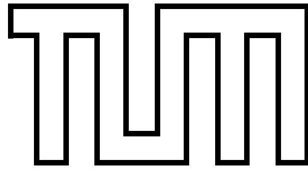


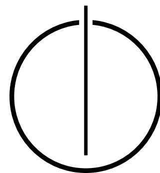
FAKULTÄT FÜR INFORMATIK

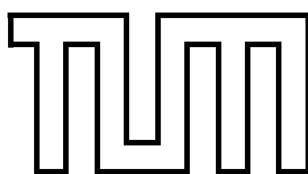
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

MPI-Parallel tuning strategies for inhomogeneous scenarios in AutoPas

Johannes Kroll





FAKULTÄT FÜR INFORMATIK

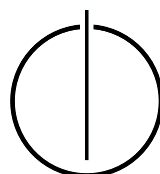
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

MPI-Parallel tuning strategies for inhomogeneous scenarios in AutoPas

MPI-Parallele Tuningstrategien für inhomogene Szenarien in AutoPas

Author:	Johannes Kroll
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisor:	Fabio Alexander Gratl, M.Sc.
Date:	16.08.2021



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

Munich, 16.08.2021 Johannes Kroll

Acknowledgements

First and foremost, I would primarily like to thank my advisors, Fabio Gratl and Steffen Seckler. Thank you for being patient, answering all my questions, and responding to emails late at night. Next, I will thank my parents, sister, and friend, Mirella, who all read through drafts of this thesis. I would also like to thank my two roommates, Kathi and Till, who gave me a pleasant time and encouraged me to work on this thesis. Last but not least, I would like to thank the Hanns-Seidel-Stiftung for supporting me throughout my past six semesters by granting me a scholarship.

Abstract

Molecular-Dynamics-Simulations are computationally highly demanding but can produce very deep insights in Chemistry or particle physics. In order to perform simulations more efficiently, a vast number of strategies and optimizations are available. Because of this large selection, the simulation must automatically be adjusted during run-time. This process of testing and selecting options is called auto-tuning. AutoPas is a C++ particle simulation library, which provides this functionality for simulations implemented by the user [7]. It is possible to parallelize this tuning process by sharing the workload with other processes. However, this was previously only possible if the simulated scenarios were homogeneous. This thesis extends this parallelization scheme in order to use it for inhomogeneous scenarios. The communication is based on MPI [6]. The new implementation is compared with the old one and also with not using parallel tuning. Eventually, significant performance improvements can be achieved, but only at the cost of more manual configuration. This could be solved in the future by also automating the selection of the new configuration parameters.

Zusammenfassung

Molekulardynamik-Simulationen sind sehr rechenintensiv, können aber sehr tiefe Einblicke in die Chemie oder Teilchenphysik liefern. Um Simulationen effizienter durchführen zu können, steht eine große Anzahl von Strategien und Optimierungen zur Verfügung. Aufgrund dieser großen Auswahl ist es notwendig, dass die Simulation während der Laufzeit automatisch angepasst wird. Dieser Prozess des Testens und Auswählens von Optionen wird als Auto-Tuning bezeichnet. AutoPas ist eine C++-Bibliothek für Partikel-Simulationen, die diese Funktionalität für die vom Benutzer implementierten Simulationen bereitstellt [7]. Es ist möglich, dieses Prozedere zu parallelisieren, indem die Arbeitslast auf andere Prozesse verteilt wird. Allerdings war dies bisher nur möglich, wenn die simulierten Szenarien homogen waren. In dieser Arbeit wird dieses Parallelisierungs-Schema erweitert, um es auch für inhomogene Szenarien nutzen zu können. Die Kommunikation basiert auf MPI [6]. Die neue Implementierung wird mit der alten und auch mit der ohne Parallelisierung verglichen. Letztendlich können signifikante Leistungsverbesserungen erzielt werden, allerdings nur auf Kosten von mehr manueller Konfiguration. Dies könnte in Zukunft dadurch gelöst werden, dass auch die Auswahl der neuen Konfigurationsparameter automatisiert wird.

Contents

Acknowledgements iv

Abstract v

Zusammenfassung vi

I. Introduction and Background 1

1. Introduction 2

2. Theoretical Background 3
2.1. Molecular-Dynamics-Simulations 3
2.1.1. The Lennard-Jones 12-6 Potential 3
2.1.2. Computational Challenges 4
2.1.3. Algorithmic approaches 4
2.2. Auto-tuning 5

3. Technical Background 7
3.1. AutoPas 7
3.1.1. Options for performance optimization 7
3.1.2. Tuning-strategies 9
3.2. Message Passing Interface 10

II. Implementation 12

4. Comparing scenarios 13
4.1. Finding parameters to determine similarity 13
4.2. Homogeneity 14
4.3. Maximum Density 17
4.4. Correlation of parameters and properties of scenarios 17
4.5. Similarity-Metric 20

5. Realization in AutoPas 22
5.1. Auto-tuning in AutoPas 22
5.2. Smoothing 23
5.3. Communication and distribution into buckets 23

III. Results 25

6. Overview 26
6.1. Expectation 26
6.2. Method and Background 26
6.3. Scenarios ans Strategies 27

7. Measurements 30
7.1. Total execution time 30
7.2. Execution time of tuning and not tuning iterations 32
7.3. Quality of tuning 34
7.4. Comparison with new set of scenarios 36

IV. Conclusion 37

8. Conclusion and future Work 38

V. Appendix 39
8.1. yaml file of scenario A 40

Bibliography 43

Part I.

Introduction and Background

1. Introduction

Simulations of particles and their interaction with one another can lead to very deep insights into Chemistry, Biology, and Physics. For Biology research, the transformation of a normal protein into a disease-causing agent can be modeled [8]. Another example is modeling various coefficients of a material, such as viscosity or thermal conductivity, to gain knowledge in material sciences [4]. Many of these simulations have a large number of particles in common, which often represent single atoms or molecules. Since iterations can occur very quickly on this small scale, the time step which each iteration of the simulation represents has to be very short [1]. This leads to a huge number of iterations needed to achieve a meaningful result. Combined with a large number of particles, such simulations can be computationally highly demanding. Therefore immense computational power is needed to conduct Molecular-Dynamics-Simulations. However, this alone will not solve a problem of high computational complexity. Optimizations and efficient algorithms are needed, which will be explained in Subsection 2.1.2 and Subsection 2.1.3.

Choosing the optimal strategy is not trivial, as the efficiency of the algorithms depends on the input or, in this case, the scenario to be simulated. The fact that many scientists who use these simulations are not themselves computer scientists does not help either. To solve this, the simulation automatically tests optimizations and strategies in order to choose the best one. This is called auto-tuning and is addressed in Section 2.2. AutoPas is a C++ software library for particle simulations, which extensively uses auto-tuning for more efficient computation. In AutoPas, tuning is done regularly during the run-time of a simulation. Details to this can be found in Section 3.1.

Particle simulations are often computed on systems with many processors and other computational units (from here on nodes). Therefore they are highly optimized to run in parallel. Also, the tuning payload is distributed to reduce overall execution time. However, this is a relatively new feature in AutoPas and is only applicable when the simulated scenario is homogeneous. This means the sub-scenarios, computed by the nodes, are very similar, and therefore tuning results apply to all nodes [16]. The goal of this thesis is to extend the parallel tuning to also work for inhomogeneous scenarios. In order to achieve this, a metric to compare scenarios have to be found. The thesis will show how much the overall performance is improved by this new feature.

2. Theoretical Background

2.1. Molecular-Dynamics-Simulations

Molecular-Dynamics-Simulations (MD-simulations) deal with the interactions of particles. These particles usually represent atoms or molecules. The interactions resulting from the forces that these particles exert on each other are also calculated by the simulation. Each particle is thereby accelerated by Newton's laws of motion and could move in the neighborhood of other particles [9]. These forces could be electrical because of a specific charge of the particles, gravitational in one direction to simulate a scenario in a gravitational field, or the strong interaction known as one of the four fundamental interactions in physics.

2.1.1. The Lennard-Jones 12-6 Potential

One model of this complex, strong integration is the Lennard-Jones 12-6 Potential. This potential is used in a wide variety of MD-simulations due to its mathematical simplicity, but realistic results [2].

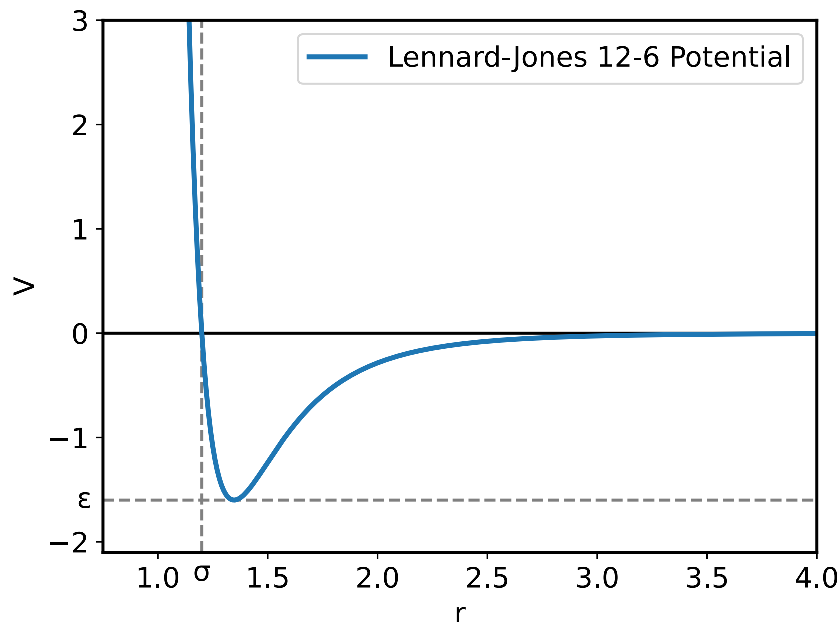


Figure 2.1.: Graph of the Lennard-Jones 12-6 Potential, $\sigma = 1.2$, $\epsilon = 1.6$

As seen in Figure 2.1 this potential models a strong repelling force at very close distances and an attracting force at a moderate distance that converges to zero very quickly with

growing distance. In the graph, σ represents the distance at which the potential and, therefore, the force between two particles is zero. ϵ, on the other hand, is the maximum force of two particles attracting each other. These two parameters must be adjusted for each molecule or element to obtain the potential for it.

The Lennard-Jones 12-6 Potential is a pair potential. To calculate the force of one particle in a domain with more than one other particle, one has to calculate the force every other particle exerts on this particle by deriving the potential with respect to the location. These forces can then be simply added up to one force vector.

2.1.2. Computational Challenges

Having to pairwise calculate forces is a property of almost all potentials used in physics and leads to the big computational problem of particle simulations, called the n-body problem. One of the first times this problem arose in the calculation of the movement of celestial objects and their gravitational force. Computationally this first and foremost means the number of calculations needed to obtain all forces and therefore also the calculations in a simulation needed for each time-steps grows with $\mathcal{O}(n^2)$, with n being the number of particles [17].

MD-simulations often include millions of individual particles and span over a large number of time steps. Therefore, it is only logical to tackle this problem with the power of High-Performance Computing (HPC). Though as for many computational problems, hardware will only get you so far because due to the computational complexity, the number of calculations needed to solve the problem is growing so rapidly. To conduct some useful simulations in an acceptable time, the number of calculations and the time per calculation needs to be reduced by optimizing the algorithms used.

2.1.3. Algorithmic approaches

To clarify how this computational problem can the tackled, some generally used optimizations will be discussed. The explicit implementation in AutoPas[1] will then be explained in Section 3.1.

A very widely used algorithmic optimization uses the principle of Newton's third law of motion. The law states that "the actions of two bodies upon each other are always equal and always opposite in direction." [12]. This means that forces are interactions between bodies, or in this case particles, that result in equal but opposite forces on these two particles. Computationally this allows to calculate the force between two particles only once and obtain the forces afflicted on both particles. Therefore only half of the force calculations are needed in comparison to a naive implementation.

Another way to drastically decrease calculations exploits the rapid convergence of the Lennard-Jones 12-6 Potential towards zero. This allows for the approximation of the potential being exactly zero for distances greater than a chosen cutoff radius r_c. Therefore, all forces between particles further apart than r_c are zero and don't have to be calculated. Choosing the value for r_c is critical: A too small r_c can negatively affect the accuracy of a simulation. That is because for particles further apart than r_c, interactions are assumed to be nonexistent, although they might just be weak. When r_c is chosen too high, the

[1] The MD-simulation framework used for this thesis

computational advantage can be very small to absent. For big scenarios with the size of the domain way bigger than r_c and r_c chosen at a suitable value with acceptable accuracy, the potential of this optimization is very high. A lot of force calculations can be skipped, and the overall computational payload can shrink drastically.

The concept of the cutoff radius r_c is the first step towards reducing the computational complexity in the ream of $\mathcal{O}(n)$. Combined with the right data structures in which the particles are stored, this can very well be achieved. Section 3.1 provides a deep insight how to this is handled in AutoPas.

2.2. Auto-tuning

Tuning or performance tuning in regards to computers is the effort to improve the overall performance of a system. This can be achieved in very different ways, one being to highly parallelize the problem and distribute the work on many processes, like in HPC. Other tuning approaches can concentrate on optimizing the code itself, using the cache more efficiently, or using load balancing. To avoid manual tuning by the user, auto-tuning uses various tuning-strategies, which can be used at three different times from building the application to executing it [3]:

1. At compile-time, the software can be optimized to the degree that it is independent of inputs but assures the best performance on the hardware architecture it is built on.
2. While the program is initialized and gets its input. It can be optimized to this given input, provided there are no changes during run-time.
3. During run-time, the program can regularly and repeatedly determine if the current configuration is still optimal.

For a lot of HPC applications, including MD-simulations, configuration optimization as described in number 3 is a big part of the tuning approach [5]. This means a very flexible software framework can be configured using a large set of parameters. One set of valid parameters then represents a configuration. Subsection 3.1.1 describes the configuration options of AutoPas and can be seen as an example. These parameters can influence the performance significantly, as they can also alter the algorithms and data structures used. It is important to note that these configurations are all qualitatively equivalent, so the precision of the end result will always be the same.

Optimization is a common problem in computer science, and a lot of optimization algorithms have been developed throughout the years. However, these solutions mostly optimize one numerical parameter. Configurations are often composed of numerical parameters reaching over a continuous interval and parameters describing a concrete selection from a discrete set of possible adjustments. This makes it very hard to use usual optimization algorithms.

Especially configuration optimization can be quite confusing for a user because of the large number of configurable parameters with very different impacts on the performance. In addition to that, some configurations might only work on certain architectures or provide a performance boost only for certain inputs. Particularly in particle simulations, the scenario to be simulated has a big impact on which algorithms will perform well. Therefore it would

be unrealistic to assume the user, who might not even be a computer scientist, would come up with the ideal configuration.

So it makes sense to let the computer do the tuning here. But such auto-tuning or normal tuning can create overhead because either some computational time will be used for optimizing or the user itself optimizes the program before run-time. Especially repeated tuning, while the program is running, has the potential to extend the run-time per iteration while tuning because inefficient or insufficient configurations will be tested in order to find the best configuration. But in most cases, the speed up achieved by the better configurations far outweigh the overhead caused through auto-tuning. How AutoPas is using auto-tuning will be described in the next chapter.

3. Technical Background

3.1. AutoPas

"AutoPas is an open-source C++ node-level performance library, which aims to provide a base for arbitrary N-body simulations." [7] To accomplish this, the user only has to implement a class for the particle interface and a functor for force calculation. Deep knowledge about optimizations or algorithms is not needed because AutoPas will select the best options via auto-tuning.

3.1.1. Options for performance optimization

To achieve a high level of flexibility, multiple options regarding algorithms, data structures and optimizations are implemented. Since it depends on the simulated scenario and used hardware which options are performing the best, tuning is needed. As mentioned in Section 2.2 MD-simulations often use a configuration optimization during run-time as an auto-tuning mechanism. Such a configuration in AutoPas is currently composed of values for the following specifications:

- **Containers**
  In order to exploit the advantages of the cutoff radius of the Lennard-Jones 12-6 Potential, the distances to all particles have to be calculated. This also comes with a huge computational cost, with the same $\mathcal{O}(n^2)$ complexity. Therefore the containers aim to store neighboring particles, particularly particles within the cutoff radius. Figure 3.1 demonstrates how three different containers try to achieve this goal.
  - *DirectSum* has the very simple approach of calculating all distances but only the forces within the cutoff radius.
  - The *LinkedCells* approach divides the simulation domain into cells with a cell width greater or equal to the cutoff radius. So only particles in bordering cells have to be considered for calculations. This can reduce the computational complexity to $\mathcal{O}(n)$ [7].
  - *Verlet-Lists* follow a similar approach with an additional list of neighbors, which might come closer to the particle. This neighbor-list isn't updated every iteration due to high computational costs. Accordingly, the yellow circle in Figure 3.1 has to be big enough, so particles don't pass through it in the inner circle without updating the list.

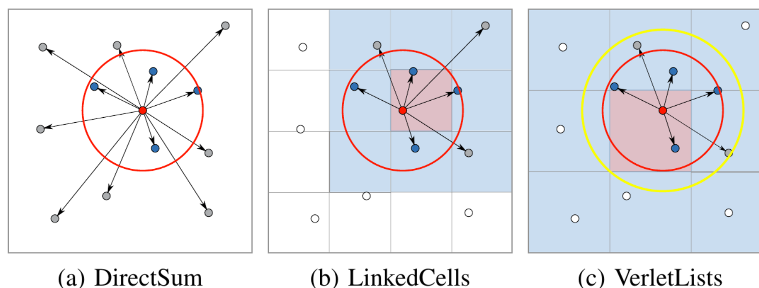
  More containers with similar functionalities can also be selected.
- **Cell Size Factor**
  This parameter can alter the size of the container-cells introduced above. This also is

the only occurrence of parameter-tuning instead of tuning a selection from a fixed set of options.

- **Traversals**
  The traversal dictates in which order forces of particles are calculated, and therefore memory is accessed. To utilize the full potential of shared memory and pluralization via OpenMP[1], one needs to find a way to access memory with as little overhead as possible and avoid collisions, where multiple threads access the same memory. For the traversals, AutoPas has by far the widest variety of options, but each traversal is only compatible with a small number of containers.

- **Data Layout**
  This option determines in which way particles and their information is stored in memory. They can be stored as Array-of-Structures (AoS) or as Structure-of-Arrays (SoA). AoS favors the access of information of one individual, whole particle. While SoA has an advantage because the same information of multiple particles has to be accessed in a more streamlined way. Figure 3.2 shows in which way AoS and SoA store particle information. A CUDA[2] layout could also be a third option if GPUs[3] are available.

- **Newton 3**
  This optimization was already explained in Subsection 2.1.3. The option here is to either use it or not, as not all user-implemented force-functors might be compatible with it.



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

Figure 3.1.: Illustration of different container types [7]. The red circle shows the cutoff-radius of the red particle in the center. Only particles inside the circle are used for force calculations. The yellow circle shows the boundary of the neighbor list of Verlet-Lists.

[1] API for shared memory programming, used to parallelize software
[2] Compute Unified Device Architecture is an API created by Nvidia to easily program GPUs
[3] graphics processing unit

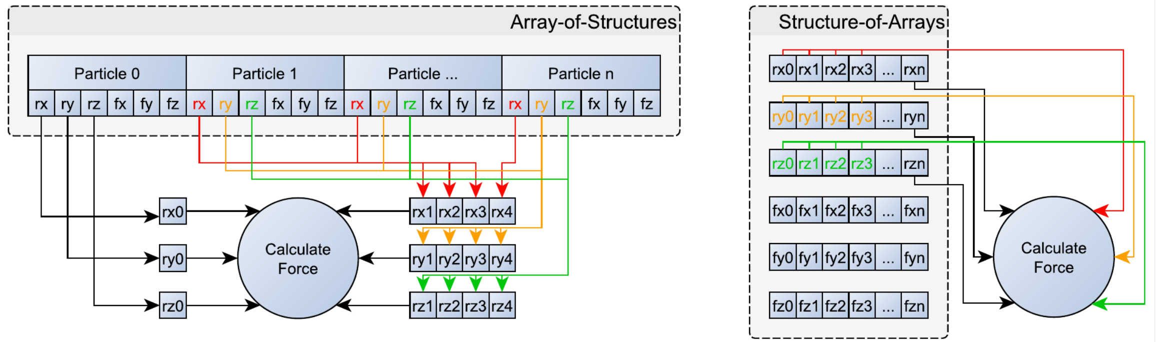


Figure 3.2.: Example how the data structures AoS and SoA access load data from memory [7]

3.1.2. Tuning-strategies

AutoPas also uses tuning at coMPIle-time, but for this thesis, tuning during run-time is the interesting part. The user starting a simulation can specify which values should be allowed for the options listed above. This set of configurations defines the search space from which the strategies are to select the best possible one. When a simulation starts, AutoPas will use a tuning-strategy to get a suitable configuration for the next x iterations. x is called *tuning-interval* in AutoPas and can also be set by the user. After those iterations, the tuning-strategy will again look for a fitting configuration and so on until the simulation is done. The iterations, while the tuning-strategy is searching for the best configuration, is called a tuning phase. In a best-case scenario, the selected configuration is optimal, but not all tuning-strategies guarantee to find the best configuration. Figure 3.3 shows roughly how auto-tuning is embedded in an AutoPas simulation.

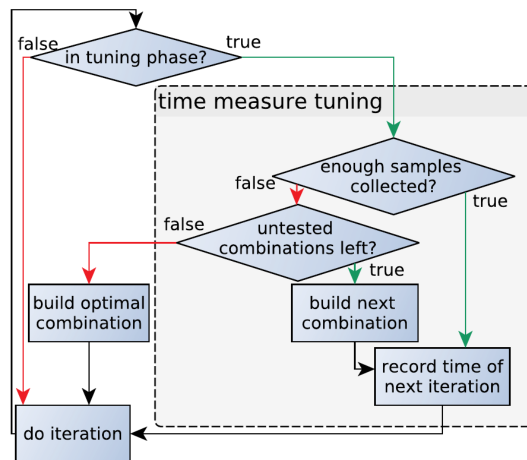


Figure 3.3.: Auto-tuning procedure in AutoPas [7]. When not in tuning phase, then compute normal simulation. Otherwise test current configuration until enough samples are collected. Then try to get next configuration to test. When search space is exhausted, then build best configuration for the simulation.

At the time of this thesis AutoPas features seven strategies, which are all based on measuring the execution times of several iterations to determine the best configuration:

- **Random-search**
  As the name implies, this strategy just randomly tests configurations until a certain number of configurations were tested. Out of those tested configurations, it selects the best one by execution time.
- **Full-search**
  The Full-search strategy will traverse the complete search space while minimizing expensive option switches, such as the containers. Because every single valid configuration will be tested, this is the only strategy that promises optimal results but also takes much more time than the others because also all inefficient configurations will be tested.
- **Predictive-tuning**
  At the start of a tuning-phase, this strategy will attempt to predict the execution times of all configurations via extrapolation and only tests the best ones. Among these configurations, the one with the lowest execution time will be the optimum. The predictions are based on performance from former tuning phases.
- **Bayesian-search &**
- **Bayesian-cluster-search**
  These strategies look for the minimum of an unknown function, which maps configurations to execution times. For this purpose, it is assumed that the performance of configurations is normally distributed. The search strategies then try to anticipate which configuration should be tested next to obtain the newest location information about the optimum of the function.
- **Active Harmony**
  Active Harmony (AH) is a universal tuning library designed to tune at run-time. AH uses itself several tuning-strategies and provides API-calls in the form of a server-client model. The strategy used in AutoPas, called Nelder-Mead, is a simplex-based method, which tries to evaluate the performance of each simplex-point and then replace it with a better one. A full explanation can be found here [11]
- **MPI-parallelized-strategy**
  This strategy was first develop by Thieme Wolf [16]. It is a wrapper strategy that is responsible for the coordination and communication with other MPI-ranks. The actual tuning is then done by one of the other tuning strategies. This also is the strategy, which was improved by this thesis.

All tuning-strategies use normal simulation iterations for the time measurements, so the simulations don't have to pause. It is assumed that the simulated scenarios do not change their behavior during one tuning phase too much so that they are comparable.

3.2. Message Passing Interface

The Message Passing Interface (MPI) is a standard to pass messages between processes, which typically run on different nodes of parallel computing architectures. This standard

does not provide an implementation, but the syntax and semantics to call message-passing functions [6]. For these MPI-calls to work, an MPI-implementation has to be installed on the system. The two most commonly used open-source implementations are MPICH and Open MPI. As a convention, processes are called ranks in MPI. Multiple ranks can then communicate via one communicator and get assigned a rank number unique in this communicator.

MPI not only allows to send data between processes but also provides functions to manipulate this data. For example, a function is implemented to reduce the data from all processes to the smallest value sent and broadcast this value back to all processes. MPI also allows methods to create new communicators or split the current communicator into multiple new ones. This will be very useful for the implementation explained in Section 5.3.

The interface provides both blocking and non-blocking versions of nearly all functions. In a blocking variant, the function call does not return until every other rank in the communicator used has also called the function. When a non-blocking function is used, the call returns immediately. It is the responsibility of the user to check if all other ranks have called the function before the data can be accessed safely. This has to be done by calling additional MPI functions.

For HPC-software, which is designed to run on multiple CPUs simultaneously, MPI adds very useful functionalities. Communication from one process on one node to another one on another node without shared memory can be very complicated. But MPI works on nearly every architecture and always chooses the fastest way of transmitting information. For this reason, the user does not necessarily need to know the architecture on which the software runs, which greatly facilitates the development of portable software.

Part II.

Implementation

4. Comparing scenarios

To achieve the main goal of this thesis providing parallelized auto-tuning for inhomogeneous scenarios, one needs to compare scenarios in any AutoPas-simulation. Comparing two numbers is quite simple because one of them will be clearly bigger than the other, or they are equal. For particle simulation scenarios, there is no simple method to compare scenarios because to completely describe a scenario one would have to describe all particles in it. Such large sets of numbers can't simply be compared in a meaningful way without consuming a huge amount of time. Hence it was necessary to come up with a way to compare scenarios.

4.1. Finding parameters to determine similarity

When faced with a complex problem in mathematics or computer science, a promising approach is to reduce the problem into (sub-)problems one knows how to solve. In computational complexity theory, a problem is reducible when the substitute problem is at least as difficult as the real problem. This thesis uses a much less sophisticated approach, as the reduction is not a complete substitution of the problem but a simplified approach.

The procedure here is to find one or a few values, which describe scenarios sufficiently. Those few values can then easily be compared with the values of other scenarios. In order to find suitable quantities, multiple scenarios have to be created and then simulated using the AutoPas demonstrator code *md-flexible*. This is a simple yet effective implementation of an MD-simulation, which comes exemplary with the AutoPas library. Some example scenarios are also provided, one of which was used as a base model for creating all other scenarios. Figure 4.1 shows the structure of the user base scenario, named exploding liquid. By altering parameters, like domain size, particle size, particle mass, or initial particle distance new, but very similar scenarios were created. Table 6.1 lists and describes all used scenarios in depth.

Many parameters were used to find values, which can describe the scenario in a meaningful way. The ones which were chosen will be discussed after this section in depth. All other parameters are briefly explained here:

- **Number of particles**
  The number of particles can heavily affect the execution time of a simulation, as more particles mean more calculations. However, the number of particles just weakly dictates which configurations are performing best. In Section 4.4 it can be seen that scenarios with different numbers of particles but similar structure use the same configurations.
- **Velocity of particles**
  The average or maximum velocity of the particles might seem like a promising factor. But the velocity is determined by the force on the particles, which itself depends on the distribution of particles. The values chosen directly describe the arrangement of all particles and are therefore suited even better.

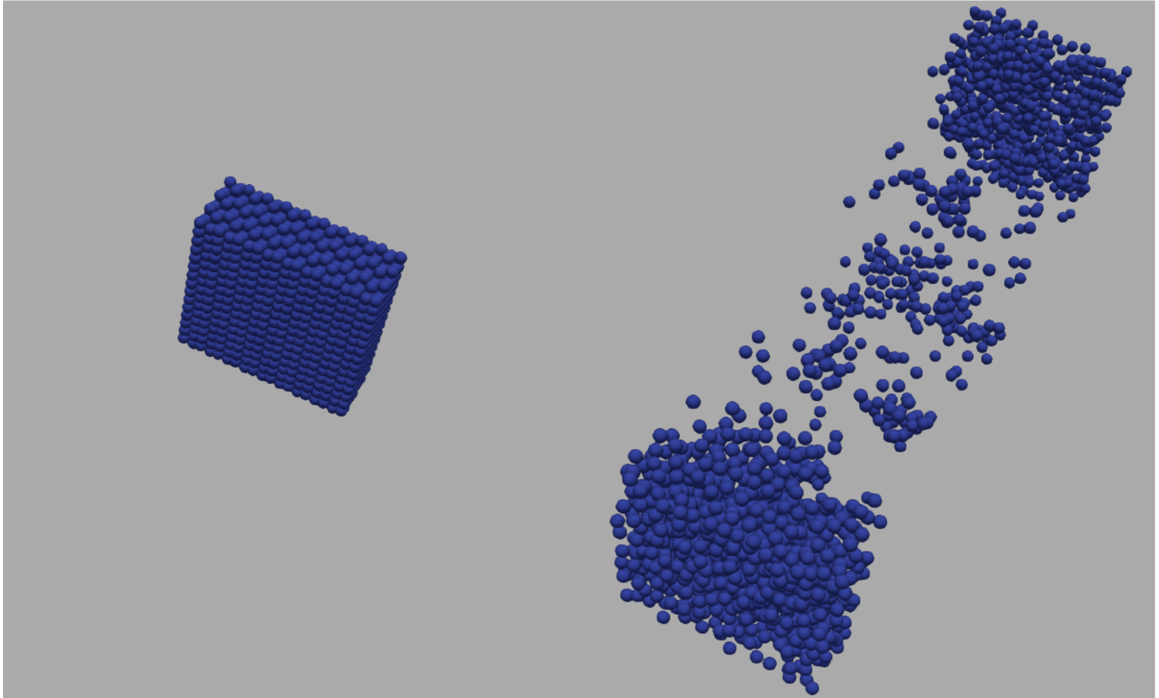


Figure 4.1.: Scenario exploding-liquid at the beginning (iteration 0, left) and the end (iteration 12000, right) of simulation. Particles spread over simulation domain.

- **Mass of particles**
  The mass is only a scalar factor on all velocities. It almost behaves like a slowing down or speeding up of the simulation and does not describe the scenario sufficiently. Since the force calculation is independent of the mass of the particles, it has no influence on the computationally complex part.
- **Average force of particles**
  The force of particles depends on their arrangement. Just like the velocity this is already covered by the chosen values.

4.2. Homogeneity

The homogeneity of a scenario describes how evenly the particles are distributed. A very homogeneous scenario would look like a regular grid of particles, which are all equidistant from each other. An example for an inhomogeneous case would show all particles of the domain close together in one corner. As previously teased, this value describes the arrangement of all particles. A very inhomogeneous scenario will overall lead to strong forces and high velocities and vice versa. To understand what a single homogeneity value means, one has to understand how it is calculated.

In short the homogeneity is the standard deviation of the particle densities in the domain. Computationally this means the whole simulation-domain is split up in n equally sized cubes,

with

$$n = \left\lceil \frac{number\ of\ particles}{10} \right\rceil \tag{4.1}$$

As not every domain can be perfectly split into equally sized cubes, cuboids at the edge are added to the cubes to form an array of cells. Then for each cell, the density is calculated as shown in Equation 4.2. The mean density is determined the same way, with the whole domain being one cell. As for the standard deviation, one only has to get the density variance using Equation 4.4 and take the square root of the value as shown in Equation 4.5.

$$cell\ density = \frac{number\ of\ particles\ in\ cell}{volume\ of\ cell} \tag{4.2}$$

$$mean\ density = \frac{number\ of\ all\ particles}{volume\ of\ simulation\ domain} \tag{4.3}$$

$$density\ variance = \sum_{all\ cells} \frac{(cell\ density - mean\ density)^2}{number\ of\ cells} \tag{4.4}$$

$$standard\ deviation = \sqrt{variance} \tag{4.5}$$

The homogeneity values usually range from 0 to 1 [1], where 0 represents the greatest possible homogeneity and the larger the number, the more inhomogeneous is the scenario. Now it should be clear that this value is a very powerful tool when it comes to overall particle arrangement and how a scenario will behave in the near future. In order to see how homogeneity behaves over time, we will look at the homogeneity values of different scenarios over the course of a simulation.

Figure 4.2 shows the homogeneity of eleven scenarios, with more or less similar starting parameters. In all scenarios, the particles are densely packed in a small region of the whole domain; hence the homogeneity value is big at the beginning. The plot shows the homogeneity in respect to iterations, but since every iteration represents a short time step, this is equivalent to time. The graphic clearly shows that the homogeneity of scenarios is constantly changing. This is important since the scenarios themselves are also continually evolving. A steady variable, therefore, wouldn't be of much use.

The particles in the simulation follow Newton's Laws of Motion, which means we can also predict the behavior using laws of physics. One law that is particularly useful here is the second law of thermodynamics. It states that an isolated system approaches thermodynamic equilibrium and thereby increases its entropy [14]. For such an isolated system, the total energy always has to stay the same; also, temperature changes from outside are not possible. For ideal gases or other groups of small particles, this means that inhomogeneities in the system will fade away. This then leads to a more homogeneous scenario, which can be seen in the value of the homogeneity. Nearly all scenarios lead towards a more homogeneous state or at least stay roughly stable in their homogeneity. In Section 4.4 we will see that scenarios with similar homogeneities also behave similarly and therefore have similar best performing configurations.

[1] but can very well get higher

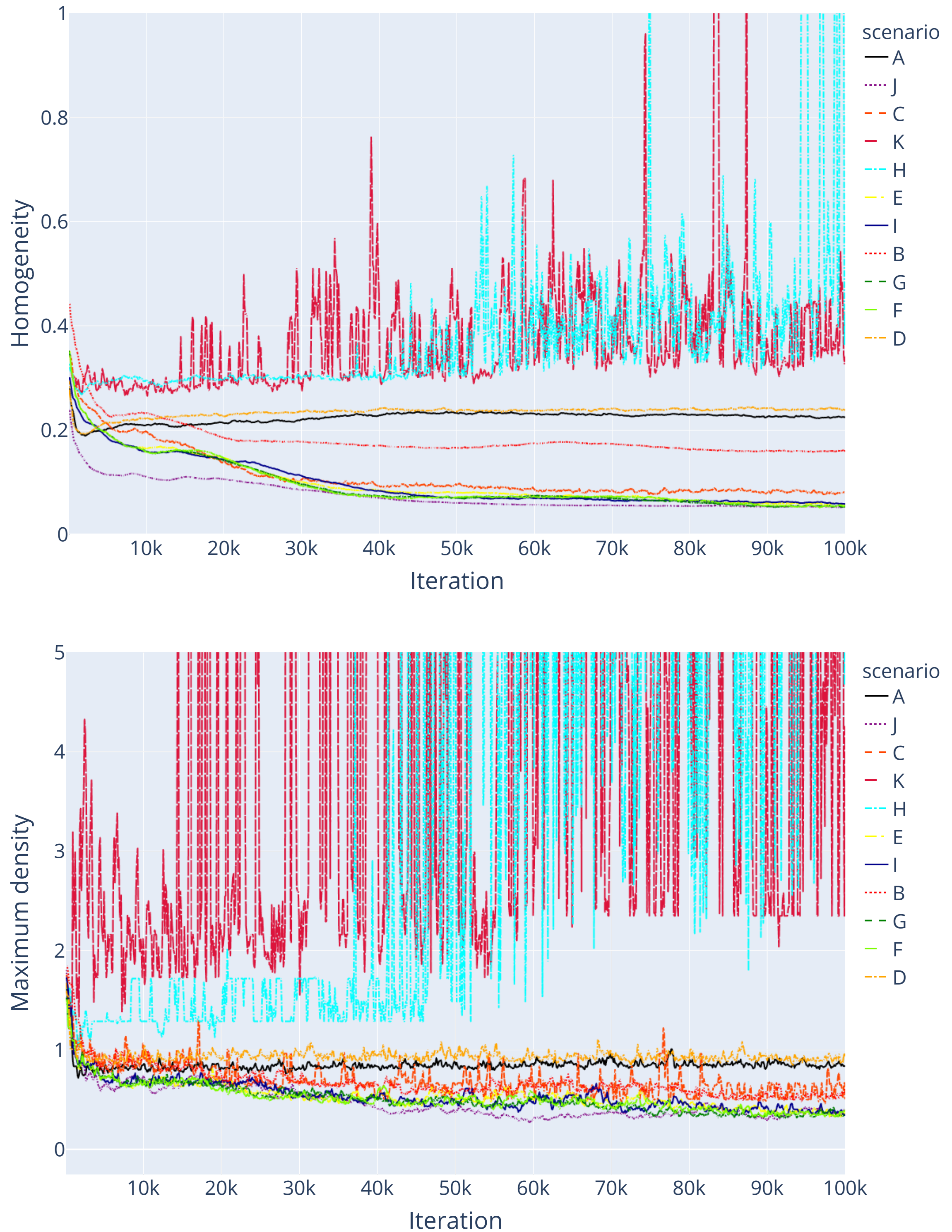


Figure 4.2.: Homogeneity (above) and maximum density (below) of different scenarios with respect to the iterations of the simulation. Overall both value shrink and gets more stable, but some scenarios have fluctuating values.

4.3. Maximum Density

As described in the previous section, to calculate the homogeneity of a scenario, one has to calculate all cell densities (Equation 4.2). During this process, the maximum density is stored separately, as it can provide additional information. Of course, those two values are strongly connected, but in some cases, the actual value of the maximum density can be beneficial.

Just like the homogeneity also the maximum density has a tendency to approach a low, stable value due to the second law of thermodynamics. A low maximum density indicates a homogeneous environment without strong inhomogeneities. **??** shows the maximum density of the same eleven scenarios, as before, over the same simulation-iterations. As stated before in the beginning, the value is quite high due to a packed particle formation. Then the particles spread out, and the density shirks. Only the two scenarios that had a relatively high homogeneity value show an increase in density. This again shows how strong those two values are connected.

To get an intuition for density values: They normally are between 0 and 2, nonetheless, very few times values way above 10^{30} are calculated. A conjecture on why some scenarios develop increasing densities is in the way of calculating them. As explained in Section 4.2, extra cells at the edges of the domain have to be added when the domain isn't easily split up. This could lead to very small cell volumes and, therefore, extraordinary high densities. However, testing this hypothesis was not yet successful.

4.4. Correlation of parameters and properties of scenarios

Correlation coefficients are widely used in science to test if some values or parameters are in any statistical relationship. Often this coefficient is a value between -1 and 1. A value of -1 stands for a perfect inverse correlation, 0 for no correlation, and 1 for a perfect correlation. For this thesis, the hypothesis has to be tested whether homogeneity and density values correlate with AutoPas' choice of algorithm configurations for a scenario. If this correlation is significant, then scenarios with similar homogeneity and density values should choose similar configurations as their optimum. This can then be exploited to parallelize the tuning workload.

A common form of calculating a correlation is the Pearson product-moment correlation coefficient [13]. This coefficient evaluates a linear correlation between two data sets. It is basically a measurement of the covariance normalized on the interval from -1 to 1. Sadly this does not help in our case, as we don't know if the connection is linear, and we are searching for a rank correlation. A rank correlation coefficient searches for similar ranks between two variables. Meaning, not the absolute values have to correlate, but their relative position[2]. This makes it very useful for enumerated variables and especially for the configuration ranking of AutoPas. Spearman's rank correlation coefficient is a measure for the correlation of rankings, using any monotonic function instead of a linear one [10]. Mathematically the Spearman coefficient mainly also uses the Pearson coefficient, but it can handle rankings and non-numerical data much better. Therefore this coefficient was used to determine the

[2] like position labels: 1st, 2nd, 3rd

relation of the chosen parameters and the container- and traversal-ranking of multiple tuning phases.

The container-ranking of a tuning phase is obtained by listing all tested configurations sorted in descending order by execution time. This way, the best configurations are at the top. Now the configuration description is reduced to the container description only. For each container, the first occurrence in this ranking determines their place in the container-ranking. Table 4.1 shows exemplarily how such a ranking is built. The ranking is then cut off after the third place, as it is more important which container is the fastest or second/third fastest than which is the slowest or second slowest. For traversals, the procedure is analogous. To disregard the actual order of the ranking, sorted versions are also used. This sorted version then only represents a set of three containers/traversals, which were the three fastest. They are called sorted because the ranking was sorted alphabetically to get rid of the execution time sorting.

1.	Linked-Cells				
2.	Linked-Cells				
3.	Linked-Cells	1.	Linked-Cells	1.	Linked-Cells
4.	Verlet-List-Cells	2.	Verlet-List-Cells	2.	Verlet-Cluster-List
5.	Linked-Cells	3.	Verlet-Cluster-List	3.	Verlet-List-Cells
6.	Verlet-Cluster-List				
7.	Verlet-List-Cells				

Table 4.1.: Example for container-ranking — left: sorted configuration list — middle: corresponding container-ranking — right: alphabetically sorted ranking

Figure 4.3 shows the heat-map for the calculated correlation coefficients. The fields are colored according to the value of the coefficient correlating the variable at the row and column head. Bright yellow represents a perfect correlation (1), and dark blue a perfect inverse correlation (-1). It is sufficient to only look at the lower-left triangle of the heat-map since it is mirrored at the yellow diagonal. The colors already give a good impression, but the most important correlation coefficients are also listed in Table 4.2. It should be quite clear that homogeneity and maximum density both strongly correlate with the container-ranking and moderately correlate with the sorted traversal-ranking. A coefficient value around 0.7 is a very good indicator for a statistical relation [15]. It should be noted that negative correlation values are just as beneficial for this connection as positive, as negative values also promise clustering of certain scenarios at certain configurations.

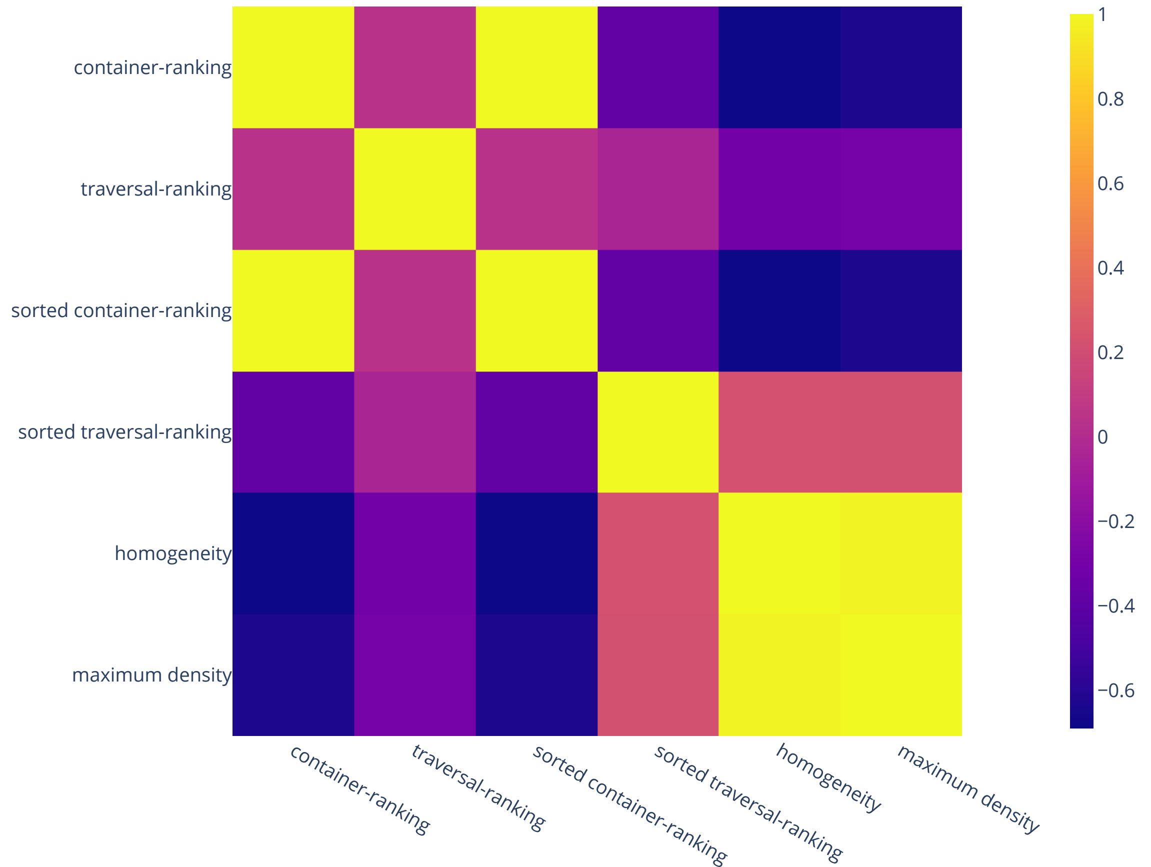


Figure 4.3.: Heat-map of correlation matrix of homogeneity & max-density with container-ranking and traversal-ranking. Values can lie between -1 (dark blue) and 1 (yellow). The higher the absolute value, the stronger the statistical relationship.

	homogeneity	max-density
container-ranking	-0.69	-0.64
traversal-ranking	-0.31	-0.30
sorted container-ranking	-0.69	-0.64
sorted traversal-ranking	0.22	0.21

Table 4.2.: Most important correlation values from Figure 4.3. Values lie between -1 and 1. The higher the absolute value, the stronger the statistical relationship.

4.5. Similarity-Metric

In order to simply compare homogeneity and max-density of multiple scenarios, both values are combined to one similarity-metric. This also allows to easily extend and modify this metric. Currently, the metric is calculated like this:

$$similarity\text{-}metric = homogeneity + weight\text{-}for\text{-}max\text{-}density * max\text{-}density \quad (4.6)$$

weight-for-max-density is a configurable weight to define how much impact the max-density values have on the similarity. The weight is needed because the variables are not in the same order of magnitude. Also, the weight can be used as an off-switch for the density value, so only the homogeneity will be used. As of now, weight-for-max-density has a default value of 0.0.

One could ask why the density is needed in this calculation; why not use just the homogeneity? Figure 4.4 and Figure 4.5 demonstrate the reason why max-density was included. They show the container-ranking of multiple tuning phases with respect to values of homogeneity and max-density. The data clearly shows that homogeneity and the ranking of containers are statistically connected. For the max-density the picture is not so clear, but if the value is very high, the containers will always be ranked in the same order. This is the main reason why the similarity-metric was introduced, and the density has an effect on it. It is important to note again that the plots show rankings of multiple tuning phases, which means this phenomenon is not unique but can be reproduced.

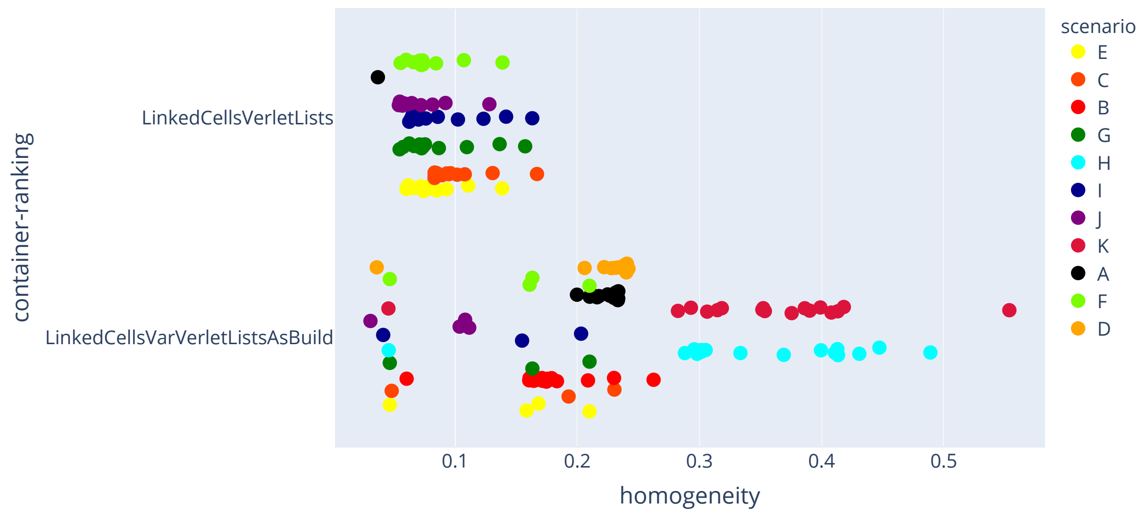


Figure 4.4.: Container-rankings of multiple tuning-phases in respect to homogeneity of scenario. Similar homogeneity values lead to similar rankings.

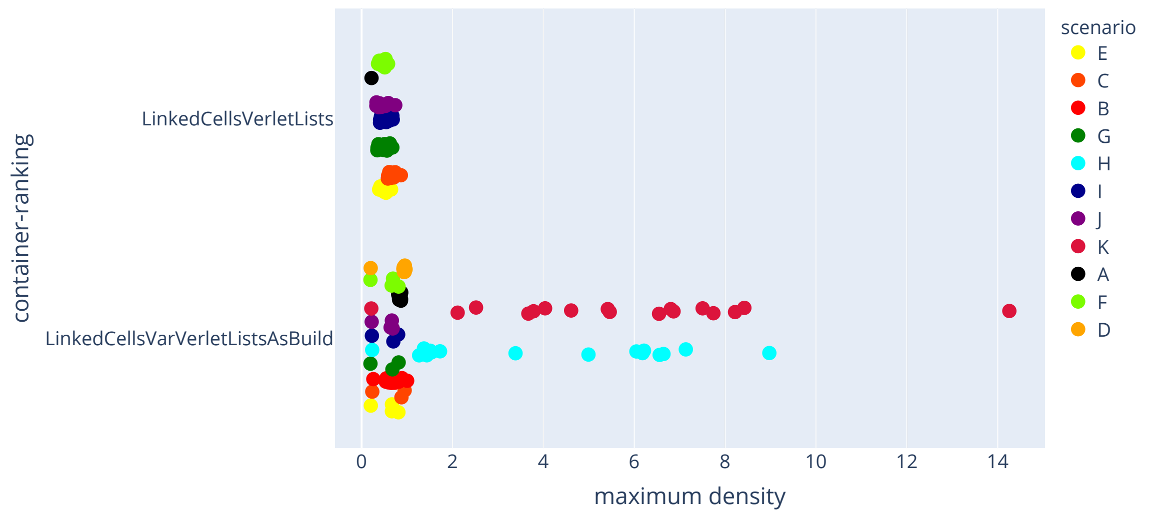


Figure 4.5.: Container-rankings of multiple tuning-phases in respect to maximum density of scenario. Extremely high values lead to the same container and traversal ranking

5. Realization in AutoPas

5.1. Auto-tuning in AutoPas

The old implementation of the MPI-parallelized-strategy was briefly introduced in Subsection 3.1.2, but will now be discussed in more depth. In order to do this, one needs to understand that every tuning-strategy is managed by the so-called auto-tuner. Before each tuning phase, the auto-tuner resets the strategy and then calls it to get new configurations to test until the strategy is done testing. As explained earlier, the MPI-strategy is only a wrapper using one of the other strategies. The MPI implementation is only responsible for equally distributing the search space to all MPI_ranks. This way, parallelization during tuning is maximal, as every rank only has to traverse a part of the global search space. After tuning, every rank gets to use the same configuration until the next tuning phase begins. However, the tuning result is only reasonable if all ranks are simulating similar scenarios. Otherwise, some ranks will use inefficient configurations. Figure 5.1 shows how the search space is distributed. Further information can be found in the thesis of Wolf Thieme [16].

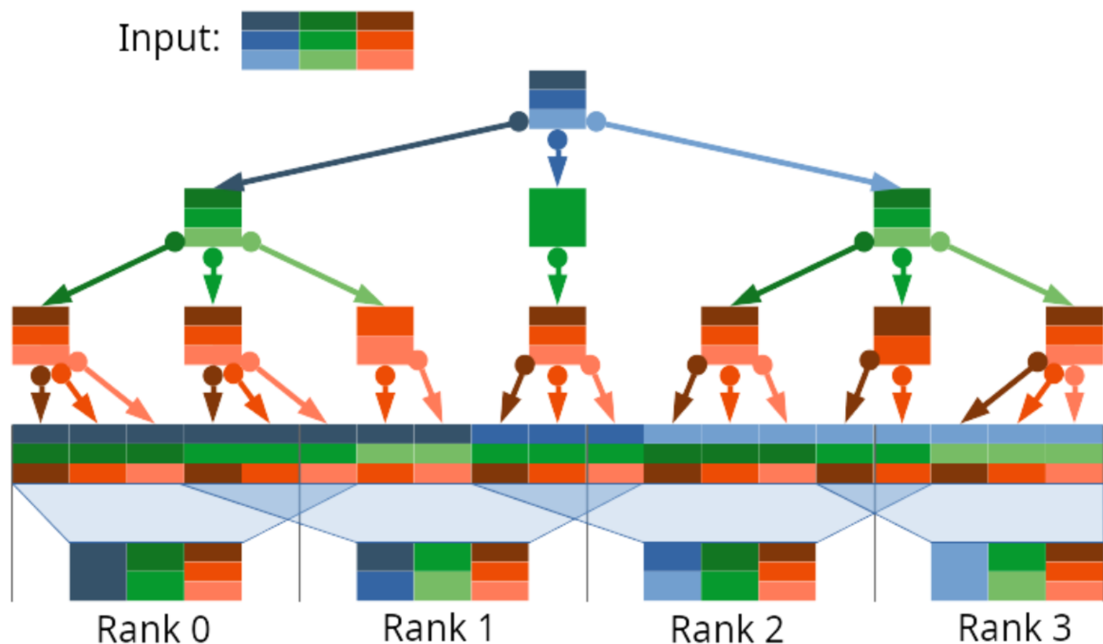


Figure 5.1.: Illustration of the distribution of configurations across four ranks. A combination of one blue, one green, and one red rectangle is a configuration. "Input:" denotes the global search space. The blue bridges show which parts correspond to a rank. [16]

5.2. Smoothing

Figure 4.2 and **??** already show a problem of the homogeneity and max-density variables. Especially the density can fluctuate drastically from one iteration to the other. Hence it is necessary to calculate these values over multiple iterations and combine them to a smooth value. As sometimes the density values fluctuate to more than 10^{30} calculating an average is not sufficient. When collecting ten samples and one of them is way too high, the average would still be in the realm of 10^{28}. A better approach is to select the median of the collected samples. This way, a correct value is selected when more than half of the samples are normal. As those extremely high values occur very rarely, this is sufficient enough.

Calculating homogeneity and max-density multiple times creates computational overhead. However, **Add figure here** shows this overhead is negligible. For a simulation lasting over 14 minutes, the maximum time used to calculate homogeneities is 0.22 seconds. This boils down to 0.026% of the total simulation time and can therefore be ignored.

5.3. Communication and distribution into buckets

The heart of the implementation is the distribution of ranks into buckets. The functionality of the MPI-parallelized-strategy stays the same for the most part. But to ensure that scenarios use efficient configurations, ranks with similar scenarios are collected into a bucket. This way the global search space will be traversed by each bucket, which leads to less parallelization during tuning. However, the goal is to distribute the ranks in a manner, which provides a perfect mixture of parallelization and efficient tuning. This should lead to an overall short execution time and, therefore, more efficient implementation.

As described in Section 5.1 the auto-tuner controls the tuning-strategy and resets it before every tuning phase. This is where the distribution is done. As part of the reset method of the MPI-parallelized-strategy the smoothed homogeneity and max-density values are calculated to a similarity-metric value. This value is then sent to all other ranks running the simulation. Simultaneously the values of all other ranks are collected. This is both covered by the blocking function call *MPIAllGather*.

After the call returns, every rank calculates which rank is going into which bucket. This sounds like unnecessary, redundant calculations, but since the simulation of all ranks is at a pause until the distribution is done, there is no simple, faster way. Also, the distribution algorithm is very lightweight and therefore does not cause a lot of overhead. The computational complete of the algorithm is $\mathcal{O}(n)$, with n being the number of ranks.

The algorithm to distribute ranks into buckets has to detect clusters within an array of values. All values, which are "close" to each other, represent a bucket. To find these clusters, all similarity-metric values are sorted, and then the difference from one to the next is calculated and stored in a separate array. These differences are then transformed into relative differences by dividing through the values. Then this array of relative differences is iterated. As long as the difference is smaller or equal to *max-difference-for-bucket*, the rank is added to the current bucket. When the difference is greater, a new bucket will be initiated. The parameter *max-difference-for-bucket* can also be configured to alter the granularity of the distribution. The default value is set at 0.3. Figure 5.2 shows three bucket-distributions of the algorithm using different values for weight-for-max-density and

max-difference-for-bucket. The color in the plot shows the selected bucket. This means for an ideal distribution, all points, which are "near" to each other regarding the x-axis have to have the same color.

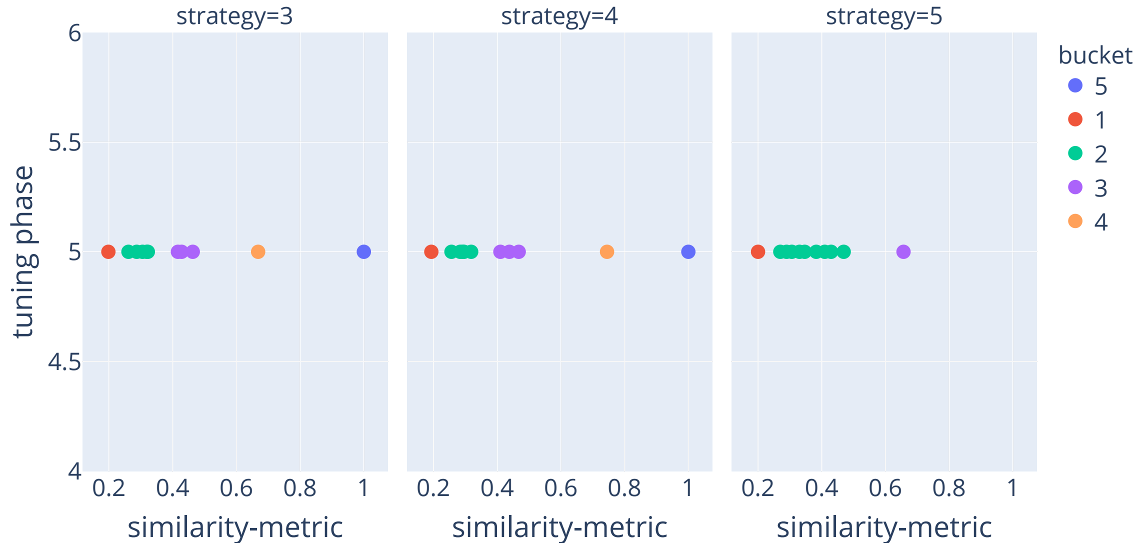


Figure 5.2.: Distribution of ranks into buckets based on their similarity-metric value. Only one tuning phase is presented. Ranks with similar values, which are "close", should have the same color.

After each rank has a bucket assigned, each bucket has to initiate its own MPI communicator. Luckily MPI implements the function *MPI_Comm_split*. This method takes in the old communicator, a color value, a key value, and a pointer for the new communicator. The color value determines in which new communicator the rank will end up in. So for this case, the color is the bucket number. The key value has an effect on the rank order in the new communicator and is not needed for this implementation. Table 5.1 shows a simple example of how this split call can be used.

After every rank has a new bucket-communicator, the tuning will work just like the old implementation explained in Section 5.1. Each bucket will distribute its own global search space on its ranks. After tuning, every rank in a bucket will be using the same configuration for the next tuning-interval. The communicator of all ranks will never be deleted. At the start of each tuning phase, the ranks will be distributed again.

MPI_rank	0	1	2	3	4
MPI_COMM_WORLD	x	x	x	x	x
bucket 0	x		x		
bucket 1		x		x	
bucket 2					x

Table 5.1.: Representation of MPI_Comm_split. MPI_COMM_WORLD is the global communicator. The buckets show the distribution after the split function is called.

Part III.

Results

6. Overview

This part will cover the results of the new implementation and highlight accomplished achievements. The results will be compared to the results of the old MPI-strategy and not using MPI for tuning at all. The outcome of the various simulations will first be presented from a high-level view. Then we will dive deeper and focus more on details to find out what the results are based on. Overall this thesis tries to implement a more efficient way to simulate particles. From the HPC-perspective, this boils down to good resource management. The two big resources in HPC are hardware and time. For the case of this thesis, the hardware was set by a fixed number of ranks, each running a specific scenario. Therefore a more efficient simulation means a lower total execution time.

6.1. Expectation

To get a good understanding of the results and to gain some context is useful to first consider what is theoretically possible. A better tuning-strategy will not magically cut the total execution time in half. One thing to examine is the maximum possible level of parallelization. This is achieved when all scenarios are in the same bucket. On the other hand, the best possible tuning will only take place when every scenario tunes on its own. One has to find the sweet spot between parallelization and qualitative tuning results, which yields the lowest total execution time.

When speaking about execution times, it is useful to split the total time into execution time of all iterations during a tuning phase (tuning iterations) and all iterations during the rest of the simulation (non-tuning iterations). To minimize the time spent for tuning iterations, all scenarios have to be placed in one bucket, as this reduces the number of tuning iterations to a minimum. But this also produces the most slow-down for the non-tuning iterations, as many scenarios will be simulated using an inefficient configuration. The fastest non-tuning iterations will occur with no MPI-tuning because each scenario will use the optimal configuration, which was found through its own tuning. But due to the lack of parallelization, the tuning phase will take longer than with any MPI-tuning strategy. The execution time of the thesis implementation will be between those two extremes for both tuning and non-tuning iterations. The goal is to achieve a total time, which is lower than the total time of the other two strategies.

6.2. Method and Background

Most measurements taken here are time measurements, as time is a valuable resource in this case and needs to be saved as much as possible. AutoPas already provides a wide variety of logging mechanisms, one of which logs times for every iteration. After a simulation, a .csv

file with the information to every iteration is provided. Among this information is also a Boolean value whether the iteration is part of a tuning phase or not.

All simulations where conducted on nodes of the CoolMUC-2. This cluster is composed of 812 nodes, each containing 28 cores with each two Threads. Each node has 64 GB of memory available. For the simulations, each scenario was simulated by one MPI-rank, and each MPI-rank was running on its own node. Furthermore, each rank had access to 28 OpenMP threads.

To test the effect of the parallelized tuning strategy, a normal simulation is not sufficient. Ideally, the simulation would divide a big scenario is small scenarios, which are each simulated on one MPI-rank. However, *md-flexible* is not yet capable of doing this. Therefore multiple simulations, each running on an MPI-rank with different scenarios as input, have to run simultaneously. When using MPI-tuning, this makes a lot of sense since the ranks will communicate regularly. For a tuning strategy not using MPI, this has no effect, as the ranks will just simulate their scenario without even knowing that other simulations are running at the same time. Since not every scenario has the same workload and therefore not the same execution time, this can lead to a measurement problem. When taking times of simulations not using MPI, the times of the scenarios will be different. But when taking times of the MPI strategies, all ranks will have roughly the same execution time, as they have to sync up every tuning phase. To solve this problem, the simulation not using MPI-tuning was modified to get a more comparable measurement. After each iteration, the simulation needed to wait at a barrier for the other ranks before moving on. This was done using the *MPI_Barrier* function. This way, synchronization of all ranks was modeled without adding overhead by actually passing data. Therefore the total execution time of all ranks was the same.

As stated in Subsection 3.1.2 the MPI-tuning-strategy is only a wrapper for an actual strategy. The strategy used in all simulations was the Full-search strategy. This promises optimal results when not using MPI and the most potential for parallelization for the MPI-strategy. In order to measure the quality of a tuning result, these results were compared to the ideal results of no MPI tuning. Equal or similar results speak for a high quality.

The new version of the MPI-tuning-strategy implements two new configurable variables. *weight-for-max-density* was introduced in Section 4.5 and *max-difference-for-bucket* in Section 5.3. In order to find a good default value for both, a parameter study with over 30 simulations was conducted. However, the plots shown in Chapter 7 will only include the two best ones and a bad example. Furthermore, the old MPI-tuning-strategy and the normal Full-search strategy with the *MPI_Barrier* will be included. The simulations for finding the default values all used the same set of scenarios, so these values might not yet be ideal for all scenarios and are therefore still configurable by the user.

6.3. Scenarios ans Strategies

Throughout this thesis, multiple scenarios were used in figures and plots. Table 6.1 offers a description for each scenario used.

The eleven scenarios A-K are all based on the scenario “light and heavy particles blending over domain”. A visualization of this scenario can be seen in Figure 6.1. This scenario was chosen because small changes in the input drastically changed the behavior of the simulation.

This promised to produce scenarios, which are somewhat similar but, after some iterations, very different. This makes it harder to predict the behavior and should therefore lead to a more robust implementation. As an example, the yaml-file describing scenario A is attached in Part V.

Scenarios L-Q are some random scenarios, including the example scenario exploding-liquid, pictured in Figure 4.1 and also "light and heavy particles blending over domain". These scenarios where first used to find comparable parameters and later to test the implementation and its default values for *weight-for-max-density* and *max-difference-for-bucket*.

A	same proportions 20% smaller
B	smaller domain overall
C	smaller domain in y direction
D	slightly less particles at normal distance
E	two particle groups further apart
F	light and heavy particles blending over domain
G	two particle groups closer together
H	slightly more particles at normal distance
I	larger domain in y direction
J	larger domain overall & more spread out
K	same proportions 20% bigger
L	slightly slower exploding-liquid, because of larger domain
M	few particles staying together and shooting two particles out
N	light and heavy particles blending over domain
O	normal exploding-liquid
P	particles spread over domain from the begin
Q	many heavy particles close together and few light particles spread out

Table 6.1.: Description of all used scenarios. Scenarios A-K are all based on the scenario F. Scenarios L-Q are some random scenarios, including the example scenario exploding-liquid and scenario F.

To produce cleaner-looking plots, the used tuning strategies also have abbreviations. Table 6.2 provides an explanation for each strategy used.

1	no MPI tuning, but barrier
2	MPI tuning for homogeneous scenarios
3	new MPI tuning , *max-difference-for-bucket* = 0.3, *weight-for-max-density* = 0.0
4	new MPI tuning , *max-difference-for-bucket* = 0.4, *weight-for-max-density* = 0.4
5	new MPI tuning , *max-difference-for-bucket* = 0.4, *weight-for-max-density* = 0.3

Table 6.2.: Description of all used tuning strategies. Strategy 1 is not using MPI for parallelization. Strategy 2 is the old implementation. Strategies 3-5 represent the new implementation.

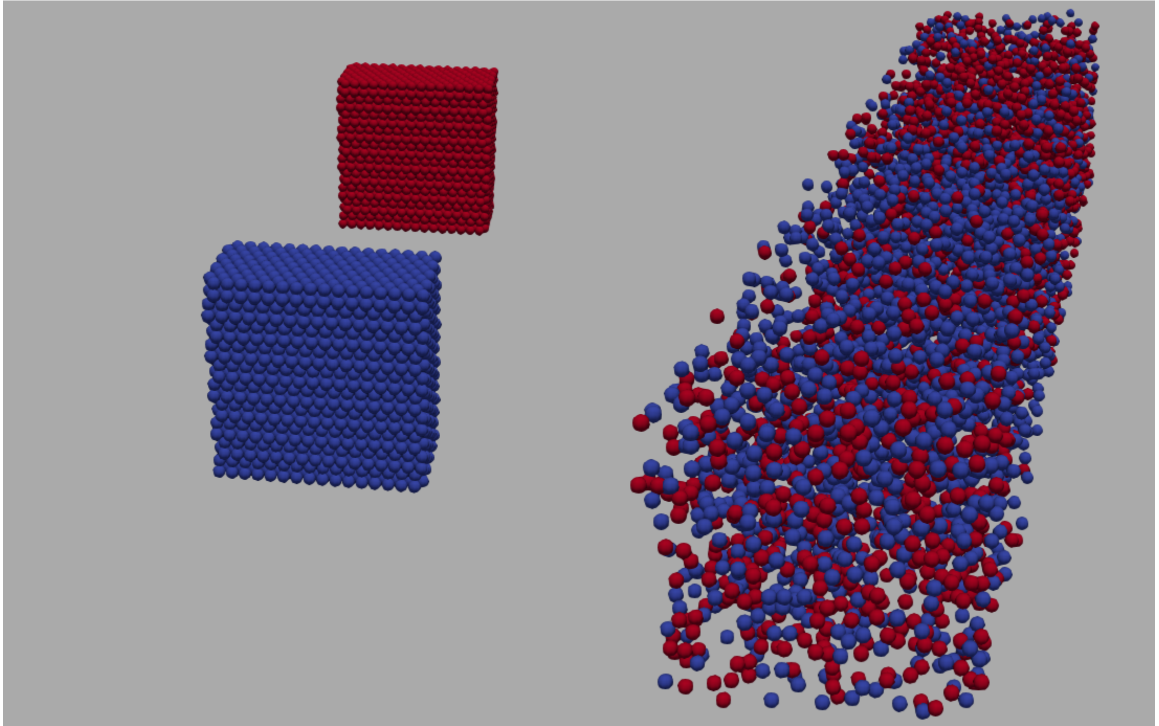


Figure 6.1.: Scenario F: "light and heavy particles blending over domain" at the beginning (iteration 0, left) and the end (iteration 100000, right) of simulation. The two particle groups mix together and spread over domain.

7. Measurements

7.1. Total execution time

Figure 7.2 shows the total execution time of the slowest scenario simulated by the five different tuning-strategies: Full-search with *MPI_Barrier* as the base-strategy, the old MPI-tuning and three times the new MPI-tuning with different values for *weight-for-max-density* and *max-difference-for-bucket*. As stated earlier, all simulations have some sort of regular sync mechanism among the scenarios. Nevertheless, the execution time of the lowest scenario is used because the simulation is only considered finished when the last rank completed its simulation. The gray color marks the base-strategy, which the others are compared to. Green represents a faster simulation than the base-strategy, and red a slower one. Figure 7.1 clearly shows that even the slowest MPI-strategy is faster then the base-strategy. However, to improve the MPI-tuning-strategy, the new implementation has to be faster than the old one. This can only be accomplished with suitable values for *weight-for-max-density* and *max-difference-for-bucket*.

Figure 7.2 is better suited to see the difference between these strategies. The plot is derived from the same data and shows the relative difference of the execution time in comparison to the base-strategy, which is the normal Full-search strategy. Therefore, the bar for the base-strategy is not present. The old MPI-tuning is considerably faster, but the new MPI-tuning with sufficient values is even faster. In numbers: the old implementation is 19.2% faster and the new one 22.9%. This means the main goal, a more efficient parallel tuning, was achieved.

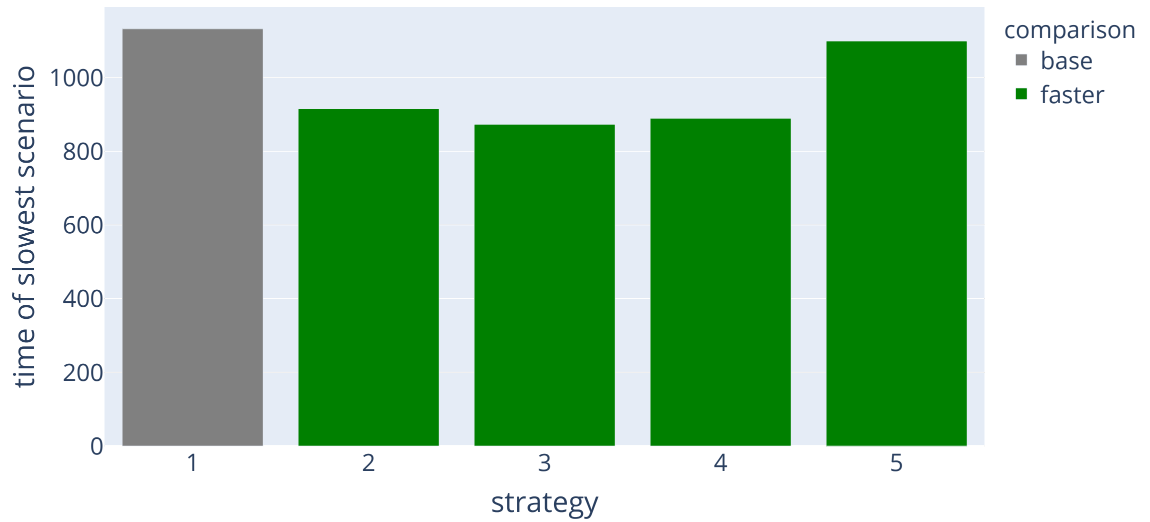


Figure 7.1.: Total execution time of slowest scenario of several tuning-strategies. Full-search without MPI is used as base strategy. Green bars represent faster strategies. All MPI-strategies are faster than the base-strategy

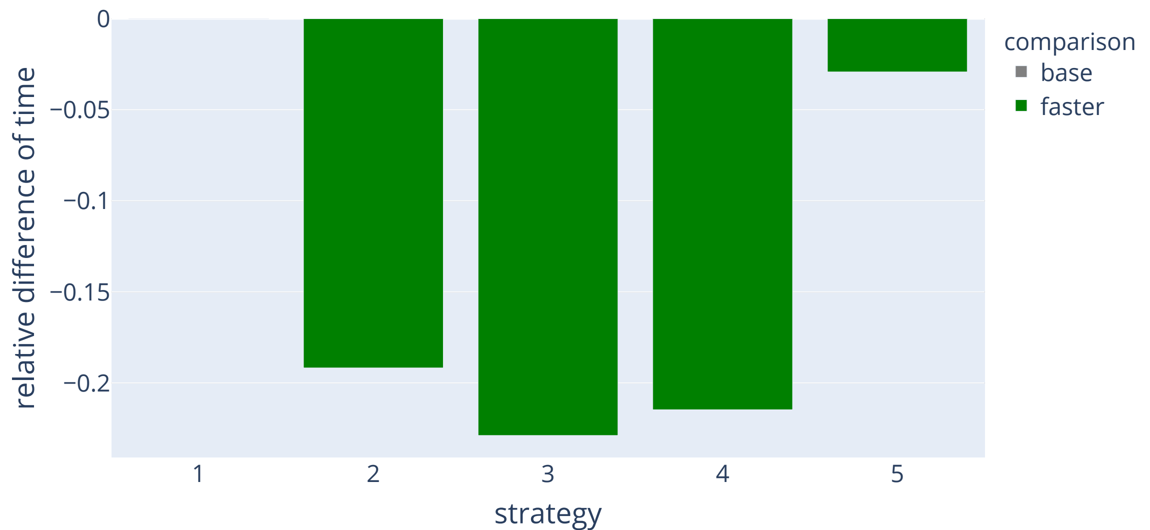


Figure 7.2.: Relative difference of total execution time of slowest scenario of several tuning-strategies. Strategies are compared to no MPI-tuning. Green bars represent faster strategies. The lower the bar, the faster the execution time.

7.2. Execution time of tuning and not tuning iterations

To gain a deeper understanding why MPI-tuning is faster, it is helpful to divide all iterations of a simulation into tuning iterations and non-tuning iterations, as explained in Section 6.1. For Figure 7.3 the execution time of all tuning iterations per scenario was added up. Then the relative difference to the base-strategy was calculated. These values are represented by the bars in the graph, which is divided into subplots for each tuning-strategy. Each bar in a subplot represents one scenario. This shows which scenario is performing the worst or the best.

As the relative difference was calculated, lower values also mean lower execution time, which is considered better. Blue colored bars represent strategies and scenarios, which were faster in either tuning or non-tuning iterations. Green bars show strategies and scenarios, which were faster in both tuning and non-tuning iterations. Lastly, red-colored bars show strategies and scenarios, which were slower in both tuning and non-tuning iterations. The vast majority of MPI tuning bars are blue because they have an advantage when it comes to tuning but a disadvantage for non-tuning iterations. This matches with the expectations stated in Section 6.1, as most of the time, the execution time for tuning iterations is lowered, but the time for non-tuning iterations is higher. The plot shows that the time during tuning for all MPI tuning strategies is nearly 90% lower than when no MPI tuning is used. This makes sense, hence the test is using 11 scenarios and, therefore 11 ranks that can share the tuning payload. As expected, the speedup of the new implementation can't be higher than the old one because the old one already exploits the full parallelization potential.

Figure 7.3 also shows a second plot, only for all non-tuning iterations. This means that all execution times outside the tuning phases were added up for this presentation. The bars also represent the relative difference to the base strategy, hence lower means faster and therefore better. The expectations here are also met, as the majority of MPI-tuning strategies perform worse than the base strategy. But the new implementation does not perform as badly as the old one, which is mostly seen for scenario K. This shows very well how the implementation has been improved, which was the essence of this thesis. By sorting ranks with similar scenarios into buckets, better tuning results are achieved. This leads to lower execution times for non-tuning iterations.

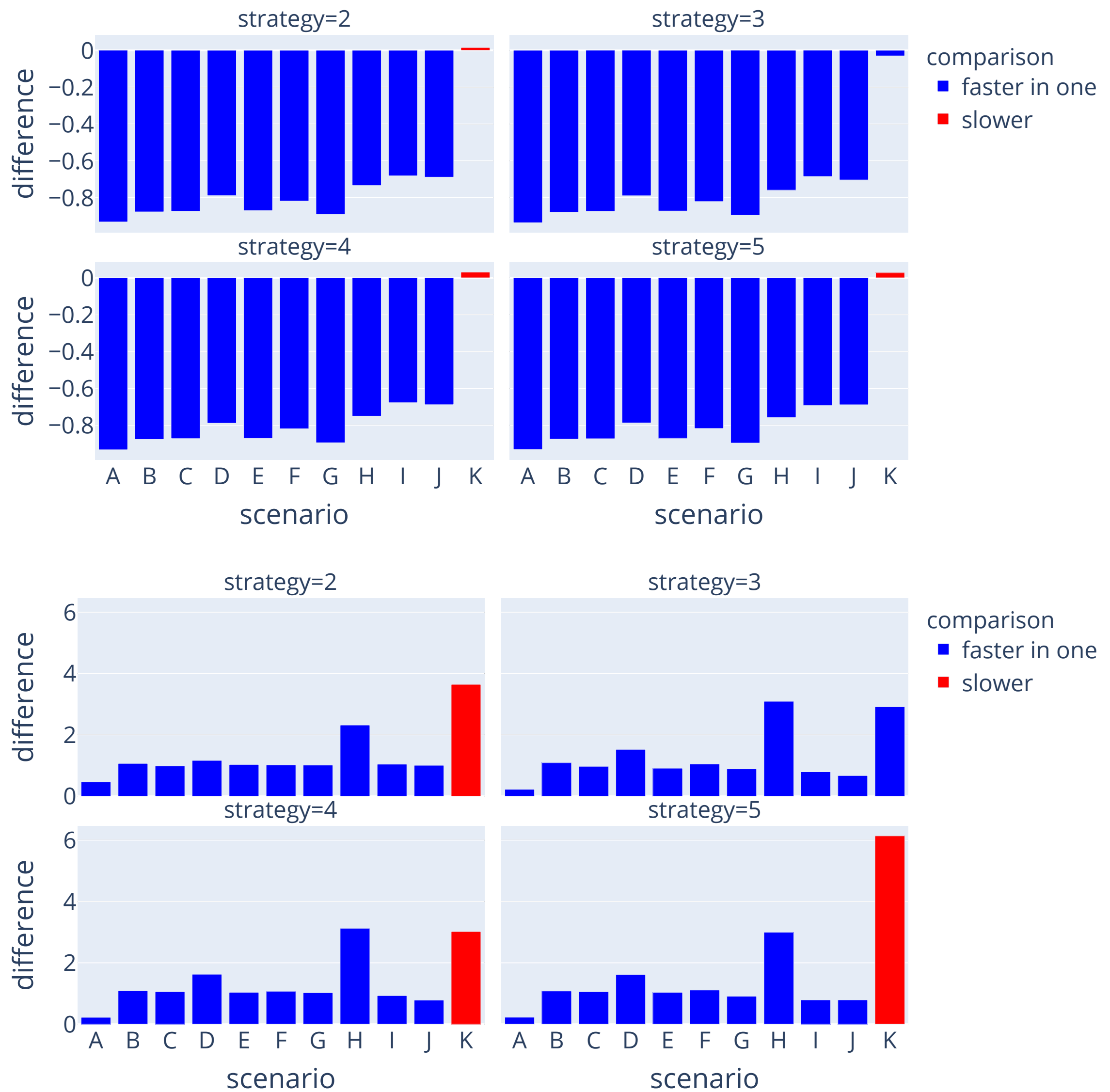


Figure 7.3.: Above: Relative difference of tuning time of several tuning-strategies. Below: Relative difference of non-tuning time of several tuning-strategies. Times are compared to no MPI-tuning and relative differences are calculated. Red bars represent slower scenarios, blue bars represent scenarios, which are faster in either tuning or non-tuning iterations green bars represent scenarios, which are faster in both.

7.3. Quality of tuning

The quality of tuning results can be expressed in more ways than just a lower execution time. Since it is assumed that the full-search strategy without MPI leads to an optimal tuning result, one can compare the bucket distribution of ranks and their tuning results with the best possible result. To keep this comparison comprehensible, only the results of one tuning phase will be plotted. Figure 7.4 aims to do exactly that for the results of the sixth tuning phase of the simulations. Just like the charts earlier, this one is divided into subplots for each tuning-strategy. However, the y-axis indicates the bucket a rank is sorted into, and the x-axis represents the 11 scenarios, which are each simulated by one rank—the color shows which Traversal was select as a tuning result. Using the Traversal here is more meaningful, as most of the time, the Traversal already indicated which Container is used and therefore provides a more distinct description.

The tuning-strategy without MPI has no bucket distribution, but since every rank does the tuning on its own, it acts like every rank is in its own bucket. Contrary to this, the old MPI tuning acts like all ranks are in the same bucket. That is why those two strategies also have a bucket distribution in the plot without actually sorting ranks into buckets.

One thing this illustration shows clearly is that all ranks in one bucket have the same tuning results, which is exactly what was intended by the implementation. The goal of achieving tuning results similar to the base strategy is represented here by having as many ranks as possible using the same Traversals as the base strategy. Which itself is illustrated in the figure by the same color. This leads to the conclusion that strategies with similar colored dots for each scenario should have similar execution times. When comparing Figure 7.4 with the execution times in Figure 7.1 this makes even more sense as strategies 3 and 4 picture the exact same results and have very similar execution times. Those two strategies are also more or less similar to the ideal result of the base strategy, which explains the low execution time. Furthermore, the results of strategy 5 are really not similar to the results of the base strategy, which leads to a longer execution time for non-tuning iterations, as shown in Figure 7.3.

This plot supports the premise of the thesis quite clearly. Tuning results heavily affect the execution time of a simulation. In order to get good results while using MPI, only similar scenarios must be compared. But the optimal result can only be achieved when every scenario is doing its own tuning.

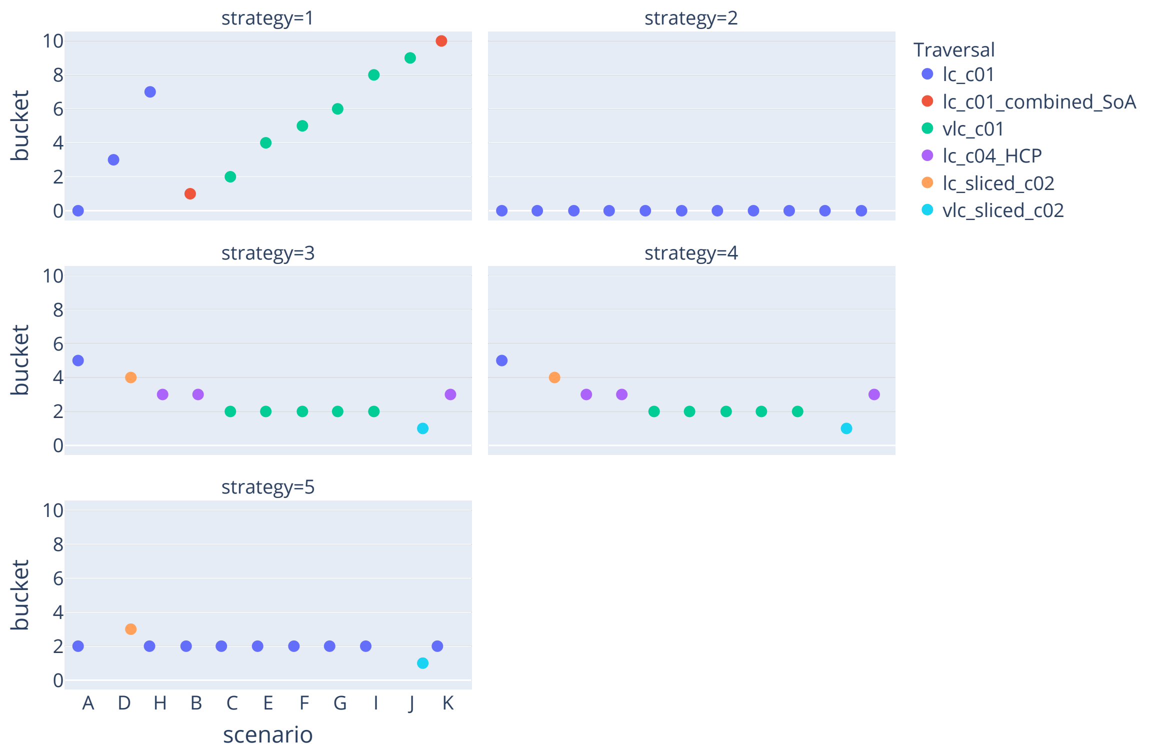


Figure 7.4.: Comparison of bucket-distribution of tuning-strategies with their tuning results. Scenarios in the same bucket have the same tuning result. Colors represent the chosen Traversal of the result

7.4. Comparison with new set of scenarios

Especially for finding the best values for *weight-for-max-density* and *max-difference-for-bucket* only one set of scenarios was used. To verify the implementation and the chosen values, a new set of scenarios was simulated using the full-Search strategy, the old MPI-strategy, and the new one. The times of the total execution time can be seen in Figure 7.5. Just like Figure 7.1 the color green shows that a strategy is faster than the base strategy (full search without MPI). Although the new MPI implementation is not faster than the old one, it is still faster than not using MPI. This proves the implementation works, but also questions how universal the default values for *weight-for-max-density* and *max-difference-for-bucket* are. This result suggests that these two parameters also have to be adjusted to the set of scenarios that will be simulated.

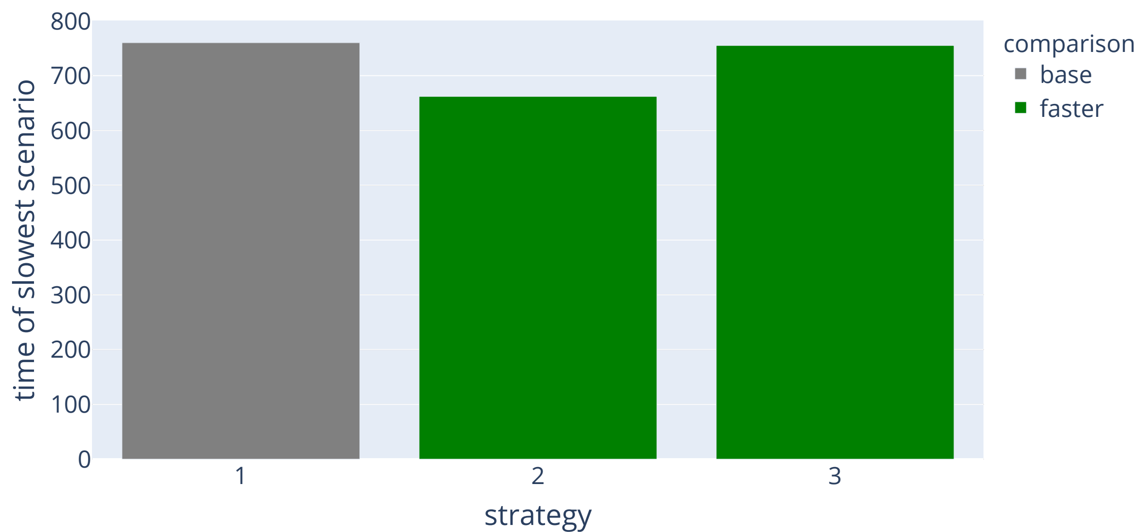


Figure 7.5.: Total execution time of slowest scenario of several tuning-strategies. Full-search without MPI is used as base strategy. Green bars represent faster strategies. Scenarios simulated are different than in the previous plots.

Part IV.

Conclusion

8. Conclusion and future Work

The MPI-parallelized-strategy of AutoPas was improved to yield reasonable tuning results even for inhomogeneous scenarios. This was achieved by finding a meaningful way to compare scenarios in AutoPas during run-time. The Similarity-Metric is mainly based on the homogeneity value of a scenario. The calculated homogeneity represents the overall structure of a scenario. It was shown that scenarios with similar homogeneity values behave similarly and produce similar tuning results.

This is exploited by sorting ranks into buckets based on the Similarity-Metric. Each bucket then distributes the search space of all valid configurations to its ranks. This way, only ranks with similar scenarios will tune in parallel and will share the next configuration. The communication for all this is done by MPI.

As expected, parallel tuning can drastically shorten tuning phases and therefore reduce the overall execution time, which is very desirable, as time is a valuable resource. This is accomplished because less inefficient configurations are tested by each rank. The results also showed that the tuning quality comes close to the quality of normal tuning. However, an improvement over the old implementation is not always guaranteed since the bucket distribution is not always ideal.

Two new configuration parameters were introduced. One of which can alter in which way the maximum density of a scenario affects its Similarity-Metric. The other one determines how different scenarios have to be to get sorted in different buckets. It was discovered that finding universal default values for both parameters is not trivial. Future work could focus on also automatically optimize these values, as parallel tuning only works with a good bucket distribution.

Also, the calculation of cell densities and homogeneity is not ideal, as both values can fluctuate very drastically. In this thesis, this was done by smoothing multiple values. Perhaps the algorithm computing these values could be improved to eliminate this behavior.

Eventually, a completely new idea for comparing scenarios or a new way to parallelize tuning for inhomogeneous scenarios could be found. This is certainly not the only approach to solve the problem. Maybe the tuning results of one rank could be used to predict tuning results of another rank, computing a similar scenario. This idea could be implemented in the Predictive-tuning strategy.

Part V.

Appendix

8.1. yaml file of scenario A

```yaml
verlet-rebuild-frequency        : 10
verlet-skin-radius              : 0.2
verlet-cluster-size             : 4
selector-strategy               : Fastest-Mean-Value
tuning-strategy                 : full-Search
tuning-interval                 : 5000
tuning-samples                  : 10
tuning-max-evidence             : 10
functor                         : Lennard-Jones AVX (12-6)
cutoff                          : 3
box-min                         : [0, 0, 0]
box-max                         : [16, 80, 16]
cell-size                       : [1]
deltaT                          : 0.0005
iterations                      : 100000
periodic-boundaries             : true
Objects:
  CubeClosestPacked:
    0:
      box-length                : [12, 6, 12]
      bottomLeftCorner          : [2, 20, 2]
      particle-spacing          : 1.
      velocity                  : [0, 0, 0]
      particle-type             : 0
      particle-epsilon          : 1
      particle-sigma            : 1
      particle-mass             : 0.5
    1:
      box-length                : [12, 6, 12]
      bottomLeftCorner          : [2, 56, 2]
      particle-spacing          : 1.
      velocity                  : [0, 0, 0]
      particle-type             : 1
      particle-epsilon          : 1
      particle-sigma            : 1
      particle-mass             : 2
log-level                       : debug
no-flops                        : false
no-end-config                   : true
no-progress-bar                 : false
mpi-strategy                    : divide-and-conquer
max-difference-for-bucket       : 0.3
weight-for-max-density          : 0.0
```

List of Figures

2.1. Lennard-Jones 12-6 Potential 3

3.1. Illustration of different container types 8
3.2. Data structures AoS and SoA 9
3.3. Auto-tuning in AutoPas 9

4.1. Scenario exploding-liquid 14
4.2. Homogeneity and maximum density of different scenarios 16
4.3. Heat-map of correlation matrix 19
4.4. Container-ranking in respect to homogeneity 20
4.5. Container-ranking in respect to maximum density 21

5.1. Illustration of configuration distribution 22
5.2. Bucket Distribution of one tuning phase 24

6.1. Scenario F: “light and heavy particles blending over domain” 29

7.1. Total time of several tuning-strategies 31
7.2. Relative difference of total time of several tuning-strategies 31
7.3. Relative difference of execution time of several tuning-strategies 33
7.4. Comparison of bucket-distribution of tuning-strategies 35
7.5. Total time of several tuning-strategies with other scenarios 36

List of Tables

4.1. Example for container-ranking 18
4.2. Correlation values of homogeneity and maximum density 19

5.1. Representation of MPI_Comm_split 24

6.1. Description of all used scenarios 28
6.2. Description of all used tuning strategies 28

Bibliography

[1] *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, New York, NY, USA, 2009. Association for Computing Machinery.

[2] B. J. Alder and T. E. Wainwright. Studies in molecular dynamics. i. general method. *The Journal of Chemical Physics*, 31(2):459–466, 1959.

[3] P. Balaprakash, J. Dongarra, T. Gamblin, M. Hall, J. Hollingsworth, B. Norris, and R. Vuduc. Autotuning in high-performance computing applications. *Proceedings of the IEEE*, 106, 07 2018.

[4] K. Binder, J. Horbach, W. Kob, W. Paul, and F. Varnik. Molecular dynamics simulations. 16(5):S429–S453, jan 2004.

[5] P. Cao, Z. Fan, R. X. Gao, and J. Tang. Solving configuration optimization problem with multiple hard constraints: An enhanced multi-objective simulated annealing approach, 2017.

[6] M. P. I. Forum. *MPI: A Message-passing Interface Standard, Version 3.1 ; June 4, 2015*. High-Performance Computing Center Stuttgart, University of Stuttgart, 2015.

[7] F. A. Gratl, S. Seckler, N. Tchipev, H.-J. Bungartz, and P. Neumann. Autopas: Auto-tuning for particle simulations. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 748–757, 05 2019.

[8] T. Hansson, C. Oostenbrink, and W. van Gunsteren. Molecular dynamics simulations. *Current Opinion in Structural Biology*, 12(2):190–196, 2002.

[9] T. Hansson, C. Oostenbrink, and W. van Gunsteren. Molecular dynamics simulations. *Current Opinion in Structural Biology*, 12(2):190–196, 2002.

[10] A. D. Myers, Jerome L; Well. *Research design and statistical analysis*. Inquiry and Pedagogy Across Diverse Contexts. Mahwah, N.J. : Lawrence Erlbaum Associates, 2003.

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

[12] I. Newton. Philosophiae naturalis principia mathematica, 1687.

[13] K. Pearson. Note on Regression and Inheritance in the Case of Two Parents. *Proceedings of the Royal Society of London Series I*, 58:240–242, Jan. 1895.

[14] C. W. R. Frigg. Entropy – a guide for the perplexed, 2010.

[15] D. J. Rumsey. *How to Interpret a Correlation Coefficient r.* 2015.

[16] W. Thieme. Parallelization of existing tuning strategies in autopas using mpi. Bachelorarbeit, Technical University of Munich, Sep 2020.

[17] M. Trenti and P. Hut. Gravitational n-body simulations, 2008.