FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Application and Evaluation of Auto-Tuning Tools in Molecular Dynamics Simulations

Jakob Andreas Englhauser

FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Application and Evaluation of Auto-Tuning Tools in Molecular Dynamics Simulations

Anwendung und Auswertung von Auto-Tuning Tools in Molekulardynamiksimulationen

Author:	Jakob Andreas Englhauser
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisor:	Fabio Alexander Gratl, M.Sc.
Date:	February 17, 2020

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

Munich, February 17, 2020 Jakob Andreas Englhauser

Abstract

The simulation of molecular dynamics with high numbers of particles is a very computationally intensive task. To keep resource and time consumption minimal, it is, therefore, necessary to use algorithms that are well suited for the specific simulation environment. One library that can be used by N-body simulation programs is AutoPas, which can automatically select optimal combinations of algorithms during runtime through different tuning strategies. In this thesis, external auto-tuning tools, namely Active Harmony, were integrated into AutoPas and evaluated by simulating a common physical mechanism, the spinodal decomposition. In this simulation, Active Harmony was able to achieve a lower runtime than previous, in AutoPas implemented tuning strategies by optimizing configuration selection during the tuning process.

Zusammenfassung

Molekulardynamiksimulationen mit hoher Anzahl an Partikeln stellen eine sehr rechenintensive Aufgabe dar. Um Resourcen- und Zeitverbrauch zu minimieren, ist es daher notwendig Algorithmen einzusetzen, die für die jeweilige Simulationsumgebung möglichst gut geeignet sind. Eine Bibliothek, die von N-body Simulationsprogrammen verwendet werden kann, ist AutoPas, welche zur Laufzeit automatisch optimale Kombinationen von Algorithmen auswählt, indem verschiedene tuning Strategien benutzt werden. In dieser Arbeit wurden externe Werkzeuge für auto-tuning, namentlich Active Harmony, in AutoPas integriert und durch die Simulation eines geläufigen physikalischen Mechanismus, dem Spinodalen Zerfall, ausgewertet. In dieser Simulation konnte Active Harmony die Laufzeit im Vergleich zu vorherigen, in AutoPas implementierten tuning Strategien verringern, indem während dem tuning Prozess bessere Konfigurationen zur Auswertung gewählt werden.

Contents

Abstract vii

Zusammenfassung ix

I. Introduction and Background 1

1. Introduction 2

2. Theoretical Background 3

2.1. Lennard-Jones Potential 3
2.2. Newton's Third Law 4
2.3. Algorithms 4
2.3.1. Direct Sum 5
2.3.2. Linked Cells 5
2.3.3. Verlet Lists 7
2.4. Data Layout 9
2.5. Algorithm Selection 9

II. Active Harmony 10

3. Introduction to Active Harmony 11

3.1. Concept of Active Harmony 11
3.2. Processing Layers 12
3.3. Search Strategies 13

4. AutoPas 16

4.1. Introduction to AutoPas 16
4.1.1. AutoPas Interface 16
4.1.2. Tuning 16
4.2. Integrating Active Harmony into AutoPas 17

5. Evaluation 20

5.1. Simulation Environment 20
5.1.1. md-flexible 20
5.1.2. Linux-Cluster 20
5.2. Simulation of Spinodal Decomposition 20
5.2.1. Equilibration 21

5.2.2. Decomposition 21
5.3. Results 22

III. Conclusion **24**
5.4. Conclusion 25

IV. Appendix **26**

A. YAML-input files **27**

Bibliography **31**

Part I.

Introduction and Background

1. Introduction

Observation of natural phenomena is an important part of most natural sciences to improve understanding of the world. However, not every experiment is feasible to run in the real world, as the cost of required materials or equipment can become too high. In such cases, the simulation of these experiments on a computer can present a viable alternative and as improvements in computer technology are made, increasingly complex scenarios can be simulated within a reasonable time frame. But still, the computation of high numbers of particles in molecular dynamics simulations is a very hardware-intensive and time-consuming task, what makes it necessary to optimize the program as much as possible. However, it is impossible to create a configuration of algorithms and data structures that will be optimal for every simulation [Bro87] and not every non-computer scientist can be expected to perfectly judge the advantages and disadvantages of all algorithms to achieve optimal performance.

For reasons like this, the C++ library AutoPas was created. It aims to select the best options for several parameters during a simulation of molecular dynamics without further user interactions, a process called auto-tuning. This is done dynamically during runtime to adapt to changes in the simulation.

But particle simulations are not the only application of auto-tuning. Technically every program with parameters that can influence its performance can be subjected to auto-tuning. This led to the creation of general-purpose auto-tuning tools such as Active Harmony.

The goal of this paper is to bridge the gap between general-purpose tools and AutoPas by integrating them into AutoPas to improve the performance of programs using the AutoPas library to auto-tune N-Body simulations.

2. Theoretical Background

2.1. Lennard-Jones Potential

Newton's second law [New87] states that the sum of forces F on an object is equal to the product of its mass m and its acceleration a:

$$F = m \cdot a \tag{2.1}$$

This means to calculate a molecule's acceleration — and therefore its movement — one needs to know the forces acting on the molecule. As it is impracticable to exactly calculate every interaction between microscopic particles, the overall force on the molecule is approximated by using various potentials. One such potential, frequently used in molecular dynamics simulations, is the Lennard-Jones 12-6 potential [LJ24]

$$U_{LJ}(r_{ij}) = 4\epsilon \left(\left(\frac{\sigma}{r_{ij}} \right)^{12} - \left(\frac{\sigma}{r_{ij}} \right)^{6} \right) \tag{2.2}$$

where r_{ij} is the distance between two particles.

The Lennard-Jones potential consists of one attracting and one repulsing part.

The attracting part is the van-der-Waals potential U_1, which is a result of the van-der-Waals force.

$$U_1 = -4\epsilon(\frac{\sigma}{r_{ij}})^6 \tag{2.3}$$

Electrons in an atom can move relativly freely within the shell of the atom. So when two atoms come closer, since two equal charges will repulse each other, the electrons of one atom will be pushed away from the other, while the electrons of the other, attracted by the positive nucleus, will move in between both atoms, thus creating two temporary dipoles, that will attract each other.

The repulsing part U_2 models the Pauli repulsion, that is created by two electron-clouds overlapping [BZBP14].

$$U_2 = 4\epsilon(\frac{\sigma}{r_{ij}})^{12} \tag{2.4}$$

The parameters ϵ and σ describe the strength of the bond between the two particles and the distance at which the van-der-Waal force becomes stronger than the Pauli repulsion respectively [GKZ07]. Figure 2.1 visualizes the Lennard-Jones potential with $\sigma = \epsilon = 1$.

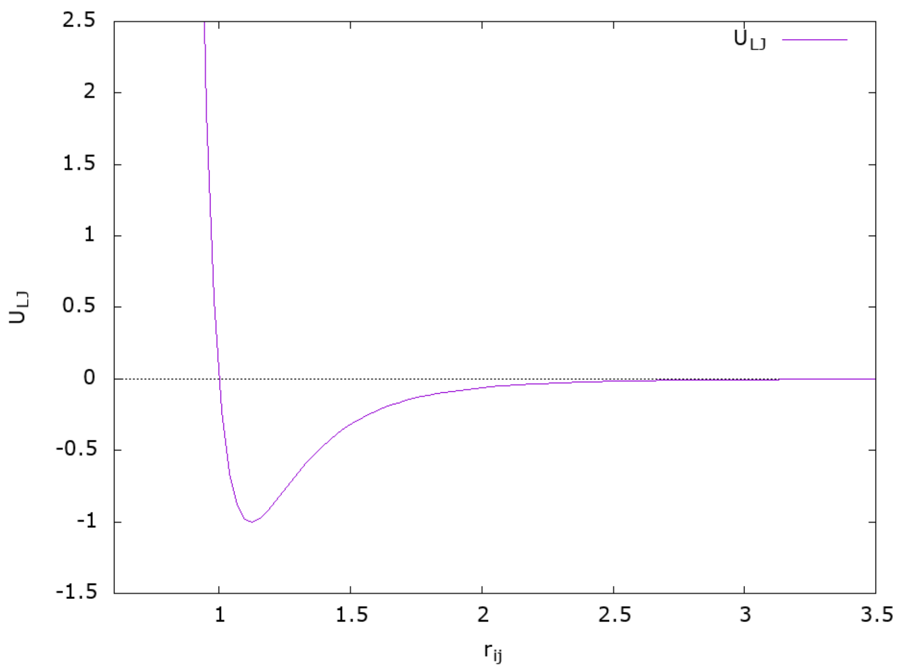


Figure 2.1.: Lennard-Jones 12-6 potential for $\sigma = \epsilon = 1$.

Since $(\frac{\sigma}{r_{ij}})^{12} > (\frac{\sigma}{r_{ij}})^6$ for $0 < \frac{\sigma}{r_{ij}} < 1$ and $(\frac{\sigma}{r_{ij}})^{12} < (\frac{\sigma}{r_{ij}})^6$ for $\frac{\sigma}{r_{ij}} > 1$, is the Lennard-Jones potential negativ, thus attracting, if $r_{ij} > \sigma$ and positiv, thus repulsing, if $r_{ij} < \sigma$. Furthermore, as r_{ij} approaches positive infinity, the potential will approach zero:

$$\lim_{r_{ij} \to \infty} U_{LJ}(r_{ij}) = 0 \tag{2.5}$$

2.2. Newton's Third Law

Newton's third law of motion states that to every action there is a reaction of equal magnitude and opposing direction [New87]. For the simulation of molecular dynamics, this means that the force exerted from a particle A on a particle B can be calculated by simply negating the force exerted from B on A, which effectively cuts the number of necessary distance- and force-calculation in half. Since those calculations take up the largest part of the runtime, this optimization can theoretically reduce the total runtime by up to 50%.

2.3. Algorithms

The simulation of molecular dynamics relies heavily on computing forces between two particles. How the simulation iterates through them can be done by different algorithms, three important ones described here.

2.3.1. Direct Sum

The simplest approach to computing intermolecular forces is the Direct Sum algorithm. This algorithm iterates through all possible combinations of two (different) particles and calculates the pairwise forces between them. Due to its lack of a complex overhead, this method can be efficient for small simulations.

However, even when making use of Newton's third law, a simulation of N particles results in $N * \frac{N-1}{2}$ unordered pairs, that all require a calculation of their distance. Therefore, the time complexity of this algorithm is $\mathcal{O}(N^2)$, which makes it unsuitable for larger simulations.

2.3.2. Linked Cells

As mentioned in Section 2.1 the force between two particles approaches zero, the further apart they are, so if the distance is sufficiently high, the interaction between particles barely influences their behavior and can, therefore, be neglected. Regarding molecular simulations, this means you can define a cutoff radius r_c limiting the distance after which forces are no longer calculated.

One algorithm to implement this optimization is the Linked Cells algorithm [GST+19], which divides the domain of the simulation into cubes (cells) with a side length, equal to (or slightly bigger than) the cutoff radius. It is also possible to choose a smaller cell-size to more closely approximate the interaction sphere and therefore further reduce the amount of unnecessary distance calculations while increasing the number of neighboring cells [MR99].

At the beginning of the simulation, all particles are assigned to different cells based on their position and every time a particle moves, it potentially needs to be moved to a different cell. During force-calculation, only particles within the same or neighboring cells need to be considered due to the aforementioned property of the Lennard-Jones Potential. By choosing the amount of cells proportional to the total number of particles, the number of particles within each cell, and therefore the possible pairs for every particle, remains constant, resulting in a time complexity of $\mathcal{O}(N)$.

While it should be clear, that this approach greatly reduces the amount of unneeded distance calculations compared to the Direct Sum algorithm, the probability of two particles in neighboring cells being within cutoff radius remains very low. With a cell size equal to the cutoff radius r_c, the probability can be calculated as follows:

$$\frac{CutoffVolume}{SearchVolume_{LC}} = \frac{\frac{4}{3}r_c^3\pi}{(3r_c)^3} \approx 0.155 \tag{2.6}$$

This means that there is only a 15.5% chance the particles are sufficiently close, leaving 84.5% unrequired calculations. Since maintaining the cells also adds a small memory and computational overhead, this algorithm is only beneficial for large enough simulations. Figure 2.2 visualizes the Linked Cells.

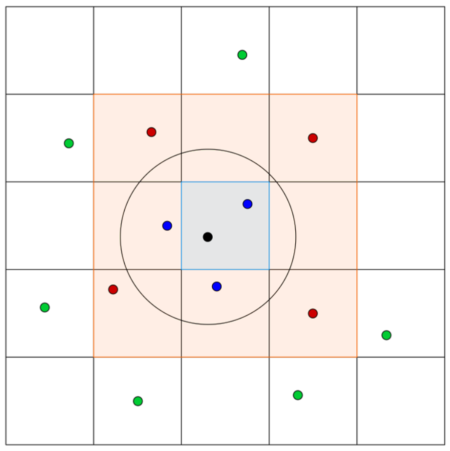


Figure 2.2.: Interactions between particles in the Linked Cells algorithm. Blue points are within the cutoff radius of the black point, red points are in neighboring cells but outside the cutoff radius, green points are completely outside interaction range.

Special care needs to be taken when running the simulation on multiple threads, as race conditions can occur when multiple threads try to update particles simultaneously. Two exemplary parallelization techniques are c08 and sliced:

c08 While processing one cell without further optimizations, particle updates are possible in the base cell and all 26 neighboring cells, resulting in 27 cells that cannot be modified by other threads. This number can be reduced by half by applying Newton's third law, so only cells with a higher index than the current one are considered for force calculations. Interactions between particles of the base cell and cells with a lower index are subsequently computed while processing the lower indexed cell. However, it is possible to reduce the number of modified cells even further. By calculating interactions between certain cells other than the base cell, only a square of 8 cells needs to be locked, which is visualized in Figure 2.3. The c08 traversal [TWG$^+$15] assigns every base cell one of eight colors in a way, that no two neighboring cells have the same color, as can be seen in Figure 2.4a. This allows all cells of the same color to be processed simultaneously in the described method without the possibility of race conditions. Since individual cells can be scheduled dynamically with OpenMP, this technique can yield good load balancing, while at the same time requiring an increased overhead [GST$^+$19].

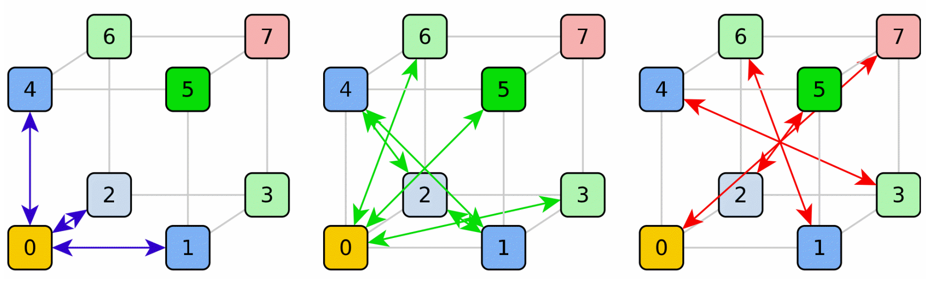
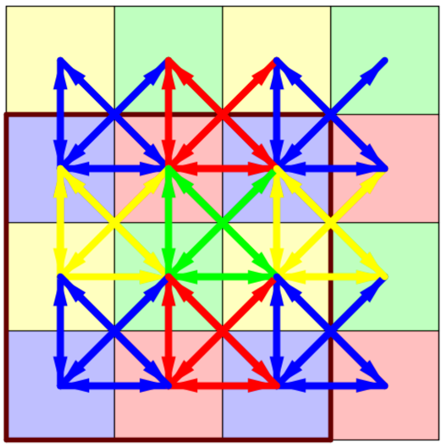


Figure 2.3.: Basic cell interactions. By calculating all displayed interactions for every cell, all interaction in the simulation are accounted for.
Source: [GST+19]

sliced This traversal divides the domain into several slices along one axis, which are then processed sequentially by one thread. In every slice the last plane is locked until the first plane in the neighboring slice is fully processed to prevent race conditions. Figure 2.4b shows a division of the domain into three slices. Due to the lack of a complex scheduling overhead, this approach works well in a homogenous system, as the last plane for each slice will probably be already unlocked when the thread reaches a cell in it. However, since the sliced traversal does not allow load balancing, it can be suboptimal in an inhomogeneous system. Furthermore, the total number of threads is limited by the size of the domain, as every slice has to be at least two cells wide [TSH+19].


(a) c08: Cells outside the brown box are halo cells.

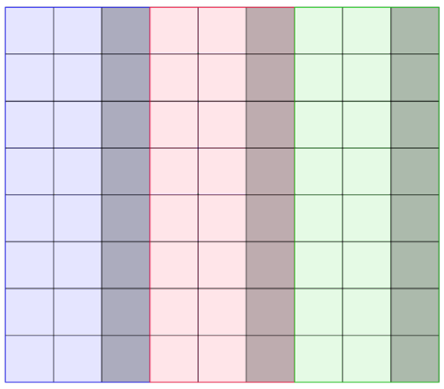

(b) sliced: Dark cells can only be processed after the next column has been processed entirely.

Figure 2.4.: Parallelization methods for the Linked Cells algorithm.

2.3.3. Verlet Lists

The Verlet Lists algorithm [Ver67] saves neighbor lists for each particle containing all particles within a specified radius equal to the cutoff radius plus a positive skin radius, as shown in

Figure 2.5. These lists need to be frequently updated, otherwise particles not initially inside the Verlet skin could have moved inside the cutoff radius without being included in force calculations.

The frequency of list updates depends on the velocity of particles and the skin radius. Every timestep the distance between two particles can be reduced by not more than $2 * d_{max}$, where d_{max} is the maximum distance traveled by any particle in a single timestep. The factor two is a consequence of the possibility of two particles moving directly towards each other. Therefore, the (additional) skin radius r_s has to be at least $r_s = 2 * d_{max} * q$, where q is the number of consecutive timesteps without rebuilding the Verlet Lists. If, for example, the Verlet Lists are updated every tenth timestep, then $q = 9$.

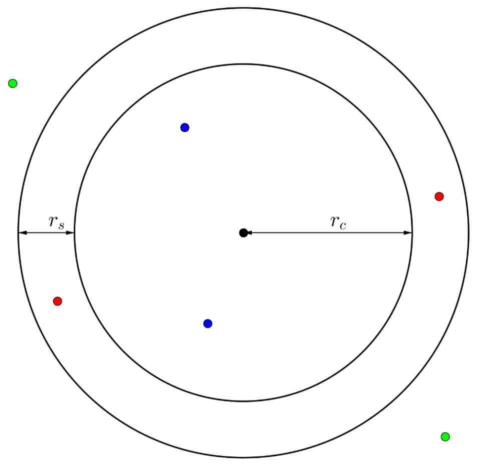


Figure 2.5.: Particle (black) with cutoff radius r_c and skin radius r_s. Blue and red particles are stored in the Verlet List and thus require distance calculations but only blue particles are considered for force calculations. Green particles are ignored entirely.

The probability of a particle in the Verlet List also being within cutoff radius can be calculated the same way as in Subsection 2.3.2:

$$\frac{CutoffVolume}{SearchVolume_{VL}} = \frac{\frac{4}{3}r_c^3\pi}{\frac{4}{3}(r_c + r_s)^3\pi} = \frac{1}{(\frac{r_c+r_s}{r_c})^3} \tag{2.7}$$

It becomes apparent, that the probability depends on the relative size of the skin radius r_s. For example, a skin radius $r_s = 0.2 * r_c$ results in a probability of $\frac{1}{1.2^3} \approx 0.579$, a significant increase compared to the Linked Cells algorithm. However, the need for maintaining the Verlet Lists adds extra computational and memory overhead.

Building the Verlet lists requires to compute the distance between all particles, so it has complexity $\mathcal{O}(N^2)$ if done unoptimized. It is, however, possible to use the Linked Cells algorithm to limit the number of distance calculations [YWLC04].

2.4. Data Layout

Intuitively one would store the data representing the state of the simulation as an array of particle objects. This layout is called Array-of-Structures (AoS). On the other hand, it is possible to instead store the individual attributes in their own arrays. The values representing one particle can then be found at the same index in those arrays. This layout is called Structure-of-Arrays (SoA).

Most modern processors offer the possibility to apply the same operation on multiple values at once, these operations are called Single Instruction Multiple Data (SIMD) instructions. Since most calculations within the simulation have to be repeated for multiple particles, this can be used to reduce runtime. However, when using AoS layout the data needed for those calculations are spread across the memory, hindering the effectiveness of this feature. This problem is resolved when SoA layout is used instead, as the data is already stored consecutively in memory.

2.5. Algorithm Selection

With several algorithms introduced, the question remains on how to decide which combination of algorithms to use, as the relative performance of different algorithms depends on the state of the simulation, as well as the employed hardware. Therefore, it is impossible to select one algorithm, that is guaranteed to always be the best.

When trying to find the best algorithm for a given problem, one first needs to determine what exactly shall be optimized. In simulations, two main aspects need to be considered. First, resource consumption should be minimized. These resources can be runtime, usage of memory or bandwidth, or other hardware related aspects. Second, the solution quality, as in the accuracy of the simulation, should be maximized. Often one aspect can be improved by worsening another, for example, can abstractions and approximations, like the Lennard-Jones potential, reduce runtime, while also reducing the precision of the result [Ewa12].

Therefore, it is necessary to define, which aspects are more important than others, or to select limits, determining acceptable values. In this context, optimizing the runtime of the simulation will be the main focus. All presented simulation options provide the same accuracy.

After specifying, how to evaluate different algorithms, one has now two options on how to perform the evaluation. It might be possible to judge an algorithm solely based on theoretical information about the algorithm and the problem definition. As an example, different parallelization methods of the Linked-Cells algorithm can be expected to perform better or worse depending on how homogenous the particles are distributed over the domain [GST^{+}19]. However, with an increasing amount of algorithms and millions of particles in the simulation, it becomes increasingly difficult to make definite statements about the relative performance of different methods.

Alternatively one can gather empirical data to evaluate algorithms. It would be simplest to only test each algorithm once at the beginning of the simulation, however over time the state of the simulation changes, which can lead to suboptimal performance. Therefore, it makes sense to regularly probe each algorithm and select the optimal, which is exactly what is currently done by the AutoPas library, although in a suboptimal manner.

Part II.

Active Harmony

3. Introduction to Active Harmony

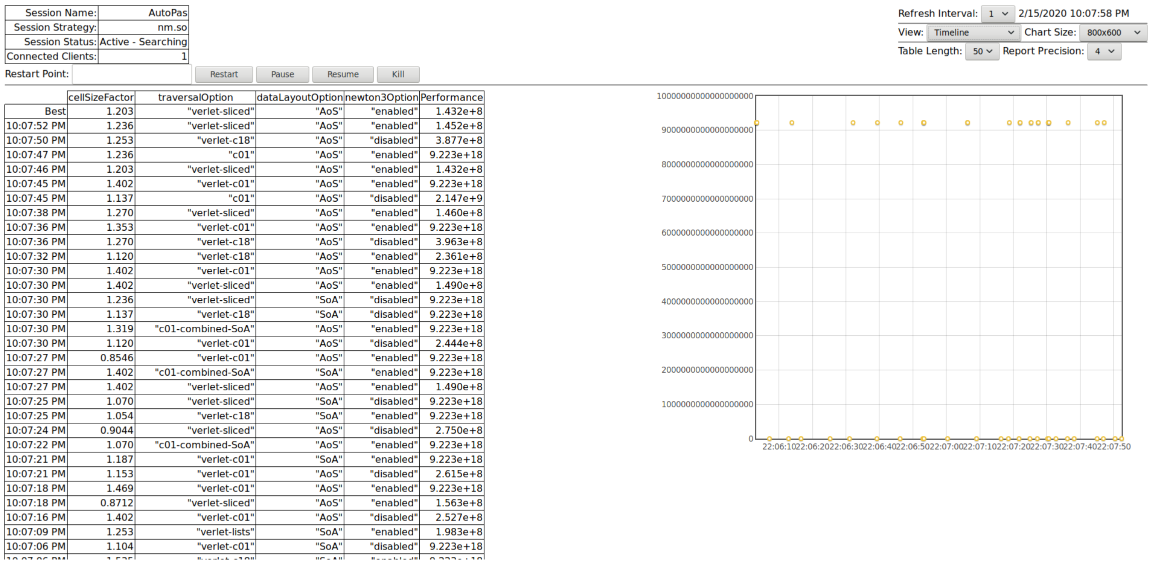
3.1. Concept of Active Harmony

Active Harmony [1] is a C library developed by the Paraydn[2]/Dyninst[3] project, used to find optimal values for parameters of another application, a process known as auto-tuning. These parameters are represented in Active Harmony as tuning parameters of type int, real, or enum. The int type and the real type are defined by a minimum value, a maximum value, as well as a step range, while the enum type is an explicit list of possible values. The cross product of all variables forms the search space.

Tuning begins with the tuning session informing the client application about the new point, so it can measure the performance of the specified configuration. This performance is then reported back to the tuning session, where it is analyzed according to the processing layers and search strategies described below. Active Harmony always aims to minimize the observed performance value. Therefore, if the client wants to instead maximize the performance value, it has to be negated before being passed to the tuning session.

One key feature of Active Harmony is, that it allows multiple clients to be part of the same tuning task. This is achieved by running an Active Harmony server, that handles the tuning process. Multiple clients can then connect to the server, to receive configurations and report performances. The hostname and port of the server are handled by environment variables HARMONY_HOST and HARMONY_PORT. One server can also handle multiple different tuning tasks simultaneously. By connecting to the server with a Javascript-enabled web browser, one can access detailed information as well as a visualization of the tuning process. Figure 3.1 shows an example of the browser interface.

[1] `https://github.com/ActiveHarmony/harmony/`
[2] `http://www.paradyn.org/index.html`
[3] `https://www.dyninst.org/`



Session Name:	AutoPas
Session Strategy:	nm.so
Session Status:	Active - Searching
Connected Clients:	1

Restart Point:

Refresh Interval: 1 2/15/2020 10:07:58 PM
View: Timeline Chart Size: 800x600
Table Length: 50 Report Precision: 4

	cellSizeFactor	traversalOption	dataLayoutOption	newton3Option	Performance
Best	1.203	"verlet-sliced"	"AoS"	"enabled"	1.432e+8
10:07:52 PM	1.236	"verlet-sliced"	"AoS"	"enabled"	1.452e+8
10:07:50 PM	1.253	"verlet-c18"	"AoS"	"disabled"	3.877e+8
10:07:47 PM	1.236	"c01"	"AoS"	"enabled"	9.223e+18
10:07:46 PM	1.203	"verlet-sliced"	"AoS"	"enabled"	1.432e+8
10:07:45 PM	1.402	"verlet-c01"	"AoS"	"enabled"	9.223e+18
10:07:45 PM	1.137	"c01"	"AoS"	"disabled"	2.147e+9
10:07:38 PM	1.270	"verlet-sliced"	"AoS"	"enabled"	1.460e+8
10:07:36 PM	1.353	"verlet-c01"	"AoS"	"enabled"	9.223e+18
10:07:36 PM	1.270	"verlet-c18"	"AoS"	"disabled"	3.963e+8
10:07:32 PM	1.120	"verlet-c18"	"AoS"	"enabled"	2.361e+8
10:07:30 PM	1.402	"verlet-c01"	"AoS"	"enabled"	9.223e+18
10:07:30 PM	1.402	"verlet-sliced"	"AoS"	"enabled"	1.490e+8
10:07:30 PM	1.236	"verlet-sliced"	"SoA"	"disabled"	9.223e+18
10:07:30 PM	1.137	"verlet-c18"	"SoA"	"disabled"	9.223e+18
10:07:30 PM	1.319	"c01-combined-SoA"	"AoS"	"enabled"	9.223e+18
10:07:30 PM	1.120	"verlet-c01"	"AoS"	"disabled"	2.444e+8
10:07:27 PM	0.8546	"verlet-c01"	"AoS"	"enabled"	9.223e+18
10:07:27 PM	1.402	"c01-combined-SoA"	"SoA"	"enabled"	9.223e+18
10:07:27 PM	1.402	"verlet-sliced"	"AoS"	"enabled"	1.490e+8
10:07:25 PM	1.070	"verlet-sliced"	"SoA"	"disabled"	9.223e+18
10:07:25 PM	1.054	"verlet-c18"	"SoA"	"enabled"	9.223e+18
10:07:24 PM	0.9044	"verlet-sliced"	"AoS"	"disabled"	2.750e+8
10:07:22 PM	1.070	"c01-combined-SoA"	"AoS"	"enabled"	9.223e+18
10:07:21 PM	1.187	"verlet-c01"	"SoA"	"enabled"	9.223e+18
10:07:21 PM	1.153	"verlet-c01"	"AoS"	"disabled"	2.615e+8
10:07:18 PM	1.469	"verlet-c01"	"AoS"	"enabled"	9.223e+18
10:07:18 PM	0.8712	"verlet-sliced"	"AoS"	"enabled"	1.563e+8
10:07:16 PM	1.402	"verlet-c01"	"AoS"	"disabled"	2.527e+8
10:07:09 PM	1.253	"verlet-lists"	"SoA"	"enabled"	1.983e+8
10:07:06 PM	1.104	"verlet-c01"	"SoA"	"disabled"	9.223e+18

Figure 3.1.: Active Harmony browser interface

3.2. Processing Layers

Processing layers can be used to modify data or perform operations before a strategy receives a performance report or right after the point is generated. Each layer generally consists of an analyze- and a generate-phase. Whenever the tuning session receives a performance report, it will pass through the analyze-phase of all layers, before being handed to the search strategy. After the next point has been selected, this point will pass through the generate-phase of all layers in the opposite order than during analyzation, before reaching the client application [Par]. The most important layers, that come bundled with Active Harmony, will be described below.

Aggregator can be used to reduce the impact of outliers by forcing multiple evaluations of the same configuration. Once the specified number of trials for one point is reached, a single performance report will be calculated according to the selected aggregation function. Available functions are minimum, maximum, mean, and median.

Point Caching/Replay layer saves reported performances in its analyze-phase, so this value can be reported back to the tuning process in its generate-phase if the same configuration is chosen later. Therefore, the target application only needs to evaluate every configuration once.

Input Grouping layer allows the user to split the input variables into groups, which will then be tuned individually. This means variables in one group will be adjusted until an optimal configuration is found, while variables in other groups remain unchanged.

Point Logger writes reported performance/configuration pairs to a specified file.

XML Writer same as Point Logger except the log file is in XML format.

Omega Constraint layer allows the user to define dependencies between multiple tuning variables. If a point generated by the search strategy does not fulfill those dependencies, it will be rejected and a new point will be selected. This processing layer requires an external program, the Omega Calculator[4], to work.

3.3. Search Strategies

Active Harmony has four built-in search strategies: exhaustive, random, Nelder-Mead and parallel rank order [Par].

Exhaustive explores the entire search space and selects the optimal configuration afterward. It is possible to specify how often the entire search space has to be evaluated before convergence.

Random randomly chooses points within the search space. It will never converge, but keep generating new points, even after the entire search space has been evaluated. However, it is still possible to manually interrupt the search task and select the so far best configuration.

Nelder-Mead this algorithm [NM65] uses a simplex defined by $(n + 1)$ points to find the local minimum of an n-dimensional function, which in this case is the function assigning every configuration its reported performance. In Active Harmony, this means that every point in the initial simplex and every newly calculated point has to be passed to the client application for evaluation before the next phase in the algorithm can begin.

In every cycle, the worst, meaning highest function value, point P_h is replaced by a better point through reflection, expansion, or contraction. The first step is always to calculate the point P_r by reflecting P_h on the point $\bar{P}$, which is defined as the centroid of all points in the simplex except P_h:

$$P_r = (1 + \alpha)\bar{P} - \alpha P_h \tag{3.1}$$

Here $\alpha > 0$ denotes the reflection coefficient, which is a multiplier for the distance between $\bar{P}$ and P_r. If the function value of P_r is lower than the maximum but higher than the minimum among the points of the simplex excluding P_h, then P_h is replaced by P_r and the cycle restarts by performing the reflection on the new simplex.

If, however, P_r is better than all other points, then the expanded point P_e is calculated through

$$P_e = \gamma P_r + (1 - \gamma)\bar{P} \tag{3.2}$$

with $\gamma > 1$ being the expansion coefficeent, that denotes the relative increase in distance between $\bar{P}$ and P_e. Then the cycle is restarted by replacing P_h with either P_r or P_e, whichever has the lower function value.

If P_r is worse than any other point ignoring P_h, then P_c is calculated by contraction:

$$P_c = \beta P_h + P(1 - \beta)\bar{P} \tag{3.3}$$

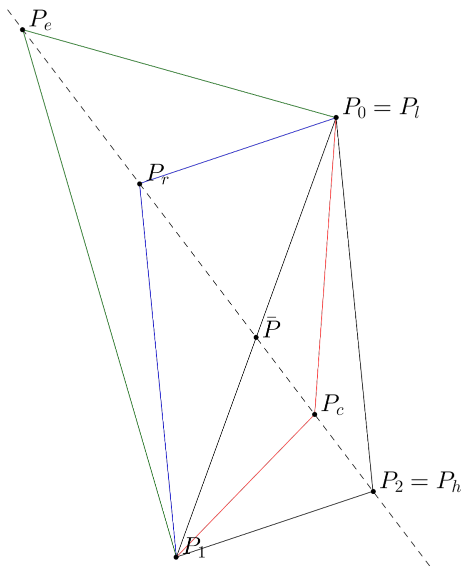
[4]`https://github.com/davewathaverford/the-omega-project/`

with $0 < \beta < 1$ as contraction coefficient. If P_c is better than both P_h and P_r, the cycle restarts with P_h being replaced by P_c

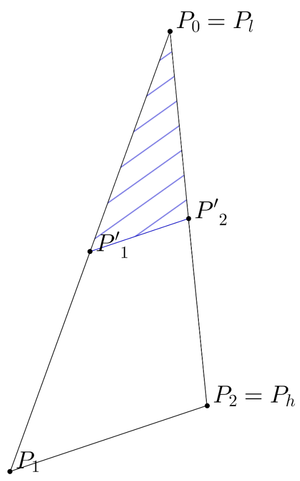
Otherwise, none of the above operations could produce a better point than P_h. In this case the entire simplex is shrunk by replacing every point P_i, except for the best point P_l, with P_i' defined by

$$P_i' = P_l + \sigma(P_i - P_l) \tag{3.4}$$

with $i \neq l$ and $0 < \sigma < 1$ as shrinking coefficient. The resulting simplex is then used to restart the cycle.



(a) Example for reflection (blue), expansion (green) and contraction (red) with $\alpha = 1$, $\beta = 0.5$, $\gamma = 2$



(b) Example for shrinking with $\sigma = 0.5$

Figure 3.2.: Examples for Nelder-Mead operations with a two-dimensional simplex.

This is repeated until a given convergence criterion is reached. One such criterion is the difference of function values falling below a certain threshold, which is also one of two criteria Active Harmony is using. The second criterion for Active Harmony is the relative volume of the simplex becoming too small. Figure 3.2 shows a visual example for all operations on a two-dimensional simplex.

Active Harmony uses $\alpha = 1$, $\beta = 0.5$, $\gamma = 2$, $\sigma = 0.5$ as defaults. Also, the Nelder-Mead algorithm is defined for continuous numbers, but ActiveHarmony has a discrete parameter space. This issue is resolved by rounding the resulting values to the next integer [Chu04]. Enum tuning variables are assigned an integer value for every option, starting with zero and increased by one in the order, in which the options are defined.

Parallel Rank Order works similarly to the Nelder-Mead strategy, but is better suited for parallel tuning, as it allows evaluating multiple points in the search space simultaneously. However, as the tuning within AutoPas is done on a per-timestep basis and timesteps have to be computed sequentially, the Nelder-Mead strategy is used in this implementation.

4. AutoPas

4.1. Introduction to AutoPas

AutoPas[1] — a part of the TalPas[2] project — is a C++ library aimed at optimizing node-level performance for particle simulations [GST+19]. This is achieved by implementing multiple algorithms for force calculations and selecting an optimal configuration during runtime to adapt to changing simulation environments. AutoPas acts as a black box to provide optimal performance for users without a need for an in-depth understanding of the advantages and disadvantages of different algorithms.

4.1.1. AutoPas Interface

The main interface between the AutoPas library and the user is the AutoPas class, which offers functionality to insert and remove particles from the simulation, as well as to iterate over all particles. It also handles general simulation parameters, such as domain size, cutoff radius, tuning interval, and allowed options for tuning parameters described in Subsection 4.1.2.

To make the library adaptable to different simulations types, users can implement their own particle- and functor-class by inheriting from their respective base-class. A particle class has to have member variables for at least position, velocity, acting force and ID of the particle, as well as if it belongs to the current AutoPas object. Additional requirements depend on the used functor. The functor defines how to compute interactions between particles, which is usually based on various forces and potentials to simulate different physical phenomenons. A functor and particle class for the Lennard-Jones 12-6 potential is part of the default AutoPas distribution.

4.1.2. Tuning

AutoPas will begin the tuning phase at simulation start or after a specified amount of non-tuned iterations have passed. During the tuning phase, a configuration is generated by the chosen tuning strategy. The configuration defines values for the following parameters:

Particle Container is the basic algorithm such as the ones described in Section 2.3.

Traversal specifies in which order particles are traversed. Particle containers can allow different traversal, but every traversal can only be used in combination with one container. Subsection 2.3.2 specifies two different traversals for the Linked Cells container.

Data Layout AoS or SoA as described in Section 2.4.

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

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

Cell Size Factor multiplier for the sidelength of cells in the Linked Cells algorithm.

Newton-3 wether the optmization using Newton's third law (see Section 2.2) should be used or not.

If the configuration is invalid, for example, because the traversal does not support the selected Newton-3 or Data Layout option, a new configuration is generated until a valid one is found. This configuration is then used for a predefined number of timesteps to measure its performance. Multiple measurements will be reduced to a single one according to a specified strategy, such as calculating the minimum, mean, or median of all measurements.

This process is repeated until the tuning strategy indicates that it found an optimal configuration, which is then used for a fixed number of time steps.

Currently, AutoPas offers three tuning strategies apart from Active Harmony:

Full Search works just as Active Harmony's exhaustive search by measuring the performance of every configuration and selecting the best afterward.

Random Search similarly to Active Harmony's random search configurations are chosen randomly, but unlike Active Harmony's version, this one stops after a predefined number of configurations were tried and selects the best of those.

Bayesian Search models the stochastic distribution of performances as a Gaussian Process to estimate the benefit of measuring a certain configuration next. It stops after evaluating a specified number of configurations.

4.2. Integrating Active Harmony into AutoPas

To integrate Active Harmony into AutoPas a new class ActiveHarmony inheriting from the TuningStrategyInterface was created. It acts as an additional TuningStrategy option, which can be chosen at initialization.

Active Harmony requires an environment variable pointing to the installation directory to be set. Therefore, cmake was used to define a macro containing the path to the Active Harmony directory, which is then used to set the environment variable.

The Active Harmony search task is set up by first initiating the Harmony descriptor

```
hdesc = ah_alloc();
if (hdesc == nulltpr) {
  exception("Error allocating Harmony descriptor.");
}
```

Listing 4.1: Allocation of the harmony descriptor

which is then connected to a newly started Harmony session:

```
if (ah_connect(hdesc, nullptr, 0) != 0) {
  exception("Error connecting to Harmony session.");
}
```

Listing 4.2: Connection to the Harmony session

At this step, it is also possible to connect to an already running Active Harmony server by providing its address and port. However, as tuning in AutoPas is currently done within a single instance of AutoPas, a local session suffices.

After setting up the tuning session, one now has to create a task definition, holding information about tuning variables, potential processing layers, and the search strategy. The definition also assigns a name to the search task, which is used to identify different tasks if a server is hosting multiple tuning processes.

```
hdef_t *hdef = ah_def_alloc();
if (hdef == nullptr) {
  exception("Error allocating definition descriptor");
}

if (ah_def_name(hdef, "AutoPas") != 0) {
  exception("Error settings search name");
}

ah_def_strategy(hdef, "nm.so");
ah_def_cfg(hdef, "INIT_RADIUS", "0.5");
```

Listing 4.3: Setup of the tuning task definition

Most tuning variables were defined as an enum, as shown in Listing 4.4, with the allowed cell size factors being the only special case for now. Allowed cell size factors can be defined by a finite or infinite number set. Since both number types of Active Harmony (int and real) are defined as a range of equidistant points, a finite number set has to be declared as an enum, just like the other parameters. Infinite number sets, however, are defined as a real with step size $\frac{max-min}{100}$, with min being the lower bound of the allowed cell size factors and max the upper bound. This adjusts the step size to always fit 100 sample points into the number set.

Trivial parameters — parameters with only one possible value — are not forwarded to Active Harmony, instead, they are automatically set, when the selected configuration is handed to AutoPas. Another parameter, that is set automatically, is the particle container option. As mentioned in Subsection 4.1.2, every traversal is only applicable for one particle container, therefore it makes sense to only tune the traversal, and set the container accordingly.

```
void configureTuningParameter(hdef_t *hdef, const char *name, const std::set<O
    > options) {
  if (options.size() > 1) {
    if (ah_def_enum(hdef, name, nullptr) != 0) {
      exception("Error defining enum.");
    }
    for (auto &option : options) {
      if (ah_def_enum_value(hdef, name, option.to_string().c_str()) != 0) {
        exception("Error defining enum value.");
      }
    }
  }
}
```

Listing 4.4: Definition of tuning variables

Tuning variables can also be provided with a — depending on the tuning variable type — string, int, or double pointer, which allows Active Harmony to update variables automatically upon configuration retrieval.

As of now, no processing layers (see Section 3.2 for detailed information) are used, as the aggregation of time measurements is already handled by AutoPas and logging by md-flexible. The Point Caching/Replay layer would be useful, however, the necessity of sometimes needing to restart the Harmony search task during one tuning phase requires a manual implementation of this feature.

This behavior is a consequence of the search space containing invalid configurations, usually because the selected traversal does not support the selected data layout option. In most cases, this is no problem, since reporting the worst possible performance — the upper bound of the double data type — ensures, that every valid performance is better and therefore, the invalid configuration will not be selected as the optimal configuration. However, if the initial simplex in the Nelder-Mead algorithm consists of solely invalid configurations, it will immediately converge and be locked in a state, where all considered configurations are invalid. The only way to resolve this is by restarting the search task to create a new simplex. Therefore, all evidence is stored in a map, mapping configurations to their recorded performance. Every time a configuration with an entry in this map is generated by the search strategy, its performance is immediately reported back to Active Harmony, to create a new configuration.

```
htask = ah_start(hdesc, hdef);
ah_def_free(hdef);
if (htask == nullptr) {
  exception("Error starting task.");
}
```

Listing 4.5: Start of the tuning task

5. Evaluation

5.1. Simulation Environment

5.1.1. md-flexible

The program md-flexible [Fot19] was used to run the simulation. It is an example of a simulation program using the AutoPas library and is part of the AutoPas repository. Particles are initiated as a grid in the shape of a cube or sphere. It is also possible to distribute particles randomly with uniform or Gaussian distribution within a cube. The specification of the simulated scenario can be provided either via the command line or as a YAML[1] configuration file. To make it easier to re-run the simulation with small changes, YAML files were used as input.

5.1.2. Linux-Cluster

All simulations were run on the CoolMUC-2 segment of the Linux Cluster of the Leibniz Rechenzentrum (LRZ). The Hardware of the cluster is described in Table 5.1.

	CooLMUC 2
CPU	Intel Xeon E5-2690 v3
CPU Architecture	Haswell
Frequency	2.6 GHz
Number of Nodes	384
Cores per Node	28
Hyperthreads per Core	2
Memory per Node	64

Table 5.1.: CoolMUC-2 overview.
Source: `https://doku.lrz.de/display/PUBLIC/CoolMUC-2`

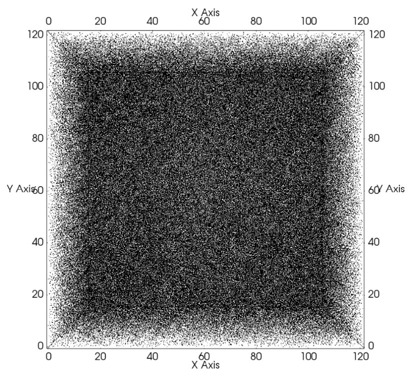
5.2. Simulation of Spinodal Decomposition

To compare the Active Harmony implementation with existing tuning strategies, a simulation of spinodal decomposition was performed. This phenomenon describes the rapid unmixing of a homogenous solution, which results in the creation of two different thermodynamic phases [CHK+94]. In the simulation, this experiment has to be performed in two parts: An initial equilibration phase, followed by a decomposition phase.

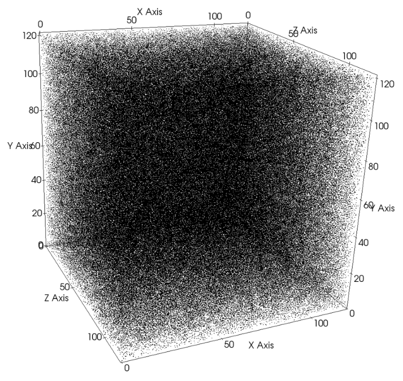
[1] `https://yaml.org/`

5.2.1. Equilibration

The simulation is initiated with a grid of 512000 particles, which are then equilibrated. This is done by initializing the velocity of the particles according to Brownian motion and then running the simulation for 100000 timesteps while keeping the temperature constant. After the particles are equilibrated a vtk-file is created, that is used by the second simulation as a checkpoint. The state of the simulation at the end of the equilibration can be seen in Figure 5.1.



(a) frontal view

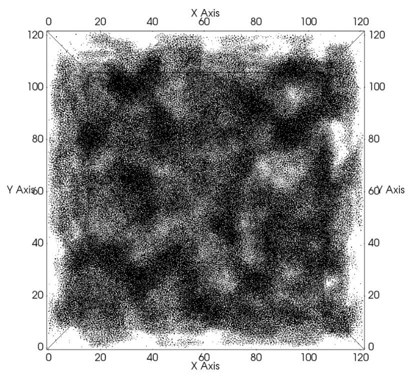


(b) side view

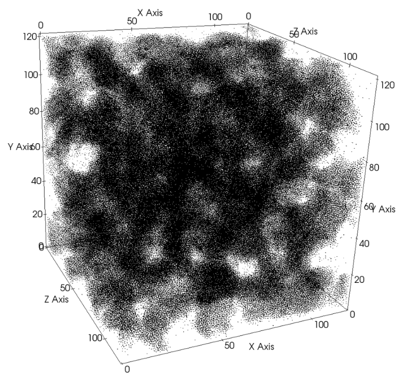
Figure 5.1.: End of equilibration phase with 512000 particles.

5.2.2. Decomposition

For the simulation of the actual decomposition, the previously generated checkpoint is used to initialize the particles. The simulation is then continued for 30000 timesteps with the same parameters as the equilibration, except with half the temperature. This causes the particles to form clusters while creating large areas where only a few particles are located, which can be seen in Figure 5.2. It is because of this change into an inhomogenous state, that this experiment is a well suited application for autotuning as requirements to the algorithm change during the simulation.



(a) frontal view



(b) side view

Figure 5.2.: End of decomposition phase with 512000 particles.

5.3. Results

To evaluate the performance of Active Harmony, the same simulation was also performed using the already in AutoPas existing tuning strategies bayesian-search and full-search. The tuning parameters mentioned in Subsection 4.1.2 were mostly set to allow all possible options. Only the particle container option — and subsequently the traversal option — was restricted to all options except DirectSum, VerletClusterLists, and VerletClusterCells. The first was excluded to improve overall performance, the latter two because they were dysfunctional at the time of the simulation. Every 1000 non-tuning iterations a new tuning phase was performed, in which every configuration was evaluated three times using the minimum as the final value. All simulation parameters were defined by a YAML file, which can be found in Appendix A.

Table 5.2 shows performance measurements for the equilibration phase of the simulation and Table 5.3 for the decomposition phase. One immediately notices that using full-search led to a significant increase in runtime compared to both other tuning strategies. This is hardly surprising since it is guaranteed to select every suboptimal algorithm multiple times while adjusting other tuning parameters. Although full exploration of the search space guarantees to find the optimal configuration, which results in full-search showing the lowest average time per iteration outside the tuning phase, this cannot compensate for the time lost during the tuning phase. Also, having such a large search space, and therefore a long tuning phase, contributes to the poor performance of full-search. Furthermore, the combination of the length of the tuning phase and the high runtime during it, makes full-search perform particularly poor with small tuning intervals.

	harmony	bayesian	full
Total time spent on force calculations in hours	12.26	18.30	*terminated after 96 h*
while tuning	2.44	4.52	
while not tuning	9.82	13.76	
Iterations used for tuning	6588	2940	
Average time per iteration during tuning	1.336	5.530	
Average time per iteration during non-tuning	0.378	0.511	

Table 5.2.: Runtime analysis of the equilibration phase of the simulation

	harmony	bayesian	full
Total time spent on force calculations in hours	5.75	7.02	32.68
while tuning	1.18	2.11	30.50
while not tuning	4.58	4.91	2.18
Iterations used for tuning	2502	900	15000
Average time per iteration during tuning	1.692	8.453	7.320
Average time per iteration during non-tuning	0.599	0.607	0.523

Table 5.3.: Runtime analysis of the decomposition phase of the simulation

The difference between bayesian-search and active-harmony is smaller but with a 50% higher runtime during equilibration and a 22% higher runtime during decomposition still noticeable. Upon further inspection, this can be largely accredited to a better exploration of the search space by active-harmony. This means active-harmony selected less suboptimal configurations during tuning, which is shown by the 78% lower average time per iteration, while maintaining, or even improving, the quality of the ultimately chosen configuration. The second part is explained by the different criteria on when to finish the tuning process. Bayesian-search always stops after a specified number of configurations have been tested, which means a close to optimal configuration might have not been found before tuning is interrupted. On the other hand, active-harmony keeps generating new configurations, until it can no longer significantly increase the performance of the optimal solution. Furthermore, the fact that active-harmony performed relatively well during the tuning phase suggests that it is particularly well suited for fast-changing simulation environments, which require frequent tuning to achieve optimal performance.

Part III.

Conclusion

5.4. Conclusion

In this thesis, the auto-tuning library Active Harmony was successfully integrated into AutoPas. The simulation of the spinodal decomposition mechanism was then used to evaluate this library, which showed a significant performance increase over other tuning strategies that are currently available in AutoPas. Compared to the bayesian-search strategy, for example, a reduction in total runtime of about 18% was achieved. This can mostly be explained by Active Harmony's low average time per iteration during the tuning phase, which suggests that it generates only a few bad configurations by using the Nelder-Mead algorithm. Therefore, it can be expected that Active Harmony's advantage over the other tuning strategies will be particularly noticeable when more frequent tuning is required.

Another interesting part of Active Harmony, that might be subject to future work, is its ability to tune multiple clients at once, especially since molecular dynamics simulations often undergo heavy parallelization. This feature could potentially be used to allow similar subdomains of a large simulation to be tuned together, which would significantly reduce tuning time.

Part IV.

Appendix

A. YAML-input files

```
container                        :   [LinkedCells, VerletLists,
    VerletListsCells, VarVerletListsAsBuild]
verlet-rebuild-frequency         :   10
verlet-skin-radius               :   0.3
verlet-cluster-size              :   4
selector-strategy                :   Fastest-Absolute-Value
data-layout                      :   [AoS, SoA]
traversal                        :   [c08, sliced, c18, c01, verlet-sliced,
    verlet-c18, verlet-c01, cuda-c01, verlet-lists, c01-combined-SoA, c04,
    var-verlet-lists-as-build, c04SoA]
tuning-strategy                  :   active-harmony
tuning-interval                  :   1000
tuning-samples                   :   3
tuning-max-evidence              :   10
functor                          :   Lennard-Jones (12-6)
newton3                          :   [enabled, disabled]
cutoff                           :   2.5
box-min                          :   [-0.25, -0.25, -0.25]
box-max                          :   [121.25, 121.25, 121.25]
cell-size                        :   [0.34-2]
deltaT                           :   0.00182367
iterations                       :   100000
periodic-boundaries              :   true
Objects:
  CubeGrid:
    0:
      particles-per-dimension    :   [80, 80, 80]
      particle-spacing           :   1.5
      bottomLeftCorner           :   [0, 0, 0]
      velocity                   :   [0, 0, 0]
      particle-type              :   0
      particle-epsilon           :   1
      particle-sigma             :   1
      particle-mass              :   1
thermostat:
  initialTemperature             :   1.4
  targetTemperature              :   1.4
  deltaTemperature               :   2
  thermostatInterval             :   10
  addBrownianMotion              :   true
vtk-write-frequency              :   100000
vtk-filename                     :   SpinodalDecomposition_equilibration
```

Listing A.1: YAML input file for the equilibration phase of the spinodal decomposition

```
container                        :   [LinkedCells, VerletLists,
    VerletListsCells, VarVerletListsAsBuild]
```

```
verlet-rebuild-frequency          :  10
verlet-skin-radius                :  0.3
verlet-cluster-size               :  4
selector-strategy                 :  Fastest-Absolute-Value
data-layout                       :  [AoS, SoA]
traversal                         :  [c08, sliced, c18, c01, verlet-sliced,
    verlet-c18, verlet-c01, cuda-c01, verlet-lists, c01-combined-SoA, c04,
    var-verlet-lists-as-build, c04SoA]
tuning-strategy                   :  active-harmony
tuning-interval                   :  1000
tuning-samples                    :  3
tuning-max-evidence               :  10
functor                           :  Lennard-Jones (12-6)
newton3                           :  [disabled, enabled]
cutoff                            :  2.5
box-min                           :  [-0.3, -0.3, -0.3]
box-max                           :  [121.3, 121.3, 121.3]
cell-size                         :  [0.34-2]
deltaT                            :  0.00182367
iterations                        :  30000
periodic-boundaries               :  true
thermostat:
  initialTemperature              :  0.7
  targetTemperature               :  0.7
  deltaTemperature                :  2
  thermostatInterval              :  10
  addBrownianMotion               :  false
vtk-filename                      :  SpinodalDecomposition
vtk-write-frequency               :  30000
checkpoint                        :  SpinodalDecomposition_equilibration_100000
    .vtk
```

Listing A.2: YAML input file for the decomposition phase of the spinodal decomposition

List of Figures

2.1. Lennard-Jones potential 4
2.2. Interactions between particles in the Linked Cells algorithm 6
2.3. Basic cell interactions 7
2.4. Parallelization methods for the Linked Cells algorithm 7
2.5. Verlet Skin 8

3.1. Active Harmony browser interface 12
3.2. Examples for Nelder-Mead operations 14

5.1. End of equilibration phase 21
5.2. End of decomposition phase 21

List of Tables

5.1. CoolMUC-2 overview 20
5.2. Runtime analysis of the equilibration phase of the simulation 22
5.3. Runtime analysis of the decomposition phase of the simulation 23

Bibliography

[Bro87] Frederick P. Brooks. No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, 20:10–19, 1987.

[BZBP14] Hans-Joachim Bungartz, Stefan Zimmer, Martin Buchholz, and Dirk Pflüger. *Modeling and simulation.* Springer Undergraduate Texts in Mathematics and Technology. Springer, 2014.

[CHK$^+$94] J. Clarke, J. Hastie, L. Kihlborg, R. Metselaar, and M. Thackeray. Definitions of terms relating to phase transitions of the solid state (iupac recommendations 1994). *Pure and Applied Chemistry*, 66(3):577–594, 01 1994.

[Chu04] I-Hsin Chung. Towards automatic performance tuning. 11 2004.

[Ewa12] Roland Ewald. *Automatic Algorithm Selection for Complex Simulation Problems.* Springer, 2012.

[Fot19] Nicola Fottner. Developing and benchmarking a molecular dynamics simulation using autopas. Bachelorarbeit, Technical University of Munich, Sep 2019.

[GKZ07] Michael Griebel, Stephan Knapek, and Gerhard Zumbusch. *Numerical Simulation in Molecular Dynamics*, volume 5 of *Texts in Computational Science and Engineering.* Springer, 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.

[LJ24] J. E. Lennard-Jones. On the determination of molecular fields. ii. from the equation of state of a gas. *Proceedings of the Royal Society of London. Series A, Containing Papers of a Mathematical and Physical Character*, 106(738):463–477, 1924.

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

[New87] Issac Newton. *Philosophiae naturalis principia mathematica.* 1687.

[NM65] J. A. Nelder and R. Mead. A Simplex Method for Function Minimization. *The Computer Journal*, 7(4):308–313, 01 1965.

[Par] Paradyn/Dyninst. Active harmony user's guide. https://dyninst.org/sites/default/files/manuals/harmony/v4.6.0/Users_Guide.pdf. [Online; accessed 2020-02-12].

[TSH$^+$19] Nikola Tchipev, Steffen Seckler, Matthias Heinen, Jadran Vrabec, Fabio Gratl, Martin Horsch, Martin Bernreuther, Colin Glass, Christoph Niethammer, Nicolay Hammer, Bernd Krischok, Michael Resch, Dieter Kranzlmüller, Hans Hasse, Hans-Joachim Bungartz, and Philipp Neumann. Twetris: Twenty trillion-atom simulation. *International Journal of High Performance Computing Applications*, 33:838–854, 09 2019.

[TWG$^+$15] Nikola Tchipev, Amer Wafai, Colin W. Glass, Wolfgang Eckhardt, Alexander Heinecke, Hans-Joachim Bungartz, and Philipp Neumann. Optimized force calculation in molecular dynamics simulations for the intel xeon phi. In *Euro-Par 2015: Parallel Processing Workshops*, pages 774–785, Cham, 2015. Springer International Publishing.

[Ver67] Loup Verlet. Computer "experiments" on classical fluids. i. thermodynamical properties of lennard-jones molecules. *Phys. Rev.*, 159(1):98–103, Jul 1967.

[YWLC04] Zhenhua Yao, Jian-Sheng Wang, Gui-Rong Liu, and Min Cheng. Improved neighbor list algorithm in molecular simulations using cell decomposition and data sorting method. In *Computer Physics Communications*, volume 161, pages 27–35, 2004.