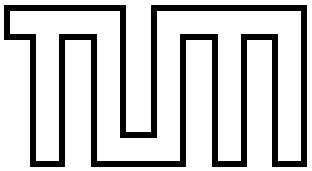


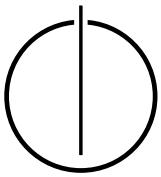
FAKULTÄT FÜR INFORMATIK

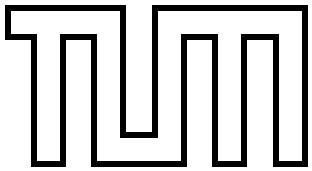
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics: Games Engineering

Parallelization of Existing Tuning Strategies in AutoPas using MPI

Wolf Thieme





FAKULTÄT FÜR INFORMATIK

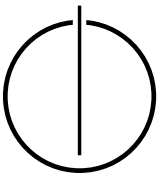
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics: Games Engineering

Parallelization of Existing Tuning Strategies in AutoPas using MPI

Parallelisierung bestehender Tuningstrategien in AutoPas mit MPI

Author:	Wolf Thieme
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisor:	Fabio Alexander Gratl, M.Sc. and Steffen Seckler, M.Sc.
Date:	15.09.2020



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

Munich, 15.09.2020 Wolf Thieme

Acknowledgements

I would primarily like to thank my advisors, Fabio Gratl and Steffen Seckler. Thank you for being patient when I still learned how AutoPas worked and for answering my E-mails even at 9 pm on a Sunday. Thank you, Steffen, for explaining many details about AutoPas and for considering my strange implementation ideas. Thank you, Fabio, for being a brainstorming partner when I was faced with challenges regarding my implementation. As the current pandemic made it impossible for me to live in Munich during the writing of this thesis, I would also like to thank my parents for housing me and allowing me to concentrate much more of my time on this thesis than it would have been possible otherwise. Lastly, I would like to thank my friend, Matthias, for reading through a draft of this thesis.

Abstract

Molecular Dynamics (MD) simulations are computationally highly involved and there exist many strategies to improve their efficiency. Because those strategies are difficult to select manually, it is desirable to automatically compare and choose them during run-time. This is called auto-tuning. AutoPas is a C++ MD-simulation library designed to do exactly that with custom simulations provided by the user[5]. However, it did not previously feature a way to parallelize the tuning process. Instead, each process used in a highly parallel MD-simulation had to find an optimal solution in isolation. This thesis implements a parallelization scheme, which allows AutoPas processes to communicate during tuning in order to find a solution more quickly. This is done using the MPI-specification[6]. As the solution has to be applicable for all processes, the simulation-domain is assumed to be homogeneous. The implementation is tested in various situations and compared to a non-parallelized execution of those same situations. Overall, it is shown that significant improvements in performance are possible, but also that they are highly situational. Therefore, future optimizations will be necessary to fully exploit the potential of parallelizing the tuning.

Zusammenfassung

Molekulardynamiksimulationen (MD-Simulationen) sind hochkompliziert zu berechnen und es gibt viele Strategien, um ihre Effizienz zu verbessern. Da diese Strategien nur schwer manuell bestimmt werden können, ist es wünschenswert, sie automatisch zu vergleichen und auszuwählen, was Auto-Tuning heißt. AutoPas ist eine C++ Bibliothek für MD-Simulationen, die eben das mit vom Nutzer eigens erstellten Simulationen tut[5]. Allerding konnte es bislang den Tuning-Prozess nicht parallelisieren. Stattdessen musste jeder Prozess in einer hoch-parallelen Simulation eine optimale Lösung selbst finden. Diese Arbeit implementiert eine Parallelisierung, die es AutoPas erlaubt, während des Tunings Informationen zwischen den Prozessen auszutauschen, um schneller eine Lösung zu finden. Dafür wird die MPI-Spezifikation verwendet[6]. Da die Lösung für alle Prozesse optimal sein soll, wird angenommen, dass die Simulationsdomäne homogen ist. Die Implementierung wird in verschiedenen Situationen mit einer nicht-parallelisierten Ausführung derselben Situationen verglichen. Im Ganzen wird gezeigt, dass dadurch signifikante Performanzsteigerungen möglich sind, aber nur in bestimmten Situationen. Deshalb werden künftige Optimierungen benötigt um das Potenzial eines parallelen Tuning-Vorgangs vollständig auszunutzen.

Contents

Part I.

Background & Introduction

1. Motivation

Complex Molecular Dynamics (MD) simulations can help scientists gain insights into many physical phenomena. These phenomena often occur on timescales of several milliseconds (10^{-3} s) and can involve millions of particles. When simulating, a distribution of particles is evolved by several discrete time steps. To yield reasonable results, a single time step will often be femtoseconds (10^{-15} s) in length [11]. For larger intervals, simulations lose accuracy because the errors in each step become larger. Femtoseconds provide a compromise between speed and acceptably accurate results. The difference between the simulated time scales and the time for a single step means that usually at least several millions of time steps need to be computed. While long-distance interactions are often ignored (see Section 2.1), a single time step still involves many thousands of interactions for every particle. In order to approach such simulations, one needs as much computational power as possible. However, simply increasing the number of cores to several thousands alone does not typically solve a computational problem. Researchers have developed several methods of improving performance, which will be discussed in more detail in Section 2.1. Unfortunately, the challenges that need to be solved depend strongly on the specific simulation.

Brooks already showed that for general software engineering problems of this kind, no universal solution can exist[3]. This is also true for MD-simulations[4]. Choosing the optimal approach for a simulation manually requires great intuition and foresight. Yet, many people who need to run MD-simulations have no computer science background. A possible solution to this is to use different strategies for parts of a simulation and automatically determine which one works best. This is called auto-tuning. AutoPas is a C++ library for particle simulations, which is based around this auto-tuning principle. More specifically, AutoPas allows users to code their own particle simulations and select which approaches should be tested. On execution, the simulation is begun with systematically measuring the performance for those approaches. When an optimum is found, it is used for many of the following time steps before beginning the procedure anew. More on this in Section 3.1.

As MD-simulations are often distributed over many compute units (called nodes from here on), it makes sense to also distribute the tuning. Before this thesis, AutoPas was not able to do this. This thesis implements a parallelization scheme which allows AutoPas to distribute the tuning-workload. The goal is to decrease the amount of time each process has to simulate with suboptimal approaches, thus improving overall execution time. The thesis limits itself to solving the parallelization for homogeneous simulation domains. This means that the optimal solution is the same everywhere in the domain. Undoubtedly, a heterogeneous simulation domain would introduce new complexity and will be left for future work. Other than that, it attempts to be as generally applicable as possible within AutoPas. This thesis will conclude that the parallelization can yield improvements in performance, but not in every scenario.

2. Theoretical Background

2.1. Molecular Dynamics & N-Body Simulations

Molecular Dynamics simulations allow important insights into the properties of certain materials or the behaviors of biophysical systems[2]. For this reason, they have been studied extensively and the High Performance Computing (HPC) Community has seen vast improvements in their efficiency and effectiveness[10]. Their complexity starts on the low end with simulating atoms as point-particles with a single kind of interaction between them. This interaction is usually modeled as a force potential, commonly the Lennard-Jones 12-6 potential depicted in Figure 2.1[5]. More complex simulations might explicitly model bonds between some number of atoms and use several potentials, sometimes between more than two particles. These are required for the modeling of many chemical processes. Some simulations even account for quantum-mechanical phenomena[2].

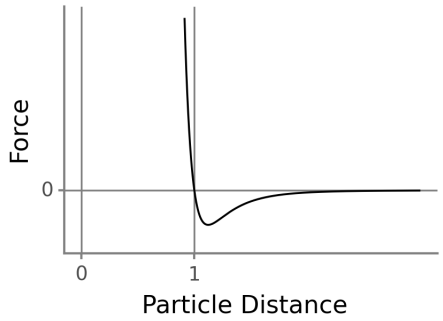


Figure 2.1.: The Lennard-Jones 12-6 Potential.
The specific values depend on two variables σ and ϵ [5]. The positive section results in a repulsion, while the negative one results in a pulling force between both particles.

In general, all MD-simulations share one common source of complexity. A physical simulation with N distinct objects, each interacting with all others via forces, is called an N-body simulation. Any such simulation has to solve what is commonly called the N-body problem[10]. The complexity arises from the fact that the number of force calculations increases quadratically with N. To alleviate this effect, optimizations, as well as approximations, are available.
A common optimization is to apply forces for two particles at a time. According to Newton's 3rd law of motion, any object acting on another with a certain force F experiences the equal but opposite force $-F$ on itself. This means that from knowing the force particle p_1 exerts upon particle p_2, one immediately knows the force p_2 exerts upon p_1. This optimization halves the number of force calculations compared to a primitive implementation. In AutoPas, it is called the Newton3-optimization.
An important approximation arises from the way short-range potentials (e.g. Lennard-Jones 12-6) behave for large particle distances. Because they commonly converge to 0 very quickly, not much is lost by assuming them to be exactly zero after certain distances. If these

cutoff-distances are significantly smaller than the simulation domain, a majority of particle interactions can be ignored. For a fixed cutoff-distance and particle density, this even means that the number of particle interactions to compute increases only linearly with N, not quadratically.

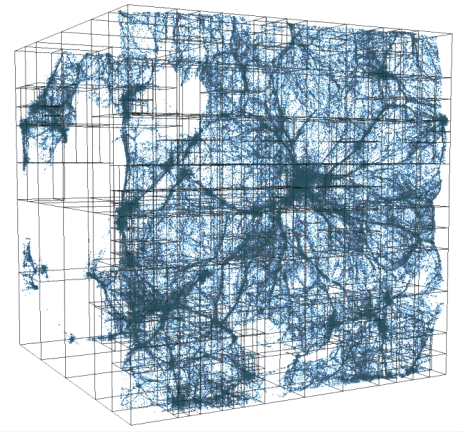


Figure 2.2.: Particles in a Visualized OCTREE. [14]

A second optimization is high parallelization [10]. This lends itself naturally to MD-simulations because the force acting between a pair of particles is independent of all other particles[1]. Another reason why parallelization follows naturally is that complex MD-simulations cannot be contained on a single node, as illustrated in Chapter 1. Specifically, there are two types of parallelizations to consider here: Shared memory parallelization on each node individually and distributed memory parallelization across all nodes. Both can also be used together. When implementing a parallel MD-simulation there are certain pitfalls to avoid, however.
For instance, when combining shared memory parallelization with the Newton3-optimization one must be careful to avoid data races. When calculating the particles in a random order, two threads might select the same pair simultaneously. In this case, the force from that interaction would be added twice per particle. For this reason, AutoPas implements several traversal patterns which determine a thread-safe order to compute the forces in.
For distributed memory parallelization, it is important to minimize communication bottlenecks. Commonly, the simulation domain will be divided into rectangular regions, for example with OCTREES [10], as depicted in Figure 2.2 or grids. Despite ignoring interactions after a cutoff-distance, there are large amounts of overlapping particle interactions between neighboring regions. The exact positions of those particles need to be updated continually on all nodes so that force calculations can remain accurate. This poses a severe challenge, as much information needs to be transmitted between many different nodes[2]. Oftentimes, the computers that run these simulations are organized in grids themselves[3], allowing for short transmission distances[6].

In conclusion, MD-simulations are very useful scientifically, but extremely difficult from a computer and software engineering perspective. Both dedicated hardware[11, 12] and advanced algorithmic solutions are being developed continually. Yet, many complex simulations are still out of reach, which underlines the importance of further research in the field.

[1] The computation of that force, however, is not independent if the Newton3-optimization is used. Details below

[2] Assuming that each sub-region in the domain is held by one node, this usually means transferring particles to up to $3^3 - 1 = 26$ neighbors in a 3-dimensional case.

[3] Technically in tori, because opposing sides are also connected.

2.2. Auto-Tuning

As indicated before, there exist several interchangeable strategies for running MD-simulations. They will be discussed in more detail in Section 3.1. For now, it is important that they are all qualitatively equivalent, assuming a cutoff-distance[4]. However, their run-times are highly dependent on certain properties of the simulation. For instance, particle density or average velocity can have a huge impact. Importantly, different strategies are affected differently by these properties, making it necessary to find the optimal choice for each simulation individually. This selection is not trivial to make, however, as the space of possible strategies is high-dimensional and MD-simulations can get very heterogeneous. Additionally, requiring a user to understand the advantages and disadvantages of all different algorithms is unrealistic and cumbersome[4]. A third problem is that the properties of a simulation can change during run-time, this changing which approach is optimal. A potent approach to solve these issues is called auto-tuning[17].

There are two primary types of auto-tuning. The first, sometimes called parameter-tuning or automatic algorithm configuration, changes certain variables that influence the performance of a given algorithm. An example of such a variable would be how far to unroll loops in auto-tuned compilers[4]. Another one is [13], where a 3D Fast-Fourier-Transform is tuned with a large parameter space. The second type, called automatic algorithm selection, is more powerful than that because it changes the algorithm altogether. This can improve performance drastically because a single algorithm with tuned variables might not be able to account for a wide range of problems (e.g. the range of all MD-simulations)[3]. Examples of possible algorithms will be discussed in Section 3.1. AutoPas includes both types of auto-tuning but relies heavily on the latter. Auto-tuning can generally occur at three different stages of a program's lifetime[5]:

1. At compile-time.
   This approach is only viable when the optimal solution does not depend on any form of input. The relevant information which informs the tuning-process here comes from the systems architecture and possibly available libraries.
2. During initialization.
   This works well for programs in which the optimal solution only depends on initial input and does not change during execution. Also, it has to be possible to infer the optimum from knowledge given during initialization. This could be achieved by doing some pre-computation on the problem and tuning based on known heuristics.
3. Anywhere throughout run-time, usually periodically or continually.
   This option becomes viable when the only way to determine which solution is optimal is trial-and-error. It becomes necessary when the optimal solution varies throughout execution. Commonly, a program will tune at the beginning and then repeat the tuning regularly to keep using the optimal approach.

It should be noted that going down this list, tuning becomes more flexible and more time-intensive as it has to be done more often. Going into a tuning phase while the old solution

[4] Otherwise, a primitive solution without cutoff-distance would be technically more precise than a solution with cutoff-distance. This is unimportant here, as AutoPas only considers short-range interactions with a cutoff-distance.

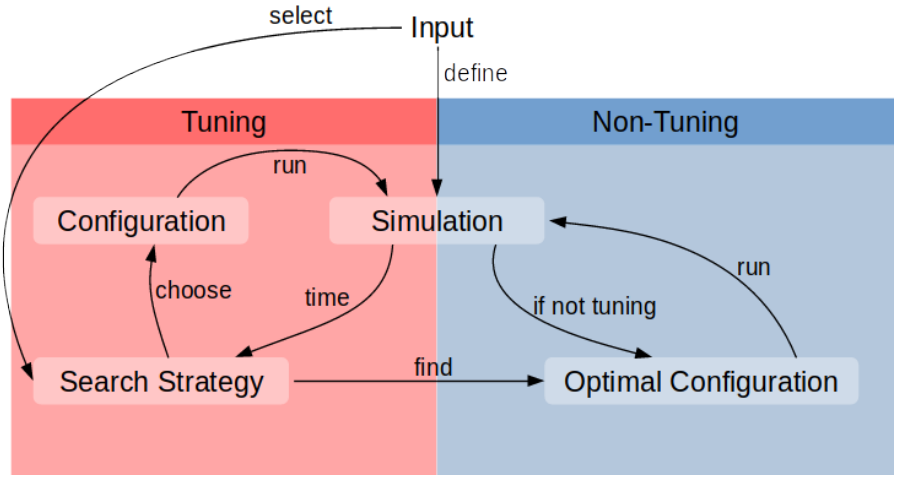
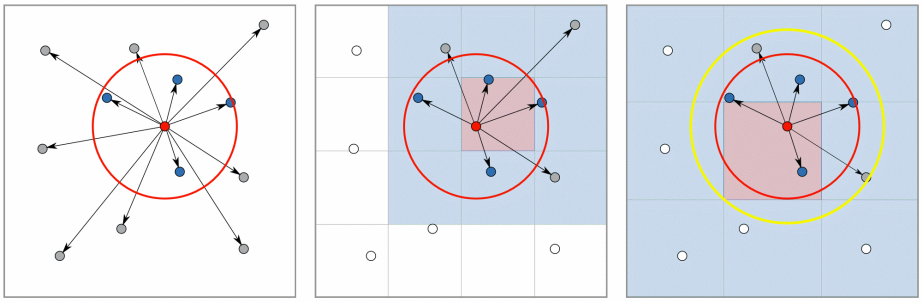


Figure 2.3.: Tuning in AutoPas, Simplified.
A configuration defines the combination of algorithms used to run the simulation. The search strategy defines which configurations to test next and how to find the optimum. More on that in Section 3.1. During a tuning phase, the search strategy tests different configurations by measuring their execution time until it is confident to select an optimal configuration. The optimum is then used until a fixed number of time steps elapsed, after which the tuning phase is repeated.

is still optimal results in unnecessary overhead. Thus, choosing to tune periodically during run-time is not a good option if the problem does not require it. An example of on-line auto-tuning is discussed in [15]. Similar to AutoPas, they distribute the tuning process over many nodes, however without algorithm selection.
MD-simulations can change drastically over time. For instance, particles could oscillate between being condensed on a small amount of space and being evenly spread over the entire domain. These two extremes have different optima, which is why auto-tuning for MD-simulations has to happen regularly. AutoPas specifically has time intervals of a set length between all tuning phases. This enables an efficient trade-off between tuning regularly and keeping tuning times low. A depiction of how auto-tuning works in AutoPas is given in Figure 2.3.

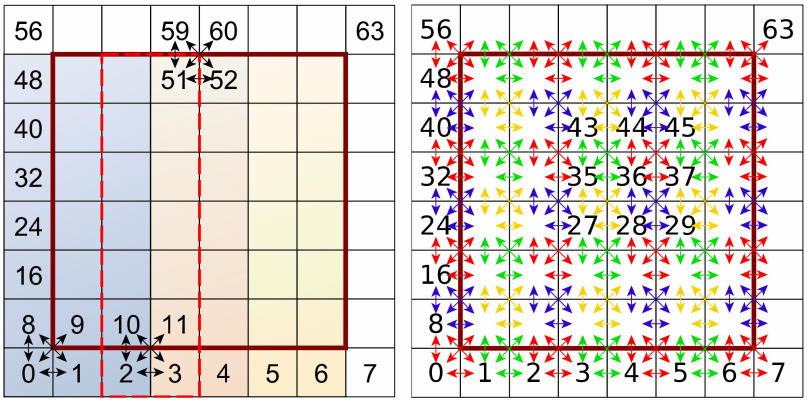
3. Technical Background

3.1. AutoPas



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

Figure 3.1.: Three Containers Visualized. [5]
The red circle shows the cutoff-distance in which forces are calculated for the red particle. For Verlet-lists, the yellow circle shows the boundaries of the neighbor list. An arrow going into a particle means that that particle is considered for force calculations.



(a) sliced (b) c08

Figure 3.2.: Two Traversals Visualized. [5]
Cells outside the red boxes are halo cells, used to handle boundary conditions.

"AutoPas is an open-source C++ node-level performance library, which aims to provide a base for arbitrary N-body simulations."[5] It approaches this goal by defining an interface for particles and their respective force calculations (via functors), which the user codes specific

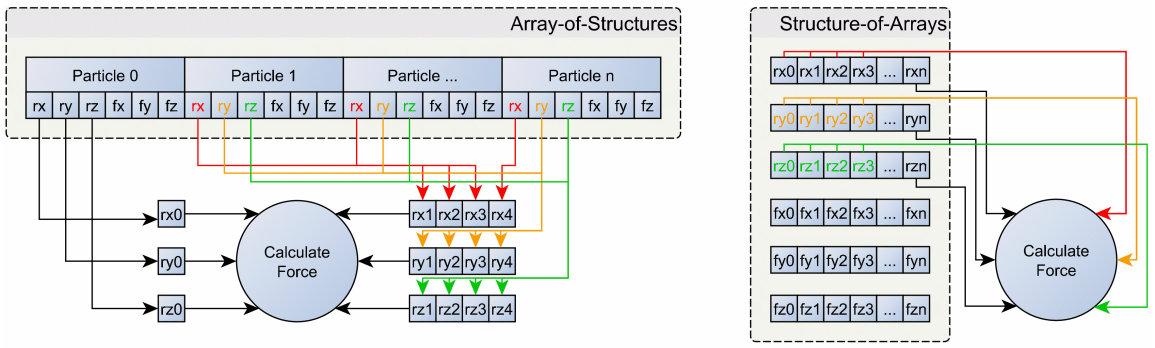


Figure 3.3.: Two Data Layout Options. [5]

implementations for. This allows for high flexibility concerning the simulation without requiring knowledge about any optimizations or algorithms from the user. Instead, AutoPas implements several options, which it automatically tunes to find an efficient execution for the given simulation. At the time of writing, there are six tuneable options with several different values each:

1. Containers:
   They determine how particle information is stored. Specifically, they keeps track of which particle pairs to consider for force calculations. Figure 3.1 shows three examples for containers. One can see that going from left to right, fewer unnecessary particles are considered. The linked-cells approach achieves this by dividing the simulation domain into squares (or cubes for the 3D case)[5]. Then, only the particles in the surrounding cells and the one that contains the red particle need to be considered. This turns the quadratic dependence on the number of particles into a linear one. The Verlet-list approach stores and regularly updates lists of neighboring particles. It can be used with linked-cells for finding the neighbors, which combines the linear dependence on N with the small number of distance computations of the Verlet-lists. There exist several other containers, many of which are variants of the mentioned ones.
2. Cell-size-factors:
   These make up the only instance of parameter-tuning in AutoPas. They correspond to how big the cells are for containers that use cells (e.g. linked-cells). Unlike the other options (which are enumerated values), these have two possible representations. They can either be stored as an arbitrarily large set of numbers or as an interval that is considered infinite in AutoPas.
3. Traversals:
   Different traversal strategies are used to avoid race conditions on a single node. They control which order particle interactions are computed in. Figure 3.2 shows two common options. The sliced traversal splits the domain into long slices which are assigned to one thread each[5]. Thus, the total number of threads is limited by the domain size. An alternative, the c08-traversal, computes interactions in non-adjacent nodes simultaneously. This means additional scheduling overhead, but very good load balancing. The traversal options are by far the most numerous in AutoPas, but most traversals are only compatible with a small subset of containers.

4. Load Estimators:
   These are used to estimate the costs associated with updating a sub-region of the domain. AutoPas currently implements two heuristics for doing so (next to not estimating at all). One is based on the number of particles, the other on the length of Verlet-lists.
5. Data Layouts:
   These options affect how particle information is vectorized. The main distinction is between Array-of-Structures (AoS) and Structure-of-Arrays (SoA), as depicted in Figure 3.3. AoS groups information by which particle it belongs to and allows quick access to other information of the same particle. SoA on the other hand groups information by its type (e.g. x-coordinate of the position vector), which allows quick access to the same information of other particles. A third option (called CUDA) is a data layout optimized for GPUs.
6. Newton3:
   This simply refers to whether the Newton3 optimization explained in Section 2.1 is utilized. While one may want to always use it, it is not compatible with all user-defined functors.

Any combination of values from those six options is called a configuration and is responsible for the exact way a simulation is executed. In a simulation with several processes, each process can use its own configuration. At the beginning of a simulation, the user may specify which values should be allowed for each option[1]. The set of (valid) configurations that arises from these values is called the search space. Determining which configuration to test next and which to select as the optimum defines a search strategy (also called tuning strategy). At the time of writing, AutoPas implements six search strategies, all of which are based on measuring execution times for configurations:

1. Active Harmony (AH).
   Active Harmony is an auto-tuning library itself[1], which has been integrated into AutoPas. AH is far more general in scope than AutoPas and thus could not make many important optimizations for MD-simulations on its own. It implements several search strategies itself, but the one used in AutoPas is the Nelder-Mead simplex method. This strategy treats the search space as a (in this case) five-dimensional simplex[2]. It then attempts to find the optimal strategy by interpreting the measured performance as the values of a continuous function on the search space. Architecturally, AH uses a server-client model. Either each AH process can use its own local server, or a server can be set up which communicates via TCP with all client instances in a global tuning session.
2. Bayesian-search &
3. Bayesian-cluster-search.
   Similar to AH, these two search strategies try to find the minimum of a function that maps configurations to measured execution times. To do this, they rely on Gaussian processes[8]. Essentially, they assume that the measured performances

[1] A common restriction is to disallow the direct-sum container to improve performance.
[2] A geometric object in five dimensions consisting of six points which are all connected to all others.

Search Strategy	Next Configuration	Finishing Criterion
Active Harmony	most promising	convergence
Bayesian (Cluster) Search		# of tested configurations
Random Search	random	
Predictive Tuning	strict order in promising set	whole promising set tested
Full Search	strict order	whole search space tested

Table 3.1.: Brief, Simplified Overview over the Functioning of all Tuning Strategies.

of a configuration are normally distributed. Then they attempt to predict which configuration would be the most useful[3] to test next. The difference between both is that Bayesian-cluster-search fixes certain values during its search.

4. Full-search.
   This is an exhaustive search through the configurations. The search space is traversed in a way that minimizes switches in options that are expensive to switch (e.g. containers). Once all configurations have been tested once, it selects the one with the fastest measured performance. Due to being exhaustive, this is the only search strategy that is always guaranteed to find the optimal solution but is also slower than most others.
5. Predictive-tuning.
   This strategy attempts to predict the performance for every configuration at the start of a tuning-phase. It then tests the configurations which are expected to perform best. When this reduced search space is depleted, it selects the configuration which was measured to be the optimum among those. There are four prediction schemes available, which use information from past tuning-phases to predict the current performance of each configuration.
6. Random-search.
   This and full-search are the two least sophisticated tuning-strategies in AutoPas. Random search selects new tuning strategies from the search space completely by chance and selects the best one after a certain number of different configurations have been tested.

A short overview over the tuning strategies is provided in Table 3.1. All auto-tuning that happens in AutoPas is based on measuring execution-times when employing different configurations. These times are then used to select an optimal configuration for the given problem. For this to be functional, it is assumed that a simulation does not change too drastically in a short number of time steps. Otherwise, the measured times could become unrepresentative before or shortly after the tuning is completed. To account for a changing simulation, tuning phases are repeated periodically.

[3] i.e. the one whose measurement would provide the most information for tuning. For instance, if changing the cell-size-factor twice has resulted in almost no performance difference, changing it again will likely not be useful.

3.2. MPI

The Message-Passing Interface (MPI) is a specification "address[ing] the message-passing parallel programming model[, extended by] collective operations, remote-memory access operations, dynamic process creation, and parallel I/O."[6] It allows for different kinds of message-based communication between several nodes. For the purposes of this thesis, a message here is simply a collection of a single type of information (e.g. a collection of float values) to be communicated between processes. The utility of MPI arises from the computations that can be done during communication. For instance, MPI defines several common reduction operations (e.g. finding the minimum of a collection of numbers) and calls to collect or distribute large data sets automatically. Among many of its stated goals are efficiency and portability, as well as supporting heterogeneous environments[6]. These make it well suited for MD-simulations in general. MPI itself is only a specification, defining requirements and interfaces for a possible implementation. Two common such implementations are MPICH [7] and OpenMPI [9].

In order to identify individual processes, MPI uses the concept of ranks, which it automatically assigns upon initialization. Ranks are usually assigned to all processes with consecutive numbers starting at zero, where by default the process with rank zero is considered the "root" node. This is important for many asymmetric MPI-calls (e.g. a broadcast from one process), although MPI always allows any other node to be specified as root. Complex MD-simulations are usually carried out on supercomputers, whose nodes are often configured in grids or torus-shapes. Therefore, to handle communication efficiently, a linear numbering is often insufficient. MPI accommodates for this, by allowing the user to define a custom node topology [6].

Many of the calls it defines come in blocking and non-blocking versions. Blocking here means that the code after the call is not executed until the call is finished. Importantly, this means that all processes need to participate simultaneously for it to succeed. A non-blocking call on the other hand returns a request handle. This handle can be used later to test if the call has already finished or to wait for it to finish. For non-blocking communication, it is unimportant when each rank joins in, but it can only finish once all ranks have done so. By their nature, non-blocking calls allow processes to execute other code while waiting.

Part II.

Implementation

4. Overview

As stated in Chapter 1, this thesis aims at distributing the tuning workload in AutoPas across several processes. Before, AutoPas was unable to communicate tuning information in a distributed memory environment. As MD-simulations are commonly executed on many nodes at a time, AutoPas spent much time measuring configurations whose performance was already known. When the underlying simulation is sufficiently homogeneous, the measurements of one configuration do not differ significantly across the nodes. This can be exploited by only measuring each configuration on a single node and communicating the result globally. To reduce communication overhead, one could broadcast only the time of the optimal configuration of each node. Since sub-optimal configurations should not be used, their times are not important to find the globally optimal one. Some search strategies (e.g. predictive-tuning), however, require the measurements of all configurations to function properly. This problem can be avoided by giving each node only a subset of the total search space. Then, each process can fully explore its search space with any strategy and does not need to know of other configurations. For this to work as intended, the union of all local search spaces needs to equal the global one. Once the local optima of all nodes are found, they can be compared to find the global optimum. This approach can be implemented well by wrapping the existing search strategies rather than integrating new code into them. Thus, a new wrapper search strategy, called *MPIParallelizedStrategy* was implemented. It implements the same interface as all other search strategies but holds a reference to another, local, search strategy which does most of the tuning. Its functioning will be described in more detail in Chapter 5.

The types of distributed communication necessary for this are comparatively uncomplicated. For the tuning itself, it is necessary to find the minimum in a distributed set of numbers (time measurements) and to broadcast the optimal configuration. Since non-serialized objects cannot generally be transmitted between nodes, the configurations are serialized and deserialized locally. To avoid race conditions and deadlocks, it is also needed to synchronize the control-flow of different processes. To this end, MPI was used. As it is a widely accepted standard, there are several efficient and well-supported implementations available. Additionally, all aforementioned use cases can be realized in MPI with very few calls.

5. Parallel Tuning Algorithms

5.1. Distribution

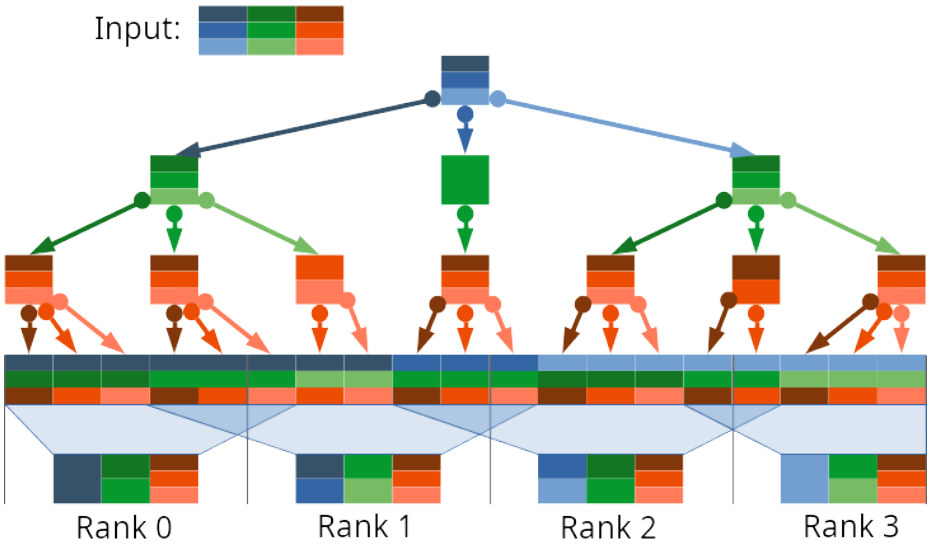


Figure 5.1.: Simplified Depiction of the Distribution of Configurations Across four Ranks. A combination of one blue, one green, and one red rectangle is a configuration. "`Input:`" denotes the global search space. The tree structure shows how all configurations are built from the input. "`Rank X`" denotes the output for Rank X, with the light blue bridges showing which parts of the array the options correspond to.

The way configurations are distributed across ranks is influenced by three considerations:

1. Load Balancing.
   For this, it is assumed that all ranks run on equally powerful machines with similar CPU time assigned to them. While it could be more efficient to assign larger search spaces to faster processes, implementing this would have resulted in more complex code and computation overhead in other areas. A big reason for this is that processes are synchronized every few time steps, as will be explained below. Therefore, it is attempted to partition the search space into equally large subsets.
2. Load Reduction.
   Full-search traverses its search space by exhausting options that can quickly switch between values before changing other, "heavier" options. For example, containers are switched as seldom as possible, because doing so takes more time than for any other option. Motivated by this, the distribution scheme assigns as few "heavy" options

to one rank as possible, while still maintaining equally large subsets. To do so, it traverses the search space in the same way as full-search does, when distributing. This way, all search strategies can benefit from a reduced load when switching between configurations.

3. Storage.
   Most search strategies in AutoPas do not store a set of configurations. Instead, they store a set of values for each of the six options. From here on, storing sets of values for options will be called the implicit representation, while storing configurations will be called explicit. The benefit of the implicit version is that no redundant information is stored. The user enables or disables values for options instead of individual configurations, so it is usually of no value to have any finer control over the search space. Also, some search strategies (e.g. Active Harmony, Bayesian-search) require the implicit representation. However, for splitting the search space at arbitrary points, an explicit approach would be necessary. As a compromise, the search space is first split according to the two considerations above. For this, the explicit representation is never actually built, but six iterators are used to traverse the search space. This has the same effect as building the explicit representation without the memory overhead. Then, the implicit representation is created per rank by storing all unique values that are used in configurations for the rank. Due to using the implicit representation, some overlapping configurations need to be accepted.

These points lead to the distribution algorithm depicted in Figure 5.1. Note that the array of configurations does not equal the Cartesian product because not all combinations are valid. Thus, its size is 19 instead of $3^3 = 27$. The values of the blue option are grouped, as it represents a "heavy" one. The values of the green option are grouped within that because they are the next "heaviest". The global search space is split equally[1] across four ranks, as can be seen by the vertical lines below it. From this, the implicit representation is generated by storing for all options the values that appear in each sub-array. In this example, even though only 19 unique configurations exist in total, there are 8 which are not uniquely assigned. Thus, adding the sizes of all four local search spaces up results in 27 instead of 19. That this is equal to the size of the Cartesian product without eliminating invalid configurations is a coincidence.

Realistically, there would be significantly more than 3 options with 3 values each. At the time of writing, options hold between 2 and 21 values. Cell-size-factors can even hold arbitrarily many values, with intervals being considered infinitely large. In the case of intervals, the distribution scheme ignores them altogether. Only if more ranks than configurations exist, are they split evenly between ranks that store the same single configuration.

One problem that can arise from using this distribution scheme with many ranks is that some of them might end up without valid configurations. Even though it is avoided to assign invalid configurations, some may only be found to be invalid after tuning already begun. Due to the way AutoPas is currently implemented, this is an unfortunate, but unavoidable consequence. Section 5.2 discusses how this situation is dealt with.

[1] As the total number of configurations is not divisible by four, Rank 3 has one fewer than the other ranks.

The involvement of MPI in this is limited to providing the number of processes and the rank of each one. From this, each process can generate the explicit representation and find its respective sub-array on its own. Alternatively, it would have been possible to dedicate a root node to the computation and use MPI to distribute the final results to all processes. This was avoided, because the computation phase would take the same amount of time, but would be followed by a large communication call. The total time would thus increase. Because the non-root nodes would not have been able to do other computations in the meantime, no workload reduction would have occurred. A second option might have been to use a dedicated MPI-call for the distribution. To do this, however, the explicit representation would need to be stored completely. Also, all configurations in it would need to be serialized, and deserialized after the communication. This would also have resulted in unneeded overhead and more complex code. Since this code is only called once per simulation, the potential performance increase was not a reasonable justification for this.

5.2. Tuning

Algorithm 1: Outline of the *tune()* method in the wrapper. Called once per tuning step and whenever an invalid configuration has been selected. Delegates tuning to the local search strategy until that finished or fails. Then uses MPI to find the globally fastest solution. *optimalConfig* is initially unassigned.

```
   Input  : Whether the currently selected configuration is invalid.
   Output : Whether the tuning phase needs to be continued.
 1 Function tune(invalid: bool):
       // handling fallback options
 2     if localStrategy.invalid() then
 3         usingFallbackOptions ← True
 4         localSearchFinished ← True
 5     if usingFallbackOptions and invalid then
 6         optimalConfig ← findNextFallbackOption()
           // only enter global synchronization once per tuning step
 7         return True
       // handling local tuning
 8     if not localSearchFinished then
 9         localSearchFinished ← not localStrategy.tune(invalid)
10         if invalid then
               // only enter global synchronization once per tuning step
11             return True
       // handling global synchronization and tuning
12     allLocalFinished ← synchronizeAndGetAllLocalFinished(synchronizationHandle)
13     if allLocalFinished then
14         if localStrategy.invalid() then
15             localOptimum ← null
16             localBestTime ← ∞
17         else
18             localOptimum ← localStrategy.optimalConfig
19             localBestTime ← localStrategy.bestTime
20         optimalConfig ← findGlobalOptimum(localOptimum, localBestTime)
21         return False
22     testGlobalSearchFinished(localSearchFinished, out synchronizationHandle)
23     return True
```

The primary principle behind the tuning is to tune completely locally until an optimum is found. Then, all local optima are compared to find a global optimum. As mentioned before, some ranks might end up without valid configurations. For this case, *MPIParallelizedStrategy* stores the global search space. If its local strategy fails, it declares the local tuning phase as completed and uses the global search space as a source of valid configurations. If the local strategy has not failed yet, the tuning is delegated to it. This behavior is depicted until line 11 in Algorithm 1.

Once every tuning step, all processes are synchronized (line 12). This is used to communicate globally if all local tuning phases are finished, via the *synchronizationHandle*. The handle is set in a non-blocking MPI-call (depicted in line 22), which reduces all provided Boolean values via a logical and-operation. The call is non-blocking to allow for computation-communication overlap, which improves performance[2]. It has been attempted to only synchronize once at the end of the tuning phase. This could in theory allow for less time spent waiting when different ranks use slow configurations during different time steps. However, since simulation codes also often need to communicate between processes, it would have been difficult to avoid deadlocks. When testing this approach, deadlocks often occurred when attempting to synchronize the ranks after all had finished their local tuning. Another factor for not pursuing this approach further is that MPI does not need to evaluate non-blocking calls immediately. This means that more time than necessary could have been spent with suboptimal configurations when waiting for the non-blocking MPI call to finish. Lastly, distributed simulation domains need to be synchronized anyways to remain accurate. This means that not much time can be saved by a non-blocking approach.

5.3. Optimization

Algorithm 2: Outline of the *findGlobalOptimum(...)* method as used in Algorithm 1. Compares each rank's best time to find the optimum. Then broadcasts the optimum from the respective rank to all others with *getConfigFrom(...)*. Serialization and deserialization were omitted to simplify the code.

Input : The optimum found in the local tuning phase and the time measured for it.
Output : The globally optimal configuration.

```
1 Function findGlobalOptimum(localOptimum: Configuration, localTime: time):
2     bestTime ← findBestTime(localTime)
3     bestRank ← communicateBestRank(bestTime)
4     return getConfigFrom(bestRank)
```

Finding the globally optimal configuration is achieved by comparing the measured times of all local optima. If two processes measured the same time for their respective optimum (which is unlikely), the one with the lower rank is chosen, as depicted in Algorithm 2. It would have been possible to combine the calls to *findBestTime* and *communicateBestRank* into one. However, the gain in performance would have been negligible, because no time is wasted waiting between both calls. On the other hand, the programming demand would have increased. MPI does provide a call to automatically find the minimum of a set of numbers and the rank which holds this minimum[3]. Unfortunately, the necessary operator is not defined for the data type of the time measurements (*size_t*). The operator would have had to be implemented and integrated into MPI beforehand, which results in the increased

[2] As the call is rather uncomplicated, the benefit is not noticeable. But a blocking alternative would not have resulted in significantly simpler code either.

[3] More specifically, it provides the operator *MPI_MINLOC* which can be used with a regular reduction call to achieve this.

programming demand.

findGlobalOptimum is fully blocking. A non-blocking implementation has also been considered. However, MPI does not provide a sufficiently powerful call to handle the entire global optimization at once. Thus, it would have to be done over several tuning steps. Since every tuning step also consists of several time steps, this would mean many more time steps of using suboptimal configurations. Simply optimizing in one set of blocking calls is much quicker in comparison. Also, at this part of the *tune()*-function, all processes are already synchronized due to the reasons outlined in Section 5.2. Thus, a non-blocking version could only have been useful if *tune()* was entirely non-blocking as well.

6. Integration into AutoPas

6.1. Changes to Tuning Strategies

As mentioned in Section 3.1, Active Harmony (AH) can use a TCP-server to handle distributed tuning sessions[1]. This is achieved by providing each AH-client with the hostname and port that the server can be contacted with. For this, two environment variables, *HARMONY_HOST* and *HARMONY_PORT* are used. However, setting up a global tuning session was not supported in AutoPas before this thesis. The reason is that to do so, some prior communication between the processes is necessary, which is usually done via MPI. Instead, every AutoPas process would set up its own, local AH-server for tuning. Thus, the first step was to implement global tuning sessions via the AH-server. For this code to execute, two conditions need to be met: MPI must be enabled in AutoPas, and the mentioned environment variables need to be set.

	Host & Port Set	Host & Port not Set
MPI enabled	global AH-server	local AH-server & MPI-tuning
MPI disabled	local AH-server	

Table 6.1.: Summary of the Results of all Combinations between MPI and a Global AH-Server. "MPI enabled" and "MPI disabled" here mean whether *mpi.h* and the appropriate MPI-strategy is used. More on this in Section 6.2.

Table 6.1 shows how AutoPas behaves depending on whether MPI is enabled and whether the environment variables for an AH-server are set. With both disabled, AutoPas functions as before. With both enabled, it ignores the MPI-wrapper strategy and only uses a server-based AH-tuning session. This also includes the initial distribution of the search space, because the server needs to be set up from a single node. This root-node thus needs to know the entire search space. Without MPI, it is not possible to set up a global tuning session, because the number of participating processes needs to be known beforehand. Thus, when a server is provided, but MPI disabled, the server has to be ignored and each AH-client creates its own, local server. With MPI enabled, but no server provided, AH is treated like any other search strategy and global MPI-tuning as described in this thesis is used.

Another change had to be made in random-search. As mentioned in Section 5.1, the division of the search space can result in processes with few or no valid configurations. This is handled with the fallback-behavior explained in Section 5.2. In order to determine when a search strategy has failed, two signals are used: Either the strategy throws an exception, or the strategy selects an invalid configuration as its optimum. Random-search did none of those things by default. For too few[1] valid configurations, it could never finish a tuning

[1] by default less than 10

phase, because it could not collect enough time measurements. With no valid configurations it would not even finish the first time step, but instead just keep randomly selecting invalid configurations. To fix this, random-search now keeps track of configurations that are already proven to be invalid. With this, it can test whether there are any untested configurations left. If there are not, it tries to select the optimal configuration from the valid, tested ones.

6.2. Other Changes

In order to not interfere with user-level MPI-calls, AutoPas will use a copy of the global MPI-communicator by default. This guarantees that AutoPas still largely functions as a black-box, which is one of its stated goals[5]. The user may execute their own MPI code in combination with AutoPas without needing to know which MPI-calls it uses. However, the user still needs to understand that AutoPas does use MPI and that the processes are synchronized regularly. This is important for avoiding deadlocks. Also, to adhere to xSDK-policies, it is possible to set the MPI-communicator that will be used for all internal operations[16]. This leads to AutoPas not copying the default communicator. To deallocate the default communicator and avoid memory leaks, AutoPas now features a *finalize()* method, which must be called before *MPI_Finalize()*.

There is now a new Option, *MPIStrategyOption*, which allows the user to decide between MPI-distributed or isolated execution. The new option was introduced to be able to use MPI in AutoPas without automatically using the new parallelization scheme. Future work could also use it to differentiate between more nuanced MPI-parallelization schemes. It is also provided to the AH-based search, where it is used to determine whether a server-based communication can be set up. Along with the option came a new compiler flag for enabling or disabling MPI usage in AutoPas.

A wrapper for common MPI-calls, *WrapMPI.h*, is used to reduce code complexity by providing default behaviors in cases where MPI is disabled. Since xSDK-policies demand that no valid MPI-installation should be assumed[16], it would normally take many preprocessor-commands to properly implement both cases. A wrapper reduces that number to a minimum because the non-MPI version of an MPI-call is usually clear without context[2]. With MPI enabled, the wrapper adds no functionality and thus no overhead. With MPI disabled, it provides primitive replicates of the actual MPI-calls based on the assumption that the current process is the only one.

[2] e.g. For an MPI-call which broadcasts a configuration from one rank to all others, the non-MPI version is usually to simply copy the configuration between buffers.

Part III.

Results

7. Measurements

7.1. Expectation

To analyze the expected speed-up during tuning, a simplified model of a tuning phase is assumed. In this model, there is an exact correspondence between configurations and their measured times. This assumption is roughly realistic, as the measurements for a single configuration tend to stay at similar values for consecutive time steps. The model also assumes that the used configuration changes after every time step and that no configuration is ever computed twice. This is less realistic, as AutoPas always takes several measurements per configuration. The difference is negligible, however, because the number of measurements per configuration tends to stay the same throughout a tuning phase. Therefore, combining several time steps into one "tuning step" per configuration, as the model essentially does, is valid. Random-search usually also computes some configurations several times. Yet, as it is no practically important search strategy, this fact can be ignored. Another assumption is that there exists no overlap of configurations between MPI-parallelized processes. This is not true, as discussed in Chapter 5. Rather, it provides a lower bound for the actual number of configurations per rank. Let n denote the number of concurrent AutoPas process and m the total number of valid configurations. Lastly, the model assumes that in a parallel setting, each tuning step takes as long as the slowest participating configuration. This is realistic, as the processes are synchronized at each tuning step. Firstly, a special case is considered: full-search.

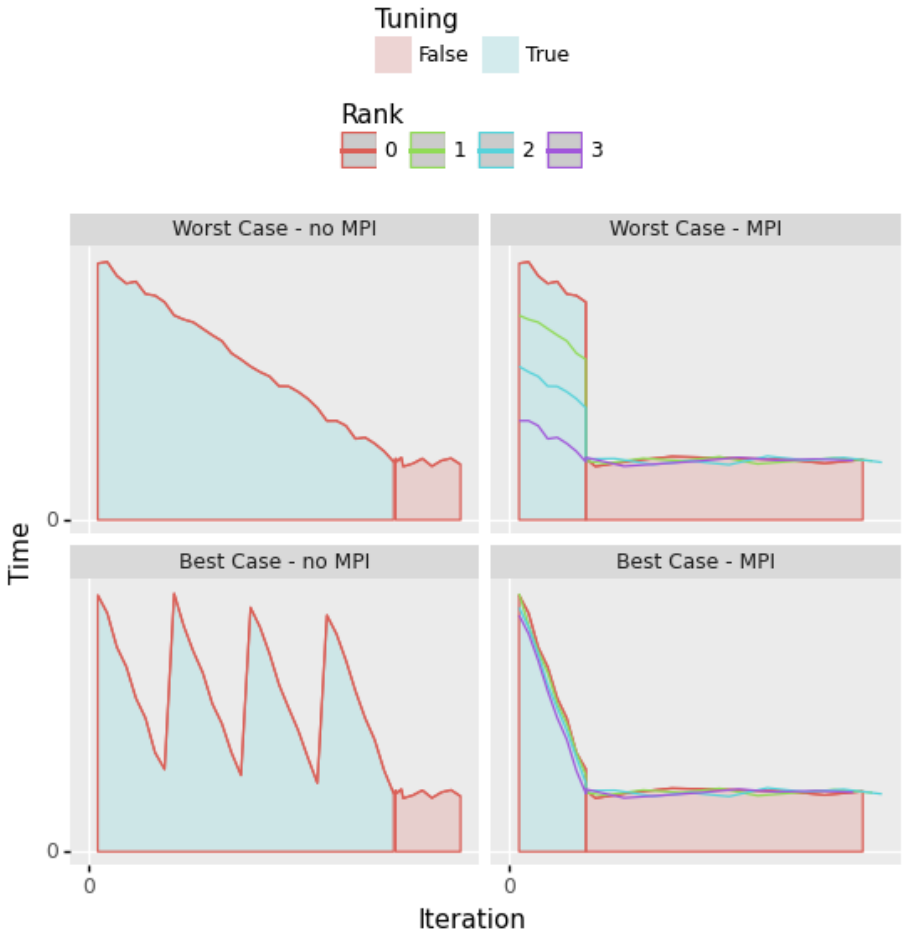


Figure 7.1.: Comparison of Worst-Case and Best-Case Scenarios for an Idealized Tuning Phase with Full-Search.
The "no MPI" sub-plots show a sequential execution of all configurations. The others depict a parallel execution with four processes. The lines for each rank in them show the time of a configuration without subsequent synchronization. The actual execution time is the maximum of those for all ranks, shown by the blue and red areas. All sub-figures use the same configurations with the same times, but ordered differently. The overlapping for the MPI cases is based on splitting the tuning phase into four equal intervals. This works, because the distribution algorithm traverses the search space in the same order as full-search, as described in Section 5.1.

This strategy is unique in that it definitely searches through all configurations and is thus guaranteed to find an optimal solution. This means that no speed-up can be expected outside of tuning phases, but rather that the tuning phases are shorter. For the case where MPI-communication is inactive, full-search has exactly m tuning steps. Therefore, the total time for tuning is the sum over all m tuning steps' times.

Figure 7.1 depicts four tuning phases using full-search after the stated model. As can be seen, the total tuning-time depends strongly on the order of configurations. While the number of tuning steps is the same in both scenarios, the sum of their measured times differs strongly. The worst-case scenario is one where $n-1$ fast configurations run concurrently with one slow configuration for each iteration, as depicted in the "Worst Case" sub-plots. This would result in a tuning-time of the sum of the slowest $\lceil \frac{m}{n} \rceil$ configurations. The best-case tuning-time results from n similarly fast configurations being concurrent per tuning step, as depicted in the "Best Case" sub-plots. To analyze the resulting tuning time, one can imagine an array of all configurations, decreasingly sorted by their speed. The result is the sum of the times of every n-th configuration, starting with the first. The mean of tuning step times, in this case, can be interpreted as an upper bound for the mean of all m times.

Let M be the total number of iterations (tuning steps and non-tuning iterations). Also let $t_{fastest}$ be the time of the fastest configuration, t_{mean} the mean time of all configurations, and $t_{slowest}$ the mean time of the slowest $\lceil \frac{m}{n} \rceil$ configurations. In a realistic scenario, a configuration would be used for several iterations. To reflect this, a constant c is defined for the number of iterations per configuration. Therefore, let $M_{parallel} = \lceil \frac{m}{n} \rceil \cdot c$ be the number of tuning iterations for the parallel case. Similarly, $M_{sequential} = m \cdot c$ corresponds to the number of tuning iterations in the sequential case. The predicted speed-up, in this case, is $\frac{M_{sequential} \cdot t_{mean} + (M - M_{sequential}) \cdot t_{fastest}}{M_{parallel} \cdot t_{slowest} + (M - M_{parallel}) \cdot t_{fastest}}$ in the worst-case.

Example

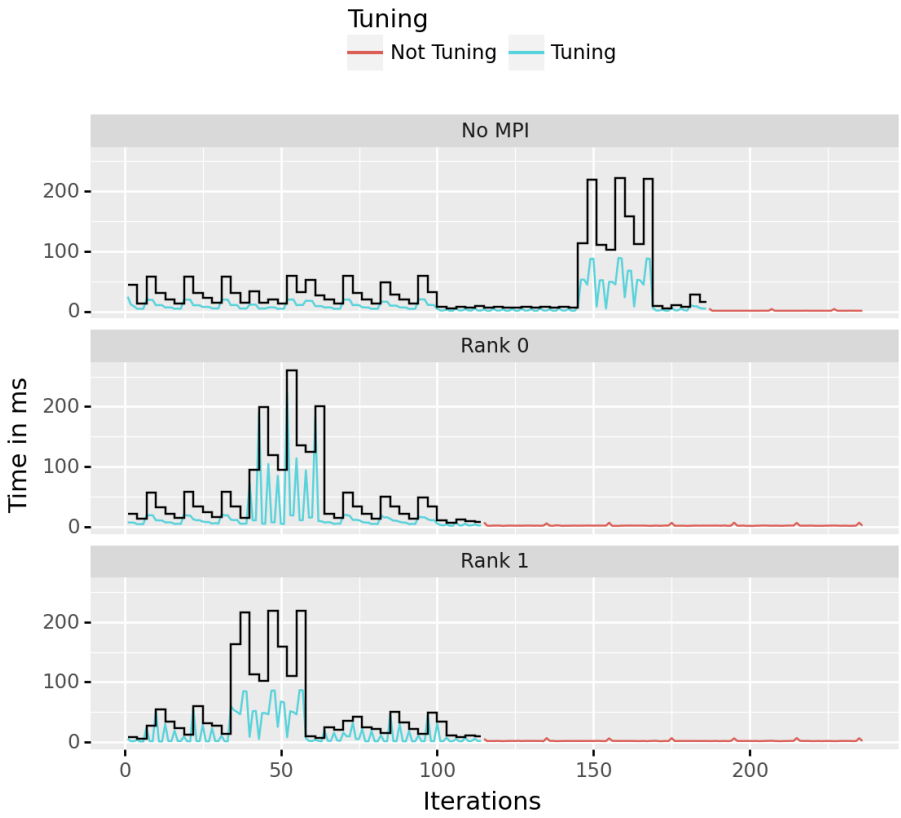
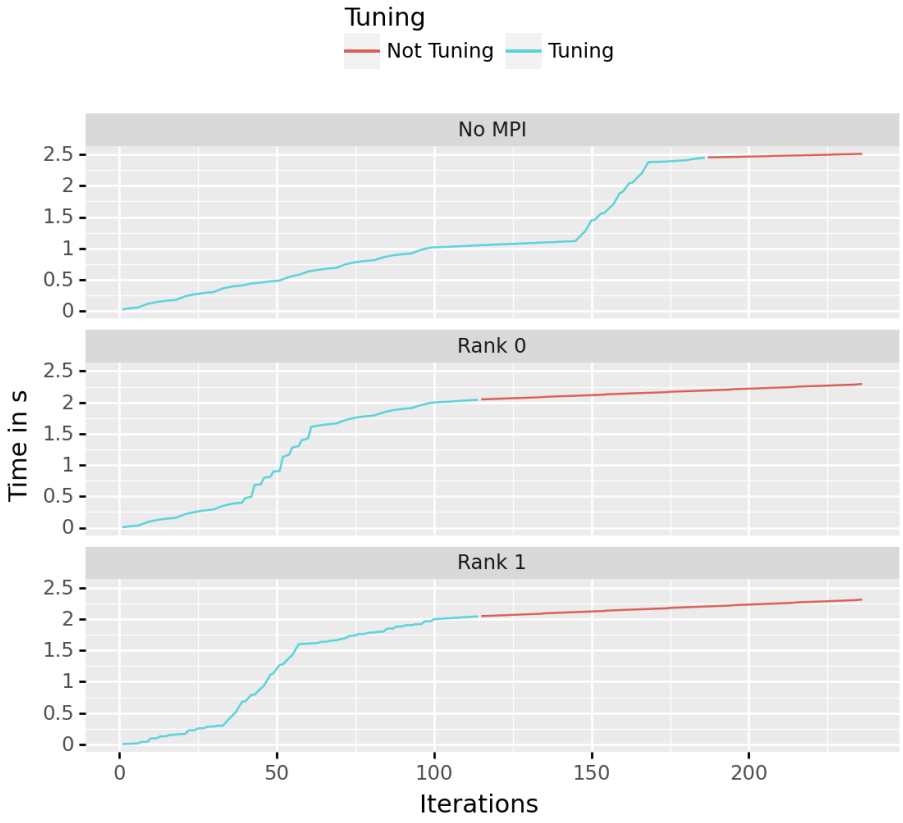


Figure 7.2.: Example Tuning Iterations Using the Default Values for MD-Flexible (Section 7.2).
The tuning strategy is full-search. The upper sub-plot shows a sequential execution, while the lower two show both ranks of a single parallel execution. The red and blue lines show the measured time for each iteration including synchronization. The black, stepped plot shows the time per tuning step (three iterations), i.e. per configuration.



.

Figure 7.3.: Example Tuning Iterations with Accumulated Times.
The data is taken from the same executions as in Figure 7.2. The right-most y-value corresponds to the total execution time of the simulation without initialization.

Figure 7.2 shows an example of one tuning phase, followed by some non-tuning iterations. As can be seen, the graphs for the parallel case roughly show the maximum of overlapping the left and right half of the serial case. Differences arise from the overlap of configurations between the ranks. Another factor for differences is the synchronization, which causes the spikes in the blue portion of the parallel graphs. Synchronization only happens when changing to a new configuration. Thus, a spike appears after one configuration has been significantly slower than the corresponding configuration of the other rank. For this reason, the stepped line appears to fall one configuration behind for whichever rank is faster than the other at a given tuning step. Figure 7.3 shows the same scenarios, but with the times accumulating from left to right.

Since in this example, all the most time-intensive configurations are in the search space of rank one, we expect a speed-up close to the worst-case scenario from above. Here, n is two and m is 62[1]. The slowest $\lceil \frac{m}{n} \rceil = 31$ configurations took approximately $23.0ms = t_{slowest}$ in the

[1]Technically there are more configurations in the search space, but not all are applicable in this example.

mean. AutoPas had every configuration run for three iterations, so $c = 3$. Lastly, $t_{fastest}$ has been measured to about $1.2ms$ in the mean, and t_{mean} to approximately $13.2ms$. Using the formula from above results in an estimated minimal speed-up of $\frac{186 \cdot 13.2ms + 50 \cdot 1.2ms}{93 \cdot 23.0ms + 143 \cdot 1.2ms} \approx 1.065$ or 6.5%. This does not take any overlap into account. The measured total times for *No MPI*, *Rank 0*, and *Rank 1* approximate 2.51, 2.29, and 2.31 respectively. The actual speed-up thus was $\frac{2.51}{2.31} \approx 1.087$ or 8.7%. This corresponds to the fact that the scenario was close to, but not entirely the worst-case[2].

If the strategy is not full search, the tuning phases are usually significantly shorter, at the cost of optimality. Thus, the expected speed-up in these cases is even more difficult to quantify, as it depends on the specifics of the strategy. For some, the tuning phase is independent of the search space size (e.g. Bayesian-search, random-search). In those cases, the parallelized tuning phases could even be assumed to be longer than the non-parallelized ones, because the execution time for a tuning step is the maximum of all of its ranks and the number of tuning steps is the same. However, some of these strategies provide options for the user to manually shorten the tuning phases. The speed-up then is determined by the balance of short tuning phases and sufficiently accurate results. When simulating with a tuning phase designed for single-process execution, enabling parallelization could therefore even result in a slow-down. In conclusion, the biggest speed-up is expected for search strategies that either benefit from smaller search spaces (e.g. full-search) or which have a low chance for optimality (e.g. random-search and Bayesian-search with short tuning phases).

[2]The estimated best-case scenario speed-up from this data is 1.74 or 74%.

7.2. Method

Testing was done with two setups (Table A.1) on two different machines (Appendix B). All tests essentially compare the execution time of certain simulations with and without the MPI-parallelization. Tests were done with 2, 16, and 32 concurrent processes. Dual-process tests were executed on a Notebook, while the other tests were run on the CoolMUC-2 cluster segment of the Leibniz Rechenzentrum (LRZ), both described in Appendix B. Another distinction is the use of OpenMP. The two-process tests used no OpenMP-parallelization, whereas the 16- and 32-process tests used 7 OpenMP-threads. This was mainly due to hardware limitations of the Notebook and to provide a more realistic use-case for the 32-process tests. The outcome of the measurements stays unaffected by this, as no comparisons are made between the run-times of setups with different numbers of processes. All tuning strategies were measured, except for Bayesian-cluster-search, as it was dysfunctional at the time of writing. Measurements have been repeated at least four times and up to tens of times each.

AutoPas provides two example simulations: sph (two versions: one with MPI, one without) and md-flexible. Although sph-mpi is the only example that uses MPI to communicate particle information between processes, md-flexible was chosen for the tests. This is because it is far more easily configurable. While sph-mpi might give a more realistic use case, both are equivalent for time measuring purposes. The only assumption this thesis makes about a simulation is that it is homogeneous. Since the md-flexible simulations for all processes are exact copies of each other, this assumption is necessarily met for md-flexible. Sph-mpi on other hand has the potential to become at least slightly heterogeneous, depending on the particle distribution and run-time. By default, md-flexible spawns particles in a grid pattern across the entire domain.

The two setups mentioned before are md-flex-100Phases and md-flex-100000Iterations. They both represent one of two extremes with respect to the ratio of tuning iterations compared to non-tuning iterations. Since md-flex-100Phases defines a tuning interval of 200 iterations and full-search with the default options tunes for 186 iterations, almost 50% of iterations can be spent tuning with this strategy. On the other hand, md-flex-100000Iterations only has a single tuning phase initially. After that, it runs for almost 100000 iterations without changing configuration. This means that the total execution time for the former setup is strongly influenced by how fast tuning can be finished. For the latter, it is dominated by how optimal the found solution is.

7.3. Results

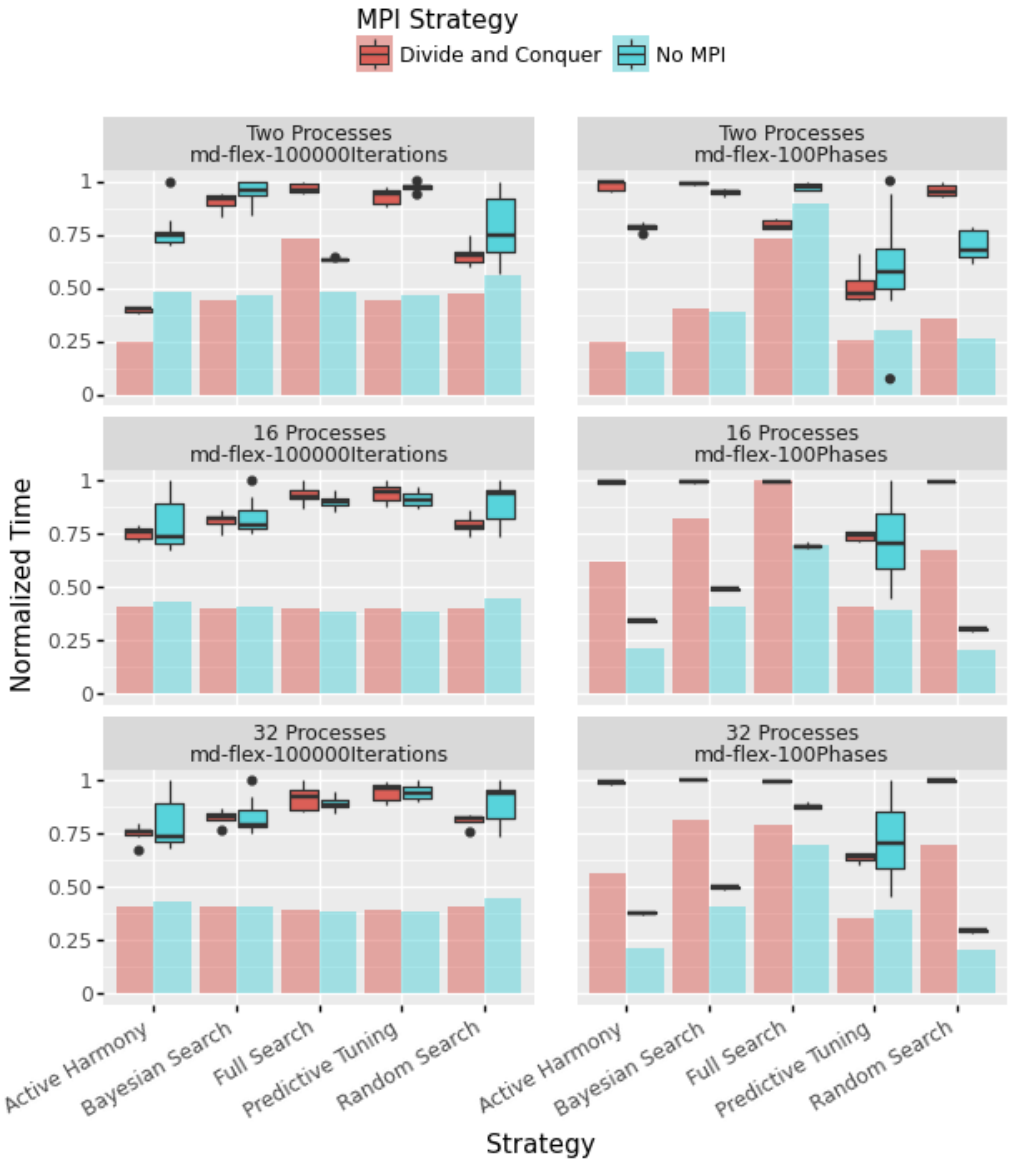


Figure 7.4.: Time Measurements for MD-Flexible Simulations with two Different Setups. The red boxes show the measurements for the given processor count, tuning strategy and setup with MPI-parallelization. The blue boxes show the same without MPI. Since md-flexible does not use MPI either, the blue boxes are the same for 16 and 32 processes. The blue boxes for two processes differ because they were measured on the notebook rather than the Linux-cluster. Dots show outliers. The values for the boxes have been normalized to 1 within their respective column. The bar plots show the means of those same values, but normalized to 1 across all sub-figures.

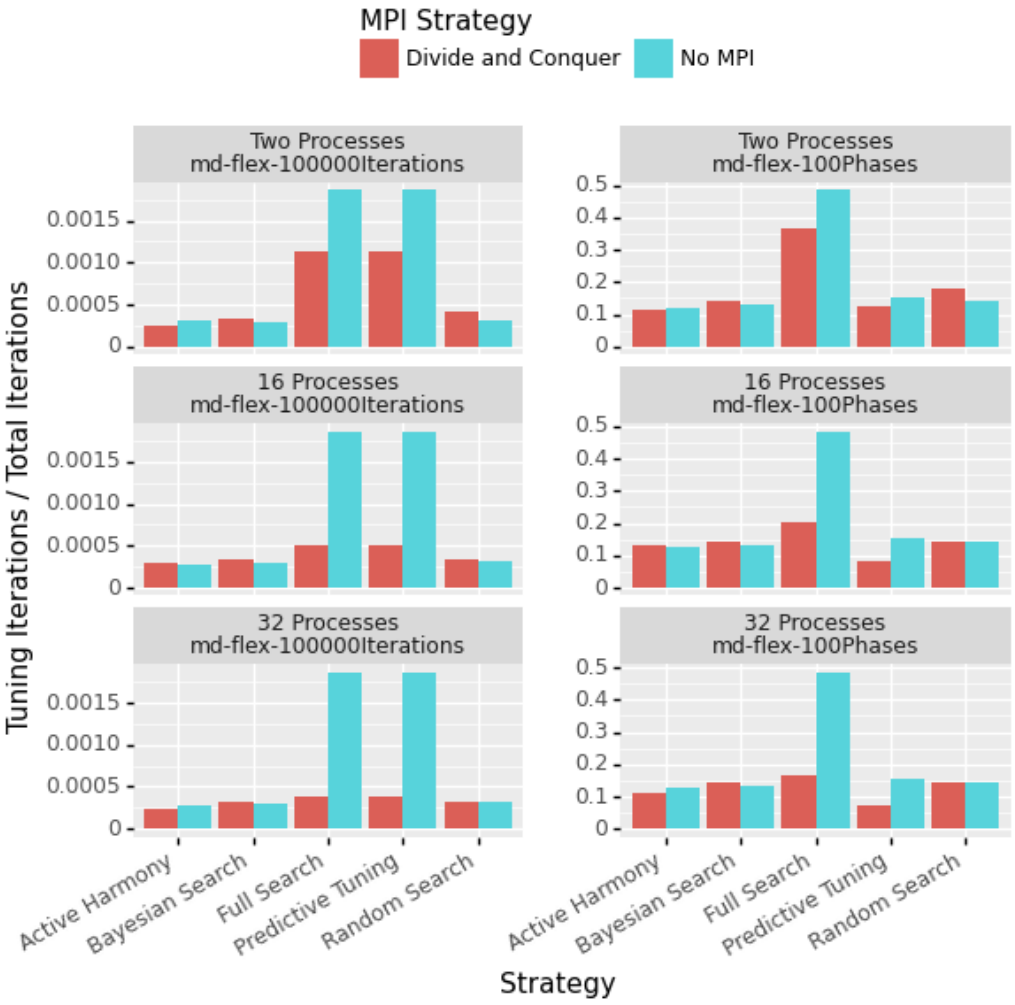


Figure 7.5.: Ratio of the Number of Tuning Iterations to Total Iterations for several scenarios.

As shown in Figure 7.4, the divide-and-conquer MPI-strategy does not ubiquitously result in a speed-up. In many cases, it can even stagger the performance. This is partially expected, as described in Section 7.1. An unexpected result is that full-search for md-flex-100Phases slows down for high numbers of parallel processes. Similarly, predictive-tuning hardly benefits from parallelization. Figure 7.5 shows that those two search strategies do indeed tune with significantly fewer tuning iterations when parallelized. Section 7.1 concluded that this would result in overall shorter execution times, which is not the case.

To explain the poor performance for full-search and predictive-tuning, a new consideration was made. As could be seen in Figure 7.2, for the default md-flexible simulation, all of the slowest configurations occur in a fairly consecutive block. This block consists of those configurations which share the verletClusterLists-container. For cases with many processes, when distributing the configurations initially (Section 5.1), many of these will be assigned

to several ranks due to overlap. Since usually no two ranks share the exact same search space, the slowest configurations are not used simultaneously. To reuse the terms from Section 7.1, this means that the worst-case scenario changes due to the overlap. Instead of a tuning phase taking $\lceil \frac{m}{n} \rceil$ tuning steps with the $\lceil \frac{m}{n} \rceil$ slowest configurations, it takes possibly more tuning steps with a slow subset of those slowest configurations. When a small number of configurations are significantly slower than all others, as in the default md-flexible simulation, the performance loss can then exceed the gain from a smaller search space. The idea that the loss in performance is explained by this overlap is supported by the fact that a slowdown for full-search using md-flex-100Phases can only be seen for higher process-counts in Figure 7.4. Fewer processes result in smaller overlaps.

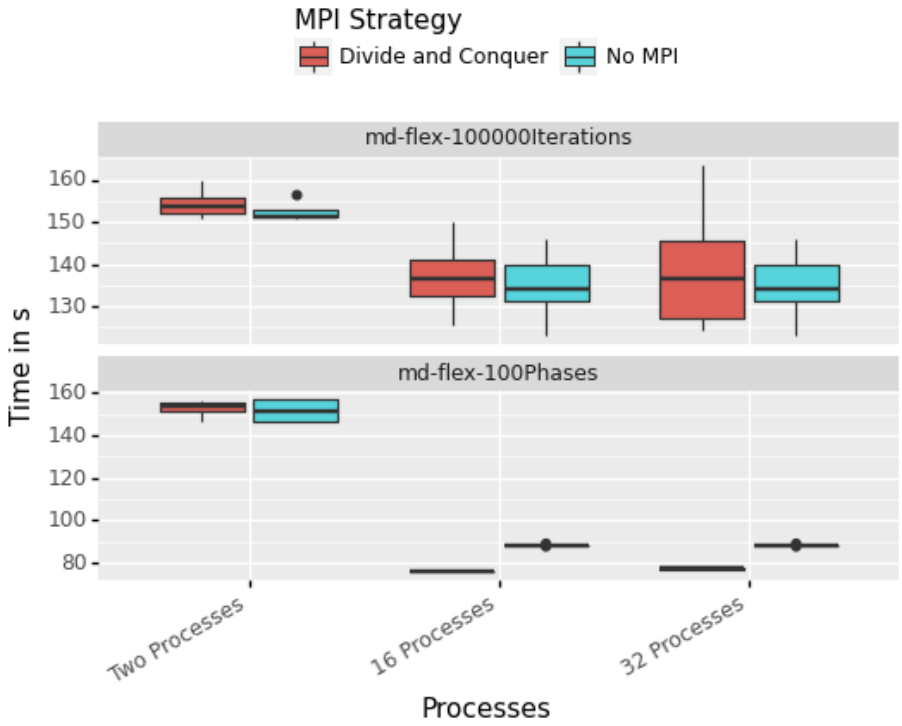


Figure 7.6.: Full-search Measured without the VerletClusterLists-Container.

To further test this, full-search was measured without verletClusterLists. The expectation was that performance of the MPI-case would improve relative to the non-MPI case. Figure 7.6 shows the results. For md-flex-100Phases, there now is a small performance gain when using MPI as opposed to a performance loss that was measured with the verletClusterLists-container. This optimization is of course not feasible for real operation. Knowing which configurations and options to remove beforehand would diminish the reason to use auto-tuning in the first place. The decrease in execution time does still not scale with the decrease in the number of tuning iterations (Figure 7.5). This is partially due to non-tuning iterations, which remain unchanged with parallelization. The other reason is that waiting times still have an impact and there is still a difference between the fastest and slowest configurations.

Figure 7.5 also reveals that the number of tuning iterations is not always reduced when employing MPI with sufficiently many processes, as mentioned in Section 7.1. Rather, AH, bayesian-search, and random-search tune for a number of tuning steps independent of search space size. Bayesian-search and random-search have a variable that directly controls the number of tested configurations. For AH, the length of a tuning phase depends on how quickly the search converges, which appears to stay fairly constant for varying search space sizes. For those strategies, where the length of the tuning phases is independent of the search space size, this explains the loss in performance. The reason for longer total times (as in the case for Bayesian-search with 32 processes using md-flex-100Phases for example) is that some processes may have only slow configurations. This means that the number of tuning-iterations stays roughly the same but all of them are spent with slow iterations because the fast processes have to wait for the slow ones. Therefore, performance could be improved for those strategies, by manually shortening their tuning phases.

To test the above hypothesis, Bayesian-search was tested with a short tuning phase. Bayesian-search was chosen because its tuning phases are easier to control than for AH and because it is more optimized than random-search[3]. Figure 7.7 shows that for md-flex-100Phases there is a significant performance increase between a non-parallelized execution with 30 iterations per tuning phase and a parallelized execution with nine per tuning phase. For md-flex-100000Iterations, no speed-up was measured, which is expected. In the case of using only two parallel processes, the tuning phases for the MPI-case were too short to reliably find an optimal solution, which is why they are slower than the non-MPI-case. For more processes, the execution time is dominated by non-tuning iterations. Since both the MPI and non-MPI version could reliably find a solution close to the optimum, no difference in speed should be expected.

[3] Although, for such small search spaces there would not have been much difference between Bayesian-search and random-search.

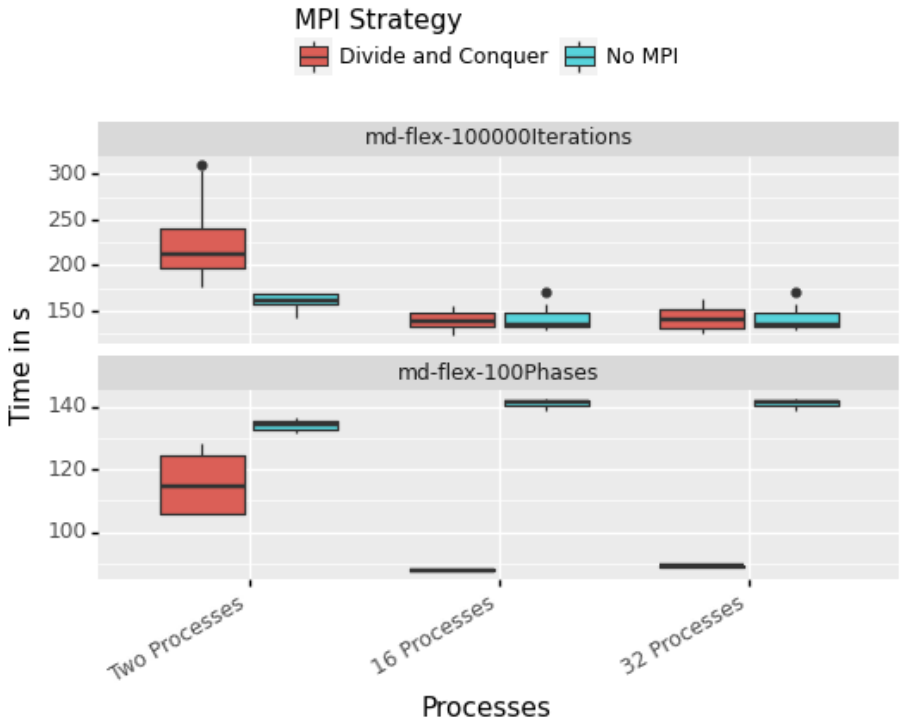


Figure 7.7.: Measured Times for Bayesian-Search with Reduced Search Spaces in the MPI-Case.

The executions without MPI used the standard max-evidence value of ten. The executions with MPI used two instead, meaning that only two configurations had to be tested per rank for each tuning phase. This usually results tuning phases of three tuning steps, i.e. nine iterations. For a max-evidence value of ten, a tuning phase has ten tuning steps, i.e. 30 iterations. The reason one more tuning step is used than the max-evidence value for the MPI-case is that the parallelization adds another tuning step at the end of each phase. Again, the case for two processes was measured on the notebook in Appendix B and the others on the Linux-cluster.

Part IV.

Conclusion & Future Work

8. Conclusion

As initially planned, this thesis implements a wrapper search strategy, which uses MPI to connect several local tuning sessions. The search spaces of all participating processes get reduced and the results are compared to find the optimal configuration for a homogeneous simulation domain. Unlike expected, the reduction of all search spaces does not uniformly result in faster execution. There are two primary factors for this:

1. Many tuning strategies tune for a set number of iterations independently of search space size. In these cases, it is usually possible to force shorter tuning phases in order to fully benefit from the MPI-parallelization. However, as this does not happen automatically, the user is still required to have a rough understanding of how many configurations there would be per rank and how long a tuning phase should take accordingly.
2. Due to overlapping configurations and the orders in which configurations are used, much time is spent waiting for slow processes. As AutoPas is currently implemented, there is no clear way to solve these issues. One main reason for this is the implicit representation of configurations.

It is possible to achieve very sizeable speed-ups using the implemented parallelization. However, this is highly situational and it is often not clear whether using it would improve or worsen performance. Some questions to ask are:

- Which tuning strategy is being used?
- How many parallel processes are involved?
- How many valid configurations are in the search space?
- How much time is spent tuning?
- How slow are the slowest configurations compared to all others?

Unfortunately, some of these questions are not trivial to answer and it is not obvious whether to use MPI in any given scenario. Additionally, the later stages of this thesis revealed that most search strategies are poorly equipped to handle small search spaces. This further complicates matters, as the integration of MPI could not be done seamlessly.

In hindsight, a different approach to this thesis might have been promising: Instead of writing a wrapper for the already existing tuning strategies, a new, independent strategy could have been implemented. This could have had the following benefits:

1. Overlapping configurations between ranks could be fully avoided by storing the explicit representation of configurations.

2. The order of ranks could be controlled freely. Normally, tuning strategies store sets of either options or configurations. With an independent strategy, it could be possible to store an ordered list of configurations.
3. Instead of having to work around the inner functioning of a generic internal strategy, it could be directly aimed at solving heterogeneous domains. Specifically, it could exchange useful information about the performance of configuration during tuning, not only at the end of a tuning phase. The approach might work similarly to the current Bayesian-search or the Nelder-Mead simplex method used in AH but fully designed for parallelism and heterogeneous MD-simulations.

An independent MPI-strategy would not be free of draw-backs. A clear one is that different tuning strategies could only be provided by implementing all of them separately. Additionally, the strategy could not benefit from any developments made for serial tuning. For instance, if a novel search strategy was implemented that would lend itself well to parallelization, doing so could not be done easily.

9. Future Work

As was argued before, a smarter distribution of configurations should be the first step to improve performance. Currently, options are ordered by the difficulty associated with changing them. This approach works fine for full-search in a non-parallelized setting, as the total simulation time is not affected by the configuration's order. With this thesis' MPI-parallelization, however, it could be useful to sort options by the impact on simulation time associated with them. Alternatively, one could sort the configurations themselves, allowing for more control over their order. This could lead to a speed-up closer to the best-case scenario, as described in Chapter 5. Whether and in which cases this might be beneficial could be an area of further research.

A second consideration with respect to the distribution is the overlap created by building the implicit representation. In most cases, this will not have drastic consequences. Figure 7.5 shows an example in which the distribution works well for reducing the search space. However, as the options are grouped within each other when traversing the search space, the worst-case behavior is severe. Whenever a container between two consecutive configurations changes, all other options can change as well. This means that from two configurations in the explicit representation, two different values for each of the six options could be generated. This results in $2^6 = 64$ configurations in the implicit representation. Although many of these would usually be invalid, this should be avoided. A possible alternative to the current order used in the distribution scheme might be one that changes only two or three options between two consecutive configurations.

Another important area for improvement is the fallback behavior. Intuitively one might hope that increasing the number of processes for a program would improve execution-time. As demonstrated, this is not necessarily the case. When increasing the number of ranks above a certain value, some ranks will end up without valid configurations. In these cases, the process will use the first (according to the ordering mentioned above) valid configuration possible. As this might be a configuration that has already been measured to be very slow, this strategy can slow the tuning phases down without providing any new information to the tuning process. An alternative solution could be to communicate already tested, fast configurations to those processes during tuning. A second option is to transfer untested configurations from another process to the one in question. Both approaches could shorten tuning phases, assuming a low overhead for the communication. Unfortunately, the second option is not easily compatible with many tuning strategies (e.g. AH).

An interesting extension of the current system would be one that can work in heterogeneous domains. However, a simple time measuring approach would not work for this case. Thus, a first important step is to find a good model to derive information about one region of the domain with a characteristically different region. Alternatively, the domain might be split

into several, roughly homogeneous sub-problems. Each would then be tuned independently of the others. The latter option could be implemented using different MPI-communicators, but requires expensive pre-computation and good heuristics.

The independent MPI-tuning-strategy outlined in Chapter 8 could also be considered for future work. Instead of replacing this thesis' work, it could also be added as an alternative. Using the current code, this would a candidate for a third MPI-strategy option. Then, the MPI wrapper could, with some of the optimizations outlined above, function for parallelizing existing strategies for homogeneous domains, while the independent MPI strategy could solve heterogeneous cases.

As has been shown in Section 7.3, whether the implemented MPI-parallelization results in a speed-up or slow-down differs between scenarios. An opportunity to take the auto-tuning idea a step further would be to turn this parallelization and any future parallelization schemes into a tune-able parameter. This would without a doubt mean much refactoring work in AutoPas. Unlike all other tune-able parameters, it could not be tested independently of other configurations. Thus, the tuning-process might be divided into several layers. The parallelization might then be tested concurrently with the configurations, as it does not affect force calculation times.

Part V.

Appendix

A. Testing setups

Setup	command

md-flex-100Phases:
 	
Many tuning phases with short times between them. Almost 50% of iterations are tuning iterations when using full-search without MPI. As the other values were left at the default, they can be found in Listing A.1.
	`./md-flexible`
	`--tuning-phases 100`
	`--tuning-interval 200`
	`--mpi-strategy <mpi>`
	`--tuning-strategy <tuning>`

md-flex-100000Iterations:
 	
A single tuning phase followed by many iterations without tuning. As the other values were left at the default, they can be found in Listing A.1.
	`./md-flexible`
	`--iterations 100000`
	`--tuning-interval 1000000`
	`--mpi-strategy <mpi>`
	`--tuning-strategy <tuning>`

Table A.1.: Description of Testing Setups

```
container                    : [LinkedCells,
                                VerletLists,
                                VerletListsCells,
                                VerletClusterLists,
                                VarVerletListsAsBuild,
                                VerletClusterCells]
verlet-rebuild-frequency     : 20
verlet-skin-radius           : 0.2
verlet-cluster-size          : 4
selector-strategy            : Fastest-Absolute-Value
data-layout                  : [AoS, SoA]
traversal                    : [c08, sliced, c18, c01,
                                directSum, verlet-sliced,
                                verlet-c18, verlet-c01,
                                cuda-c01, verlet-lists,
                                c01-combined-SoA,
                                verlet-clusters, c04,
                                var-verlet-lists-as-build,
                                verlet-clusters-coloring,
                                c04SoA, verlet-cluster-cells,
```

```
21                                        verlet-clusters-static,
22                                        balanced-sliced,
23                                        balanced-sliced-verlet,
24                                        c04HCP]
25 tuning-samples                    :    3
26 tuning-max-evidence               :    10
27 functor                           :    Lennard-Jones (12-6)
28 newton3                           :    [disabled, enabled]
29 cutoff                            :    2
30 box-min                           :    [-0.56125, -0.56125, -0.56125]
31 box-max                           :    [10.6638, 10.6638, 10.6638]
32 cell-size                         :    [1]
33 deltaT                            :    0.001
34 periodic-boundaries               :    true
35 Objects:
36   CubeGrid:
37     0:
38       particles-per-dimension     :    [10, 10, 10]
39       particle-spacing            :    1.1225
40       bottomLeftCorner            :    [0, 0, 0]
41       velocity                    :    [0, 0, 0]
42       particle-type               :    0
43       particle-epsilon            :    1
44       particle-sigma              :    1
45       particle-mass               :    1
```

Listing A.1: md-flexible default values

B. Hardware Specifications

Model	Lenovo ideapad 320
Cpu model	Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz
Cpu cores	2
Threads per CPU core	2
OS	Manjaro Linux 20.0.3 Lysia
MPI	MPICH 3.3.2

Table B.1.: Hardware Specifications of the Notebook Used for Testing

Number of nodes	812
Cores per node	28
Threads per CPU core	2
Core frequency	2.6 GHz
OS	SLES15 SP1 Linux
MPI	Intel MPI 2019

Table B.2.: Hardware specifications of the CoolMUC-2 cluster segment at LRZ

List of Figures

List of Tables

Bibliography

[1] *Active Harmony User's Guide*. accessed on 24.08.2020. 2016. URL: https://www.dyninst.org/manuals/harmony.

[2] H.M. Aktulga, J.C. Fogarty, S.A. Pandit, and A.Y. Grama. "Parallel reactive molecular dynamics: Numerical methods and algorithmic techniques". In: *Parallel Computing* 38.4 (2012), pp. 245–259. ISSN: 0167-8191. DOI: https://doi.org/10.1016/j.parco.2011.08.005. URL: http://www.sciencedirect.com/science/article/pii/S0167819111001074.

[3] Brooks. "No Silver Bullet Essence and Accidents of Software Engineering". In: *Computer* 20.4 (1987), pp. 10–19. DOI: 10.1109/MC.1987.1663532.

[4] R. Ewald. *Automatic Algorithm Selection for Complex Simulation Problems*. 1st ed. Vieweg+Teubner Verlag, 2012. DOI: 10.1007/978-3-8348-8151-9.

[5] F. A. Gratl, S. Seckler, N. Tchipev, H. Bungartz, and P. Neumann. "AutoPas: Auto-Tuning for Particle Simulations". In: *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2019, pp. 748–757.

[6] W. Gropp, M. Schulz, R. Rabenseifner, R. L. Graham, J. M. Squyres, D. Holmes, G. Bosilca, T. Hoefler, P. Balaji, J. Hammond, D. Solt, Q. Koziol, K. Mohror, and R. Thakur. *MPI: A Message-Passing Interface Standard*. 2015. URL: https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf.

[7] *Guides — MPICH*. accessed on 09.08.2020. URL: https://www.mpich.org/documentation/guides/.

[8] J. Nguyen. "AutoTuning using Bayesian Statistics in AutoPas". IDP-Arbeit. Technical University of Munich, Feb. 2020.

[9] *Open MPI v4.0.4 documentation*. accessed on 09.08.2020. URL: https://www.open-mpi.org/doc/current/.

[10] R. Prat, L. Colombet, and R. Namyst. "Combining Task-Based Parallelism and Adaptive Mesh Refinement Techniques in Molecular Dynamics Simulations". In: *Proceedings of the 47th International Conference on Parallel Processing*. ICPP 2018. Eugene, OR, USA: Association for Computing Machinery, 2018. ISBN: 9781450365109. DOI: 10.1145/3225058.3225085. URL: https://doi-org.eaccess.ub.tum.de/10.1145/3225058.3225085.

[11] D. E. Shaw, R. O. Dror, J. K. Salmon, J. P. Grossman, K. M. Mackenzie, J. A. Bank, C. Young, M. M. Deneroff, B. Batson, K. J. Bowers, E. Chow, M. P. Eastwood, D. J. Ierardi, J. L. Klepeis, J. S. Kuskin, R. H. Larson, K. Lindorff-Larsen, P. Maragakis, M. A. Moraes, S. Piana, Y. Shan, and B. Towles. "Millisecond-Scale Molecular Dynamics Simulations on Anton". In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. SC '09. Portland, Oregon: Association for

Computing Machinery, 2009. ISBN: 9781605587448. DOI: 10.1145/1654059.1654099. URL: https://doi-org.eaccess.ub.tum.de/10.1145/1654059.1654099.

[12] D. E. Shaw, J. P. Grossman, J. A. Bank, B. Batson, J. A. Butts, J. C. Chao, M. M. Deneroff, R. O. Dror, A. Even, C. H. Fenton, A. Forte, J. Gagliardo, G. Gill, B. Greskamp, R. C. Ho, D. Ierardi, L. Iserovich, J. Kuskin, R. H. Larson, T. Layman, L. Lee, A. K. Lerer, C. Li, D. Killebrew, K. M. Mackenzie, S. Y. Mok, M. A. Moraes, R. Mueller, L. J. Nociolo, J. L. Peticolas, T. Quan, D. Ramot, J. K. Salmon, D. P. Scarpazza, U. B. Schafer, N. Siddique, C. W. Snyder, J. Spengler, P. T. P. Tang, M. Theobald, H. Toma, B. Towles, B. Vitale, S. C. Wang, and C. Young. "Anton 2: Raising the Bar for Performance and Programmability in a Special-Purpose Molecular Dynamics Supercomputer". In: *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis* (2014), pp. 41–53.

[13] S. Song and J. K. Hollingsworth. "Computation–communication overlap and parameter auto-tuning for scalable parallel 3-D FFT". In: *Journal of Computational Science* 14 (2016). The Route to Exascale: Novel Mathematical Methods, Scalable Algorithms and Computational Science Skills, pp. 38–50. ISSN: 1877-7503. DOI: https://doi.org/10.1016/j.jocs.2015.12.001. URL: http://www.sciencedirect.com/science/article/pii/S187775031530048X.

[14] S. Szabo. "Speeding up exact cover algorithms by preconditioning and parallel computing". In: June 2011.

[15] D. Varagnolo, G. Pillonetto, and L. Schenato. "Auto-tuning procedures for distributed nonparametric regression algorithms". In: July 2015, pp. 640–647. DOI: 10.1109/ECC.2015.7330614.

[16] *xSDK Community Package Policies*. accessed on 24.08.2020. 2019. URL: https://figshare.com/articles/xSDK_Community_Package_Policies/4495136.

[17] H. Yu, D. Zhang, and L. Rauchwerger. "An adaptive algorithm selection framework". In: *Proceedings. 13th International Conference on Parallel Architecture and Compilation Techniques, 2004. PACT 2004.* 2004, pp. 278–289.