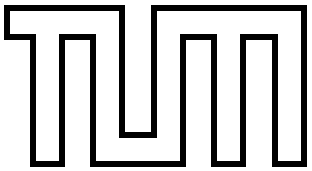


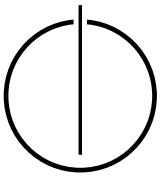
FAKULTÄT FÜR INFORMATIK

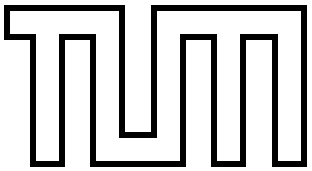
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Efficiency Analysis of Shared Memory Parallelizations from the Algorithm Portfolio in Autopas

Twain Mark Thomas Henkel





FAKULTÄT FÜR INFORMATIK

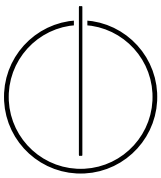
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Efficiency Analysis of Shared Memory Parallelizations from the Algorithm Portfolio in Autopas

Effizienz Analyse von Shared Memory Parallelizations des Algorithmus Portfolio in Autopas

Author:	Twain Mark Thomas Henkel
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisor:	Fabio Alexander Gratl, M.Sc.
Date:	17.08.2020



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

Munich, 17.08.2020 Twain Mark Thomas Henkel

Abstract

AutoPas is a C++ library that provides different algorithms to efficiently solve N-body problems. For molecular dynamics simulations, such algorithms are especially important to quickly calculate large amounts of short-range forces. This thesis adapts the existing sliced traversal into a 3-dimensional slicing algorithm and evaluates its efficiency in comparison to the existing algorithm. The resulting analysis shows some increases in efficiency, but also some different shortcomings.

Zusammenfassung

AutoPas ist eine C++ Bibliothek, welche verschiede Algorithmen zum effizienten Lösen von N-Körper Problemen bereitstellt. Für Molekulardynamik Simulationen sind solche Algorithmen besonders wichtig um schnell eine große Menge an Kräften in kurzer Reichweite zu berechnen. Diese Arbeit passt das existierende Schnitt Traversal zu einem 3 dimensionalen Schnitt Traversal und evaluiert dessen Effizienz im Vergleich zum bestehenden Algorithmus. Die resultierende Analyse zeigt manche Verbesserungen in der Effizienz, aber auch manche verschiedene Mängel.

Contents

1 Introduction

Molecular Dynamic (MD) simulations are used to simulate future states of a given molecular system by calculating all interactions between comprising molecules or atoms. Such systems are used in a wide range of fields where direct analysis of physical systems is not feasible. The efficiency of such simulations is very important because the number of particles in such systems can be incredibly large. Even a droplet of water with the weight of $0.18g$ exists of roughly $6.022 \cdot 10^{21}$ [VMS+19] molecules. In comparison, the current fastest supercomputer in the world can calculate $4.88 * 10^{17}$ FLOPS [fCS20]. Simple iteration over that water drop would take longer than 3 hours or $1.2 \cdot 10^5$ seconds. While this is an acceptable amount of time, it is usually necessary to calculate many multiple steps into the future to ascertain useful information. Usually, the particles can interact with each other. This alone makes any MD Simulation an N-Body Problem, with the complexity of $\mathcal{O}(n^2)$.

Therefore the efficiency of MDs is a vital part to ensure problem applicability by reducing the complexity. One such library dedicated to reducing that is AutoPas. AutoPas [GS+20] is one code library providing algorithms under the TaLPas (Task-based Load Balancing and Auto-tuning in Particle Simulations) Project [tal20] for MD Simulations run in HPC. AutoPas provides auto-tuning at the node level and has various approaches for Shared Memory Parallelizations. In this thesis, we analyse efficiency on the implementation of AutoPas.

2 Theoretical Background

Molecular Dynamics Simulations are N-body Simulations calculating pairwise interactions between bodies. As the affected area is on a molecular level, the bodies are usually particles. While the simulations themselves should always be as accurate as possible, it is often feasible to abstract or ignore some data and still achieve correct simulation output under given starting parameters. Such strategies could include principles such as coarse-graining [HEHB15, p. 12] or truncating the Lennard-Jones-12-6 Potential [GST+19]. Some coarse-graining strategies might change the simulation results, which might be acceptable depending on the simulation objective. AutoPas provides a particle interface, where the user can define their particles, whether they are single atoms, larger molecules, astrophysical objects or a mix, preferably in roughly the same size range. As the practical work of this thesis built on the AutoPas Library and coarse-graining efficiency depends on the strategy of the user, any specific particle-based optimizations will not be discussed. As AutoPas is only a library, we test our MD simulations with example software using AutoPas, named md-flexible.

2.1 Particle Simulation

Technically all particles in a given simulation environment would be interacting with each other. Usually, in MD simulations this is avoided by using the Truncated-Shifted Lennard-Jones-12-6 Potential. Different interactions require different minimal timesteps to still preserve accuracy. The timesteps can be integrated using the Störmer-Verlet method, as is done in md-flexible. In md-flexible the interactions between particles are calculated only in short ranges, leaving the Lennard-Jones-12-6 Potential and Newtons Third Law of Motion as the main calculations between forces.

2.1.1 Lennard Jones Potential

The Lennard Jones Potential [len24] describes an additive force combination between particles i and j. The Equation models Van der Waals forces and the Pauli repulsion:

$$U(r_{ij}) = 4\epsilon \left(\left(\frac{\sigma}{r_{ij}} \right)^{12} - \left(\frac{\sigma}{r_{ij}} \right)^{6} \right)$$

with r_{ij} being the length between i and j. The strength of the attraction or repulsion ϵ between i and j is also called the well depth. Leaving the Van der Waals radius σ as the distance at which $V(r_{ij})$ becomes zero.

The Lennard-Jones Potential quickly convergences towards a 0 limit, allowing a truncation at radius r_c.

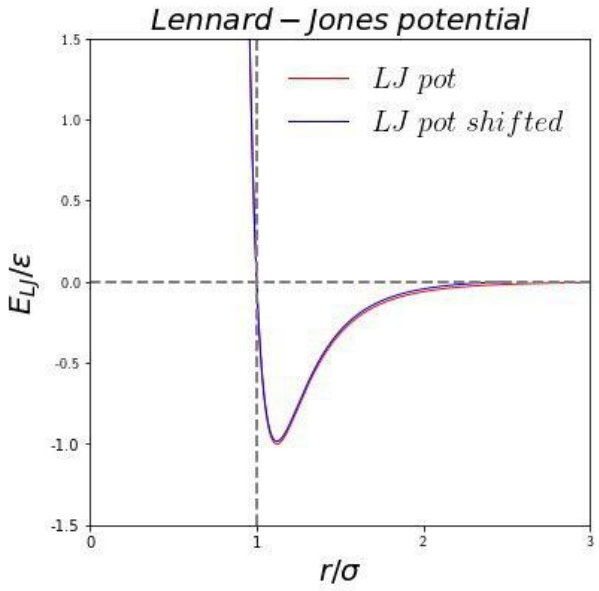


Figure 2.1: From [Mar17]. The Truncated Lennard-Jones Potential-12-6 Potential, truncated at 2.5 with $\sigma = 1.0$ and $\epsilon = 1.0$

.

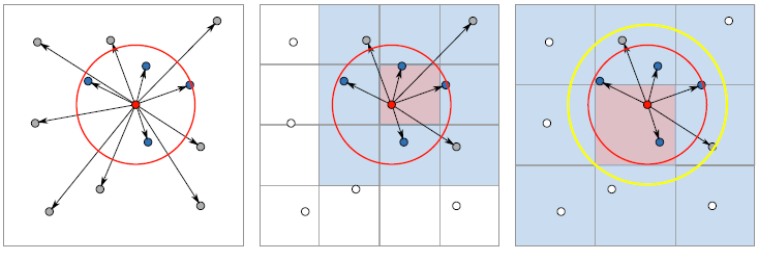
This can be used to reduce the number of pairwise calculations for each particle significantly while neglecting a possible force, that converges towards 0. AutoPas allows this cutoff radius to be variable, by setting the cutoff radius as required. Any mention of a cutoff radius in this thesis always refers to the r_c cutoff radius of the Lennard-Jones Potential, unless explicitly otherwise stated.

2.1.2 Newtons Third Law

Newtons Third Law says that every action, there is an equal and opposite reaction. In a molecular dynamics environment this saves half of the inter-particle calculations as $U(r_{ij}) = (-1) \cdot U(r_{ji})$. The force affecting particle i, is equal to the negative force on particle j.

2.2 Force Calculation

There are multiple algorithms to calculate the forces for particle pairs. All of them provide the same results and only differ in their approach. AutoPas includes the following:



(a) DirectSum (b) LinkedCells (c) VerletLists

Figure 2.2: From [GST+19]. Illustration of the different interaction possibilities. Connection by arrows indicates necessary distance evaluations. The red circle marks the cutoff radius.

2.2.1 Direct Sum

An intuitive algorithm simply iterating over all existing particles. It checks each particle if it is within distance once per iteration. The runtime complexity for n particles is therefore $\mathcal{O}(n^2)$. Such complexity is not useful for larger amounts of particles. But as the overhead is rather small, it can be advantageous to use for very few particles. Nevertheless, it serves as a perfect worst-case benchmark for comparison.

2.2.2 Verlet Lists

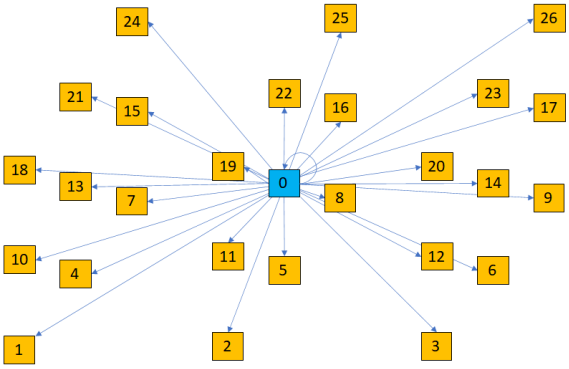
The Verlet Lists Algorithm now combines the Direct Sum with the cutoff radius. Each Particle holds a list of all particles close enough to be in their cutoff radius. To calculate the force for a single particle it is only necessary to check each particle on its Verlet-list. If the particles are moving, they can move in or out of the cutoff radius. To mitigate recalculation of the verlet list every step, the cutoff is expanded by a skin length, to account for movement. The complexity to build the lists for n particles is in $\mathcal{O}(n^2)$ because each particle compares their distance to all other particles. Depending on the size of the Verlet-skin though, it is possible to keep the list for multiple time steps, before a rebuild is necessary. The build complexity could be further reduced if it would not be necessary to search through all particles on rebuild. Which is what Subsection 2.2.3 is about.

2.2.3 Linked Cells

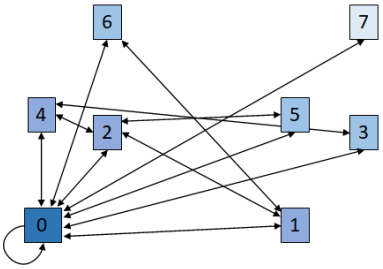
The Linked Cell Algorithm expands on the Direct Sum incorporating the cutoff radius to organise the simulation environment. The simulation environment is structured into cubes with the edge length of the cutoff r_c. If there are different particles, usually the longest cutoff is used. Instead of each particle having to check all other particles to find if they are inside their cutoff, they now have to check only their Cell and its neighbours. This reduces the complexity down from $\mathcal{O}(n^2)$ to $\mathcal{O}(c \cdot n)$. Of course only under the assumption that the particles are spread over the environment and do not cluster into a few single cells or that all cells of the domain are neighbours to each other. The latter could happen if r_c is too large because of a freak particle or because of a very small size of the environment. The iteration actually does not iterate over particles, but cells and conducts the calculation for all particles in the cell.

2.3 Cell-Handling

AutoPas provides multiple algorithms on how the cells should be iterated and the particles inside each cell handled.



(a) The c01 interaction pattern for a single cell.



(b) The c08 interaction pattern for a single cell.

C01

The c01 algorithm reads all neighbouring cells to calculate the forces for the current inner cell. Except for the border, this necessitates checking 27 cells, while the force is only calculated for 1 cell per step. Without applying newtons third law it is rather easy to parallelize, as one does not have to worry about race-conditions.

C08

The c08 cell handling uses less cells than c01 per step. This is possible through not only calculating the influential force from the neighbour cells to the current cell but also by applying the forces of the current cell to the neighbour cells. If Newtons Third Law is applied, the calculation of opposite forces for the particles in the neighbouring cells often becomes trivial. For parallelization, it is necessary to be aware of the writing process, which occurs into the neighbouring cells, or else race-conditions can appear. Hence all cells need to be locked until the interactions as shown in Figure 2.3b for the specific cell have been calculated. Through the application of Newtons Third Law the cells can be unlocked after they have been processed. This means that cell 0 locks all cells $0 - 7$, but if 0 has been processed already cell 1 would only need locks on cell 2 and 6.

2.4 Traversals

A traversal parallelizes a different cell handling base-step. Strategies to achieve parallelization include different colouring schemes for the cells and locking mechanisms.

2.4.1 Coloured Traversals

A colouring strategy allocates colour to cells or regions of cells, depending on the traversal. The colouring mechanisms only allow one colour to be processed until all threads finished their coloured region of cells. Cells which can not influence each other in a single cell handling step can be coloured in the same colour. Each connected set of cells with the same colour is then processed by a single thread. As an example, a traversal using the c08 cell handling can pass by with 8 different colours. This usually generates some minimal idle time for the threads before switching colour.

2.4.2 Sliced Traversal

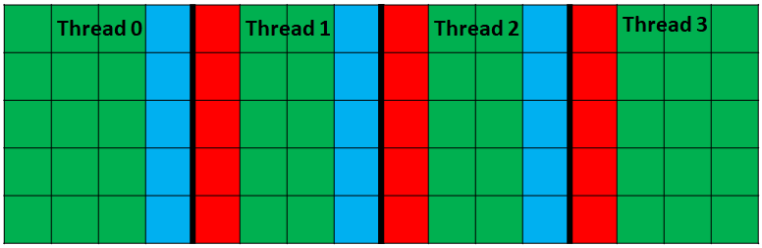


Figure 2.4: The sliced traversal over 16x5 cells with 4 threads. Blue cells are locked at the start and are freed as soon as the neighbouring red cells have been processed.

A different approach would be to separate the simulation domain into different areas or slices. The Sliced Traversal in AutoPas divides the longest domain by the number of available threads. Each thread then becomes a slice to process. To avoid race-conditions the borders between the cells are locked. In Figure 2.4 the blue cells are locked at the start by the neighbouring thread. Each thread will traverse first over their red cells, to free the neighbouring blue cells as soon as possible. This additionally gives a direction in which the thread traverses over the cells, as the last cells to traverse should be the blue cells to avoid waiting times because of locks still held by neighbour threads.

3 Implementation

All the different strategies of the traversals are trying to optimize the speed of the force calculation between the particles. It seems unlikely that there would be a single strategy that is always the fastest in any situation, but we can certainly assume that there are algorithms, which are better suited in specific particle distributions than others [Tch20]. Hence the question to identify certain conditions under which the traversal algorithms can be easily distinguished in performance. For best application, those conditions should be as simple and come with as little calculation effort as possible, while still being good indicators which traversal to prefer currently. Therefore it is necessary to analyse those algorithms that we have already and to find out under which situations they are efficient. Building on that we can expand to judge the traversals against each other and decide in which scenarios it might be best to use traversal A over B.

3.1 Context of Efficiency

We define a program as efficient if it wastes as little time as possible. This is not a simple task though, as it can be favourable to increase time usage in some parts of a program to achieve a significant gain in the end. And just because some time can be saved in a subprocess, does not necessarily apply to time saved in the end. In consequence, the aim is to analyse efficiency, which translates directly into minimizing total execution time. A good starting point is the force calculation as it is the most time-consuming part of each MD Simulation. Consequently, the main goal is to achieve the utmost efficiency of the force calculation, even if possibly creating a comparable small increase in other time expenditures. It is important to note, that this thesis is about the time efficiency of traversals, which focus on the parallelization of cell handling.

3.1.1 Minimizing Thread Waiting Time

A theoretically most efficient parallelization algorithm would not simply avoid any thread idle time. It would have the smallest sum of idle time, aggregated over all threads. Additionally, any overhead processing from actually traversing should also be minimal. This generates a direct competition between:

- Maximizing Time spent on Force Calculation
- Higher Parallelization or using more Threads
- Minimizing Thread Idle Time
- Minimizing Overhead Processing

At the very beginning, there are a few things we can not avoid right from the start. There will always be small differences in the simulations as particles move and are not necessarily

always spread equally over threads. With a different time to finish for each thread even in a theoretically optimal solution, there would always be some waiting time in the end until the last thread is finished. Also, there might always be some overhead at the start of the traversal, which might not be parallelizable.

This thesis will try to improve an already existing traversal by decreasing its thread waiting time. The optimization builds on the sliced traversal as the sliced traversal splits the simulation domain only by one (the longest) axis. Allowing for possible optimization by splitting all three axes instead of only one. The increased amount of possible threads should be an optimization the larger the simulation domains gets. As the load is also balanced better threads might idle less.

3.2 Sliced Block Traversal

The Sliced Block Traversal is a 3-dimensional extension based on the original sliced traversal and a proposed 2-dimensional extension "sli_blk" in [Tch20, p. 127].

The general idea is to slice each dimension of the simulation into blocks and assign each block bijective to one thread, allowing finer granularity. For this, it uses the available amount of threads to slice each dimension, which results in blocks, and assign the number of slices $length_i$ to dimension i. The amount of slices per dimension are then the i-th root of the dimensional length: $l_i = \lceil \sqrt[max(i)]{n} \rceil$ with n being the number of available threads and i the number of dimensions. If l_i is not a natural number, we round down and expand the last block in the dimension towards the end of the domain length.

For the Sliced Block Traversal, we use 3 dimensions, as AutoPas and most molecular dynamics simulations only simulate in 3 spatial dimensions, while accounting for time differently. Therefore we only need the cubic root $l_{x,y,z} = \lceil \sqrt[3]{n} \rceil$ of available threads n to calculate the number of slices per dimension.

While the normal sliced approach only slices in one dimension, it must slice along the longest axis to use the maximum amount of threads possible. In the sliced traversal, the amount of usable threads scales only with one axis. Additionally, the length of each slice has to be at least 2 for reasons of locking, as parallelization with slice length 1 would become serialization. See also Figure 2.4. The maximum ratio of cells to threads for the Sliced traversal is therefore:

$$\frac{c}{\frac{l_i}{2}} = \frac{2 \cdot c}{l_i}$$

with c being the total amount of cells and l_i the maximum axis length in cells.

The Sliced Block Traversal slices in all dimensions though with the maximal ratio:

$$\frac{c}{\frac{l_x}{3} \cdot \frac{l_y}{3} \cdot \frac{l_z}{3}} = \frac{1}{27} \cdot c$$

We divide each axis length by 3, because we still want to form cubes with a middle cell for parallelization optimization explained later in this section. This has no disadvantageous ratio of thread to cells if the simulation environment approaches a cubic form. Assuming l_x is the longest axis, we can compare the number of cells per thread directly:

$$\frac{27 * c}{y \cdot z} = 2 \cdot c$$

$$2 \cdot y \cdot z = 27$$

The sliced traversal scales rather badly for cubic form domains, whereas the sliced block traversal theoretically always uses 27 cells per thread. In practice, this is, of course, limited by the physical maximum of threads available. We can assume the sliced block traversal will converge a lot faster on using the maximum amount of threads than the sliced traversal, at least for cubic form domains.

While it illustrates the expected gain, it is important to remember, that with any dimension length smaller than 3 cells the actual Sliced Block Traversal implementation is not applicable. In the rare case that the other 2 dimensions are so large that a gain from 2-dimensional slicing would be advantageous to the normal slicing, we would propose to stretch the single or double cells to a 3 cell length by filling with empty space if possible. Nevertheless, the worst-case slicing of the Sliced Block Traversal is still a good slicing for the normal Slicing Traversal, excluding the aforementioned exception.

Different dimensional sizes lead to different lengths of the slices or blocks, which in turn lead to different block dimensions, rather than perfect cubes. Especially as the number of threads is most likely not divisible without remainder, the last blocks towards a dimensional border must be expanded towards the dimensional border.

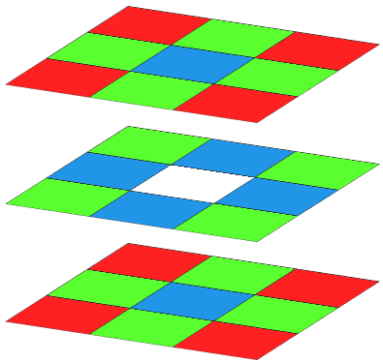
3.2.1 Splitting the blocks into sub-blocks

We can not easily iterate over each block with each thread, as it would cause race-conditions if two threads would operate on neighbouring cells. To mitigate this we split the blocks into the thinnest possible surface areas around a large middle cuboid. The depth of the surface area needs to be the rounded up cutoff r_c length in cells.

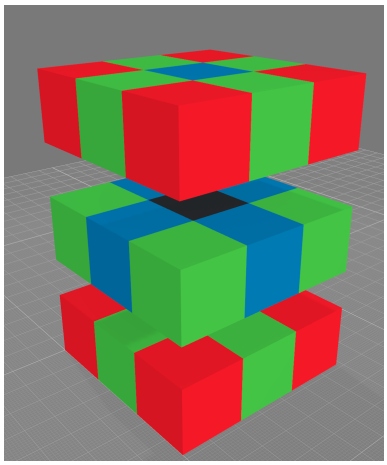
This provides us with two questions: First, how do we split up the surface area to lock neighbouring blocks? Secondly, how do we iterate over the surface area to minimize locks and locked time?

For the answer to the first question, we need to remember that we want the inside area of each block to be as big as possible because this is the area with the least amount of locks needed. This is the case because a thread does not need to lock cells, which can only be accessed by itself. If we imagine equally sized blocks stacked to fit a large space, we analyse the areas where each block would overlap with other blocks. In a 3 dimensional slice, we would have areas or sub-blocks:

We see that the overlap differs in the corners, edges and the surface. Including the inner part of our block, we can split one block into 27 sub-blocks, 8 Corners, 12 Edges, 6 Surfaces and 1 inner part as seen in Figure 3.1a The size of these depends on the length of our overlap measured in cells. If we have an overlap of 1 or a larger overlap, the size of the sub-blocks changes. But the size of the surface areas for a given block varies only depending on the overlap. With that, we can distinguish the sub-blocks in a single block in the following way:



(a) A minimal 3 sided cube. The Corners (red) touch 8 other blocks, the Edges (green) 4, the Surfaces (blue) 1 and the Middle (white) none.



(b) A 3 dimensional render of a minimal block.

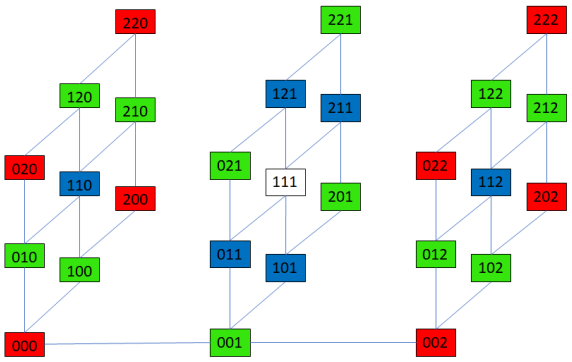


Figure 3.2: Each Subblock with a unique id in relation to their orientation inside the block. Again Red = Corner, Green = Edge, Blue = Surface, White = Inner Part.

To calculate the sub-blocks now, we only need the starting coordinates of the sub-block, the length x_i of the block in each dimension and the overlap size.

Size Corner:	$overlap_x * overlap_y * overlap_z$
Size Edge:	$overlap_i * overlap_j * (blocklength_k - overlap_k * 2)$
Size Surface:	$(blocklength_i - overlap_i * 2) * (blocklength_j - overlap_j * 2) * overlap_k$
Size Middle:	$(blocklength_i - (overlap_i * 2)) * (blocklength_j - (overlap_j * 2)) * (blocklength_k - (overlap_k * 2))$

Table 3.1: Volumes per Sub-Block in cells

3.2.2 Sub-Block Traversal and Locking

First, we do not want to lock cells inside the sub-blocks, as we have already established minimal areas for locking. Secondly, the surface areas are rather small themself, with the corners being exactly a single cell for $r_c = 1$. This in turn should lead to quick traversal through the surface areas and a long traversal through the inner sub-block. Assuming that each cell needs to lock all neighbouring cells necessary for parallel force calculations, each

sub-block would have to lock every neighbouring sub-block. To avoid this our traversal will use the c08 cell-handling as described in Section 2.3. Therefore the number of locks decreases significantly depending on the number of cells the cell-handler uses to make the calculation. This corresponds to 8 locks per sub-block traversal.
Because the areas are rather large, we want to avoid a colouring approach as the possible waiting times per thread could be drastic. Technically we can let the thread freely choose when to iterate over which sub-block, as long as it can lock the corresponding neighbours. And depending if we want that or not we have two options to choose from:

Asynchron We allow each thread to freely choose when to work on each sub-block and queue each sub-block for each thread. If one thread can not process a sub-block because it is locked, that thread can then choose the next sub-block in the queue (unless it is the last one) and continue calculating.

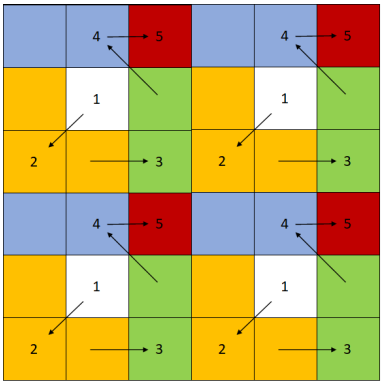
Synchron If we do not allow a free sub-block sequence, we can save locks by giving a chronological order. As each thread has to only lock those own sub-blocks, which have not yet been processed and can only request locks for neighbouring sub-blocks, which have been processed already.

Asynchronous Sub-Block Handling

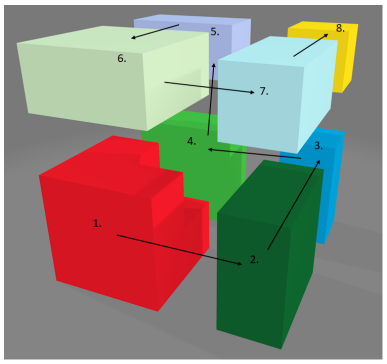
This option varies greatly depending on the underlying cell handler. In the case of the c08, each cell needs to lock 7 neighbour cells and itself. This corresponds exactly to the same neighbouring and own sub-blocks as the cells used by the cell handler to iterate over. The c08 cell handler also has the other advantage of corresponding directly to a simple shifting of the blocks. The asynchronous handling has the advantage, that a thread is not stuck waiting for a neighbour to finish some of the shared surface cells unless everything else is finished already.

3.2.3 Synchronous Sub-Block Handling

The main idea about this is to use the locks as indicators to show where in the sub-block order a thread is. The immediate drawback is clear as we can have waiting times for other blocks. Our synchronous locking idea is based on the sli-blk [Tch20, p. 127] using one lock per dimension per block. Therefore each block has a x, y and z lock. This is a small number of locks and therefore less overhead than the asynchronous handling. To use this to the full advantage we would take multiple sub-blocks in one single step. We want to allow threads to continue processing sub-blocks as long as they can and only stop if they meet a neighbouring surface which is currently being processed by a neighbouring thread. Additionally, we want to switch the surfaces, through which we iterate, so that a single thread is not stuck multiple times behind locks of the same neighbour thread in succession.
This leads to us to have one lock per dimension per block.
While both strategies have their advantages, the asynchronous could possibly reduce waiting times by a lot more than the synchronous handling. In comparison with the gain from less overhead, it seems the reduced waiting time is the more efficient choice. Therefore we implemented the asynchronous sub-block handling. Nevertheless, there can be no certainty until the efficiency has been analysed for certain scenarios.



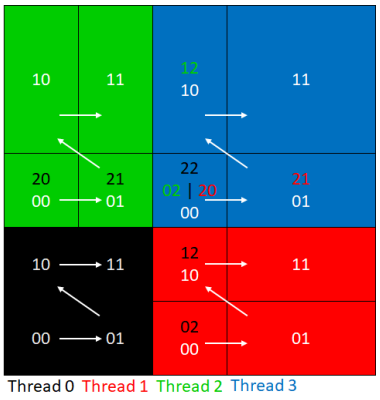
(a) Top Down View on a minimal cube, with 2D traversal order, based on locks.



(b) A blow-up view of a minimal block with a possible iteration order for the sliced block synchronous algorithm. This shows only the surface areas. Each cube in the figure corresponds to a sub-block.

20	21	22 20	12	22
10	11	12 10	11	12
20 00	21 01	22 02 | 20 00	21 01	22 02
10	11	12 10	11	12
00	01	02 00	01	02

(a) A 2D example of overlapping blocks with corresponding sub-blocks.



(b) Each thread iterates only over their coloured area. Arrows indicate inital iteration order in queue.

3.2.4 Parallelization

Now we have different options regarding our parallelization mechanisms. But first, we want to decrease the total amount of different locks per block from 27 to 8. We do this by overlapping the blocks on their surface areas.
This compression increases the parallelization even further, by decreasing the number of cells per block. Each thread also has to iterate over fewer sub-blocks. The locking uses the actual cell coordinates of the closest cell to point of origin of a sub-block. We achieve asynchronous handling by using the OpenMP "omp_test_lock" function, which returns true if a lock could be set and false otherwise. Now it is possible to test if all sub-blocks could be locked and if so, we iterate over the cells in the sub-block with the c08 handling. If not all neighbour sub-blocks could be locked, we free the sub-blocks, then push back our sub-block in the queue and test the next sub-block in the queue.
Nevertheless we could still achieve deadlocks. Avoiding them is not a trivial task though, as the following deadlock scenario might demonstrate:

Block 0 starts: Locks its own sub-blocks 000, 001, 010, 100, 110, 101, 011, 111 (equal to traverse order)
Block 0 is quick and can continue to traverse all sub-blocks until 011 is finished.
Block 0 wants to lock the corresponding sub-blocks for 111 now.
Block 1 is now finished with its sub-block 000 and wants to go into 001.
Block 1 sub-block 001 corresponds to Block 0 sub-block 201.
Now Block 1 locks Block 0: 201 (which is fine) and Block 0: 211 (= 1: 011).
Now Block 0 locks Block 0: 111 and 112 and 212. Where 0: 212 (= 1: 012).
Now we have a deadlock, because Block 1 needs 0: 212 and Block 0 needs 1: 012.

To avoid creating any deadlocks while locking, we use a single masterlock. To avoid any deadlocking while setting the locks, only the thread holding the masterlock is allowed to test for locks. This obviously does not keep us from initially queueing the sub-blocks with the possible order of the synchronous handling. And it neither keeps us from setting the first locks in the initialization before the masterlock is needed. As there are only 8 locks to be tested, the possible waiting time on the masterlock seems still fairly short in comparison to waiting for a cell filled with particles. Unless of course there is no other sub-block left, besides a neighbour locked one.

4 Analysis

4.1 Tools

4.1.1 VTune

We used the Intel VTune Profiler to generate and analyse the efficiency of our algorithm.

4.1.2 Computational Platform

All tests have been run on the Linux Clusters of the Leibniz-Rechenzentrum (LRZ) of the Bavarian Academy of Sciences.

CoolMUC2

The LRZ Linux Cluster CoolMUC2 was mainly used for running the tests. It has multiple partitions with different amounts of shared memory. In total, the cluster has 812 Intel 28-way Haswell based nodes, each with 28 cores and 2 hyperthreads per core. Each node has 64 GB DDR4 memory. The core nominal frequency is 2.5 GHz, with a total peak performance of the complete system of 1400 TFlop/s. For our test execution this means we have 56 threads per node.

CoolMUC3

The LRZ Linux Cluster CoolMUC3 has 148 Intel 64-way Knights Landing 7210-F many-core processors, each with 64 cores and 4 hyperthreads per core. Each node has 96 GB DDR4 memory. The core nominal frequency is 1.3 GHz, with a total peak performance of the complete system of 459 TFlop/s. For our test execution, this means we have 256 threads per node.

4.2 Theoretical Analysis

We already guessed in Section 3.2 that the sliced block traversal should be able to utilize more threads than the sliced traversal. This should become visible for increased larger simulation domains. Additionally, we want to see if the masterlock and lock testing has had positive or negative repercussions. Also, the overhead through the generation of the sub-blocks has increased drastically and is, therefore, a possible issue to check. The granularity of sliced block should also be better than that of the sliced traversal, which might be visible in direct comparison.

Cluster	CoolMUC2	CoolMUC3
Dimensions	200 x 200 x 200	50 x 25 x 13
Particles:	0	0
Iterations:	100	100

Table 4.1: Reference Test Parameters

4.3 Tests

4.3.1 Reference Test for Overhead Operations

A reference test with 0 particles was run at the beginning on each Cluster. Following parameters were used:
The first results on CoolMUC3 were astonishing because sliced block was not only faster but also had less overhead. The problem that occurred was that the length of each slice became less than 2 using more a serialization of the slices, rather than parallelization.

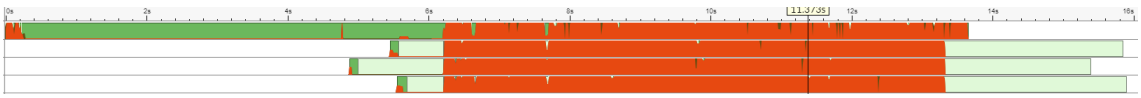


Figure 4.1: Profile of 4 threads out of 256 running the sliced traversal in CoolMUC3.

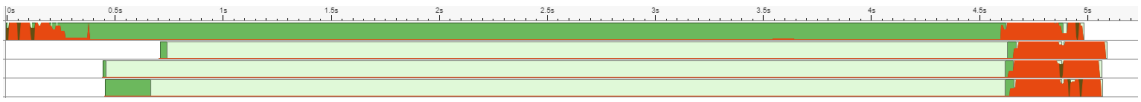


Figure 4.2: Profile of 4 threads out of 256 running the sliced block traversal in CoolMUC3.

All threads except the master thread (Top bar in the profiles) showed similar behaviour and are therefore shortened. The colours have the same meaning and do not change over the tests. Red marks the spin and overhead time, light green is waiting time, dark green is run time and brown colour is CPU utilization time.

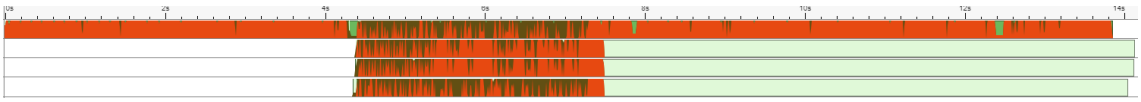


Figure 4.3: Profile of 4 threads out of 56 running the sliced traversal in CoolMUC2.

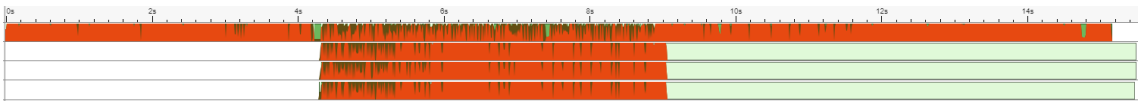


Figure 4.4: Profile of 4 threads out of 56 running the sliced block traversal in CoolMUC2.

The CoolMUC2 results were more to the initial expectations. Sliced took 14.168 seconds and sliced block took 15.505 seconds.

Particle	Dimensions	Iterations	TimeToFinish sliced	TimeToFinish sliced block
0	50 x 25 x13	100	1.337	0.223
16	50 x 25 x13	100	0.72	0.2
64	50 x 25 x13	100	0.705	0.201
256	50 x 25 x13	100	1.106	0.2
1024	100 x 50 x 25	50	0.778	1.947
4096	100 x 50 x 25	50	1.046	2.25
16396	100 x 50 x 25	50	3.133	4.19
65536	200 x 100 x 50	25	9.533	7.844
131072	200 x 100 x 50	25	24.769	18.397
262144	200 x 100 x 50	25	48.861	41.964
524288	200 x 100 x 50	25	102.64	79.718
1048576	200 x 100 x 50	5	194.687	143.305
2097152	200 x 100 x 50	5	378.763	285.997
4194304	200 x 100 x 50	5	778.627	567.727
8388608	200 x 100 x 50	5	1132.888	1219.315

Table 4.2: Gaussian distributed tests over a cuboid.

4.3.2 Homgenous Tests

The homogenous tests had the following parameters and were executed on a single node on CoolMUC2 with using 56 threads.
There are a lot of interesting results. First, the bad runtime for the sliced traversal seems to have the same reason as in the reference tests. This is attested by the better runtime once the dimensions increase at 1024 particles. Interestingly the increase for the sliced block traversal increases immensely at the same point.

Figure 4.5: Profile of 4 threads with the 3rd from top being the master thread. Test has 256 particles in a cuboid of 100 x 50 x 25 cell dimensions.

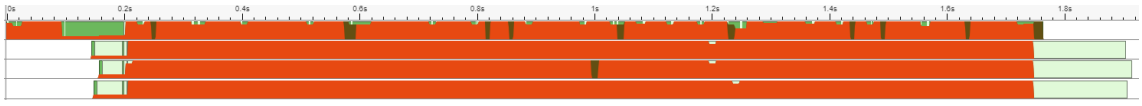


Figure 4.6: Profile of 4 threads running a test with 1024 in a cuboid of 100 x 50 x 25 cell dimensions.

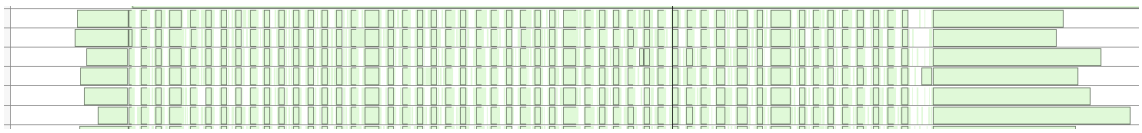


Figure 4.7: Profile of only the waiting time of 6 threads running a test with 1024 in a cuboid of 100 x 50 x 25 cell dimensions.

Particle	Dimensions	Iterations	TimeToFinish sliced	TimeToFinish sliced block
0	50 x 50 x 50	100	0.89	3.325
16	50 x 50 x 50	100	0.871	3.266
64	50 x 50 x 50	100	0.889	3.244
256	50 x 50 x 50	100	0.906	3.264
1024	100 x 100 x 100	50	2.003	3.205
4096	100 x 100 x 100	50	2.198	3.298
16396	100 x 100 x 100	50	2.815	3.964
65536	200 x 200 x 200	25	14.932	14.141
131072	200 x 200 x 200	25	20.439	17.575
262144	200 x 200 x 200	25	32.175	26.105
524288	200 x 200 x 200	25	71.616	52.74
1048576	500 x 500 x 500	5	227.346	203.114
2097152	500 x 500 x 500	5	303.136	249.984
4194304	500 x 500 x 500	5	548.944	387.837
8388608	500 x 500 x 500	5	693.957	787.501

Table 4.3: Gaussian distributed tests over a cuboid.

What looked weird at the start is easily explained by Figure 4.7 and the stacktraces mainly show waiting for joining the endTraversal and only very very rarely an actual lock because of the master lock.
The rest of the simulation shows that the sliced block traversal is gaining efficiency the larger the dimensions become and the more particles are used. It is not simply the dimensions, which increase efficiency, but by increasing the number of particles the larger overhead each step is easier justified. For the largest dimensions in our test, it is a bit surprising that sliced block traversal is quicker, as the sliced traversal can also use all its threads and does not have the overhead. At this phase, most waiting was done for memory calls. A possible explanation for this would be the better predictability, which cells need to be preloaded for the sliced bulk traversal, as the cells for a single thread are closer together than for the sliced traversal.

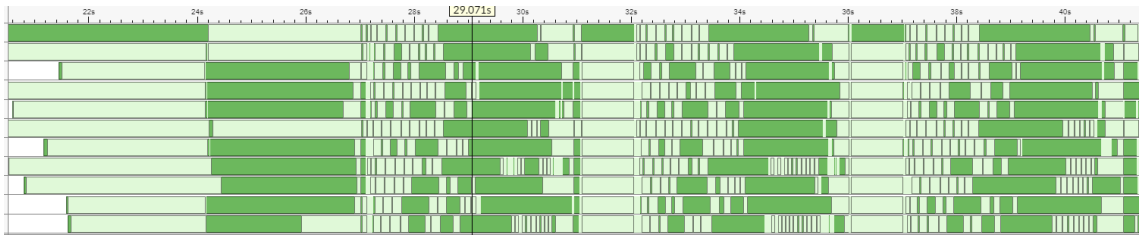


Figure 4.8: Profile of waiting times for threads at the start of the homogenous simulation running 2097152 particles on the CoolMUC3 Cluster.

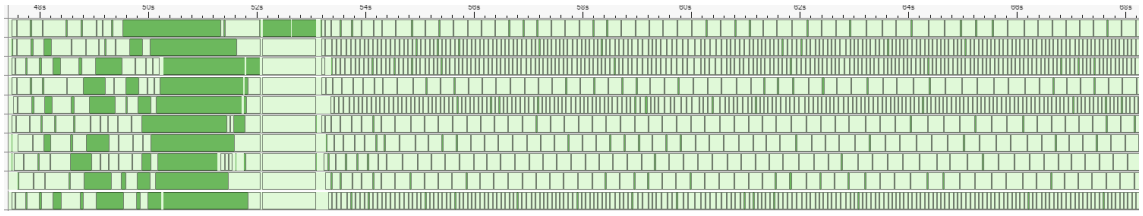


Figure 4.9: Profile of waiting times for threads at the middle of the homogenous simulation running 2097152 particles on the CoolMUC3 Cluster.

Additionally, a subset of the homogenous cubic tests was run on CoolMUC3 to see if the masterlock might cause locking problems. The total amount of waiting for locks did increase, as shown in Figure 4.8. It is certainly guaranteed to increase further with more threads, but the biggest bottleneck was waiting for memory, again. This, in turn, triggered a high waiting period for some threads, possibly while the locking uses actual cell coordinates, which in turn needs memory, which might cause the thread holding the masterlock to wait for memory. Therefore the Masterlock certainly remains a weak link, especially if memory needs to be loaded regularly.

4.3.3 Inhomogenous Tests

The inhomogenous tests were executed with equal parameters as the homogenous tests but included an additional dense part of particles in the domain. The dense part was generated as a sphere with radius either being 20, 40, 60 or 80 corresponding to the increase in dimensions. While the theory behind the test was to try to find out if behaviour changes drastically compared to the homogenous tests, it did not. Runtime increased significantly because of the higher amount of particles as did possible waiting time on the threads inside the spheres.

5 Conclusion

In this thesis, a new traversal was introduced building on the sliced traversal with focus on higher efficiency. Using AutoPas as a basis to develop, the number of applicable threads for parallelization could be significantly increased in comparison to the base. The new traversal slices with a lot finer granularity, and additionally allows for basic wait time avoidance through the lock testing functionality of OpenMP. The traversal was then tested on the CoolMUC2 and CoolMUC3 Clusters regarding efficiency. Those tests succeeded under certain conditions, while not being a perfectly efficient traversal. The analysis provided insight, which can be used to further optimize the traversal and to correct some shortcomings as the masterlock. All in all the new sliced bulk traversal certainly provides an efficient traversal, if the dimension is big enough and the amount of used threads is not too high in combination with frequent memory calls.

5.1 Optimizations

5.1.1 Masterlock

In theory, only neighbouring cells need to share a masterlock. It also should be possible for threads to test for two masterlocks, if it is always in the same order. An example would be building masterlock areas like the slices from the sliced traversal and all threads in such a slice share a masterlock and the masterlock of the neighbouring slice.

5.1.2 Expand the Surface Areas

Having each sub-block assigned only to a single thread allows for more selection in the queue. Additionally, it allows for each inner sub-block to be free of locks, as the only neighbouring cells are assigned to the same thread. This allows a single sub-block in the queue, which is the largest and can always be processed. The increase in total locks per thread might be negligible.

List of Figures

List of Tables

Bibliography

[fCS20] RIKEN Center for Computational Science. Supercomputer fugaku, 2020.

[GS+20] Fabio Gratl, Steffen Seckler, et al. Autopas, 2020.

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

[HEHB15] Alexander Heinecke, Wolfgang Eckhardt, Martin Horsch, and Hans-Joachim Bungartz. *Supercomputing for Molecular Dynamics Simulations*. Springer International Publishing, 2015.

[len24] On the determination of molecular fields.—i. from the variation of the viscosity of a gas with temperature. *Proceedings of the Royal Society of London. Series A, Containing Papers of a Mathematical and Physical Character*, 106(738):441–462, October 1924.

[Mar17] Jakob Martin. Molecular dynamics in python, 2017.

[tal20] Talpas project, 2020.

[Tch20] Nikola Plamenov Tchipev. *Algorithmic and Implementational Optimizations of Molecular Dynamics Simulations for Process Engineering*. Phd thesis, Institut für Informatik 5, Technische Universität München, Garching, 2020. unpublished thesis.

[VMS+19] Ed Vitz, John W. Moore, Justin Shorb, Xavier Prat-Resina, Tim Wendorff, and Adam Hahn. Number of atoms in a drop of water, 2019.