.
- a parallel reduction called *parallel_reduce*.

The parameters for the *parallel_for* operation are a n-dimensional range policy and a lambda function with n parameters of type unsigned long. The range policy defines the range of indices for every dimension. The lambda function takes the n loop indices, and its body refers to the body of a conventional for loop.

Kokkos does not guarantee how many iterations of the loop are executed at the same time as well as in which order these iterations are processed. [TLGA+22]

The *parallel_reduce* operation works similar to the *parallel_for* operation by taking a range policy and a lambda function. However, the reduction operation additionally takes a reduction variable by reference, which stores the final reduced value after execution, and a reducer.

Kokkos provides a set of predefined reducers for simple reduction operations, like summations, products, and logical binary operations. For the complete list please refer to the Kokkos github repository [5]. Furthermore, the lambda function is now required to not only take the set of indices, but also an intermediate reduction variable by reference with the same type as the global reduction target. [TLGA+22]

[5] `https://github.com/kokkos/kokkos`

4. Related Work

Cabana

The only project that draws similarities to the thesis is the library Cabana and the molecular dynamics simulator built with it: CabanaMD. Cabana uses a similar list of algorithms that are also implemented in AutoPas (see Subsection 3.1.1). However, while AutoPas changes its configuration at runtime, the Cabana configuration has to be defined at compile time.

The process of simulation also works quite differently compared to AutoPas. Cabana only takes care of the data structure and leaves the traversing part to the user. This is done by providing two components:

- a central particle storage
- a collection of containers which only sort the central particle storage and creates a metadata object. When using, for example, the linked cells container, the particle storage is sorted in such a way that particles within the same cell are next to each other in memory. The metadata object then points to the beginning and end of each cell in the central particle storage.

As described in the next chapters, a similar process was chosen for the KokkosLinkedCells container in the AutoPas Kokkos integration.

Therefore, Cabana's sorting algorithm for the central particle storage was closely inspected. It features parallel out-of-place sorting and binning of particles. Sorting particles creates a deterministic order of particles, whereas binning only makes sure that particles are sorted with respect to an assignment, like a cell index, but their order within the assignment is random.

Cabana uses a combined AoSoA datalayout as central particle buffer that was also a particularly interesting idea. Its keynote is to store the particles in an array of SoAs, with each of the SoAs being large enough to fit exactly one vector of a vector machine. Most modern CPUs are capable of vectorization, so this is a promising idea for performance portability, as this layout can be optimal for mixed cases with vectorization capable multicore systems.

Part II.

Integrating Kokkos into AutoPas

5. Implementation

This chapter covers the changes that were made to integrate Kokkos in AutoPas. A prerequisite for the platform indepentent execution was to rework the process of accessing particles via iterators, as iterators provide pointers to data that can be inaccessible to some execution spaces. Afterwards, the real integration could start by designing the general structure of Kokkos based containers. Lastly, the implementation of two containers is detailed.

5.1. Prerequisite: Replacing iterators

As described in Section 3.1, AutoPas used iterators to perform operations on particles by returning them to the user code. The left visualization of Figure 5.1 shows, that when using a device memory space these pointers reference data that is not available in the host execution space the user code is running in. The logical counterpart of bringing data to the user code, was, to bring the user code to the data by using lambda functions. The right part of Figure 5.1 also nicely shows that the iteration does not require any context switches, as the entire iteration is performed on the device.

Iterators were used to update singular particles by executing user code. One example is updating the particle velocities and positions after the forces are calculated. Because these iterators are meant to only affect singular particles, the given user code has to be threadsafe, to not create problems via race conditions. In case an iterator is used in a non threadsafe manner, for example when writing output to a file, when reducing particle values, or when using nested iterations, it can specifically be used in serial mode. Additionally, a user can choose which particles should be included in the iterator based on particle ownership and position. For a matching lambda function based solution, all these details needed to be replicated.

To update particles, *forEach* functions with *inRegion* and *parallel* options were added to the AutoPas API. Each of these have the particle ownership for the target particles as a parameter as well as a bounding box for the *inRegion* vartion. Additionally a lambda function is passed to this API. The lambda function should be defined to take and process a particle, and produces no output. See Figure A.1 in the appendix for an example call to the forEach API.

For reduction operation over particles a *reduce* call with the same *inRegion* and *parallel* options as for *forEach*. Again each of these calls took a lambda function and the particle ownership as parameters as well as the bounding box for the *inRegion* variation. In addition to these, the global reduction target for the reduction needs to be passed, and the lambda function needs a local reduction target as parameter. See Figure A.2 in the appendix for an example call to the reduce API.

Within AutoPas, the lambda function and execution parameters are passed to the active container. The container then checks which of its cells fit the execution parameters and

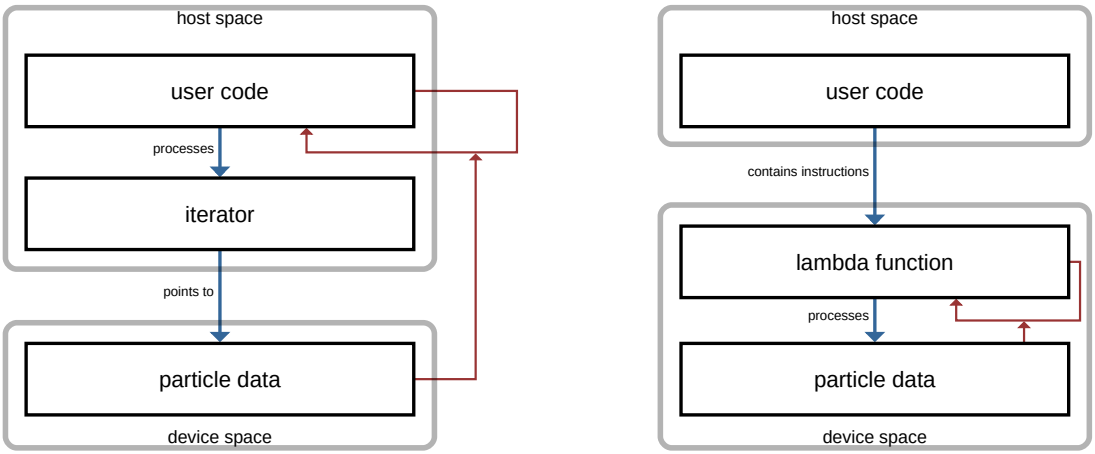


Figure 5.1.: A visualization to show the iteration over particle; the old iterator based approach on the left and the lambda function based approach on the right.

iterates over these cells. Each cell then iterates over all particles, checks again if they fit the execution parameters, and executes the lambda function on those which do.

With its available meta information, the container optimizes this process by determining the necessity of some checks. If a cell, for example, only contains particles of one ownership type, the check for ownership does not need to be executed on a per particle basis. Or if the cell is fully within the bounding box of the *forEachInRegion* call, this check can be left out of the iteration within the cell.

The actual iteration over particles and code execution is performed by the particle cell.

5.2. The General Structure of AutoPas with Kokkos

Each cell based container in AutoPas stores a list of particles or particle references in a list of cells. In Subsection 3.2.1 it was explained why storing views inside views is not recommended so keeping the current structure is impossible.

This section starts by explaining the implementation details of the container, as it is the central component in AutoPas, followed by the particle cell, the new central particle storage and the Kokkos based traversals.

5.2.1. The Kokkos Cell Based Particle Container

All the particle and metadata that is required during the simulation needs to be available in the targeted execution space. Consequently, the datastructures that store such metadatadata would have to be replaced with Kokkos::views.

Instead, a different approach was chosen for the Kokkos cell based containers: all particles are stored in a central particle view, similar to the *ReferencedLinkedCells* container, and cells only contain indices to a range of particles in this central particle view similar to the approach in Cabana. For these references to work, the central particle view also needs some

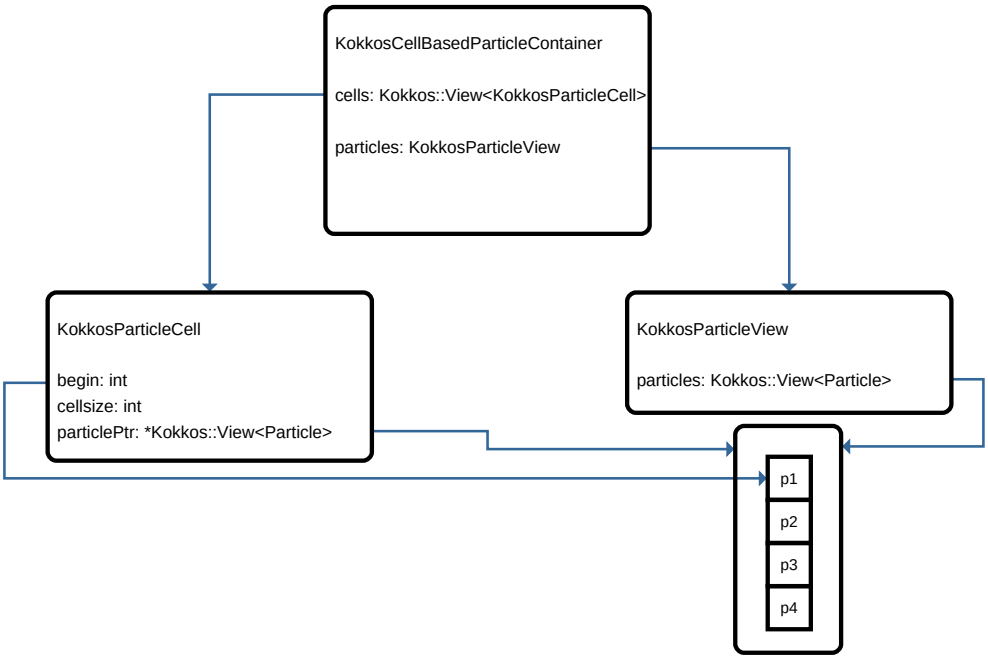


Figure 5.2.: The new structure of AutoPas with *KokkosCellBasedParticleContainers*.

sort of particle sorting. After these changes, the Kokkos cell based particle container logically holds one list of particles and one list of cells. Figure 5.2 visualizes this new structure.

The tasks of the container remain largely unchanged from the base *CellBasedParticleContainer*. Each implementation of the *KokkosCellBasedParticleContainer* requires have its own particle-to-cell assignment. This assignment is used when updating the container, i.e. resorting the central particle storage and building the metadata information.

One additional task comes from the fact that the stored particles need to be in the correct order, or else the metadata is incorrect. Therefore, a *dirty* flag was added to the container. Unlike in the previous structure, particles cannot be added directly to their correct cell but have to be appended at the end. So, whenever particles are, added this flag is set to *true*. Only after updating the container, i.e. resorting the particles, it is set to *false* again.

The forEach logic has not been changed apart from checking the dirty flag when trying to use the cell metadata for optimizations.

Initializing traversals also has not been changed from regular AutoPas.

5.2.2. The Kokkos Particle Cell

The *KokkosParticleCell* is drastically reduced in complexity compared to the other cells, because it does not store any particles. Due to the datatype restriction for Kokkos::views, the cell is turned into a simple struct that only stores three data points:

- **Begin** is an unsigned long value that references the starting index of the cell in the central particle storage.

- **CellSize** is an unsigned long value that counts the amount of particles in the cell, and together with *begin* indexes the end of the cell in the central particle storage.
- **ParticleViewPtr** is a pointer to the Kokkos::View inside the central particle storage. This is required as the traversal, these cells are passed to, has no direct access to the central particle buffer.

5.2.3. The Particle View

The *ParticleView* is the new central particle storage of the *KokkosCellBasedParticleContainer*. As Kokkos::views are fixed in size, some workarounds for this needed to be implemented. Kokkos offers a resize for views which was used to implement a basic growable particle storage.

For this, the *ParticleView* holds two variables: size and capacity. The size describes the amount of particles that are currently in storage. It changes every time particles are added to or removed from the simulation. Additionally, it indicates the field where new particles should be stored.

The capacity describes the actual size of the Kokkos::view and the maximal amount of particles that can be stored in it. Every time a particle should be added to the view, even though its capacity is already reached, the capacity is doubled and the Kokkos::view resized, allowing for more storage.

After the particle storage were sorted out, the additional responsibilities of the *ParticleView* had to be implemented.

The first responsibility is sorting particles according to the assignment function of the container. This is done in four steps. Figure 5.3 shows a visualization of this process.

Firstly, the cellsizes of each cell have to be computed by iterating over all particles and counting how often each cell is assigned. This is necessary as particles cannot be inserted in between other particles, so the *begin* indices of each cell have to be known before the sorting can take place.

Secondly, the new position for each particle has to be found and stored in a permutation vector. The index of a particle is calculated by first finding its cell with the assignment function, then incrementing that cell's *cellsize*, and lastly adding *begin* and *cellsize*. The work in this step is performed in parallel, and to avoid race conditions the increment has to be done by an atomic *fetchAdd* function.

Afterwards, the particles are permuted by moving them into an intermediate copy of the original particle view. As this is also done in parallel, the permutation cannot be in place, i.e. within one view, to avoid race conditions.

Lastly, the content of the intermediate view has to be copied back into the main particle view concluding the permutation process.

The second additional responsibility of the *ParticleView* is to perform the individual particle updates with the previously implemented lambda based solution (see Section 5.1). In the original implementation, this was located in the particle cells. So in addition to the ownership and location specific functions, a third set of parameters was necessary to cover the entire functionality: an index range to iterate over all particles of one cell.

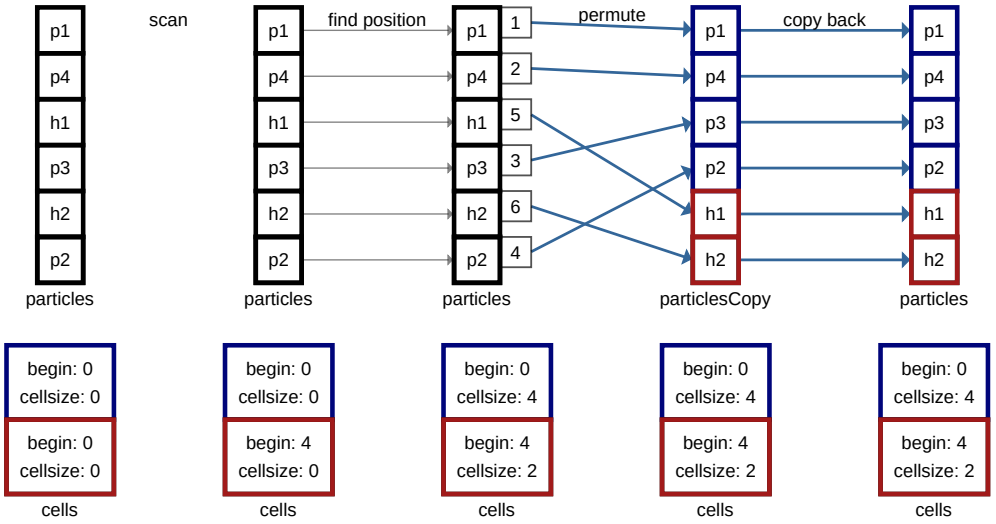


Figure 5.3.: A visualization for sorting particles and halo particles into their respective owned (blue) and halo (brown) cells. The process is composed of a scan for cellsizes, a permutation into a copy of the original view, and a copy back operation.

Every check is hidden behind a templated boolean variable set to true if this check has to be performed. Templated boolean variables mean that for each state of the variable a separate function is compiled, so the checks are not even in the code when disabled.

The same applies to the parallel and serial execution mode. Kokkos only supports parallel iteration, so the Kokkos::parallel_for body contains another serial for-loop. The ranges for these two loops are interchanged based on the templated *parallel* boolean variable. When set to true, the *range policy* of the parallel for loop is set to the desired particle range, whereas the sequential for range is set to {0, 1} and vice versa.

Figure A.3 shows a pseudo code simplification of this function.

The same changes to the *forEach* function were applied to the *reduce* implementation.

5.2.4. The Kokkos Color Based Traversal

The CBasedTraversal only includes a three-fold for-loop to iterate over all cells which was parallelized with OpenMp. Differences between the colored traversals only lie in the stride and offset variables (see Subsection 3.1.3).

This means that apart from replacing the three-fold OpenMp for loop with the Kokkos parallel for loop taking one RangePolicy per dimension nothing had to be changed to create the *KokkosCBasedTraversal* interface.

5.3. The implemented KokkosCellBasedContainers

This section describes the implementation of two *KokkosCellBasedContainers*: the *KokkosDirectSum* container and the *KokkosLinkedCells* container.

```
1 assignDirectSum(Particle p){
2     if (p.getOwnershipState().equals(owned)){
3         return indexOwnedCell;
4     } else {
5         return indexHaloCell;
6     }
7 }
```

Figure 5.4.: The assignment function for the *KokkosDirectSum* container.

5.3.1. KokkosDirectSum

Because of its logical simplicity, the DirectSum container was chosen as a first candidate for conversion. However, it shares implementation details with more advanced concepts. So, the idea was to use the gained insight when building the *KokkosLinkedCells* container.

The DirectSum container is simple because it only uses two cells: one for the owned particles and one for halo particles. Owned particles are the particles that are targeted by the simulation while halo particles only set the boundary conditions. This results in a very simple assignment function that is shown in Figure 5.4 and an ordering of the central particle storage that is easily testable.

Additionally, there is no further metadata of the particles' positions stored, so the forEach and reduction lambda function only check the targeted ownerships and then pass the requests on to the lower components.

As AutoPas parallelizes the force calculation process by iterating over multiple cells at the same time, which simplifies creating a traversal for the direct sum container. Only the owned cell has to be processed, meaning the force calculations within the owned cell and between the owned and halo cell. The result of this is the *KokkosDSSequentialTraversal*.

5.3.2. KokkosLinkedCells

When creating a cell based container, the first thing to do is create the cell data structure. In the original *LinkedCells* container, this is done by the *CellBlock3D*, a helper object that handles the logical mapping of the simulation space to cells. This helper object was completely reused for the *KokkosLinkedCells* container. It features functions to:

- calculate the required amount of cells for the given cellsize and simulation domain size,
- to map the three dimensional spacial cell indices to the one dimensional indices of the cell datastructure,
- and, to map points in the simulation space to cells indices.

Thereby, the *CellBlock3D* already provides most of the logic that is required for the *KokkosLinkedCells* container. When creating the container, the cellblock can be used to spawn the correctly sized Kokkos::View to store the particle cells. The assignment function only calls the cellblock's *get1DIndexOfPosition*. And lastly, the cellblock can differentiate whether cells contain halo particles which is useful for optimizing the *forEach* particle updates.

To calculate the pairwise forces between particles, this container offers parallel colored traversals. Because of time constraints, only a one colored traversal was implemented, the c01 traversal. This, however, was trivial and promises a fast transition of additional colored traversals that are already available for the conventional linked cells container. The conventional linked cells c01 traversal defines start and stride vectors for the one color and uses the functor to iterate over all particles of neighboring cells. It only had to be copied and inherits the *cTraversal* method of the *KokkosCBasedTraversal* instead of the conventional one.

6. Results

This chapter analyses the previously described implementation, starting with an example simulation that primarily shows the capabilities of md-simulations but also serves as a sanity test for the KokkosLinkedCells container.

Afterwards, the performance of the new KokkosLinkedCells container and its traversal is compared with its logical predecessor. Firstly, the general strong scaling of both containers is compared. Secondly, the scaling of both is investigated in multiple scenarios, including the distribution of time spent in all steps of a simulation. Afterwards, the total simulation time of very large simulations is studied. Lastly, the impact of resorting the particles in the KokkosLinkedCells container is analysed.

Most of the measurements in this chapter have been taken on the cm2_tiny partition of the CoolMUC-2 cluster[1]. This partition of the cluster offers four nodes with up to 56 parallel threads each[2]. However, as these analyses focus on shared memory parallel performance, only one node was used for the measurements.

6.1. Example Rendering

An examplary simulation that can be conducted with AutoPas' md-flexible is displayed in Figure 6.1.

This simulation was performed with 2^{14} particles over 75 timesteps with a $\Delta t = 1 * 10^{-8}[s]$.

owned particles	2^{14}
iterations	75
cellsize	one times the cutoff $c = 1$
simulation domain size	10^3 cells

The vtk prints in Figure 6.1 are snapshots taken at iterations $t = 0$ (top left), $t = 25$ (top right), $t = 50$ (bottom left), and $t = 75$ (bottom right). In addition to the particle positions, the visualization colors the particles based on the magnitudal force that is exerted on them.

Initially, the paricles are arranged according to a three dimensional Gaussian distribution. This means that the particle density peaks in the center of the domain and becomes less at the edges of the simulation domain. All particles are colored equally blue, as this is the initial state before any forces are calculated.

After a few iterations, some particles have started to rush outward, but most are still located near the center, which can be seen in the second picture. Additionally, all particles are colored individually based on their magnitudal force.

The third and fourth images show an increasingly uniform particle distribution.

[1] `https://doku.lrz.de/display/PUBLIC/Linux+Cluster`
[2] `https://doku.lrz.de/display/PUBLIC/CoolMUC-2`

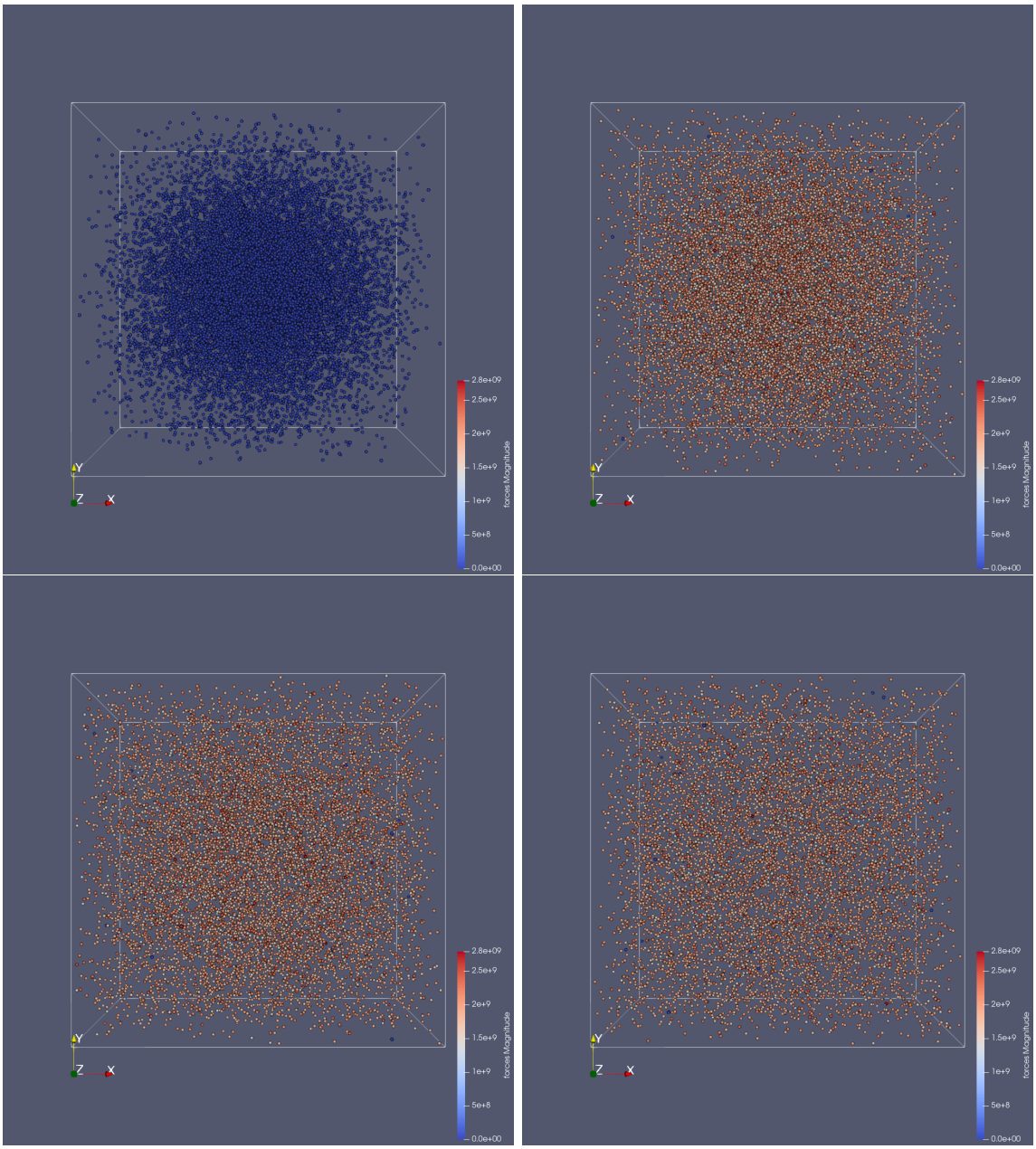
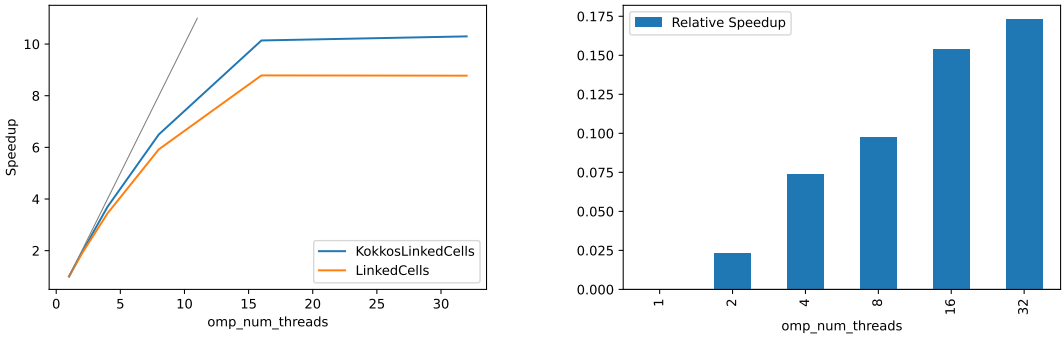


Figure 6.1.: The visual output of a md-simulation performed with AutoPas. Starting with particles that are arranged according to a gaussian distribution in the top left and the particles spreading out the longer the simulation runs. The particles' color represent the magnitudal force that is exerted on them at that point in time.



(a) Speedup

(b) Relative Speedup

Figure 6.2.: The left diagram shows the speedup of the KokkosLinkedCells container, the LinkedCells container, and the theoretically optimal speedup in grey. The right diagram shows the relative improvement in speedup of KokkosLinkedCells over LinkedCells.

6.2. Strong Scaling Analysis

Strong scaling uses Amdahl's law to calculate the speedup of an application when using more and more parallel computing resources. The speedup

$$S(N) = \frac{t_1}{t_N} \tag{6.1}$$

describes the relative performance of an algorithm when using N parallel processes compared to its serial execution. The optimal speedup is a N-times faster total execution time for N-times the parallel processing units. However, because of serial parts of applications and parallel overhead, this optimal speedup is unreachable.

For this analysis, a simulation with the following configuration was chosen:

owned particles	$2^{16} = 65536$
iterations	20
cellsize	$1\times$ cutoff $c = 1$
particle distribution	uniform
simulation domain size	10^3 cells
newton3	disabled (because c01 does not support it)
datalayout	AoS (because it was not implemented yet)
parallel threads	$2^x \vert x \in [0..5]$

The uniform particle distribution should focus the measurements completely on parallel performance, as load balancing is trivial in this situation.

Note that the given particle count always refers to owned particles. So, including halo particles the total particle count is closer to 2^{17}.

The graph in Figure 6.2a shows the speedup of both containers when scaling from one to 32 parallel shared memory threads. Both start to stagnate with 16 processing units with the KokkosLinkedCells container having a slightly higher cap than the LinkedCells container.

The bar chart in Figure 6.2b shows the improvement in speedup of the KokkosLinkedCells (klc) container relative to that of the LinkedCells (lc) container

$$\frac{S_{klc}(N) - S_{lc}(N)}{S_{lc}(N)} \tag{6.2}$$

With 32 parallel threads, the kokkos implementation finishes 10.3 times faster compared to the regular LinkedCells' speedup of 8.8 times.

This shows that for small examples the kokkos implementation parallelizes the work more efficient than the regular C++ and OpenMP one.

6.3. In Depth Scaling Analysis

The weakness of a relative analysis like strong scaling is that the better speedup can come from either better parallel or worse serial performance. For this reason this section focuses on a comparison of the actual runtimes of both containers in differenct scenarios.

All measurements in this chapter have been taken from a series of simulations, generated with a combination of the following configurations:

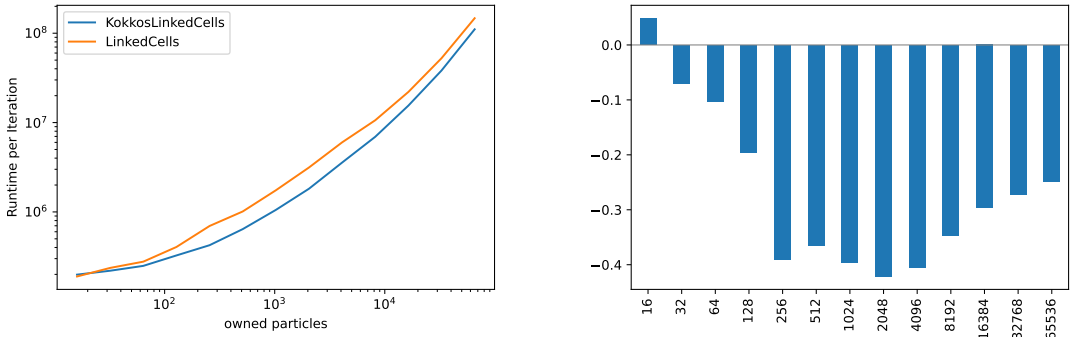
owned particles	$2^x \vert x \in [4..16]$
iterations	between 20000 for simulations with few particles to reduce noise in the data and 20 for a lot of particles to reduce the total simulation time
cellsize	$cs \in \{1c, 2c\}$ with cutoff $c = 1$
particle distribution	uniform or gaussian
simulation domain size	10^3 cells
traversal	c01
newton3	disabled
datalayout	AoS
parallel threads	56

These configurations should cover most areas where differences in the implementations could cause differences in runtime, like parallel overhead for simulations with less particles, parallel efficiency with a lot of particle interactions, and load balancing. Additionally, building the container and sorting the central particle storage was done once in the beginning and not updated afterwards.

To highlight differences in runtimes in certain areas of the following analysis, this equation is used to calculate the relative improvement of the KokkosLinkedCells (klc) container over the LinkedCells (lc) container:

$$\frac{t_{klc}(N) - t_{lc}(N)}{t_{lc}(N)} \tag{6.3}$$

where $t_x(N)$ is the time t it takes to perform a certain computation with the container x dependent on the number of particles N.



(a) Runtime per Iteration

(b) Relative Difference in Runtime

Figure 6.3.: The left graph shows a log-log plot of the average duration of a single iteration for both containers when increasing the particle size and using a cellsize equal to the cutoff distance. The right chart shows the relative difference in runtime per iteration between the two comtainers, using Equation 6.3. Negative values favor the KokkosLinkedCells container.

6.3.1. Runtime Per Iteration Analysis for Uniformly Distributed Particles

First of all, the average duration of a single iteration step is compared when increasing the particle size and using a cellsize equal to the cutoff distance. The graph in Figure 6.3a shows a log-log plot of that average simulation step duration. The LinkedCells container is faster per iteration for the first simulation with only 16 owned particles. This can be explained with the larger parallel overhead of the kokkos parallel function dispatches compared to the little computational work that is done in this simulation. 16 particles spread uniformly over 1000 cells results almost no particle interactions.

For larger simulations, however, the KokkosLinkedCells container pulls ahead and is, according to Figure 6.3b, over 40% faster in some scenarios.

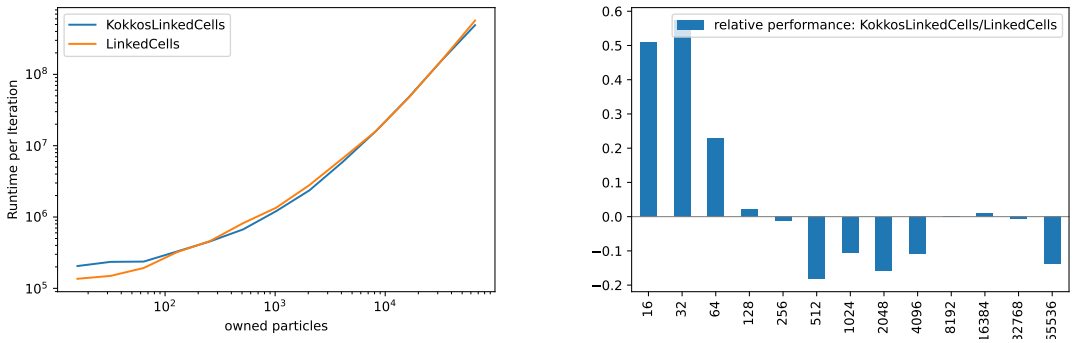
When increasing the cellsize to twice the cutoff distance, the previously observed advantage of the LinkedCells container is increased to about 50%. For more particles, the runtimes of the KokkosLinkedCells container improve relatively to the LinkedCells container, similarly to the $cellsize = cutoff$ example. The KokkosLinkedCells container even overtakes the LinkedCells container in the range between 2^8 and 2^{12} particles. Overall the performance is closer for the larger cellsize.

One curious result is that the biggest improvement of the KokkosLinkedCells container is found in these medium sized examples. Beyond that amount the difference in per iteration performance decreases again. This applies to both cellsize factors.

Analysis of Runtime Portion of Individual Parts per Iteration

As each iteration step consists of differently scaling parts, it is hard to determine the reasons for this observation from the runtime of an entire simulation step. Therefor, this subsection isolates each of the four parts and analyses them individually.

The steps that each simulation iteration has to process are:



(a) Runtime per Iteration (b) Relative Difference in Runtime

Figure 6.4.: The left diagram shows a log-log plot of the average duration of a single iteration for both containers when increasing the particle size and using a cellsize equal to twice the cutoff distance. The right chart shows the relative difference in runtime per iteration between the two comtainers, using Equation 6.3. Negative values favor the KokkosLinkedCells container.

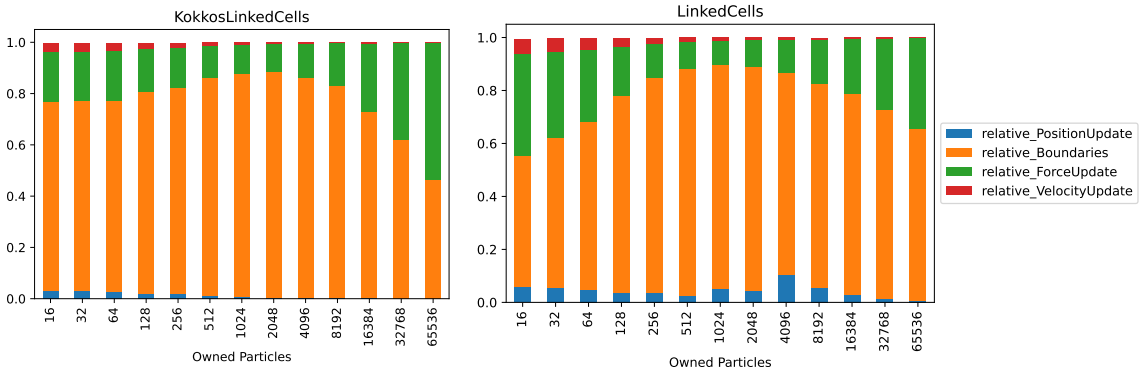
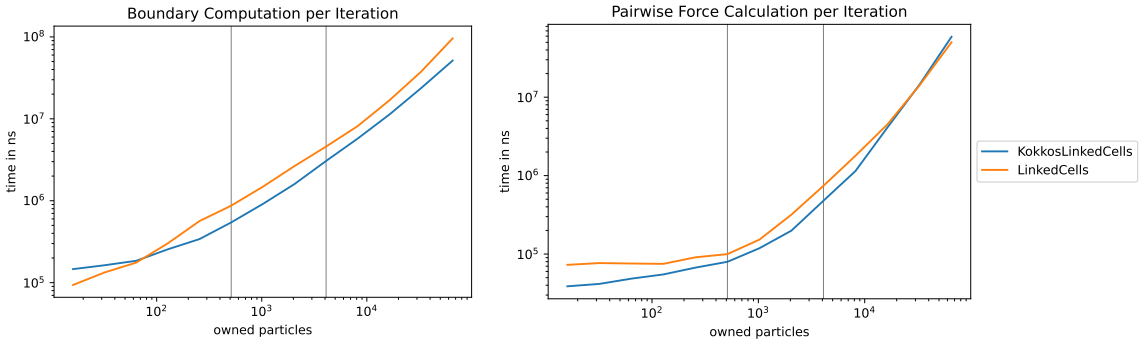


Figure 6.5.: The time distribution for each iteration step of a simulation with both, the KokkosLinkedCells container (left) and the LinkedCells container (right), scaled by owned particle count.



(a) Boundary Updates

(b) Pairwise Force Calculation

Figure 6.6.: The graph shows the comparison in scaling of the boundary update (left) and the pairwise force calculation (right) for both containers. The vertical lines mark the particle count interval in which the total simulation step performance advantage of the KokkosLinkedCells container is the biggest.

- the position updates of all particles,
- the updating of halo particles,
- the pairwise force calculations,
- and the velocity updates of all particles.

Figure 6.5 shows the portion each of these four parts take in a simulation step for both containers. The performance of both containers is mainly driven by the updating of halo particles and the pairwise force calculations. This is because both position and velocity updates only scale linearly with the amount of particles, but the pairwise force calculation scales quadratically with the amount of total particles when keeping the simulation and cellsizes fixed and to update halo particles, the matching particle has to be found among existing halo particles first, this also scales quadratically with the amount of halo particles.

The left diagram of Figure 6.5 shows that up to $2^{11} = 2048$ particles around 80% of the runtime is spent updating halo particles, and around 20% of the time is spent calculating the pairwise forces. This 2048 particle mark relates to an average of two particles per cell leading to an average of 55 distance and eight force calculations per particle. Both double with each doubling of the total particle count for larger simulations leading to a more and more dominant amount of force calculation work.

Like in the scaling comparison in Figure 6.3 and Figure 6.4, it is around that tipping point where the KokkosLinkedCells container has the biggest performance advantage over the regular LinkedCells container, for both cell sizes.

Analysis of Total Runtime of Individual Parts per Iteration

The following comparisons of absolute runtime of the bounday update and pairwise force calculation parts should shed further light on this performance difference. Figure 6.6b shows the scaling of boundary update part of a simulation step which make up the biggest chunk of time of a simulation step. The runtime of this part is dominated by the updating

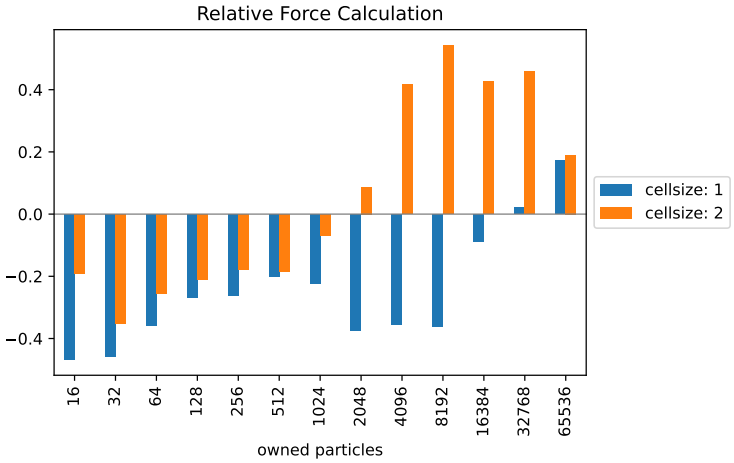
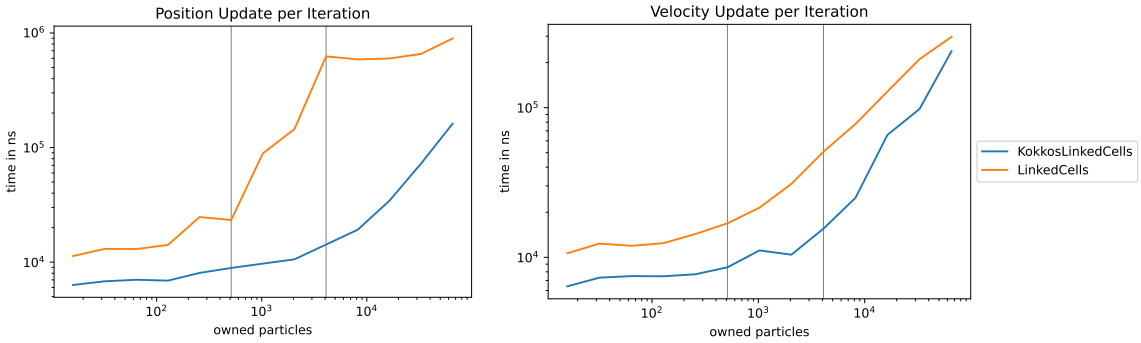


Figure 6.7.: This bar chart shows the relative difference in isolated pairwise force calculation runtime of both containers, grouped by cellsize. These values were calculated using Equation 6.3. Negative values mean an advantage for the KokkosLinkedCells container.

of halo particles. To begin this part, all halo particles are deleted and then replaced with the updated set of halo particles. The KokkosLinkedCells container relies on fixed particle indices so that its index based metadata structure remains valid. Other containers, like LinkedCellsReferences or VerletLists, cross reference particles and need to keep the particles to avoid invalidation of these references. For this reason halo particles cannot be deleted and added each simulation step. Alternatively, halo particles are only marked as deleted but kept in the particle storage. Then, when the updated halo particle set has been received, the old halo particles are searched for matches and when found are updated and reactived again. This saves rebuilding the metadata structures every time step, but comes with the cost a quadratically scaling search and update operation.

The new KokkosLinkedCells container makes use of the new parallel for-each implementation to parallelize this process, whereas the regular LinkedCells container uses the old iterater based approach. Structural design reasons lead to each dimension triggering one parallel operation to update halo particles, resulting in three parallel dispatches per iteration. The first datapoints of Figure 6.6b (16 to 64 owned particles) show that for very small examples the parallel overhead of the three Kokkos' parallel dispatches result in a worse performance compared to the LinkedCells container. But for all simulation sizes beyond that, the parallel efficiency outweighs this overhead and the KokkosLinkedCells container is consistently around 35% faster.

The second biggest factor in the simulation step performance are the pairwise force calculations for owned particles. Figure 6.6a shows the scaling of these force calculations for both containers. In contrast to the boundary updates, the KokkosLinkedCells container performes better for simulations with fewer particles. For the scenarios with more than $2^{13} = 8192$ particles, the performance of both containers gets closer with the LinkedCells improving to a performance advantage of around 18% for the simulation with the highest particle count.



(a) Position Update per Iteration

(b) Velocity Update per Iteration

Figure 6.8.: These graphs show the comparison in scaling of position (left) and velocity (right) updates for both containers.

Figure 6.7 shows how this changes with the larger cellsize factor. For a larger cellsize, the tipping point is earlier, meaning with fewer total particles. Additionally, the performance advantage of the LinkedCells container stabilizes at around 40%. When linking the total particle count and cellsize factor to determine the average particle count per cell, it becomes clear that this tipping point is at around eight to sixteen particles per cell. At this point both implementations use effectively the same computing code as there were no changes made within the cell iteration. So, a possible explanation for this is that the parallel computing efficiency of the regular OpenMP parallelization is superior compared to that of the parallel Kokkos operations.

As seen previously in Figure 6.5, the position and force updates only have a minor impact on runtime each iteration. However, Figure 6.8 shows that the Kokkos implementation is faster throughout the entire series of simulations. This again highlights the improvement of the parallel for-each solution over the original iterator one.

In conclusion, the overall performance of the KokkosLinkedCells and the regular LinkedCells container is very close for uniformly distributed particles, with the Kokkos implementation drawing most of its succes from the new for-each parallelization, but the original one still excelling at the raw parallel efficiency during the pairwise force calculations.

6.3.2. Runtime Per Iteration Analysis for Unevenly Distributed Particles

This section shows the effect of load balancing on the overall performance of the KokkosLinkedCells container. As previously mentioned in Section 3.2, Kokkos does not guarantee a specific order in which the cells are traversed in the parallel operation. Additionally, Kokkos by default uses static scheduling. As a result, the parallel efficiency is expected to drop when using simulation scenarios with unevenly distributed particles.

Figure 6.9 compares the performance of the KokkosLinkedCells container on a uniform and a gaussian particle distribution. Note that this time the scaling is done for the total particle count, not the amount of owned particles. This is because the AutoPas scenario generator extends the particle distribution in the halo cells, so using a uniform distribution leads to more halo particles and, therefor, a larger total particle count. Figure 6.9 shows that for small particle counts the performance is similar, but for larger total particle counts

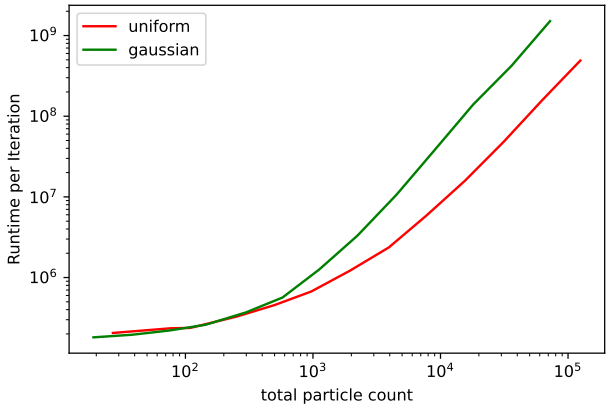


Figure 6.9.: This graph compares the performance of the KokkosLinkedCells container when using a uniform or a gaussian particle distribution.

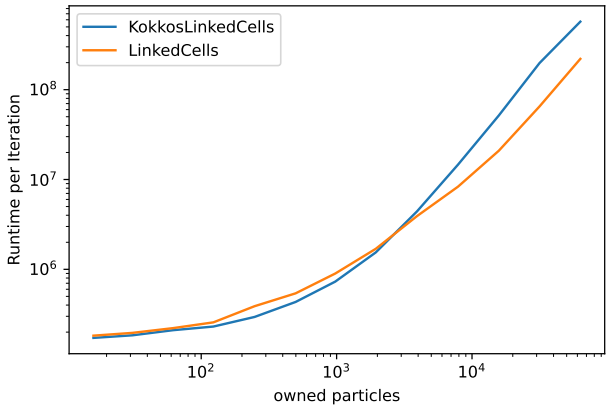


Figure 6.10.: This graph compares the performance of the KokkosLinkedCells and regular LinkedCells container when using a gaussian particle distribution.

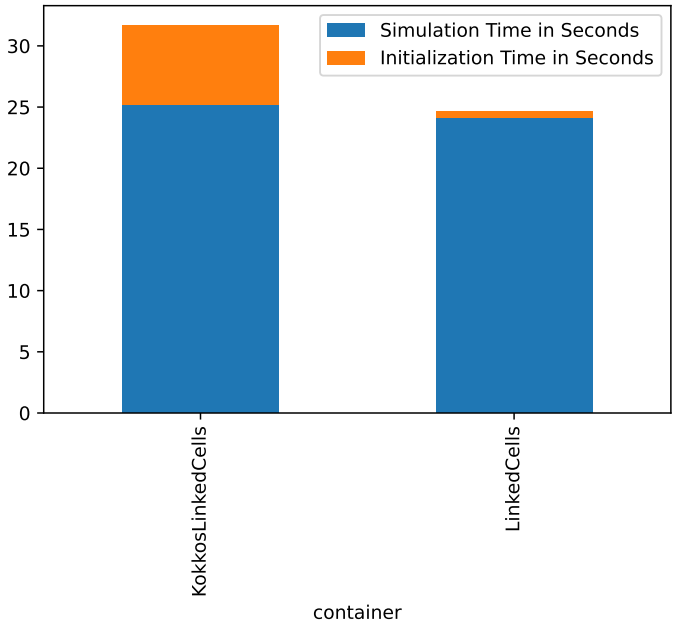


Figure 6.11.: This chart shows the total simulation times of a compute-intensive scenario for both containers: KokkosLinkedCells (left) and LinkedCells (right).

the parallel efficiency is harmed by the forced bad load balancing.

The regular LinkedCells container has integrated load balancing for every traversal that were not yet implemented for the KokkosLinkedCells container. So as expected, Figure 6.10 shows an increasingly longer runtime of the KokkosLinkedCells container compared to its logical predecessor.

Overall, this is a field of the transition that still requires work to make the KokkosLinked-Cells container competitive with the regular LinkedCells one.

6.4. Total Simulation Time Comparison

This section compares the total simulation time of the Kokkos- and regular LinkedCells container when simulating a larger scenario. The used scenario was configured with the following options:

particles	1×10^9
iterations	20
cellsize	1× cutoff
particle distribution	uniform
simulation domain size	25^3 cells
newton3	disabled
datalayout	AoS
parallel threads	56

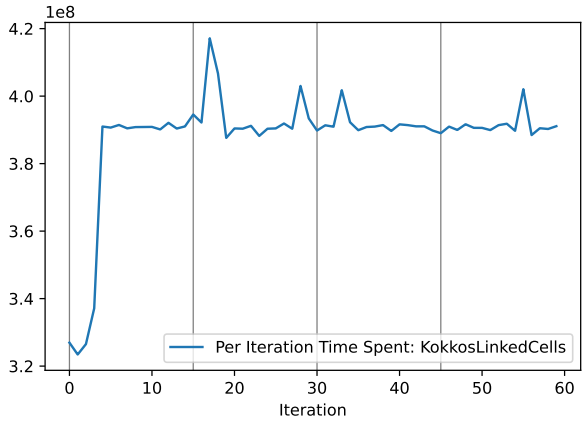


Figure 6.12.: This graph shows the duration of the pairwise force calculations in each simulation iteration. The grey lines mark the iterations where a container rebuild takes place.

Figure 6.11 shows that the total simulation time for the KokkosLinkedCells container is around 32 seconds, whereas the LinkedCells container takes around 25 seconds to complete the simulation. However, the figure also shows that the difference of these two total simulation times is made up almost entirely by the seven seconds of initialization time for the Kokkos implementation.

This initialization phase of the simulation has not yet been covered by any of the measurements. It is the step where the owned particles are generated and then added to the first container. Particles are added one by one, and adding singular elements to a Kokkos::View with Kokkos' deep copy operation is very inefficient.

Apart from the initialization, the total simulation times are close with a slight edge given to the regular implementation. These close times are a further highlight of the trend that was already observed in Subsection 6.3.1.

6.5. The Impact of Resorting the Central Particle Storage

The Kokkos implementation of a container in AutoPas has changed multiple performance critical parts of a simulation. The last one that has not yet been analysed is the impact of resorting the central particle storage and rebuilding the metadata structure.

Both simulations that contributed to measurements for this section were not ran on the lrz linux cluster, but on a local personal computer. So, these measurements are not directly comparable to any of the other ones that were previously shown in this chapter. The CPU in this computer is an Intel Core i7-9700 with a max core clock frequency of 4.7 ghz.

The two simulations for the measurements of this section were configured with the following options:

particles	32000 for the small example and 250000 for the large example
iterations	75
cellsize	$1 \times cutoff$
particle distribution	uniform
simulation domain size	10^3 cells
newton3	disabled
datalayout	AoS
parallel threads	56
container rebuild frequency	once every 25 iterations

Figure 6.12 displays the duration of each of the 75 simulation iterations. This includes all previously analysed parts of a simulation iteration and the rebuilding of the container. Grey vertical lines mark the iterations when container rebuilds took place. However, the graph shows that the performance is barely impacted by these rebuilds, and based on duration alone it is impossible to determine when they happen. On average, the entire resorting and updating of metadata structures takes only 7% of the pairwise force calculation duration. For the larger example, this portion is reduced even more to only about 1%, and the iterations with resorting are even less identifiable.

In conclusion, the resorting of the central particle storage has only a minor impact on the overal simulation performance.

Part III.

Summary and Future Work

7. Summary

The goal of the integration of Kokkos into AutoPas, was, to improve AutoPas' performance portability. But the current state of the implementation does not allow for execution on anything but conventional multi-core systems. The implementation for this thesis got AutoPas closer to such a state, but is not there yet.

The prerequisite of the implementation was to replace the old iterator approach for mass particle updates with lambda based for-each functions. This conceptual change is necessary to run these updates on hardware systems with separated host and device memory spaces and required similar changes in all AutoPas containers.

After this prerequisite was met, the structural base for Kokkos-based containers was designed and implemented. It uses a central particle storage and a metadatastructure that is based on indices in that central particle storage. This metadatastructure is built according to an assignment function, for example position or ownership based, that has to be implemented by each container.

With this KokkosCellBasedParticleContainer in place, it was trivial to create the KokkosDirectSum container and the KokkosLinkedCells container and a subset of their traversals from the base AutoPas version.

When looking at the performance of this implementation, the overall results were good. The largest improvement was accomplished by the for-each functions for parallel particle updates that are used in multiple parts throughout a single simulation. In scenarios in which manual load balancing is not required, the performance of the KokkosLinkedCells container is close to its logical predecessor, the LinkedCells container. However, in cases with unevenly distributed particles, and therefor differing particle counts per cell, the new KokkosLinkedCells container would profit from better load balancing, for example, via dynamic scheduling.

8. Future Work

The first aspect is getting this implementation ready for execution on GPGPU based hardware systems. This feature was the reason to integrate Kokkos into AutoPas and without it most of the implementation for this thesis is obsolete.

To improve the performance of the current implementation, some additional features have to be added. Firstly, implementing a way to add multiple particles to the simulation increases the efficiency of adding particles to the central particle storage for KokkosCellBasedParticleContainers. Therefor, the initialization time of simulations with large amounts of particles will be reduced significantly. Secondly, adding the eight color based traversal for the KokkosLinkedCells container allows newton3 improvements and add a traversal with inherently good load balancing.

As the LinkedCells container is the base for more complex cell based containers, it is now possible to add the Kokkos variant of other cell based AutoPas containers like VerletListsCells.

Part IV.

Appendix

A. Code excerpts

```
autoPas.forEachParallel([=] (Particle p) -> void {
    process(p);
}, IteratorBehavior::owned)
```

Figure A.1.: An example call to the forEach function in the AutoPas api.

```
autoPas.reduce([=] (Particle p, double &reductionTarget) -> void {
    reductionTarget += p.getVelocity();
}, IteratorBehavior::owned, globalReductionTarget)
```

Figure A.2.: An example call to the reduce function in the AutoPas api.

```
template <bool parallel,
          bool ownershipCheck,
          bool regionCheck>
void _forEach(forEachLambda,
              ownershipTarget,
              particleRange,
              boundingBox) {

    if (not parallel) {
        serialForRange = particleRange;
        parallelForRange = {0, 1};
    } else {
        serialForRange = {0, 1};
        parallelForRange = particleRange;
    }

    parallel_for (i in parallelForRange) {
        serial_for (j in serialForRange) {
            index = parallel ? i : j;
            if ((not ownershipCheck) or ownershipTaret.equals(p.ownership)) {
                if ((not regionCheck) or boundingBox.contains(p.position) {
                        forEachLambda(particles[index]);
                    }
                }
            }
        });
    fence();
}
```

Figure A.3.: This is a pseudocode simplification for the internal forEach function in the *particleView*.

List of Figures

Bibliography

[Gho12] Jayshree Ghorpade. Gpgpu processing in cuda architecture. *Advanced Computing: An International Journal*, 3(1):105–120, Jan 2012.

[GKZ07] M. Griebel, S. Knapek, and G. Zumbusch. *Numerical Simulation in Molecular Dynamics: Numerics, Algorithms, Parallelization, Applications*. Texts in Computational Science and Engineering. Springer Berlin Heidelberg, 2007.

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

[GSBN21] Fabio Alexander Gratl, Steffen Seckler, Hans-Joachim Bungartz, and Philipp Neumann. N ways to simulate short-range particle systems: Automated algorithm selection with the node-level library autopas. *Computer Physics Communications*, 273:108262, 2021.

[LJ31] J E Lennard-Jones. Cohesion. *Proceedings of the Physical Society*, 43(5):461–482, sep 1931.

[nvi] Nvidia toolkit documentation: Unified memory programming. `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#um-unified-memory-programming-hd`. Accessed: 2022-01-03.

[TLGA+22] Christian R. Trott, Damien Lebrun-Grandié, Daniel Arndt, Jan Ciesko, Vinh Dang, Nathan Ellingwood, Rahulkumar Gayatri, Evan Harvey, Daisy S. Hollman, Dan Ibanez, Nevin Liber, Jonathan Madsen, Jeff Miles, David Poliakoff, Amy Powell, Sivasankaran Rajamanickam, Mikael Simberg, Dan Sunderland, Bruno Turcksin, and Jeremiah Wilke. Kokkos 3: Programming model extensions for the exascale era. *IEEE Transactions on Parallel and Distributed Systems*, 33(4):805–817, 2022.