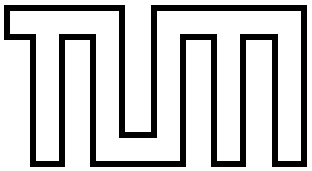


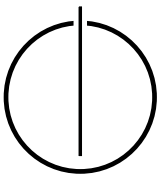
FAKULTÄT FÜR INFORMATIK

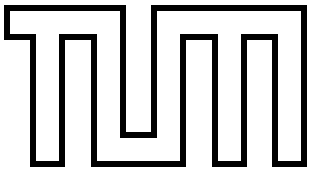
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Parallelizing Particle Simulations with Kokkos

Maximilian Geitner





FAKULTÄT FÜR INFORMATIK

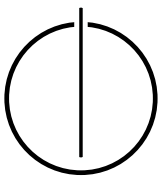
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Parallelizing Particle Simulations with Kokkos

Parallelisierung von Partikelsimulationen mit Kokkos

Author:	Maximilian Geitner
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisor:	Fabio Alexander Gratl, M.Sc.
Date:	16.08.2019



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

Munich, 16.08.2019 Maximilian Geitner

Acknowledgements

For this thesis, I appreciate the support from the people who supported me throughout this thesis. First of all, I want to thank the chair for scientific computing for offering me the possibility for this thesis and providing me access to the computation platform used in this work. Also, I want to express my gratitude to Fabio for his guidance, great explanations, helpful discussions and constructive feedback throughout this project. Last but not least, I want to thank my family for the continuous support during the thesis.

Abstract

The naive approach and the linked cell method are two approaches for computing short-range interactions such as the Lennard-Jones potential in the domain of MD simulations. In order to compute the interaction between particle pairs, it is necessary to parallelize the computations. Therefore, the usage of tools such as OpenMP or CUDA are quite useful for parallel executions of applications. However, each toolkit provides its own directives and requires the re-implementation of algorithms for each target platform.

This thesis describes the usage of the library Kokkos in the AutoPas project[1] from the chair of scientific computing at the Techncal University of Munich and compares it to already existing implementations in OpenMP. Kokkos is a programing model in C++ developed by the Sandia National Laboratory and focuses on the deployment of portable performance applications to all major HPC platforms. Kokkos provides its own data management and directives for parallelized execution which are modified during compile time and optimized for the specific target platform. Kokkos currently offers implementations for OpenMP, Pthreads and CUDA. AutoPas is an application which consists of many different configurations of traversal methods and other options, the goal is to select the most efficient configuration to calculate a given simulation consisting of particles.

Two different implementation strategies of Kokkos are tested with the execution platform OpenMP in AutoPas with several different configurations on the CoolMUC2, a linux cluster based on the Haswell architecture at the LRZ, and compared to an already existing application using OpenMP.

In the detailed analysis of the results, it is shown which weaknesses each Kokkos implementation has and how good the parallelization of the N-body algorithms work in practice. The performance is highly dependent on the chosen configuration, but there are cases in which Kokkos can compete with the native OpenMP implementation.

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

Zusammenfassung

Der naive Ansatz und die Linked-Cell Methode sind zwei Möglichkeiten sogenannte kurzreichweitigen Potentiale wie das Lennard-Jones Potential im Bereich der MD Simulation zu berechnen. Dabei ist es sinnvoll die Berechnung der Potentiale zwischen zwei Partikeln zu parallelisieren. Werkzeuge wie OpenMP oder CUDA sind zwar hilfreich für die Implementierung parallelisierbarer Applikationen, aber haben das Problem jeweils eigene Direktiven zu verwenden und setzen somit oft eine erneute Implementierung von existierenden Anwendungen voraus.

Diese Bachelorarbeit beschreibt die Anwendung von der Bibliothek Kokkos in Verbindung mit dem AutoPas Projekt[2] am Lehrstuhl für wissenschaftlichen Rechnen and der Technischen Universität München und vergleicht die Ergebnisse mit einer bereits existierenden OpenMP Implementierung. Kokkos ist eine C++ Bibliothek vom Sandia National Laboratory und ermöglicht portable Applikationen für sämtliche HPC Plattformen zu schreiben. Für diesen Zweck bietet Kokkos ein eigenes Datenmanagement und Direktiven zur Parallellisierung an, die dann zur Compilezeit an die jeweilige Plattform angepasst wird. Kokkos besitzt derzeit Implementierungen für OpenMP, Pthreads und CUDA. AutoPas ist eine Applikation, die aus einer Vielzahl an möglichen Konfigurationen zwischen Traversierungsmethoden und anderen Parametern wählen kann und eine Simulation berechnet. Das Ziel des Projektes ist es basierend auf einer Konstellation von Partikeln die optimale Strategie für eine schnelle Berechnung zu wählen.

Es werden zwei verschiedene Ansätze für die Integration von Kokkos in das AutoPas Projekt mit der Zielplatform OpenMP getestet. Die unterschiedlichen Konfigurationen werden auf dem CoolMUC2 am LRZ, einem Linux Cluster basierend auf der Haswell-Architektur, ausgeführt und mit einer existierenden OpenMP Implementierung verglichen.

In einer detaillierten Analyse werden die Ergebnisse auf Schwächen der Kokkos Implementierungen hin untersucht und geprüft wie gut die Parallelisierung mit Kokkos funktioniert. Das Ergebnis weist auf starke Schwankungen je nach Wahl der Parameter hin, manche Konfigurationen können fast mit der nativen OpenMP Implementierung mithalten.

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

Contents

1 Introduction and motivation

In particle simulations, the amount of particles used can range from a few thousand up to millions of particles. One step of the simulation usually consists of the following parts: Firstly, we have to calculate the interaction between all particles and update values like the force of the particle. Secondly, from these values the position or the velocity can be computed, which is done for each particle individually. Finally, the current state of the particles can be analysed and we could compute variables of interests such as the density in certain boundaries or how much time the computation needed. [Til87]

Modern numerical methods can be characterized by many parameters like the accuracy of the method, the reproducibility of the computation and their efficiency. [Rap97] Modern computer systems are usually multi-core systems and offer the possibility to run several tasks concurrently.

In the field of high-performance computing (HPC), there are solutions like OpenMP that can help building applications running on more than a single thread. Another possibility is the use of so called graphical processing units (GPUs), which are very efficient for the execution of many parallel tasks at once. OpenMP, OpenCl or CUDA are some popular programming toolkits and each solution has its own conventions to parallelize parts of the code. Therefore, the algorithm needs to be rewritten when the target platform changes.

In the last few years developers from the Sandia National Laboratory developed a library called Kokkos[1], which is written in the programing language C++ and provides abstractions for parallel execution of code and its own memory management. The goal of this project is to offer a portable performance application that can be deployed to major HPC platforms. Kokkos currently support the platforms OpenMP, Pthreads and CUDA. Depending on the target platform, the code will be modified at compile time and Kokkos own data structure can adapt its internal memory layout in order to increase the performance which can vary greatly between CPU and GPU layout.

This thesis focuses on the implementation of N-body simulations in the domain of molecular dynamics using OpenMP with Kokkos and compare it to a native implementation of OpenMP. The implementation is part of a project at the Chair for Scientific Computing and it is called AutoPas.

This application uses C++ and already supports different kinds of traversal methods to solve the N-body simulation, some parts used native OpenMP or CUDA features in order to increase the performance. The goal of AutoPas is to adjust the particle simulation depending on the distribution of the particles, for example it could switch between traversal options depending on the amount of particles, the distribution in the room or the selected options before the simulation started.

[1] https://github.com/kokkos/kokkos

2 Theoretical background

Particle Simulations can be deployed in many different fields of research like physics, chemistry or material science. Furthermore, the usage is not restricted to the microscopic level, therefore the definition particle does not make any specifications on its size and could also model stars, planets or galaxies in other application domains. A particle can be described with its physical properties such as mass, position, velocity or applied forces. Depending on the context of the simulation, certain attributes are enough to describe the particle properly. [GKZ07]

In this work, we specialize on particles in molecular dynamics simulations, one well-known interaction in this domain is the Lennard-Jones potential which applies a force to the particle depending on the distance to other particles. The force computation is only one part of a complete simulation step. In each simulation step, the computer calculates the interaction between each pair of particles. Then there are different ways to compute these interactions, the first described method is the naive approach which calculate the interactions between all particle pairs in order to solve the N-body simulation. Then there is the Linked Cells approach, which has the goal to reduce the interactions that needs to be considered.

2.1 Lennard-Jones Potential

Rahman carried out the first computer simulation of argon fluids in 1964 and used the Lennard-Jones potential for his studies. [GKZ07] Not only is the potential a good approximation for noble gases, it is also suitable for simulating liquid molecules and can be applied as a generic potential for qualitative explorations without referring to a specific material. [Rap97]

The Lennard-Jones potential approximates the force between two neutral atoms or molecules. It consists of one attracting and one repulsive component and only takes the distance between one particle at a time into account, the interaction with neighbouring particles are considered in another application of the Lennard-Jones potential at a different time. [Rap97]

Between all atoms exist the so-called van-der-waals force and it is based on the unequally distributed distribution of the electrons of the atoms. That weak attraction force applies to all atoms and molecules and has stronger effects at shorter distances. [Hun11] [BZB11]

This observation can be approximated with the formula:

$$U(r_{\mathrm{ij}}) = -4\varepsilon(\frac{\sigma}{r_{\mathrm{ij}}})^6$$

The distance between two particles i and j can be expressed as $r_{\mathrm{ij}} = \parallel R_{\mathrm{i}} - R_{\mathrm{j}} \parallel$, in which $\mathrm{R_i}$ and $\mathrm{R_j}$ are the respective positions of the particles.[GKZ07]

If the only component would be the van-der-waals force, we would have particles which would close the distance to each other until they reach the same position. In order to avoid

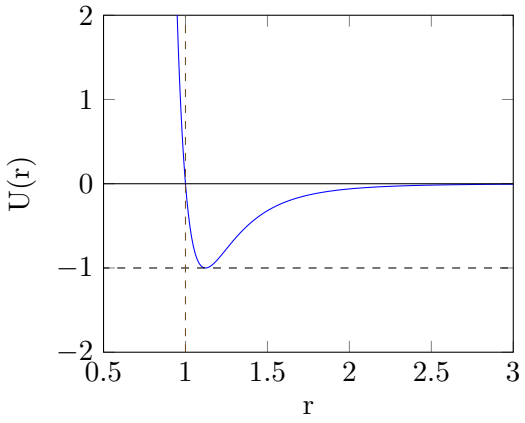


Figure 2.1: Lennard-Jones potential, $\sigma = 1$, $\varepsilon = 1$

a collapse of atoms in each other, there is a repelling component which must be stronger than the attracting force. The repulsive force is based on the effect when two electron hulls overlap and repel each other. [Cra04] This effect is called Pauli-repulsion and can be approximated by the expression:

$$U(r_{\mathrm{ij}}) = 4\varepsilon(\frac{\sigma}{r_{\mathrm{ij}}})^{12}$$

At close distances, the formula above has a stronger influence than the attracting component and as a result leads to a repelling force. At larger distances, the repelling component has only very little influence on the result. Another trait is the simple computation of the formula which is useful for the usage in simulations. [BZB11]

With the combination of both components, it results in the formula for the Lennard-Jones potential energy:

$$U_{\mathrm{LJ}} = 4 \cdot \varepsilon \left[(\frac{\sigma}{r_{\mathrm{ij}}})^{12} - (\frac{\sigma}{r_{\mathrm{ij}}})^{6} \right]$$

ε and σ depend on the materials used and ε is the depth of the potential-minimum and σ expresses the zero-crossing of the potential. [Hun11] Both values are shown in Figure 2.1 as dashed lines.

The conversion of the potential energy into the force $\mathrm{f_{ij}}$ requires the computation of the negative gradient of the potential U:

$$F_{\mathrm{ij}} = -\nabla U(r_{\mathrm{ij}})$$

After calculating $\mathrm{f_{ij}}$, the force $\mathrm{f_{ji}}$ that needs to be applied to particle j, can be determined by using Newton's third law:

$$f_{\mathrm{ji}} = -f_{\mathrm{ij}}$$

Thus, the interaction only needs to be computed once. [Rap97] In the following chapters, this optimization has the name "Newton3" and each traversal method can enable this option. This option is a trade-off between parallelization of the interaction calculation and the reduction of the computations. One the one hand, enabling Newton3 makes sense if there is

another way to parallelize the calculation and we prefer the way with the least amount of calculations. On the other hand, without this optimization we can run the calculations of the Lennard-Jones potential concurrently for each particle without risking multiple write accesses on variables from two different threads and preventing possible race conditions.

Another improvement can be derived from the Figure 2.1, in which large distances only have small influences on the result. Thus, only neighbouring particles are relevant for the computation of the Lennard-Jones potential, particles outside the cut-off radius r_{cutoff} can be neglected for the calculation. This optimization will be used in the linked cell method and this traversal method uses a modified formula with an additional cut-off parameter:

$$U(r_{\text{ij}}) \approx \begin{cases} 4 \cdot \varepsilon \ \left[(\frac{\sigma}{r_{\text{ij}}})^{12} - (\frac{\sigma}{r_{\text{ij}}})^{6} \right], & \text{if } r_{\text{ij}} \leq r_{\text{cutoff}} \\ 0, & \text{otherwise} \end{cases}$$

The cut-off optimization works well for short-range forces like the Lennard-Jones potential because ignoring particles far away only lead to very small errors in precision and can usually be neglected. In the domain of molecular dynamics, short-range forces play the major role on how the particles interact between each other. For other domains, long range-forces, such as gravitational forces, are more relevant and cannot be optimized with an cut-off radius. In these simulations, ignoring one interaction between planets very far from each other could lead to a completely different result. [BZB11]

2.2 Algorithms for N-body simulations

Each configuration of the MD simulation consists of N particles distributed across the simulation box and a mathematical description on how the interactions between two particles work. Additionally, there is the need to compute all interactions as efficiently as possible. The two used traversals in this work are the naive approach and the linked cells traversal.

2.2.1 Naive Approach

This naive approach will be referenced as Direct Sum Traversal in later chapters and contains N particles in one list. In the simulation step, this traversal method computes the interaction between each particle pair. That leads to $\frac{N \cdot (N-1)}{2}$ interactions that needs to be calculated. Depending on the Newton3 option, the traversal examines each pair once ot twice.

The run-time of this traversal amounts to a run-time of $\mathcal{O}(N^2)$ which is sufficient for a small amount of particles. Advantageous is the small overhead compared to other approaches. However, this approach is too slow for a large amount of particles and can only be parallelized by disabling the Newton3 option.

2.2.2 Linked Cells Traversal

The quadratic run-time of the Direct Sum Traversal is sub-optimal when calculating the interactions between millions of particle. Therefore, there is the need for further optimizations in the approach.

The linked cells method divides the simulation box into smaller cells, which are part of a regular sized lattice of the dimension $M \times M \times M$. The lattice does not need to be regular,

but for simplification we assume this for the applications of the linked cell method in this work. The length of the side of a cell is defined as $l = L/M$ with L the length of a side of the simulation box.

Before the use of the linked cells method, each particle needs to be assigned to a particular cell which stores all particles in a list. During each step of the simulation, the traversal calculates all interactions between particles within the same cell.

Later, the traversal computes the interactions between particles of two neighbouring cells. In the three-dimensional space, a cell has 26 direct neighbours because cells further away can be immediately excluded due to the constraints on the length of a cell which is at least as long as r_{cutoff}. If the length of the cell would be shorter, we would need to calculate interactions between non-neighbouring cells, which would make the calculation process more inefficient as we have to consider more combinations of cell pairs. One example for two dimensions is displayed in Figure 2.2 and illustrates the relationship between the cut-off radius in red and the neighbouring cells marked in blue.

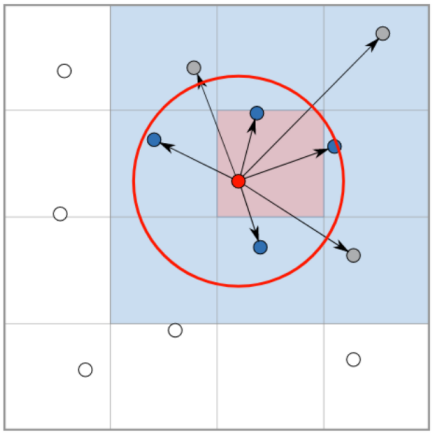


Figure 2.2: Linked Cell Method
Source: [GST+19]

In scenarios with evenly distributed particles across all cells, we can approximate the number of particles with the formula $N_c = \frac{N}{M^D}$. N stands for the total amount of particles in the simulation, M for the cells in one dimension and D for the amount of dimensions in this simulation, usually D = 2 or D = 3 is chosen.

The run-time differs from the quadratic run-time of the naive approach. In the linked cell traversal, we can assume a constant number of particles in the neighbourhood of one particle. For this particle, the traversal can compute all interactions in constant time. If we scale it up to N particles, the result is a run-time of $\mathcal{O}(N)$ and much better than the quadratic run-time from the naive approach.

But in cases with hotspots, the linked cells method degrades to checking all particle interactions again and therefore to the naive approach. Furthermore, after each simulation step the particles might have moved to another cell and has to be moved to another list. According to Allen, the linked cells method should not be used for small structures with around 1000 particles, because the overhead to maintain the structure is higher than computing all particle pair interactions. [Til87]

3 Description of Tools

This thesis is part of the AutoPas project at the chair of scientific computing which is a tool for auto-tuning particle simulations. The main aspect is the usage of the Kokkos Library developed by the Sandia National Laboratory, which offers a new approach to deploy parallelized instructions to different platforms with one code base. The execution of the experiments were done on the Haswell Nodes at the LRZ in which several configurations of the simulation were tested.

3.1 AutoPas

AutoPas consists of several layers of abstraction and provides a architecture which allows to integrate new traversals, containers or particle types. AutoPas is written in C++ and can be deployed to x86 architecture platforms, a CUDA implementation for certain algorithms can be found in the project, too.

The main aspect of AutoPas is the auto-tuning feature of this simulation program. AutoPas can switch between different containers, traversals or a newton3 option, the goal is to select the most efficient configuration in order to compute the simulation. Some traversals only work with a specific container, therefore certain configurations are restricted to use in the same configuration. In following subsections, the different layers of AutoPas will be explained.

3.1.1 Particles and Cells

On the lowest layer of AutoPas is the particle class, which consists of the properties of a single particle. The particle in AutoPas has information about its position, velocity and the currently applied force. The data is stored in arrays of the data type float or double, which can be modified by setter-methods or requested by getter-methods for each attribute.

The particles can be stored in lists, one such class is the FullParticle class that is simply a vector of the c++ standard library. The cell offers methods for the insertion of new particles and provides the particle vector for the use in later stages of the traversal.

3.1.2 Containers and Traversals

In AutoPas, there are three different types of containers: The first one is called direct sum and represents the implementation of the naive approach. The second type is the verlet list container in which each particle has access to a neighbourhood list with nearby particles. The third type is the linked cells container, that consists of many cells in which each particle is stored. In this thesis, only the direct sum and the linked cells containers were used fro the Kokkos implementation.

On each container, it is possible to apply a traversal method which processes the interactions between the particles. The traversal uses a so-called CellFunctor class that can be configured with an Functor class object which calculates the particle pair interaction, e.g. the Lennard-Jones potential. The CellFunctor class provides two methods for processing the particles within a selected cell or between particles in two different cells.

3.2 Kokkos

In this thesis we use Kokkos for MD simulations and use some of its tools that can be used for parallelized execution of code blocks. For this purpose, Kokkos provides its own data structure, so-called Views, which can store primitive data types in vectors, matrices in two to up to eight dimensions. [1] Kokkos has its own directives for declaring parallelized loops and allows to declare hierarchies in loops such as the definition of teams threads.

3.2.1 Memory Space in Kokkos

Kokkos has certain limitations on parallelized loops: It only allows to read from primitive data types and the use functions which are declared with the qualifier "const". These constraints do not affect Kokkos Views, which can only be modified in parallelized loops. The Kokkos View resides in the so-called execution space and the storage location depends on the execution platform. All other data is stored in the host space and requires deepCopy-operations for transfers to the execution space.

Kokkos Views can be created by providing a data type to the template, it works with all primitive data types, structs of built-in data types and classes which fulfil certain limitations like the existence of a default constructor, deconstructor and some requirements on method signatures. Additionally, the construction of a View needs the size of each dimension. After initialization, these parameters can be modified with a resize operation which copies the old data to the new constructed View if possible.

With regards to already existing projects, the switch to Kokkos Views can be a large burden and seems quite exhausting at the beginning. But the reason for Kokkos Views is the adaption to the target platform, which could be CPUs or GPUs. Views change the internal storage layout depending in which execution space it is used. This change can prevent cache-misses, which would happen by using the wrong layout in the wrong execution space. For each execution platform there is already a default layout specified, but it is also possible to manually change the layout structure.

Data transfers between data in the regular memory space and the Views can be achieved with the deepCopy-operation. Its purpose is to transfer data in one direction, either from or to the Kokkos View. Each transfer begins with the declaration of a so-called host mirror, the initialization requires one Kokkos View as input parameter. As a result of the initialization the host mirror has the same dimensions like the Kokkos View and a deepCopy-operation can be performed between each other. For transfers to the Kokkos View, the information can be copied to the host mirror by a value assignment at the correct location, it is later copied to the execution space by the deepCopy-operation. The reverse process begins with

[1] https://github.com/kokkos/kokkos/wiki/View

the deepCopy-operation to the host mirror. After the transfer, the data can be extracted from the host mirror.

3.2.2 Parallel Dispatching in Kokkos

Depending on the task, we can choose from a variety of parallelized loop types in Kokkos. Depending on the workload, Kokkos decides by estimating the amount of workload, whether the loop should be run concurrently or only run in sequential order. The sequential execution is useful when the task would cause more overhead with parallelization than just running it on one thread.

Parallel Operation Name	Description
parallel_for	Implementation of a "for-loop" with independent iterations
parallel_reduce	Implementation of a reduction
parallel_scan	Implementation of a scan operation

Table 3.1: Overview of Parallel Operations in Kokkos

An application of the reduce operation would be the calculation of the dot product between two vectors. The parallel scan operation works like the reduce-operation, but would return a result after each iteration. One example for the usage of the scan-operation is the calculation of the prefix-sum or the factorial function.

For each loop, there are several possibilities to declare this loops. [2] The two options are the declaration as a functor with an operator() method in a structure or the implementations with lambda expressions which were quite simple to understand. An additional parameter can specify how Kokkos should handle nested loops [3]. We can define thread teams, a collection of threads sharing a so-called scratch pad and other types of nested loops. One important qualifier in this work is the MDRangePolicy, which is suitable for tightly nested loops.

3.2.3 Deployment to Target Platform

Currently Kokkos offers the following target platforms: Serial, OpenMP, PThreads and CUDA. As the name suggests, Serial does not parallelize and runs on only one thread. OpenMP and PThreads make use of multi-core systems. The setup is quite easy and only requires recent compiler which support OpenMP.

The execution on CUDA devices, GPUs from Nvidia from the Fermi generation onwards, is compared to the other platforms more complicated to use. Kokkos provides a Wrapper which in combination with the CUDA toolkit and a compiler modifies the code during compilation. The Wrapper adds directives to Kokkos loops in order to make the execution on CUDA devices possible. Furthermore, it is required to specify the generation of the device, e.g. Kepler, Maxwell, etc. at compile time.

[2] https://github.com/kokkos/kokkos/wiki/ParallelDispatch
[3] URL: https://github.com/kokkos/kokkos/wiki/HierarchicalParallelism

3.3 Computation Platforms

For MD simulations, it is quite interesting to measure the performance on the Kokkos library compared to our previous OpenMP implementation of the naive approach and the linked cell method. OpenMP is an API for multi-core systems and can be used in the field of HPC. We have decided to run the following experiments on the CoolMUC2 from the LRZ.

3.3.1 CoolMUC2 / Haswell Nodes

The CoolMUC2 is a linux cluster at the LRZ in Garching, which provides the service to assign jobs to a node for a limited amount of time.

Many different configurations of the Kokkos implementation and the native OpenMP implementation will be evaluated, the only similarity is the Haswell Node. The node has got two Xeon processors with 14 cores each, which amounts to 28 threads. The Xeon processor can execute AVX-2 instructions, that means it can vectorize parts of the code. However, the vectorization is dependent on the compiler and has problems in optimizing Kokkos instructions.

Feature	CoolMUC2
Number of Nodes	384
Processor	Intel Xeon E5-2697 v3 ("Haswell")
Cores per Node	28
Threads per Core	2
Frequency [GHz]	2.6
Vector Instruction Extension	AVX-2
RAM per Node [GB]	64

Table 3.2: Overview of the technical details of the used platforms. Data mainly taken from: https://www.lrz.de/services/compute/linux-cluster/overview/ and taken from: https://doku.lrz.de/display/PUBLIC/CoolMUC-2

We conducted all experiments on only one node and had therefore 28 cores to use. All jobs were configured with the environment value OMP_NUM_THREADS set to the value 28 and the executables for Kokkos and the native OpenMP application were created with the compiler Clang in the version 6.0.1.

4 Kokkos in AutoPas

Kokkos claims to be easy to use and as efficient as native implementations like OpenMP and CUDA. For that reason, we have rewritten our OpenMP implementation and used Kokkos Views and its parallelization methods. Maintaining code is often a exhausting task, another goal was the re-use of existing classes where possible and to avoid changes to the structure of AutoPas.

In the next two sections, we propose two different implementations which integrate the Kokkos Views into AutoPas and test the performance in the subsequent experiments. The first approach uses the Views as attributes of the Particle class. The other one initializes one Kokkos View per cell and transfer the particle attributes to the execution space before the traversal. The information will be transfered back to the individual particles after the traversal performed all its calculations.

4.1 Storage of Views in the Particle Class

This implementation revolves about the replacement of the Particle class with an class using Kokkos Views. The idea is to store all attributes directly in the execution space and save overhead costs for copy operations before and after the traversal. These new attributes also dictate changes in classes of the upper layer, in particular some traversal classes and functor classes required changes in order to use Kokkos parallelized loops and data types. Some changes also apply to the second Kokkos implementation.

4.1.1 Kokkos Particles

The class on the lowest layer of AutoPas is the ParticleBase class. Its only function is to store the position, force and velocity of one particle and provide getter and setter-methods for each member. All specialised classes inherit from this base class, e.g. a molecule class or in this case the KokkosParticle class.

In addition to the data types from the base class, the KokkosParticle declares for each attribute a View which in principle contains the same information as the previous ParticleBase class. This change has a few drawbacks: The first one is the increased usage of memory, which can be a huge problem for the simulation of very many particles. Furthermore, the getter- and setter-methods need to be changed. All setter-methods need to be overwritten due to storing the new values in the Kokkos Views.

Due to the introduction of new variables, the KokkosParticle class offers new methods for all attributes. Each attribute requires an additional getter- and setter-method, additionally the setter-method performs a deepCopy-operation which makes the information in the Views for code blocks run in the host space available. This cost-intensive approach has to be done because some parts of AutoPas require informations about the particle to perform sorting operations, e.g. placement in the correct cell of the linked cell method. The transfer to

the non-kokkos memory space in the particular getter-method cannot be done because all getter-methods are marked with the "const" qualifier and cannot return local attributes from a deepCopy-operation.

4.1.2 Particle Cell

Regarding re-use of code in AutoPas, the particle cells did not require any changes to fit the particles with Kokkos Views. The particle cells are based on a particle template and the only modification when using particle cell classes is the declaration which type of particle class should be stored. For testing our Kokkos implementation, we store the particles in the FullParticleCell class which only has a vector as attribute. This vector adjusts itself depending on the amount of particles in this particular cell. In the linked cells approach, after each simulation step the positions of particles can change and usually we cannot plan with a fixed amount of particles in a cell. Therefore, dynamic particle sizes are necessary for cells.

4.1.3 The Lennard-Jones Functor and the Cell Functor

A functor in AutoPas describes the interaction between a particle pair. For the use of Kokkos in a MD simulation, the Lennard-Jones potential, as described in Section 2.1, is useful to verify the use of Kokkos in this research field. In AutoPas, there is already an implementation for the Lennard-Jones functor which is also used for OpenMP calculations.

The Lennard-Jones functor is the corresponding class that calculates the forces between two particles based on the Lennard-Jones potential. As this interaction needs to be computed between many pairs of particles efficiently and uses the views from Kokkos, this class needs access to the position to compute the distance between two particles. Later the functor applies the changes onto the force attribute of the particles.

4.1.4 The Cell Functor

The KokkosCellFunctor class is responsible to handle a functor from Subsection 4.1.3 and provides two methods, processCell() and processCellPair(), which can be used by the traversal methods in Section 4.3. Both methods receive an appropriate input, one or two cells, and calculate all interactions fulfilling the cu-off criteria within a cell or between both cells. The cell functor processes the task based on the currently selected Newton3 option and the used Kokkos implementation approach, these options decide whether some code segments are executed concurrently and which function from the functor class should be called. Due to differences in the data layout between both Kokkos implementations, each approach uses its own implementation of the Lennard-Jones functor.

After first experiments with the Kokkos implementations, the methods processCellOneThread() and processCellPairOneThread() were added to the KokkosCellFunctor class. These methods are quite similar to the methods in the paragraph above, the only exception is the strict sequential execution within a cell or in the calculation between two cells. The sequential execution is useful in certain configurations of the second Kokkos implementation, which did not perform well in scenarios where the parallelization of tasks was already performed in another part of the traversal.

4.2 Storage of Kokkos Views in the Cell Class

The second approach stores the data of the particles and only transfers the information to the Kokkos execution space for computations during a traversal. The only reusable class is the Particle Class, other classes require a few modifications such as new variables, method overrides or the implementation of new methods. There is another data layout in AutoPas with the name "Structure-of-Arrays" (SoA), which is based on a similar idea and also transfers data between the particles and another separate memory space.

4.2.1 Views in the Cell Class

Before we apply a traversal method, we have to initialize Kokkos data structure and fill it with the current information of the particles. We define for each cell, a container with particles in this part of the simulation, one Kokkos View as a three-dimensional matrix. The first index selects a particular particle, the second whether it is a position, force or velocity attribute, the third index is for the respective value in this dimension.

Each traversal class has the two methods initTraversal() and endTraversal(), which we will override for each traversal method and copy our data to the View or back to the host space. These two steps could cause a large overhead because each copy operation is done in sequential order and can take some time for many cells and particles.

4.2.2 Modifications in other Classes

As mentioned in Subsection 4.1.3, the KokkosCellFunctor class and the Lennard-Jones functor were modified in order to work with the second implementation approach. For the usage of the second implementation, we have to initialize all Views before the traversal and transfer the computed results back to the particles at the end of one traversal iteration. Thus, each traversal method in Section 4.3 needs to implement this functionality.

4.3 Traversals

Traversals have the task to select the order in which the interactions between the particles are calculated. Depending on the method we have to handle multiple cells and respectively the interactions between these cells. For testing the Kokkos library, we will first use only one cell and apply the naive approach on it. The other traversal is based on the linked cells method mixed with a new method which enables us to build clusters of cells and parallelize the computation of several non-overlapping clusters.

4.3.1 Direct Sum Traversal

The direct sum traversal is based on the naive approach from Subsection 2.2.1. We will define a cell which contains all particles of the simulation and a second halo cell without any particles in this cell.

The run-time of the direct sum traversal is naturally $\mathcal{O}(N^2)$ due to its similarity with the naive approach.

Initializing a direct sum traversal object needs the particle functor, in this case it is the Lennard-Jones functor adapted to use the Kokkos implementations. The functor is wrapped

in the KokkosCellFunctor from Subsection 4.1.3 which simplifies the usage of the functor for all traversal methods. The methods processCell() and processCellPair() are used in the computation phase exactly once, it is applied on the only cell which contains all the particles and for the interactions between the regular and the halo cell.

4.3.2 C08 Based Traversal

The C08 based traversal is from a paper in 2015 [TSH+19] . In this approach we divide the cells of the linked cells methods and the surrounding halo cells in clusters, usually of the size $2 \times 2 \times 2 = 8$ and compute the interactions between some cells of the cluster. We have a base cell in the cluster and only compute the interractions with cells that are in the cluster and are directly next to each other. This approach links a cluster of cells with a colour and the purpose is to calculate clusters of the same colour concurrently. It's only possible when clusters with the same colour do not overlap.

Consequently, we can parallelize all cluster calculations with the same colour and use the advantage of the linked cells methods. The Kokkos implementation parallelizes over three dimensions within the computation of one colour with a so-called MDRangePolicy provided by the library. The MDRangePolicy is a tightly nested loop and iterates over three dimensions in this case. Compared to other loop types such as Team Threads, Kokkos could parallelize the workload most efficiently with the MDRangePolicy.

4.4 Analysis

In this section, we look more into the performance of Kokkos in AutoPas. All test scenarios involve the native OpenMP implementation which is part of the executable MD-Flexible. MD-Flexible uses AutoPas and can initialize the simulation with program arguments such as which traversals should be used, how many particles and in which layout should they be placed into the simulation and many other options. In the experiments, we apply the same options to the application which uses Kokkos together with AutoPas.

All test scenarios arrange the particles in a three-dimensional grid, which has exactly the same amount of particles in each dimension. But the spacing between the particles might vary between experiments and also the simulation size depends on the experiment. The Lennard-Jones functor is always defined with the value $\varepsilon = 1$ and $\sigma = 1$ and $r_{\text{cutoff}} = 1$. In each test scenario, we will have one run with the Newton3 option enabled and one run without the Newton3 option. With an activated Newton3 option we can save exactly half of the force calculations. However, the drawback is the inability to parallelize the interaction between the involved particles and a potential loss of performance due to the decrease of multi-core performance.

Each experiment will execute the configurations ten times in a row and calculate the average time of the experiments by dividing the sum of all iterations by the amount of runs. The reason for several runs is based on potential deviations in run-time and in usual MD simulations, the force calculation is applied more than once.

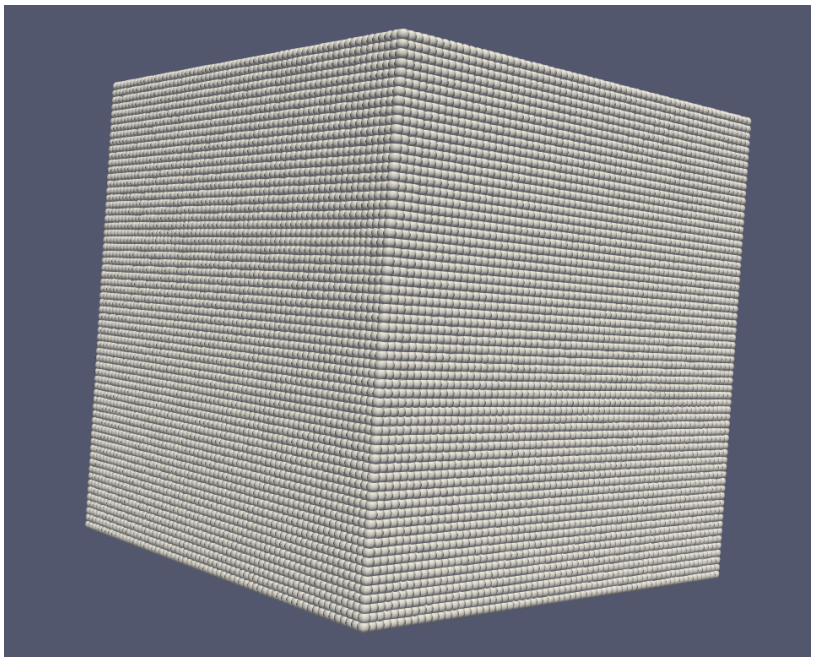


Figure 4.1: Particles in a Grid Layout

4.4.1 Comparison Direct Sum Traversal and C08BasedTraversals

Only the first experiment involves two different algorithms, which will run with between 125 and 64,000 (40^3) particles in the simulation. The spacing between particles is set to the value of 0.4, which are around 16 particles per cell. Allen mentioned in [Til87] that the naive approach outperforms the linked cells method with small structures of around 1,000 particles. He argues the overhead is causing difficulties for the linked cell method.

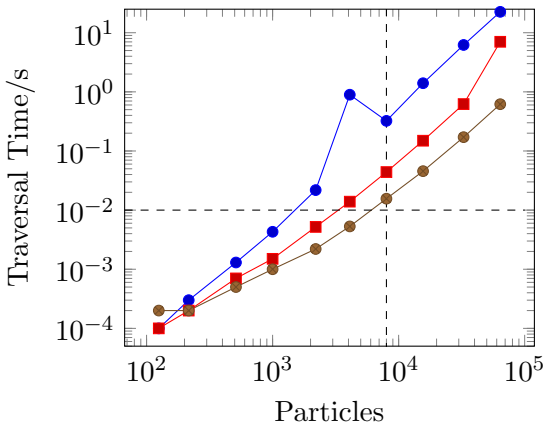


Figure 4.2: Direct Sum Traversal, Newton3 off

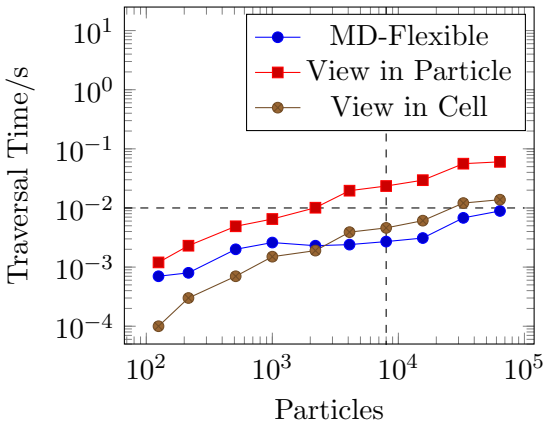


Figure 4.3: C08BasedTraversal, Newton3 off

In the case Newton3 is not enabled, the linked cell method is recommendable, apart from the View in Particle approach, for simulations with 8000 particles or above. The

Figure 4.2 shows that the MD-Flexible implementation seems rather slow compared to the other versions. The reason is the usage of only one core in MD-Flexible, the other approaches used all 28 available cores although they might be underutilised. The CellView approach is in most cases the better solution than the ParticleView approach, the cache locality is much better in the first mentioned method.

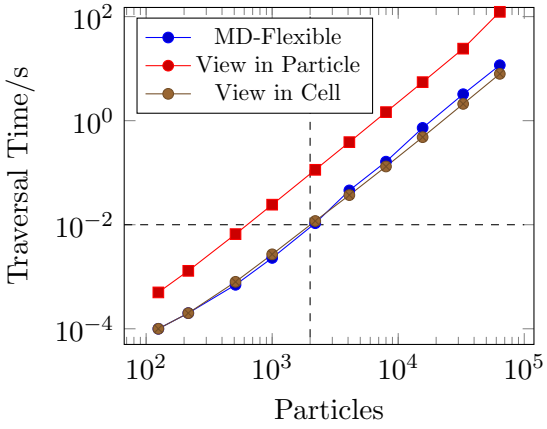


Figure 4.4: Direct Sum Traversal, Newton3 on

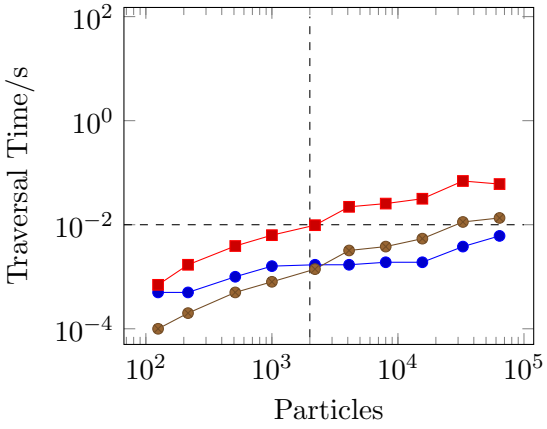


Figure 4.5: C08BasedTraversal, Newton3 on

In Figure 4.4 and Figure 4.5, we enabled the Newton3 option and observed similar results as before. After 8000 particles, every C08Based Traversal outperformed all tested Direct Sum implementations. However, all results from Figure 4.4 could not parallelize the traversal and could only use one thread. As mentioned in Section 2.1, the enabled Newton3 cannot run concurrently due to possible race conditions.Thus, the performance of the Direct Sum Traversals decreased compared to the disabled Newton3 option results in Figure 4.2

4.4.2 Domain Size

As we have seen in the first experiment, the naive approach does not work well for large amounts of particles. The next experiment focuses on the domain size of the simulation, in this case we adjust the domain size and the amount of particles, therefore maintaining a constant density in the range of 125,000 (50^3) and 8,000,000 (200^3) particles.

We assume the linked cell method scales linear in time with the amount particles, that means the increase of the particle amount by the factor two should double the amount of time for one iteration. As in the experiment before, we set the particle spacing to the value 0.4 and run all algorithms with both Newton3 options.

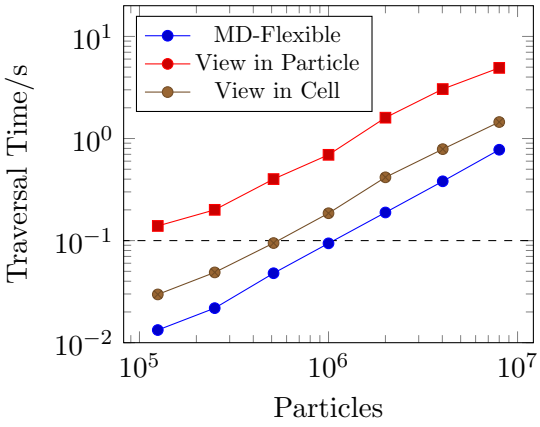


Figure 4.6: C08BasedTraversal, Newton3 on

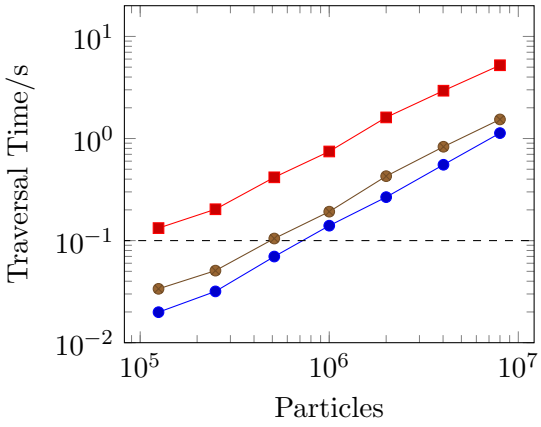


Figure 4.7: C08BasedTraversal, Newton3 off

The blue results belong to the MD-Flexible run-times and are in all tested configurations below the other competitors. The Newton3 option does not affect the Kokkos algorithms by much, but the enabled Newton3 option reduces the time of computation by 25% for MD-Flexible. When Newton3 is disabled, the best Kokkos method needs 50% more time than native OpenMP implementation. A little worse is the difference with an enabled Newton3 option, in this case Kokkos is two times slower across all sample points due to the better performance of MD-Flexible.

All algorithms scale with the input size in linear time which is to be expected for the linked cell approach. During testing, there was an older version of the CellView approach which parallelized inside the calculation of a cluster in case Newton3 was disabled. The results were drastically worse than the current iteration and took 20 seconds for one traversal with 1,000,000 particles. The newer version only parallelizes several clusters and not within the cluster and only took around 0.4 seconds until completion. This results suggest that Kokkos performs worse when trying to parallelize too many tasks and could lead to a worse cache access for the calculations. The current iteration of the ParticleView approach uses the parallelization within a cluster, but the results did not change significantly when the cluster calculation was performed sequentially. Kokkos might optimize the loops and disallowed the parallelized execution in this case or the access to the particle data was equally bad in both approaches.

4.4.3 Density of Particles in Cell

In the two experiments before, we always assumed a fixed particle spacing of 0.4 and resulting in around 16 particles per cell for the C08BasedTraversal. In the next experiment, we will change the particle density and look how the algorithms can cope with the resulting configurations. For all runs, the amount of particles is fixed at 2097152 (128^3) particles with 128 per dimension. We will have an average between four and 32 particles per cell, these configurations are achieved through the change of the particle spacing value. The value of r_{cutoff} stays at one and we run all algorithms with different Newton3 options again.

When we have the same amount of particles close to each other, the simulation box is divided into fewer cells and consequently we can parallelize fewer clusters during the

calculation of one colour in the C08Based Traversal. As a result, we have to assume we cannot create enough tasks for each thread and do not use all available computational power. Furthermore, with more particles per cell we have to consider more interactions and another reduction in performance due to more workload.

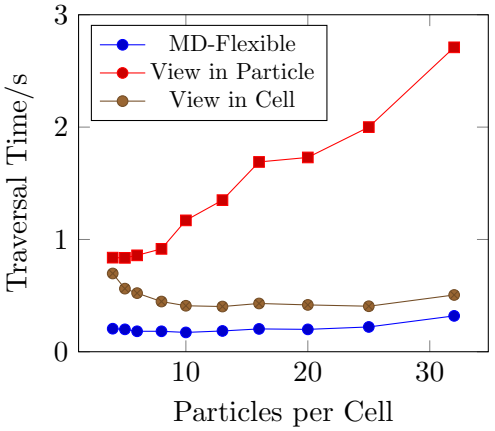


Figure 4.8: C08BasedTraversal, Newtown3 on

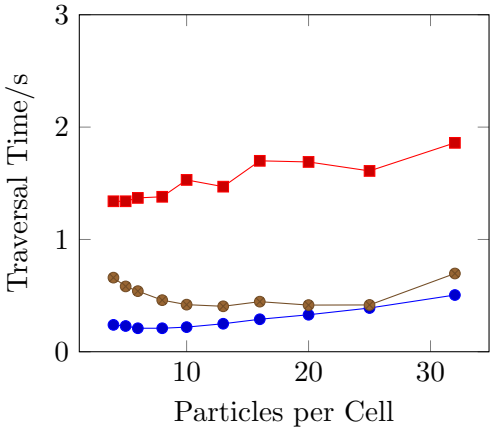


Figure 4.9: C08BasedTraversal, Newtown3 off

As in the previous results, the native OpenMP implementation wins again and outperforms both Kokkos solutions. More interesting, the CellView approach handles large particle spacings, which result in the usage of many cells, not as good as expected. That observation seems very strange due to the hypothesis the workload should actually increase with more particles next to each other and the reduced capability to parallelize. The cause for this behaviour is analysed in Subsection 4.4.4.

4.4.4 Overhead in Copying Data

One problem with the Kokkos View seems to be the initialization of its view, which might cost much time. That is the reason we do a measurement of the initialization view and after the computation we measure again how much time it took to copy the data from the Kokkos Memory Space back to the particle objects in the Host Space. The results from Table 4.1 were done with an earlier implementation of the CellView approach. The original performance is shown in Figure 4.10 and compares the current implementation and the previous algorithm with an worse initialization phase.

All experiments involved the initialization of a simulation with 128 particles per dimension in a three-dimensional grid layout, the measurements involved the creation of a new Kokkos View and the transfer from data of the host space to the View. We excluded the step to calculate the interactions between the particles and skipped to the last remaining part of a traversal, the transfer back to the Host Space. This procedure was executed five times in a row and in the following table are the records of the average times of the first and last step of a traversal.

Particle Spacing	Average Particles per Cell	Cells per Dimension (+Halo Cells)	Initialization Time Average	Finalization Time Average
1.5	0.30	194	3.16	0.27
1.25	0.51	162	2.95	0.20
1	1	130	2.89	0.15
0.75	2.37	98	1.15	0.09
0.5	8	66	0.37	0.04
0.25	64	34	0.09	0.03

Table 4.1: Results Spacing and Overhead

The results from Table 4.1 displayed a disastrous performance in the initialization phase of the traversal, which could explain the bad results when the spacing between particles was very large. Another similar observation from the ParticleView implementation suggested a similar reason due to the slow initialization of the simulation before the traversal was even applied. These results did not show up in the diagrams as there were no measurements of the AutoPas initialization. The average time for the finalization phase seems to be fine on the other hand. In order to resolve the initialization problem, the creation of a new Kokkos View should be resolved.

Instead of a initialization of the View, we can also adjust the existing Kokkos View to the current space needed to store all particle attributes. Readjustments are quite inexpensive compared to a reinitialization, especially in cases where the particle size of the cell did not change. This approach does not help us in the first iteration of the traversal where we still have an uninitialized view, but subsequent iterations can improve the first phase of the traversal by readjusting the View. Another optimization to cut down the run-time were cases in which we did not have particles in the cell. With zero particles in a cell, there is no need to readjust the Kokkos View and we could save some additional operations.

With the collection of these information above and the refined initialization sequence, we re-evaluated the experiment. As the initialization in the first iteration took significantly longer, we show the run-time for the first iteration as an own result and all following iterations as an average time. The finalization times did not change much and were in the area of measurement errors.

Particle Spacing	First Initialization Time	Other Initialization Time Average
1.5	5.94	0.26
1.25	5.97	0.17
1	5.94	0.13
0.75	2.45	0.07
0.5	0.80	0.04
0.25	0.95	0.02

Table 4.2: Results Spacing and Overhead

It is noteworthy that the refined version actually performs worse in the first iteration

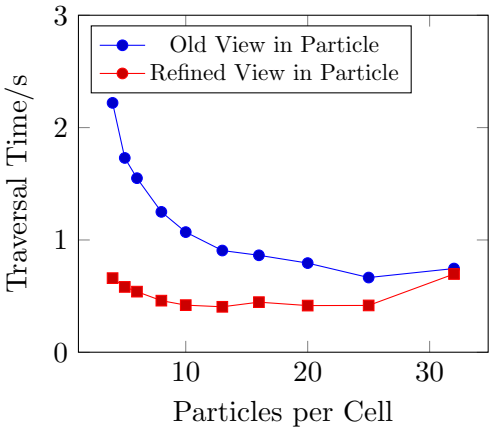


Figure 4.10: CellView Comparison, C08BasedTraversal, Newton3 on

compared to the values in table 4.1 and there are still options to improve this process. But for the other iterations, the performance increase is quite noticeable and should always be considered when using multiple traversals. The Figure 4.10 shows the old density calculation and the refined version of the CellView implementation.

4.4.5 Problems in the Integration of Kokkos

One of the most complicated task was naturally the integration of the Kokkos memory space into the project. One of the first ideas were to put objects of classes into the Kokkos View and apply a loop on the existing algorithm. Since this approach did not work out, we implemented the core elements of AutoPas in another project exclusively to try the Kokkos Views. The first working simulation used the modified KokkosParticle class with Kokkos Views as attributes and implementations of the Lennard-Jones functor and the Direct Sum Traversal adapted to Kokkos.

Regarding performance and scalability, this test environment could not give many insights about the choice to store all information separately for each particle. After first initial tests, we decided to integrate all the components into AutoPas. Because the Particle with the Kokkos memory space was using another underlying data structure, the usage of Kokkos was not integrated in the executable MD-Flexible which handle all currently usable options. All test configurations launch with another executable, similar to MD-Flexible, in the AutoPas project and can adjust similarly as the other program the available parameters regarding simulation generation.

In a project with only Kokkos as a main tool, the transfer between host space and memory space would only be needed for a few parts of the implementation. However, AutoPas is a system with several different traversal methods using CUDA, OpenMP or other instruction sets and needs to be compatible to all systems.

Another hurdle is the correct use of the parallelization techniques Kokkos provide. The basic idea Kokkos conveyed in their introductory slides to the library was how easy it was to use and how Kokkos could choose for the amount of loops the correct type of execution, which could sequential with low overhead for few iterations compared to the concurrent

executions with potentially large overhead. One problem was however, in the C08based approach it might have been a good idea to parallelize with the disabled Newton3 option and many particles in a cell. Unfortunately, cache locality was another important criteria for the execution of programs and needs to be considered when using parallelization.

During the implementation phase, we experimented which nested loop would be good for the MD simulation. As a result, nested TeamPolicy and TeamThreadRange were not the best options for the C08Based approach, but the suitably named MDRangePolicy. The MDRangePolicy is a tightly nested loop and iterates over multiple indices, in the end it was the perfect solution for using to run different clusters concurrently.

Another aspect regarding the execution of Kokkos in other execution spaces, it did not run out of the box as we imagined. The Makefiles from the tutorial made the use of CUDA quite easy, but with the integration of Kokkos into AutoPas via CMake it turned out more complicated than expected. For the compilation with CMake, we have to use the so-called NVCC_Wrapper provided by Kokkos and compile it with this script which would replace all Kokkos pragmas with suitable CUDA pragmas. Additionally, there are special traits to consider regarding the memory space and thread level rules when they are used with CUDA. In the worst case, the execution could feel like a sequentially run program.

The last point to mention is the use of Kokkos in a bigger environment. For testing on a local machine with an modern notebook processor (i7-8565U), the tool vTune by Intel showed a quite promising use for parallelization. It claimed to use between around 7 threads of the 8 threads usable on this machine. The vTune results from the CoolMUC2 were quite different and are shown in Figure 4.13 and Figure 4.12 for the MD-Flexible and for the CellView implementation. Most multi-core capabilities were unused most of the time and might be due to unbalanced dirstribution of tasks. The Kokkos implementation could often create enough tasks for 24 threads at once, but in almost no cases did the simulation use all available resources. MD-Flexible performs different compared to Kokkos and uses either all threads or only very few threads.

Another criteria for HPC environment is the capcbility to vectorize instructions. Tests on a local machine with the compiler GCC 8.3 discovered a problem with the current setup. The tool vTune detected that Kokkos code did not vectorize any instructions of the MD simulation. Retries with CMake options like the KOKKOS_ENABLE_AGGRESSIVE_VECTORIZATION flag did not help although the notebook is capable to execute at least instructions from the AVX-2 instruction set. Results with the Clang compiler or the Intel compiler may differ and show different results.

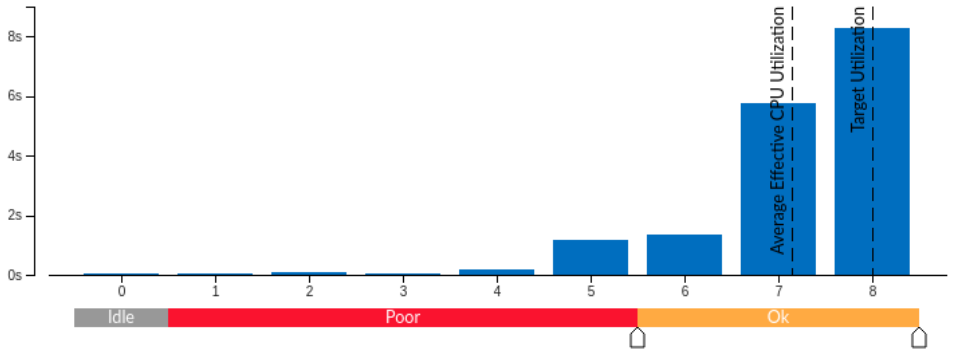


Figure 4.11: Kokkos CellView Thread Utilization: 8 threads

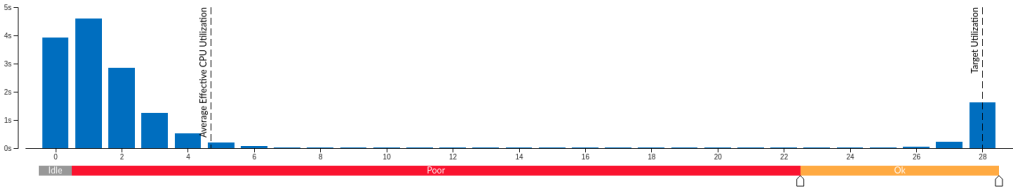


Figure 4.12: MD-Flexible Thread Utilization: 28 threads

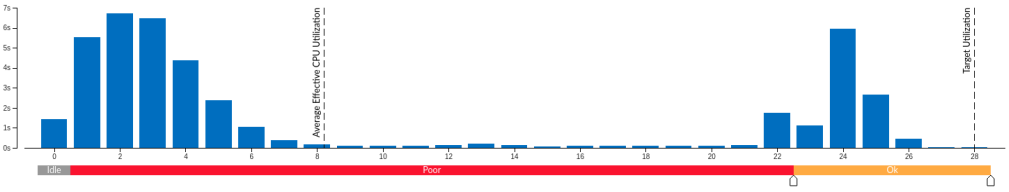


Figure 4.13: Kokkos CellView Thread Utilization: 28 threads

5 Conclusion

5.1 Summary

Kokkos provides a portable library and has definitely more potential than the results from the experiments showed. We have gained a lot of insights about the usage of Kokkos Views and different types of parallelization. We tested two different storage layouts for Kokkos, the first one stored all attributes in three views per particle. The other one only transferred the data to the View in case a traversal calculation was called, which resulted in two additional steps with more overhead to worry about.

As it turned out, creating too many separate views takes much time and can decrease the actual run-time quite drastically. Furthermore, the non-existing cache locality could be another reason this method performed quite poorly. Despite the only approach which parallelized within the clusters, it did not perform better or worse and could be explained by an optimization strategy by Kokkos which disallows too many thread levels and replaced the sequence by an sequential execution.

The second implementation with only one Kokkos View per cell performed much better despite the only approach which requires data copy operations at the beginning and at the end of the traversal. The combination of the cache locality of the data and the tightly nested loops over the clusters of the C08based approach could create enough work for over 20 threads, although in some cases the program had to wait until all tasks completed before it could start the next concurrent execution.

5.2 Outlook

In the current OpenMP implementation with Kokkos, one of the most necessary improvements would be the usage of vectorization instructions. The current circumstances may be related to the used compilers and the inability to optimize code in the Kokkos memory space. Another aspect is the inability to provide enough work for all 28 threads on the CoolMUC2, which might increase due to the selection of a larger spacing or a smaller $r_{cutoff}$and consequently more cells and also more clusters. A higher degree of parallelization comes at the cost more cells and the initialization analysis showed that the creation of many cells (with very few cells) can cause a lot of overhead.

Beside the usable OpenMP usage in this thesis, the portability of Kokkos is another important feature. With an already implemented CUDA option in Kokkos and a few modifications in the code, it should be possible to use one or more devices in parallel for the computation of MD simulations. However, as in this work the correct selection of the correct data structure and fitting nested loops needs to be researched. Compared to the OpenMP implementation, we have more device specific options in Kokkos to deploy our product to. That means, we have to select depending on the used hardware the most fitting option in

the CMake options in order to receive generation specific optimizations from Kokkos. In some cases the compilation for a Kepler generation device failed when any other generation was specified.

List of Figures

List of Tables

Bibliography

[BZB11] Hans-Joachim Bungartz, Stefan Zimmer, and Martin Buchholz. *Modellbildung und Simulation - Eine anwendungsorientierte Einführung*. Oldenburg Wissenschaftsverlag GmbH, Munich, 2011.

[Cra04] Christoph J. Cramer. *Essentials of Computational Chemistry - Theories and Models*. Wiley, West Sussex, 2004. 2nd edition.

[GKZ07] Michael Griebel, Stephan Knapek, and Gerhard Zumbusch. *Numerical Simulation in Molecular Dynamics - Numerics, Algorithms, Parallelization, Applications*. Springer-Verlag Berlin Heidelberg, Berlin, 2007.

[GST+19] Fabio Alexander Gratl, Steffen Seckler, Nikola Tchipev, Hans-Joachim Bungartz, and Philipp Neumann. Autopas: Auto-tuning for particle simulations. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Rio de Janeiro, May 2019. IEEE.

[Hun11] Siegfried Hunklinger. *Festkörperphysik*. Oldenburg Wissenschaftsverlag GmbH, Munich, 2011. 3rd edition.

[Rap97] D. C. Rapaport. *The Art of Molecular Dynamics Simulation*. Cambridge University Press, Cambridge, 1997. 2nd edition.

[Til87] M. P. Allen & D. J. Tildesley. *Computer Simulatin of Liquids*. Oxford University Press, Oxford, 1987. reprinted 2009.

[TSH+19] Nikola Tchipev, Steffen Seckler, Matthias Heinen, Jadran Vrabec, Fabio Gratl, Martin Horsch, Martin Bernreuther, Colin W Glass, Christoph Niethammer, Nicolay Hammer, Bernd Krischok, Michael Resch, Dieter Kranzlmüller, Hans Hasse, Hans-Joachim Bungartz, and Philipp Neumann. Twetris: Twenty trillion-atom simulation. *The International Journal of High Performance Computing Applications*, page 109434201881974, Jan 2019.