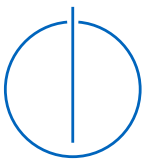
SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Optimizing Algorithm Selection in AutoPas with Decision Trees and Random Forests

Abdulkadir Pazar



SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Optimizing Algorithm Selection in AutoPas with Decision Trees and Random Forests

Optimizing Algorithm Selection in AutoPas with Decision Trees and Random Forests

Author:	Abdulkadir Pazar
Examiner:	Samuel James Newcome, M.Sc.
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Submission Date:	22.11.2024

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

Munich, 22.11.2024 Abdulkadir Pazar

Acknowledgments

I would like to express my deepest thanks to my advisor, Sam, for his guidance and insightful feedback throughout this journey. His expertise in the project itself has been fundamental in shaping the course of my work, and I greatly valued the insights I gathered from our bi-weekly discussions.

I also sincerely thank the Chair of Scientific Computing and Univ.-Prof. Dr. Hans-Joachim Bungartz for the opportunity to undertake this project. I am also grateful to the Leibniz Supercomputing Centre (LRZ) and Helmut-Schmidt-Universität Supercomputing Centre (HSU-HH RZ) for funding this thesis by providing the necessary computational resources for this research.

Finally, I am deeply thankful to my family and friends. My family's support through this journey allowed me to dedicate myself fully to my studies. At the same time, I would like to thank my friends who provided inspiration and opportunities for socialization in the form of much-needed breaks.

Abstract

AutoPas[Gra+22] is a high-performance, auto-tuned particle simulation software library for many-body particle systems, capable of dynamically switching between algorithms and data structures to achieve optimal performance during the simulation. This thesis explores a decision tree-based tuning strategy for AutoPas, allowing users to guide the tuning process by specifying custom decision tree or random forest models, which can be used to efficiently conclude the tuning phase and determine the best configuration from the search space. Efficient tuning strategies are crucial, as they allow for discarding poor configurations, thus enabling the simulation to perform most optimally.

We demonstrate that an approach where we train a decision tree or a random forest model based on data collected from already existing Full Search tuning strategy can significantly outperform existing tuning strategies on specific benchmarks by drastically reducing the tuning time up to $30\times$ and achieving a speedup in total computation time up to $1.45\times$ in certain runtime configurations.

The decision tree-based tuning strategy can drastically reduce the time spent during the tuning phases while achieving comparable tuning results; therefore, it can be an alternative candidate to the current tuning strategies.

Contents

1. Introduction

With the continuous advancements in scientific computing, scientific simulations have become increasingly important as they enable the researchers to conduct "virtual experiments" to either complement or replace "real" experiments [YR10]. In scientific simulations, efficiency and precision are of utmost importance, especially when computational methods are required to analyze complex systems. One such case of scientific simulations is many-body particle simulations, with use cases ranging from modeling of traffic to biological and medical applications to implementation of fluid flow simulations [Bun+13]. For many-body particle simulations, the choice of algorithms and data structures plays a crucial role in achieving optimal performance, especially as the scale of such simulations grows. This choice heavily depends on the current state of the simulation and, as a result, can change during the runtime of the simulation. To meet the demands of this high-performance computing (HPC) task, we turned to developing a tuning strategy for AutoPas[Gra+22], which, based on the current configurations of the simulation, selects the most suitable algorithms and data structures by utilizing one of the provided tuning strategies.

This thesis explores the development of a novel tuning strategy for AutoPas, based on decision tree and random forest models, designed to streamline the tuning process. The proposed approach utilizes a data-driven approach to first train a model using the information collected from previously developed logging tools, then use these trained models to identify optimal configurations with minimal tuning time, therefore bypassing less efficient options in favor of selections better suited to the current simulation configurations. By training decision tree or random forest models on data collected from previously developed tuning strategies, this method allows the system to learn from prior tuning runs, reducing the need for exhaustive search as before and significantly shortening tuning time.

In this work, we demonstrate that decision tree-based tuning not only provides performance on par with other tuning strategies but also reduces the total computation time on specific benchmarks by optimizing the time spent during tuning phases. The result is a tuning strategy that reduces complex simulations' run times, thereby making efficient use of computational resources. By the conclusion of this thesis, we aim to show that decision tree and random forest-based tuning strategies offer a promising alternative to current methods.

2. Theoretical Background

Before discussing how the decision tree-based tuning strategy has been implemented and integrated into AutoPas, it is important to understand the core properties and purposes of Particle Dynamics simulations and what are the intrinsic problems that make them so complex. This will help us appreciate the importance of auto-tuning and developing tuning strategies that enable the simulations to be run with the optimal configurations.

2.1. Particle Dynamics

The purpose of particle dynamics simulations is to understand the behavior of particles over time in various environments. These particles could range from gas molecules in the air with microscopic interactions between them to space debris around Earth's atmosphere and their interactions based on gravitational forces. From the given example scenarios, it can be easily seen that particle dynamics simulations can be a very useful tool for researchers conducting research in environments where practical experiments are not feasible. Using these simulation tools, the researchers can gain previously unavailable insights.

2.1.1. Lennard-Jones Potential

When implemented naively, particle simulations are n-body simulations that are computationally very expensive since their complexity is quadratic in the number of particles. This can easily be seen since, in each time step, we need to calculate forces between pairs of particles. Due to the complex quantum mechanical nature of these interactions in the case of molecular dynamics, these interactions are modeled by potential functions. Because these potential functions are approximations, there are multiple different potential functions depending on different use cases. [Bre00]. One such potential that is often used in molecular dynamics simulations and that we also use is the Lennard-Jones potential [Len31], proposed by John Lennard-Jones and John Edwards in 1924. The Lennard-Jones 12-6 potential is described as:

$$U_{L,J} = \underbrace{4\epsilon \left(\frac{\sigma}{r_{ij}}\right)^{12}}_{\text{Van-der-Walls forces}} - \underbrace{4\epsilon \left(\frac{\sigma}{r_{ij}}\right)^{6}}_{\text{Pauli repulsion}} \tag{2.1}$$

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

where r_{ij} is the distance between two molecules and ϵ and σ are constants dependent on the physical properties of the particles which the force potential is to be calculated.

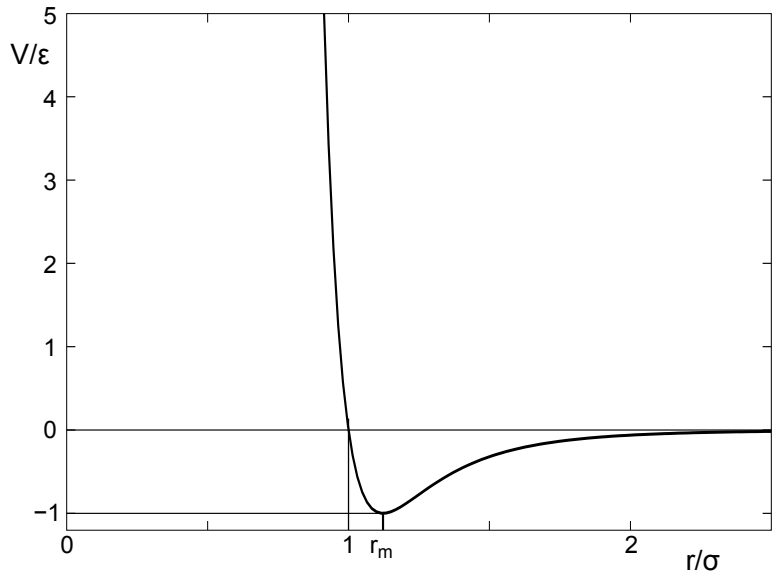


Figure 2.1.: Lennard-Jones Potential with respect to the intermolecular distance Source:[Com07]

2.1.2. Cutoff Radius

To reduce the simulation complexity, a cutoff radius r_c parameter is introduced. Since the force between the particles decreases as the distance between the particles, as seen from 2.1, long-distance interactions between particles can be omitted as their influence on the result is minimal. The range until we calculate the potential is called the cutoff radius. As such, only particles within a distance of smaller than this radius are considered in the calculation of forces. This still requires the simulation to calculate the distance between each particle pair. Still, we can avoid calculating the more expensive

Lennard-Jones potential between particle pairs with a pairwise distance larger than the cutoff radius.

The cutoff distance is a variable that can be configured by the user of the simulation software, in our case AutoPas, so that the users can increase the cutoff radius and achieve a more accurate simulation at the cost of increased computational time and energy.

2.1.3. Newton's Third Law

Another important fact that can be used to optimize the force calculations is Newton's third law of motion. This law states that for every force, a force with an equal magnitude but an opposing direction will be exerted on the other particle. For clarity, this means that for a force F_{ij}, exerted by the particle j on particle i, the equal and opposite force $F_{ji} = -F_{ij}$ is exerted on particle j. Using this law, the number of force calculations that need to be calculated for each time step can be halved. However, this introduces complexities, especially in multi-threaded environments. From now on, this will be referred to as `Newton3`.

2.2. Machine Learning

Since decision trees and random forests, which we will use for developing a new tuning strategy, have originally been developed in machine learning, we need to digress into machine learning concepts to understand them and apply them in our use cases.

Our task at hand is a *classification* [Kot07] task since we believe that a simulation's density and homogeneity parameters at a given time step are highly correlated to the optimal algorithm configuration. We can utilize ML classification methods to capture underlying correlations from a dataset of live information with the configuration values at that time as ground truth values to train a model that can infer configurations based on the live information gained from the information at that point in time.

In our special case, our task is a *multi-label* classification [TKV08] task since the configuration space is, at the time of writing this thesis, six-dimensional.

The process of learning the function $f : \mathcal{F} \mapsto Y$, where $\mathcal{F}$ is the feature space and Y is the set of possible labels or the target space, is often referred to as *training* a classifier. After training, the label of a data point $x \in \mathcal{F}$ can be retrieved by evaluating $f(x)$, a process referred to as forward-pass, inference, or prediction from now on.

2.2.1. Decision Trees

Decision trees can be described as hierarchically structured ensembles of binary decisions [Bre+84]. Since they are intuitive, scalable for very large datasets, and quick to train, they became very popular and are still widely used in numerous classification tasks. They work by partitioning the input using splits parallel to the axes to make sure the resulting subsets are as pure as possible. They try to minimize a given impurity metric, often the Gini impurity coefficient [Mur13] $I_G = \sum_{i=1}^{n} p_i(1 - p_i)$ of the subsets. As the decision trees directly partition the input space into distinct classes, they can be visualized as dividing the input space into different regions if the dimensionality of the input space allows it. One such example is in 2.2, with 2-dimensional input space divided into four classes. This recursive process results in a tree-like structure where each inner node represents a decision, and each leaf represents a class. Decision trees are known for their interpretability [SR20] as they produce a tree-like structure that allows users to understand the decision-making features easily. This feature makes them especially useful in areas where explainability of the model is necessary, as ML-based models generally are black-box models and, as such, not explainable by a clear set of rules.

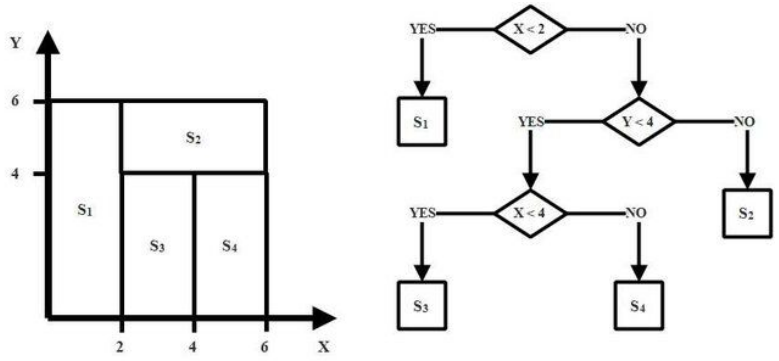


Figure 2.2.: A decision tree and the corresponding division of the space Source: [GSB16]

2.2.2. Random Forests

With the advent of ensemble learning [HTF09], constructing a collection of "weak" decision trees, namely random forests, was proposed [Ho95; Ho98]. Instead of aiming to optimize a single complex tree, we apply randomization during the learning process to create a committee of decision trees with little correlation. To achieve this, random forests use a technique called *bagging* or *bootstrap aggregation* to reduce the variance of the prediction function f. By taking a weighted average of the prediction of this

ensemble of different decision trees, random forests achieve greater generalization and, thereby, higher accuracy and robustness than single decision trees. Random forests have proven to be more robust for classification tasks and they have been shown to achieve higher accuracy and be resistant to over-fitting on various analyses on classification tasks [HTF09].

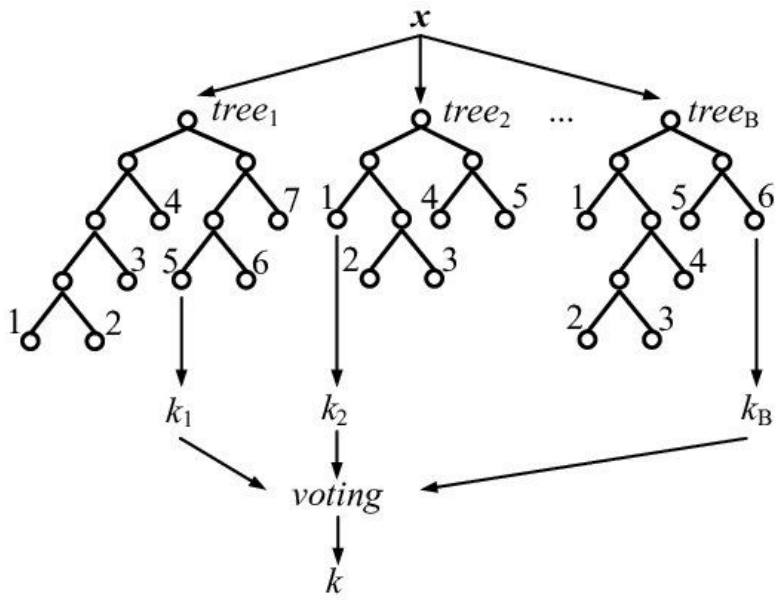


Figure 2.3.: A diagram of a random forest. Source: [Gel+14]

3. Technical Background

In the following section, we will discuss the properties of AutoPas, a library designed to overcome the task introduced in the beginning, and design a tuning strategy to tune the algorithms used based on the configuration parameters of simulation runs.

3.1. AutoPas

AutoPas is an open-source software library to achieve optimal performance for short-range particle simulations. It is designed to behave as a black box, meaning it aims to abstract the implementation details from researchers interested in using its particle simulation capabilities to run their simulations. AutoPas differs from other such libraries like LAAMPS[Tho+22] and GROMACS[Abr+24] by providing various different algorithmic implementations for the N-body problem. Since each of these different implementations has various trade-offs, one configuration that performs well for a certain particle composition would not perform similarly for other particle compositions. AutoPas can switch during the runtime of the simulation between these implementations to guarantee optimal performance during the runtime of the simulation.

Although AutoPas only provides an infrastructure to perform these simulations and does not contain the simulation logic, it comes with example applications. Throughout this work, we have primarily focused on developing a tuning strategy for `md_flexible` application, a molecular dynamics framework built to work with AutoPas. However, transferring these concepts and training new models to work with different applications will be trivial as the data used to train is not dependent on the application itself.

Now, we can discuss the auto-tuning feature and the tunable parameters featured in AutoPas.

3.1.1. Auto-tuning in AutoPas

During its runtime, AutoPas alternates between the *tuning phase*, where the software tries to find the best possible configuration to minimize a chosen energy metric, such as time or energy, and the *simulation phase*, where it uses the configuration selected in the tuning phase to run the simulation until the next tuning phase.

However, as mentioned before, as the simulation progresses through iterations and time steps, the characteristics of the system will change. As such, the chosen configuration may no longer be the optimal configuration [Gra+22]. To solve this issue, AutoPas alternates between these phases to ensure that the system runs in a configuration that is as close to the optimal for the full simulation.

Since it includes the implementations of various different approaches, this enables the developers to have tunable parameters that can mostly be combined freely and, as a result, a vast search space corresponding to these parameters. This gives the developers the opportunity to develop various different tuning strategies to trim this enormous search space.

Below, we will discuss the tunable parameters currently existing in the AutoPas library as well as the presently existing tuning strategies. For brevity's sake, descriptions here will be made to get a general understanding of these parameters. For more detailed descriptions of these parameters, refer to [Gra+22], [Sec+21], and [New+23].

3.1.2. Tunable Parameters

The current version of AutoPas provides six tunable parameters. We will refer to a combination of parameters as *Configuration*. Each configuration contains the following parameters:

Container Options

For every particle dynamics simulation, identifying the neighboring particles is crucial to efficiently compute the short-range force potentials. The goal here is to find all particles within the cutoff radius and, therefore, should be considered for the force calculation for all particles. Below, the four prototype algorithms included in AutoPas are briefly introduced:

1. **Direct Sum:** This is the 'naive' approach, and it does not use any additional data structures to store the particles. Therefore, it relies on brute-force calculations of force potentials between all particle pairs and requires $\mathcal{O}(N^2)$ distance checks in each iteration. Naturally, this container option performs very poorly as the number of particles grows and is, in turn, impractical beyond tiny systems [VBC08].

2. **Linked Cells:** This container consists of vectors of uniform cells. Each cell holds a vector with the particles it contains. Because of this data structure, particles close in space also end up close in memory. This allows for efficient iteration of sub-regions and for efficient vectorization of particles. However, since this

approach actually models the space and not the particle relations itself, if there is a region without particles in the domain, any traversal for this container still has to check the empty cells in that region. So, there is a possibility for improvement as the constant overhead of $3 \times 3 \times 3$ cell grid is going to be unnecessarily high as there are likely going to be many particles further away than the cutoff radius in the cell grid around the current cell [Gra+22]. Despite these issues, it is still generally good for large homogeneous systems of particles.

3. **Verlet Lists:** A completely different approach to the linked cells would be not to rely on spatial information but to keep track of a particle's interaction partners. So, for each particle, we keep a Verlet [Ver67] or neighbor list that contains references to all particles within the cutoff range. However, since generating these lists every iteration would be infeasibly expensive, we also include particles slightly outside the cutoff radius to use the lists as long as possible. This extension is called the Verlet skin, and a visualization of this can be seen in 3.1. This can result in a complexity of $\mathcal{O}(N)$ distance checks for each iteration. However, a trade-off exists, as having a large skin factor would cause us to include more unnecessary distance checks while requiring us to rebuild the lists less frequently. Due to these properties, it is generally good for systems with a high density of particles

4. **Verlet Cluster Lists:** To mitigate some of the aforementioned drawbacks, Verlet Cluster Lists were developed [PH13]. This container type is based on the observation that neighboring particles will have almost identical Verlet Lists. Therefore, grouping these neighboring particles into *clusters*, we can construct a single neighbor list for each cluster. This will reduce memory usage as each cluster will need only one neighbor list. Similar to Verlet Lists, it is generally good for systems with a high density of particles while benefiting from reduced memory usage compared to regular Verlet Lists.

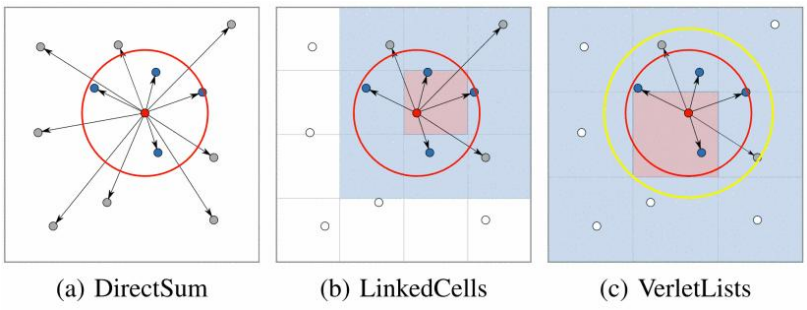


Figure 3.1.: Visualization of container options. Source: [Gra+22]

Load Estimators

The load estimator options relate to different heuristics used to estimate the load for a specific cuboid region within the domain. As we primarily focus on tuning single-node performance, we will not go into detail about these options. See [Fis20] for more details.

Traversal Options

This parameter specifies in which order the particles are visited while calculating the force potentials. In parallelized contexts, these options can significantly impact the performance. Considering the large amount of traversals included in AutoPas for varying container types, we will, for brevity's sake, only discuss the corresponding main options. For a more detailed explanation of traversal types, refer to [Gra+22].

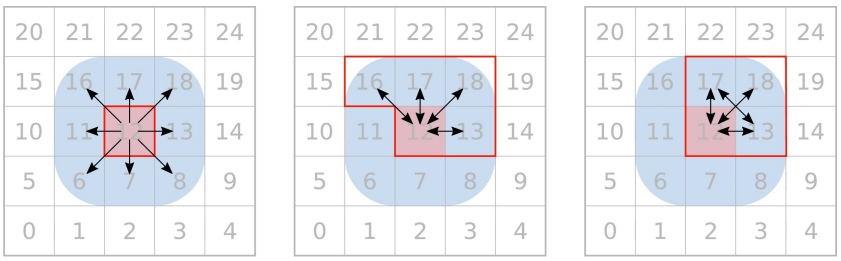
1. **Sliced Traversals:** These traversal types, as the name suggests, divide the domain into different slices. Each slice is then processed by a different thread of the application. Every thread starts with the first layer of cells in its slice and locks this section. After all cells in this plane are processed, the lock is released. Since the slices must be synchronized using a single lock, this traversal type results in only a minor synchronization overhead.

2. **Colored Traversals:** These traversal types assign a color to each cell in the container so that no same-colored cells have the same neighbor cell. During the traversal, all threads will then work on cells of the same color, which allows for high parallelism without synchronization and locks. See 3.2 for a visualization of different color-based traversal options. The following coloring options are available:

   a) **C01:** Since this traversal uses only a single color, every thread will work on all cells simultaneously, making the simulation embarrassingly parallel [HS12]. However, this comes at the cost of being forced to turn `Newton3` off as there is no way of preventing the data races.

   b) **C18:** This traversal variant divides the domain into 18 colors and only computes the interactions with the forward neighbors, meaning neighboring cells with a greater cell index. This enables the simulation to use `Newton3` as coloring prevents simultaneous access to neighboring cells.

   c) **C08:** Taking the idea of c18 base step further, this traversal variant replaces the computation for the forward diagonal interaction with the forward computation of the next cell. This allows for a higher degree of parallelism and cache reuse.



(a) c01 base step (b) c18 base step (c) c08 base step

Figure 3.2.: Visualization of color-based traversal options. Source: [Gra+22]

Data Layouts

This parameter determines how the particles are stored in memory. Two possible layouts for the data are:

1. **SoA (Structure of Arrays):** This data layout stores the properties of a particular particle in separate arrays. For example, for all particles, x-, y-, and z- positions are stored in separate arrays. When this layout is chosen, it is beneficial for vectorization as the properties of particles will be stored contiguously in memory to be loaded into vector registers with a single CPU instruction. However, this storage can also lead to inefficient cache utilization as the properties of a particular particle won't be stored close to each other in memory.

2. **AoS (Array of Structures):** This layout stores all particle properties in different structures. This, in turn, allows for efficient cache utilization as all properties are close to each other in memory. However, in contrast to SoA, loading particles into vector registers will be more complicated and can lead to inefficient vectorization for force calculations.

Newton3 Options

This option controls the aforementioned optimization technique to reduce the number of force calculations by half. Newton's third law states that for every action, there is an equal and opposite reaction. This means the magnitudes of the forces acting upon the same particle pair will be the same, with the direction being the complete opposite.

1. **Newton3 Off:** If Newton 3 is turned off, the forces are calculated redundantly for all particle pairs, resulting in a doubling of the number of force calculations.

2. **Newton3 On:** If Newton 3 is turned on, the forces between all particle pairs are calculated once. However, this means that both variables that store the

accumulated force for the respective particle have to be updated directly when computing F_{ij}, which has implications on parallelization patterns.

Cell Size Factor

The Cell Size Factor is a parameter used to determine the size of the cells in the `LinkedCells` container. If the cell size factor is high, we perform many invalid distance checks, as many particles further away than the cutoff radius are considered for the calculations. As such, it is sensible to reduce the cell size factor. However, this will increase the overhead to a point that negates the performance gain. Because of this, there exists a theoretical trade-off between reducing and increasing this parameter.

3.1.3. Existing Tuning Strategies

Full Search

The full search strategy serves as the default tuning method in AutoPas, where all possible configurations are exhaustively tested. Evaluating every potential combination ensures the identification of the optimal configuration for the current simulation state. However, this approach is typically resource-intensive and time-consuming, as it spends significant effort testing suboptimal parameter combinations. This issue becomes more pronounced as the number of parameters increases, leading to an exponential rise in possible configurations, many of which may yield poor performance. Consequently, the full search approach becomes impractical, especially as additional tunable options are introduced in AutoPas.

Random Search

The random search strategy is a straightforward tuning approach that selects a specified number of configurations randomly from the search space. This method is quicker than the full search strategy since it avoids testing every possible parameter combination. However, it does not ensure the optimal parameters for the current simulation state are found.

Predictive Tuning

The predictive tuning strategy uses prior measurements to estimate how a configuration might perform in the current simulation state. By narrowing down the search space, it retains only those configurations expected to yield good performance. These predictions

are generated using techniques like linear regression or by fitting polynomial functions to past measurement data.

Bayesian Search

AutoPas includes two implementations of Bayesian tuning, both utilizing Bayesian optimization techniques to forecast optimal configurations based on performance data from prior measurements.

Rule Based Tuning

The rule-based tuning strategy uses a set of predefined rules to automatically filter out configurations that are expected to perform poorly. The rules are passed to the simulation using a domain-specific language and are expected to be defined by expert knowledge. This method can be very effective if the rules are well-designed.

Fuzzy Logic Tuning

The fuzzy logic-based tuning strategy utilizes a data-driven dynamic rule extraction approach and fuzzy logic [NPM99] to train a fuzzy decision tree on collected data to extract fuzzy rules instead of using rules determined by expert knowledge, as is the case in rule-based tuning. The working principle of this tuning strategy is similar to the decision tree-based tuning strategy explored in this thesis in that it utilizes decision trees as part of its tuning.

4. Related Work

4.1. Works on Other HPC Software

Over the years, machine-learning-based methods have proven to be a useful approach for achieving speedups and performing optimization tasks on simulation software such as model selection, optimization of the code, and task schedulers [Che+18]. For example, in [SW23], authors use a lightweight auto-tuner driven by ML to select an optimal sparse matrix storage format from a pool of available formats to match the characteristics of the sparsity pattern, target hardware, and operation to execute. Adopting this auto-tuner results in a speedup of $1.1x$ on CPUs and $1.5x$ to $8x$ on NVIDIA and AMD GPUs, with maximum speedups reaching up to $7x$ and $1000x$, respectively.

In another work, [Kur21], the author presents "ReinforcedSel," a reinforcement learning-based approach for improving load balancing in parallel computing by automating the selection of scheduling algorithms within OpenMP. Through Q-Learning and SARSA, this method dynamically adapts to real-time performance data, achieving up to a 15.6% performance boost over traditional static scheduling. Tested on a Mandelbrot application, ReinforcedSel demonstrates adaptability comparable to expert-driven scheduling methods, highlighting reinforcement learning's potential in dynamic, heterogeneous systems. Future improvements, such as refining reward structures and expanding applications, could further enhance its effectiveness in high-performance computing.

4.2. AutoPas

Additionally, numerous studies have been developed on optimizing the tuning process of AutoPas and adding upon the existing tuning strategies, including data science-based approaches such as reinforcement learning, neural networks, and interpolation, as well as other tuning strategies using non-ML methods such as rules-based tuning or fuzzy logic.

4.2.1. "Data Science-Based" Approaches for Auto-Tuning in AutoPas

In the past, various approaches utilizing data science-based methods such as reinforcement learning, Bayesian optimization, and interpolation to auto-tune the simulation during its runtime have been developed. Below, we will discuss a series of these approaches:

Neural Networks

In [Can19], the author investigates the application of neural networks to create an auto-tuner that recommends optimal simulation configurations, aiming to reduce search time significantly while maintaining high accuracy in configuration choice. By training ML models on previous simulation data, the auto-tuner predicts configurations that balance computational efficiency and accuracy, achieving optimal choices with over 99% likelihood. The study compares different ML models, finding that classification methods generally outperform regression methods for this task.

Results from this ML-based auto-tuning approach show substantial efficiency gains, with "MachineSearch" being around 11 times faster than the exhaustive "FullSearch" approach during the tuning phases. The thesis also discusses the challenges in model integration with AutoPas. It suggests potential future improvements, including more advanced machine learning models and online learning systems that adapt to new simulation data, thus enhancing the flexibility and effectiveness of the auto-tuner in real-world applications.

However, during the development of this thesis, AutoPas didn't support advanced logging for density and homogeneity parameters of the simulation. Therefore, the author used basic features such as particle count, box length as well as cut-off and Verlet skin radii. In addition, the author didn't fully integrate the Python scripts into AutoPas and used the `frugally-deep` [Her] instead to integrate the model generated by `TensorFlow`[Aba+15] to get the prediction. However, as we will show, using Python developmental headers to communicate between scripts will enable greater flexibility for future development since knowing the model's input and output dimensions will no longer be necessary.

Reinforcement Learning

In [Lau22], the author explores the application of reinforcement learning (RL) to optimize performance. The study introduces a modified SARSA (State-Action-Reward-State-Action) algorithm to predict the optimal simulation configuration for minimizing calculation time. This approach seeks to outperform traditional tuning methods

like Full Search and Predictive Tuning, which struggle with complex, nonlinear data relationships.

The RL-based method aims to improve upon existing strategies by using prior data to predict and choose better configurations faster. The study further optimizes the SARSA algorithm with hyperparameters like learning rate, discount factor, and exploration rate, using grid search to fine-tune these parameters for better performance.

Results show that the RL-based tuning method generally performs better than Full Search and Predictive Tuning, especially in parallelized environments (using OpenMP), where it reduces the overall tuning and simulation time. Although reinforcement learning introduces some initial overhead, it ultimately demonstrates more efficient tuning over time, particularly in simulations with complex or dynamic particle distributions. The work suggests that integrating RL with more advanced algorithms or combining it with unsupervised learning techniques could improve AutoPas's performance in future work.

While other newer tuning strategies have ultimately superseded the tuning strategy developed in this thesis, this strategy shows the feasibility of an ML-based tuning strategy, namely reinforcement learning, in the task of auto-tuning the AutoPas simulation. However, decision trees serve different purposes and are primarily used for classification and regression tasks.

Bayesian Tuning

In [Ngu20], the author explores the challenge of optimizing algorithmic parameters in mixed discrete-continuous spaces using Bayesian optimization, specifically for auto-tuning tasks where configurations include both discrete and continuous parameters. Traditional Bayesian optimization, well-suited for continuous spaces, struggles with mixed-parameter spaces as discrete variables cannot be treated like continuous values. To overcome this, the author introduces a cluster-based model that organizes discrete configurations into "clusters" and optimizes continuous parameters within each cluster independently.

Each cluster is associated with its own Gaussian process model, and neighboring clusters influence each other based on similarity metrics, such as the Wasserstein distance and evidence fitting. These methods help weigh the influence of each cluster's neighbors, allowing the optimization process to incorporate useful information from similar configurations without exhaustive evaluations. Acquisition functions guide the selection of configurations, focusing on maximizing expected improvements in runtime with minimal trials, thus maintaining efficiency.

The thesis evaluates this Bayesian Cluster Search (BCS) method against other auto-tuning algorithms, including full search, traditional Bayesian search, and random

search. The BCS approach consistently performs best, achieving configurations nearly as effective as exhaustive search but with considerably reduced computation time. This cluster model offers a flexible, generalizable framework for tuning in other complex domains with mixed parameters, highlighting its potential to significantly reduce computational costs in optimization tasks.

Similar to reinforcement learning-based tuning, this tuning strategy has also been superseded by other tuning strategies. However, its results could be treated as a baseline for the performance of the decision tree-based tuning strategy.

Predictive Tuning

In [Pel20], the author introduces a predictive tuning approach using extrapolation methods —linear regression, Lagrange polynomial, and Newton polynomial— to predict the performance of different configurations. By focusing only on the most promising configurations, the predictive tuning strategy minimizes the need to test inefficient configurations, thereby reducing the overall runtime of simulations.

The predictive strategy involves an initial phase where all configurations are tested to gather baseline data, followed by extrapolation-based predictions for subsequent tuning phases. Evidence collected in early phases informs these predictions, which are then used to identify optimal configurations for testing. The effectiveness of each extrapolation method is analyzed, with linear regression emerging as the most efficient in balancing accuracy and runtime. Additionally, a "blacklist" feature eliminates consistently underperforming configurations, further streamlining the tuning process.

Evaluations using the Spinodal Decomposition simulation show that predictive tuning can reduce runtime by up to 74% compared to Full Search. The thesis concludes by suggesting potential improvements, such as dynamic adjustment of prediction ranges and incorporating error analysis into the extrapolation methods to enhance predictive accuracy.

This work shows that good tuning results can be obtained even with extrapolation techniques, which will only be able to use insights from the current simulation time. This further motivates an ML-based tuning strategy using decision trees for classification to detect underlying patterns and find the optimal configuration more efficiently.

4.2.2. Other Approaches for Auto-Tuning in AutoPas

Additionally, approaches based on other concepts, such as developing a rules-based tuning strategy to determine which configuration performs optimally on the current

simulation state and a fuzzy-tuning approach using fuzzy sets and their nuanced configurations compared to binary rules, have been developed.

Rule-Based Tuning

The report [Hum23] explores a new rule-based tuning strategy to improve the tuning phase by pre-filtering configurations based on rules derived from domain knowledge obtained from theoretical modeling, profiling, and benchmarking, thus reducing tuning overhead and improving efficiency. Extensive benchmarks with 241 simulation scenarios collected performance data to refine these rules, yielding insights such as identifying a subset of 10 optimal algorithms that perform well across scenarios.

Essentially, we aim to automate the generation of these rules in the form of decision nodes on the decision trees in this work with the decision tree tuning strategy. It is likely that the rules developed by experts can also be acquired by training a classifier on the live information collected. As the rule-based tuning strategy achieves near-optimal simulation phase runtime with a sizable tuning runtime overhead, we would try to minimize this overhead by dynamically extracting the rules from previously collected data.

Fuzzy Logic-Based Tuning

In [Ler24], the author introduces a fuzzy logic-based tuning technique, allowing the specification of custom fuzzy systems to guide tuning and prune poor configurations without exhaustive testing.

This fuzzy approach allows a more efficient exploration of the parameter space by partially activating rules based on fuzzy set logic, offering a nuanced configuration choice compared to binary rules. Through benchmarks, this technique showed speedups of up to $1.96x$ on scenarios present in the training data and up to $1.35x$ on unseen scenarios compared to Full Search. The study suggests that fuzzy tuning effectively reduces the tuning workload while maintaining comparable accuracy in simulation performance predictions, making it a promising alternative to existing strategies.

Additionally, the work explores two methods within fuzzy tuning: Component Tuning and Suitability Tuning. Component Tuning individually optimizes each parameter, simplifying the tuning process, while Suitability Tuning selects the best configuration based on the suitability score across multiple components, improving overall adaptability.

As is already suggested in the future work section of this work, the fuzzy systems could be simplified into decision trees to make the tuning process “more transparent” and “easier to understand”.

5. Implementation

This section goes over the implementation details for the decision tree-based tuning strategy. We will first discuss how the interfacing between the simulation running in C++ and the Python scripts was achieved, then we will explain the data collection process for training of the models. Finally, we will go over how the prediciton of the model is used to alter the algorithm configuration of the simulation. Refer to the GitHub Repository for the latest version of the implementation, as there can be modifications to the code.

5.1. Python/C++ Interface

The interface between Python and C++ is a crucial component in the implementation of our decision tree-based tuning strategy. We opted to use Python as the programming language to develop the ML-related parts of the tuning strategy. This decision was motivated by Python's rich ecosystem of ML libraries, such as `scikit-learn`, which facilitate the efficient training and implementation of decision trees and random forests. Furthermore, Python's adaptability makes it a strong candidate for expanding the tuning framework to include other ML-based methods in the future.

Since our simulation runs in C++, but our decision tree and random forest models are trained and run in Python, we need a seamless way to call Python scripts from within the C++ environment. To achieve this, we utilize the Python C API, specifically including the `<Python.h>` header, which provides various functions and definitions needed for embedding Python into C++ applications. The Python C API offers precise control over the Python interpreter and allows direct management of Python objects, making it ideal for performance-critical scenarios like ours. Its flexibility lets us fine-tune the interaction between the two languages without introducing unnecessary overhead.

While tools like Pybind11 simplify Python-C++ integration by automating type conversions, they come at the cost of some efficiency and control. Pybind11 is good for quickly exposing C++ functions to Python, but its higher-level abstractions can obscure critical details and add overhead. For our needs, where performance and tight integration are priorities, the Python C API was chosen as the more suitable choice.

5.1.1. Python Developmental Headers

The <Python.h> header allows direct access to the Python interpreter from C++. By including it, we gain the ability to execute Python code, call Python functions, and manage Python objects within a C++ environment. However, to make this work, the Python development headers must be present on the system, and the path to these headers must be correctly set up in the CMake configuration file.

In our CMake configuration file, we need to link the Python development libraries to ensure the compiler knows where to find the headers and libraries. We use the find_package command in CMake as follows:

```
find_package(Python3 COMPONENTS Interpreter Development REQUIRED)
include_directories(${Python3_INCLUDE_DIRS})
add_library(autopas_python INTERFACE)
target_link_libraries(autopas_python INTERFACE Python3::Python)
target_link_libraries(autopas PRIVATE Python3::Python)
```

Listing 5.1: CMake commands to include Python headers

Below is an explanation of each line in the CMake configuration file:

- find_package(Python3 COMPONENTS Interpreter Development REQUIRED): This line searches for the Python3 package on the system and specifically looks for both the interpreter and development components. The Interpreter component allows the program to invoke Python scripts, while the Development component provides the necessary headers and libraries to embed Python code within a C++ application. By adding REQUIRED, CMake will produce an error and stop configuration if Python3 isn't found. This line requires installing the package python3-dev to function correctly.
- include_directories(${Python3_INCLUDE_DIRS}): This line tells CMake to add the directory containing Python3 headers to the project's include path. This is necessary for including <Python.h> in the C++ code, allowing the C++ compiler to locate the Python headers and use Python's C API functions.
- add_library(autopas_python INTERFACE): This line creates an interface library called autopas_python. An interface library in CMake doesn't produce compiled output but allows us to define dependencies (such as linking libraries or include directories) that other targets can use. Here, autopas_python acts as a linkable target to manage Python dependencies in the project.
- target_link_libraries(autopas_python INTERFACE Python3::Python): This line specifies that the autopas_python interface library should link to the main

Python library. By linking this library, `autopas_python` gains access to the necessary functions for embedding and interfacing with Python.

- `target_link_libraries(autopas PRIVATE Python3::Python)`: This line links the main `autopas` target (our C++ application or library) with the main Python library, allowing it to call Python functions and run Python code. This makes all Python-related functions available to `autopas`, effectively integrating Python capabilities into the C++ project.

Now that Python developmental headers are successfully integrated into the building process of AutoPas, we can create a new tuning strategy utilizing Python scripts following the contribution guidelines for AutoPas.

5.1.2. `loadScript` Function

After following the contribution guidelines to implement a `DecisionTreeTuning` class that implements the `TuningStrategyInterface`, we initialize the Python interpreter by calling the `Py_Initialize()` function followed by the `loadScript()` function to load the script named `predict.py` which makes a configuration prediction using the pre-trained model based on the current state of the simulation. A listing of this function can be found in A.5.

5.1.3. `train.py` Script

A simple training script was implemented to streamline the training process using the data generated from `liveInfoLogger` and `tuningResults` loggers. By putting these logs in the directories `data/live_info` and `data/tuning_results` respectively and running this script, we obtain a trained random forest model to use for the prediction script. We ultimately decided to use a random forest model here, as the bagging approach the random forests use enables them to achieve greater generalization and, thereby, robustness. During the forward pass phase, no noticeable effect on the runtime was observed due to the prediction being made by a complex model because of using random forests instead of decision trees. A detailed overview of this script can be found in A.2.1.

5.1.4. Testing

For the testing of the implemented tuning strategy, the interface between Python and C++ was tested using a mock prediction script that always returns the same configuration as testing an ML-based method beyond splitting the dataset with train

and test sets due to their inherent randomness and limited interpretability [Wan+24]. The system was tested to verify it can handle invalid model inputs, errors during the Python script, and invalid responses by the Python script. An overview of the test cases can be found at here.

5.1.5. Memory Leak Issues

In integrating Python into our C++ application, we detected some memory leaks when enabling the address sanitizer while building even when only calling `Py_Initialize()` and `Py_Finalize()` in the constructor and the destructor of our tuning strategy, respectively. Several underlying issues in Python's interpreter can lead to residual memory retention [Fou20]. Here are some specific causes and explanations based on Python's own documentation and our observations:

- **Internal Allocations by the Python Interpreter**: When `Py_Initialize()` is called, Python allocates various global data structures and caches. These allocations support core interpreter functionality, modules, and certain optimizations. However, `Py_Finalize()` doesn't always fully deallocate these resources, especially for some standard library modules or caches created during initialization. This results in small amounts of memory remaining allocated even after `Py_Finalize()` completes.
- **Dynamically Loaded Extensions Not Unloaded**: During initialization, Python may load certain dynamic libraries or shared modules that are part of its environment. Unfortunately, `Py_Finalize()` doesn't automatically unload these modules or release all memory tied to them.
- **Circular References within Python's Core**: Although Python's garbage collector is designed to handle most circular references, it doesn't fully collect all such references, particularly those involving objects with destructors (`__del__()` methods). Circular references between core Python objects or internal data structures can thus remain after `Py_Finalize()`, resulting in memory that is not reclaimed.
- **Memory Fragmentation from Python's Internal Allocators**: Python uses its own memory management system, `pymalloc`, optimized for handling Python objects. However, this can cause memory fragmentation, especially after initializing and finalizing the interpreter. When Python requests memory from the operating system, certain memory blocks may remain fragmented and unreleased, leading to apparent memory leaks.

- **Unfreed Small Allocations by C Libraries**: In addition to Python's internal memory handling, the C standard library used by Python may retain small amounts of memory after `Py_Finalize()`.

These issues collectively contribute to minor memory leaks or unreleased memory when embedding Python, even if only `Py_Initialize()` and `Py_Finalize()` are called.

Mitigation Strategies

To address or reduce these memory leak issues, we implemented the following approaches:

- **Using a Single Initialization/Finalization Cycle**: To minimize fragmentation and the risk of leaks associated with multiple initializations, we initialize Python only once at the start of the application and finalize it only at the end, using the constructor and the destructor respectively. This strategy prevents some issues related to the reinitialization of certain modules or extensions.
- **Testing with Different Python Versions**: Testing across versions revealed variations in memory management behavior, which suggests the memory leak issue is not caused by an error on our end.
- **Analysis of the Code for Correct Dereferencing**: For the functions within the `DecisionTreeTuning` class, we carefully investigate each variable and their reference counts to make sure they reach zero by the end of the function to be deallocated by Python's garbage collector.
- **Suppressing the Memory Leaks**: After determining that the memory issues are caused by errors from the Python interpreter, we opted to suppress the memory leaks originating from `libpython` to pass the GitHub CI workflows designed to verify that the newly added parts to the software meet certain standards criteria.

5.2. Training Process

This section examines the training process for the decision tree and random forest models that are used in our tuning strategy. Here, we first describe the data collection process, followed by details on how this data is used to train and optimize the models.

5.2.1. Data Collection

To gather a comprehensive dataset for training, we incorporated a range of simulation scenarios, utilizing both standard examples from the `md_flexible` framework as well as custom configurations designed to evaluate performance under various conditions. These scenarios include:

- **Example Scenarios**: We used some example scenarios, such as `fallingDrop.yaml`, `explodingLiquid.yaml`, and `spinodalDecomposition.yaml`, as primary sources of data. These examples are standard test cases for evaluating fluid simulations and particle interactions and provide insight into the behavior of the algorithm under common conditions.
- **Custom Cube Configurations**: In addition to the standard examples, we generated custom cube simulations with varying configurations to observe performance. Specifically, we included `CubeGauss` objects with varying levels of homogeneity and `CubeUniform` objects with different densities. This allows us to observe how the model responds to controlled variations in particle distributions and densities, better training the model to recognize how algorithm selection varies based on these parameters.

For a detailed explanation of these `.yaml` files and the varied parameters during the data collection phase, see A.3.

All simulations were executed on the serial partition of the CoolMUC-2[1] cluster, and each configuration was repeated once to account for fluctuations in performance. Additionally, simulations were run with 1, 2, 4, 12, 24, and 28 threads to gather data on how parallelization affects the optimal tuning configuration.

To ensure consistency, we used the `.yaml` option `pause-simulation-during-tuning` to pause the simulation state during the tuning phases. This prevents any changes in the simulation state from influencing the measurements and guarantees that all configurations are evaluated under identical conditions. This approach is suitable during the data collection phase, where we must ensure all configurations are evaluated under identical conditions. Still, for benchmarking, we turn this option off to provide a more realistic evaluation of a tuning strategy's performance. For the data collection, we used the `FullSearch` tuning strategy, which exhaustively tests all possible configurations to obtain comprehensive data and achieves the optimal configuration at the cost of tuning overhead.

[1] CoolMUC-2 is an HPC cluster located at the Leibniz Supercomputing Centre (LRZ) in Garching, Germany. It contains 812 nodes, each equipped with 14 cores, and supports up to 28 threads per node through hyperthreading. Additional information is available at `https://doku.lrz.de/coolmuc-2-11484376.html`.

5.2.2. Training Methodology

Once the data was collected, we used it to train the random forest model. Given the nature of the problem and the variability in simulation conditions, random forests were chosen for their robustness and ability to generalize well across different conditions. With their ensemble learning approach, random forests average the predictions from multiple decision trees, which helps reduce variance and improve predictive stability.

The training process was streamlined with a custom script, `train.py`, that reads the data files from the `data/live_info` and `data/tuning_results` directories. After training, the resulting model is saved and can be loaded efficiently during the simulation runtime. During evaluation, no significant impact on runtime was observed from using the random forest model, as the prediction overhead remained minimal.

This approach allows us to benefit from the strengths of random forests without compromising performance, ensuring that the tuning strategy can adapt better to changing conditions within the simulation.

5.2.3. Limitations and Considerations

Despite collecting a broad range of data, the performance of machine learning models can degrade when faced with scenarios that diverge significantly from the training data. As such, it's essential to be aware of the limitations inherent in our dataset. Since we used only a limited number of predefined scenarios, the trained model may lack the ability to generalize confidently to new, unseen conditions. For a fair evaluation in Chapter 6, we will limit our assessment to variations of the included scenarios, which represent realistic use cases based on our training data.

In the future, incorporating additional scenarios would be beneficial to further enhance the model's robustness and extend its applicability. For example, using scenarios with varying particle types, boundary conditions, and interaction models could provide a richer training dataset and lead to a more flexible and generalizable tuning strategy.

5.2.4. Collected Information

Data collection in the simulations relied on AutoPas's existing `LiveInfoLogger` and `TuningResults` logger to capture essential metrics that describe both the configuration and state of each simulation. These tools allowed us to record a variety of metrics:

- **Configuration Metrics**: Information on the specific configuration used in each simulation run, including algorithmic parameters such as the chosen container

option, traversal option, data layout, and whether the `newton3` optimization is enabled.

- **State Metrics**: Data related to the current state of the simulation, such as the average number of particles per cell, maximum particles per cell, homogeneity of particle distribution, maximum density, standard deviation of particles per cell, and thread count.

The complete shape of the collected data can be found in Appendix A.4. We focused on a subset of relative metrics (such as `avgParticlesPerCell`, `maxParticlesPerCell`, `homogeneity`, `maxDensity`, `particlesPerCellStdDev`, and `threadCount`), which scale proportionally with the simulation. This focus minimizes overfitting risk by emphasizing relative values, enhancing the model's generalizability.

`LiveInfoLogger`

The `LiveInfoLogger` class is used to gather dynamic data during the simulation. By logging live information about the state of the simulation and the corresponding density and homogeneity-related metrics, this logger provides a detailed view of how simulation conditions evolve over time. This data was then used to train a model that accounts for the variability in real-time simulation states.

`TuningResults`

The `TuningResults` logger, on the other hand, records outcomes related to the performance of different configurations. This data includes timing information for each configuration tested, allowing us to assess the impact of various algorithmic choices on the efficiency of the simulation. Together with `LiveInfoLogger`, `TuningResults` provides a comprehensive dataset that drives the model's ability to predict optimal configurations under diverse conditions.

5.3. Getting and Applying the Predicted Configuration

After managing to train a random forest model that takes a selection of `LiveInfo` parameters as input and outputs the optimal algorithmic configuration, the remaining task is to create a Python script that the C++ part of the program interacts with by sending a JSON object and receiving the corresponding configuration for the provided `LiveInfo` as a JSON object. In this section, we will first go over the Python part, `predict.py` script, followed by the `getPredictionFromPython` method of the `DecisionTreeTuning` class.

5.3.1. `predict.py` Script

The `predict.py` script is the Python component responsible for making configuration predictions based on live simulation data passed from the C++ program. This script loads the trained random forest model and performs predictions using live information provided in JSON format. When executed by being called from the C++ class, it takes the `LiveInfo` data, pre-processes it to match the input format expected by the model, and then uses the model to output a predicted configuration in JSON format. Detailed implementation steps and the script's listing can be found in Appendix A.2.1.

5.3.2. `getPredictionFromPython` Function

The `getPredictionFromPython` function in the `DecisionTreeTuning` class is the C++ method that interacts with the `predict.py` script. This function converts live simulation data into JSON format and sends it to the Python `main` function in `predict.py` for prediction. After receiving the response, the function extracts the predicted configuration and formats it for use within the C++ environment. A full explanation and listing for `getPredictionFromPython` can be found in Appendix A.6.

6. Results

In this section, we compare the performance of the developed decision tree-based tuning strategy with existing tuning approaches in AutoPas in order to evaluate its performance and feasibility. All of the benchmarks were conducted on the version with the commit id of `48db034558092a3b68348b58e5059fdf3797d77b` in the AutoPas repository[1].

To measure the performance of the developed tuning strategy, we use the scenarios present in `md-flexible` and compare the performance of existing tuning strategies within AutoPas. The benchmarking is conducted on CoolMUC-2[2][Wil+17] cluster, the same cluster architecture the data collection was conducted in. The benchmarks were repeated in 1, 2, 4, 12, 24, and 28 threads. Since no performance difference significant enough to affect the comparison of different tuning strategies was observed in the benchmarks, we will use the single-threaded performance to generate plots and figures for brevity. We use the performance reports and logs generated at the end of a simulation run to evaluate the performance of the strategies, as they are good indicators of the performance of the simulation.

6.1. Falling Drop Benchmark (In the Training Dataset)

The falling drop benchmark simulates a falling liquid drop on a liquid surface as the simulation progresses[3]. As the data of this scenario in different configurations was already included in the training data, we expect our developed tuning strategy to perform well and predict the optimal configuration reliably.

[1] `https://github.com/AutoPas/AutoPas/pull/953/commits/48db034558092a3b68348b58e5059fdf3797d77b`

[2] CoolMUC-2 is an HPC cluster located at the Leibniz Supercomputing Centre (LRZ) in Garching, Germany. It contains 812 nodes, each equipped with 14 cores, and supports up to 28 threads per node through hyperthreading. Additional information is available at `https://doku.lrz.de/coolmuc-2-11484376.html`.

[3] The `.yaml` file used can be accessed from `https://github.com/AutoPas/AutoPas/blob/09bebc93ae679e9c2517eb836e58964457f336d5/examples/md-flexible/input/fallingDrop.yaml`

6.1.1. Scenario Configuration Details

As mentioned, we have used the default falling liquid scenario with some configurations for our benchmark. Here are the key metrics for the simulation:

- **Particle Count:** 15094
- **Objects:**
  - A `CubeClosestPacked` object was used to simulate the bottom liquid.
  - A `Sphere` object was used to simulate the liquid drop.
- **Cutoff:** Lennard-Jones force cutoff was set to 3.
- **Boundary Conditions:** Reflective boundary conditions were used.
- **Gravity:** A global force of $[0, 0, -12]$ was applied to the particles to simulate gravity.

The plot in Figure 6.1 compares the total time spent running the simulation metric generated at the end of each simulation run in terms of seconds with the falling drop scenario. To account for fluctuations and fairly examine each tuning strategy, each run was conducted 2 times due to time constraints. As can be seen from the figure, our decision tree-based tuning strategy manages to achieve comparable run-times with the fuzzy logic-based tuning strategy and over-performs other tuning strategies, as both tuning strategies are capable of immensely lowering the tuning times while managing to find the optimal configuration for the shortest runtime of force calculations. For other tuning strategies, since they test significantly worse configurations during the tuning phases, they have a higher tuning time and, as a result, higher total execution time.

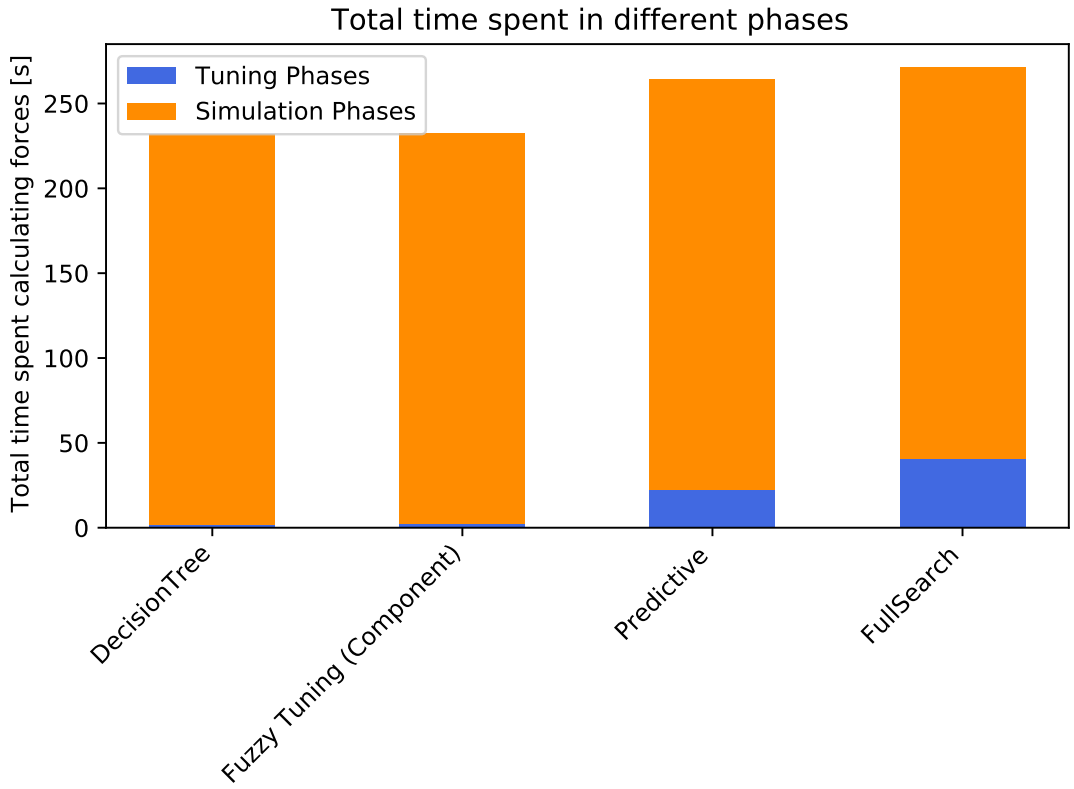


Figure 6.1.: Plot of the total time spent in different tuning strategies (Numerical Values in Table A.1)

In Figure 6.2, it can be seen that almost the entirety of the simulation time is spent on the calculation of force potentials instead of tuning the simulation, and this enables the users to run potentially more complex simulations with the same computing resources as up to 15% of the simulation time is used on tuning when `FullSearch` tuning strategy is used.

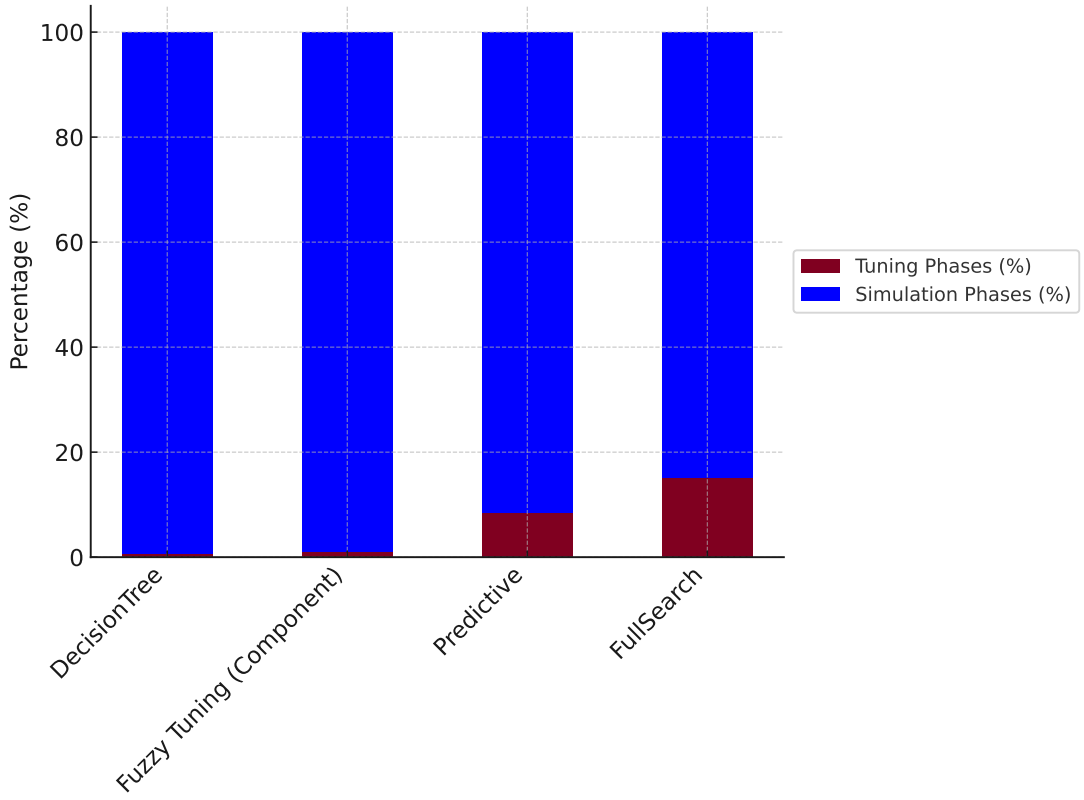


Figure 6.2.: Plot of the percentage of runtime spent on different phases (Numerical Values in Table A.2)

As a result, the decision tree-based tuning strategy achieves a speedup of $\frac{t_{\text{FullSearch}}}{t_{\text{DecisionTree}}} = \frac{40.88}{1.61} \approx 25$ in terms of time spent tuning and a speedup of $\frac{t_{\text{FullSearch}}}{t_{\text{DecisionTree}}} = \frac{271.42}{230.67} \approx 1.18$ in terms of total simulation time compared to `FullSearch` tuning strategy.

As can be understood from the figures, the lowering of the tuning overhead is the most significant factor in the performance of the decision tree-based tuning strategy. This is because, in contrast to the other tuning strategies, it does not evaluate various configurations to decide on the optimal configuration, as the main factor in the tuning overhead is the evaluation of many potentially bad configurations causing a slowdown.

6.2. Exploding Liquid Benchmark (Not in the Training Dataset)

The exploding liquid scenario [4] simulates a high-density liquid expanding outwards as the simulation progresses. To test out the robustness of the decision tree-based model, we are going to omit the data collected from this scenario, train the random forest again, and observe how it performs compared to the other scenarios. Since our training dataset now contains runs from `fallingDrop`, `spinodalDecomposition`, as well as `CubeGrid` and `CubeGauss` scenarios with different homogeneities and densities respectively (discussed in A.3), although it is possible the tuning strategy might not find the most optimal configuration, we expect it to perform well enough to have a speedup on the total run-time compared to `FullSearch` tuning strategy. We will also include a run where this scenario is included in the training set to see how the runtime is affected.

6.2.1. Scenario Configuration Details

As mentioned, we have used the default exploding liquid scenario with some configurations for our benchmark. Here are the key metrics for the simulation:

- **Particle Count:** 1764
- **Objects:**
  - A `CubeClosestPacked` object was used to simulate the exploding liquid.
- **Cutoff:** Lennard-Jones force cutoff was set to 2.
- **Boundary Conditions:** Periodic boundary conditions were used.
- **Miscellaneous:** Tuning interval and number of iterations were set to 2500 and 15000, respectively, to match the falling drop scenario for a fairer comparison.

As it can be seen in Figure 6.3, even though the exploding liquid scenario is not in the training dataset, since similar simulation configurations have been achieved in terms of trained parameters in `CubeGauss` and `CubeUniform` examples, the model has been able to learn which algorithm configuration suits best for this runtime configurations reliably. Additionally, in this scenario, the tuning overhead is larger than the falling drop scenario, presumably because the simulation state changes more drastically compared to the latter scenario due to the nature of the scenarios.

[4] The `.yaml` file used can be accessed from `https://github.com/AutoPas/AutoPas/blob/09bebc93ae679e9c2517eb836e58964457f336d5/examples/md-flexible/input/explodingLiquid.yaml`

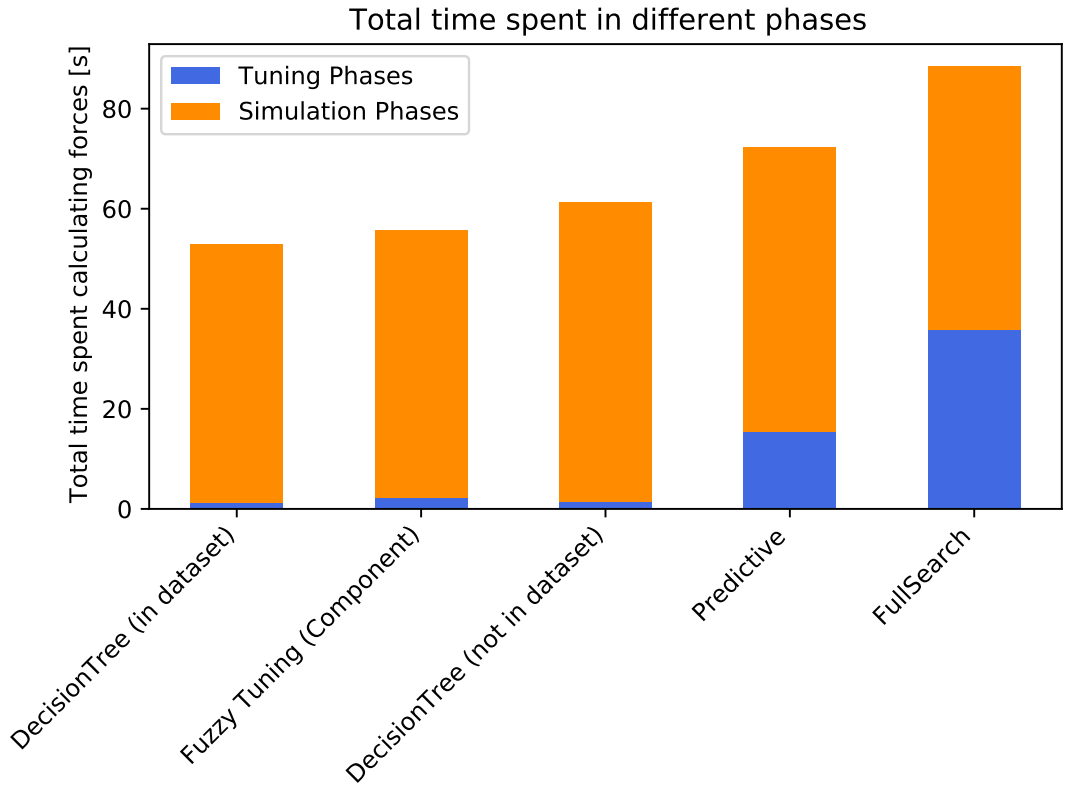


Figure 6.3.: Plot of the total time spent in different tuning strategies (Numerical Values in Table A.3)

Presumably, as we do not have the exact scenario in the dataset, the model did not select the optimal algorithm configuration for every tuning phase, resulting in slightly longer simulation times than when the scenario is in the dataset. However, because of the "data-driven" approach, we managed to lower the tuning overhead exceedingly, resulting in shorter run-times compared to predictive and full search tuning strategies.

In Figure 6.4, it can be seen that the benefit of almost eliminating tuning the overhead is much more significant in this scenario as the tuning overhead with `FullSearch` tuning can reach up to 40%. As a result, usage of the decision tree-based tuning strategy, similar to other "knowledge-based" tuning strategies, is going to be most beneficial in scenarios where the simulation state changes more than average.

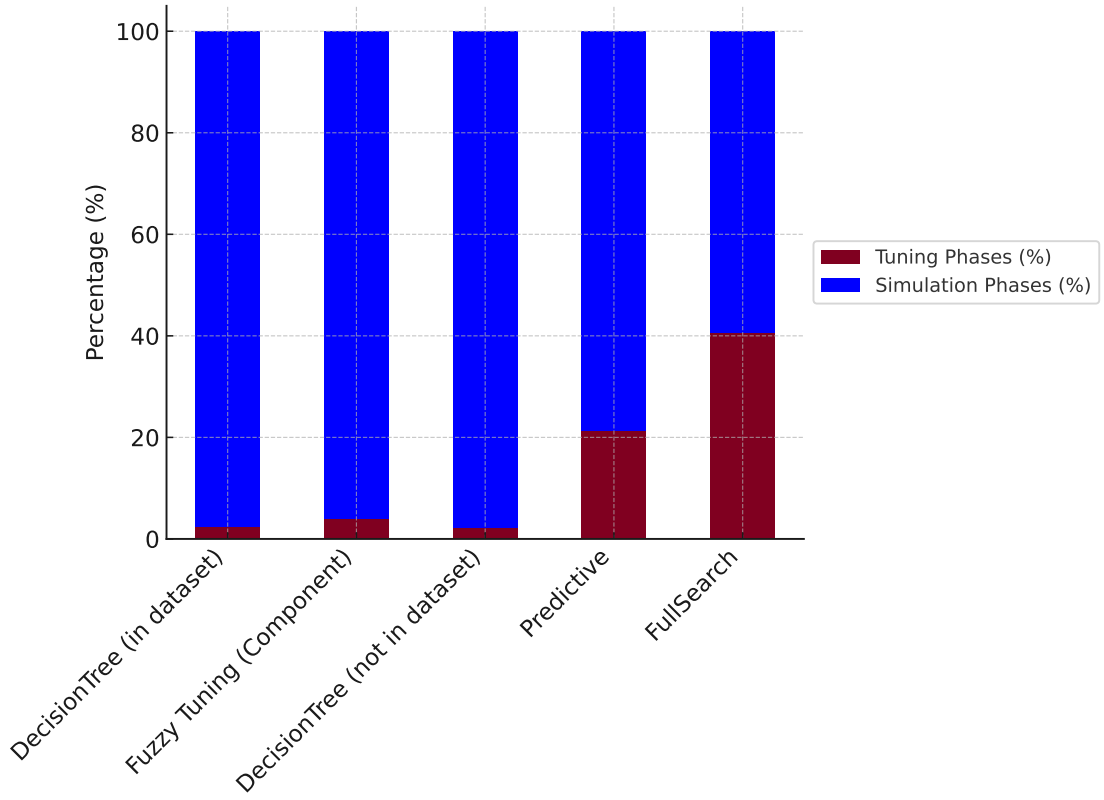


Figure 6.4.: Plot of the percentage of runtime spent on different phases (Numerical Values in Table A.4)

With this scenario the decision tree-based tuning strategy achieves a speedup of $\frac{t_{\text{FullSearch}}}{t_{\text{DecisionTree}}} = \frac{88.2}{60.6} \approx 1.45$ speedup when the scenario is not in the dataset and a speedup of $\frac{t_{\text{FullSearch}}}{t_{\text{DecisionTree}}} = \frac{88.2}{52.4} \approx 1.68$ when the scenario is included in the dataset.

6.3. Decision Trees and Random Forests Performance Comparison

During the earlier phases of development, decision trees were used instead of random forests as the model to predict the algorithm configuration. Decision trees tend to over-fit the training data, capturing noise instead of the underlying pattern, especially as the tree depth increases. This behavior was also observed in our development process, notably when the model was tested against 'unfamiliar' scenarios not included

in the training data. While the decision tree approach managed to achieve comparable results to random forests on the traditional data analytics metrics such as accuracy, F1-scores, and confusion matrices, they performed consistently worse in the scenarios where similar scenarios were not included in the training dataset. One such example benchmark can be seen in 6.5. This is due to the fact that the exploding liquid scenario was not included in the training dataset. Random forests were able to recognize similar live simulation state configurations that existed in the dataset due to their ensemble nature preventing over-fitting, managing to output optimal to near-optimal configurations from unfamiliar scenarios with the help of the `CubeGauss` and `CubeUniform` runs with differing homogeneities and densities, respectively, in the dataset, and producing better run-times than decision trees. However, this comparison and the decision to move on to random forests as the classification model was made earlier in the development phase when the training dataset was smaller than the other benchmarks were conducted. It is to be expected that the performance of decision trees and random forests ultimately depends on the quality of the dataset in covering the target space and the selected feature parameters.

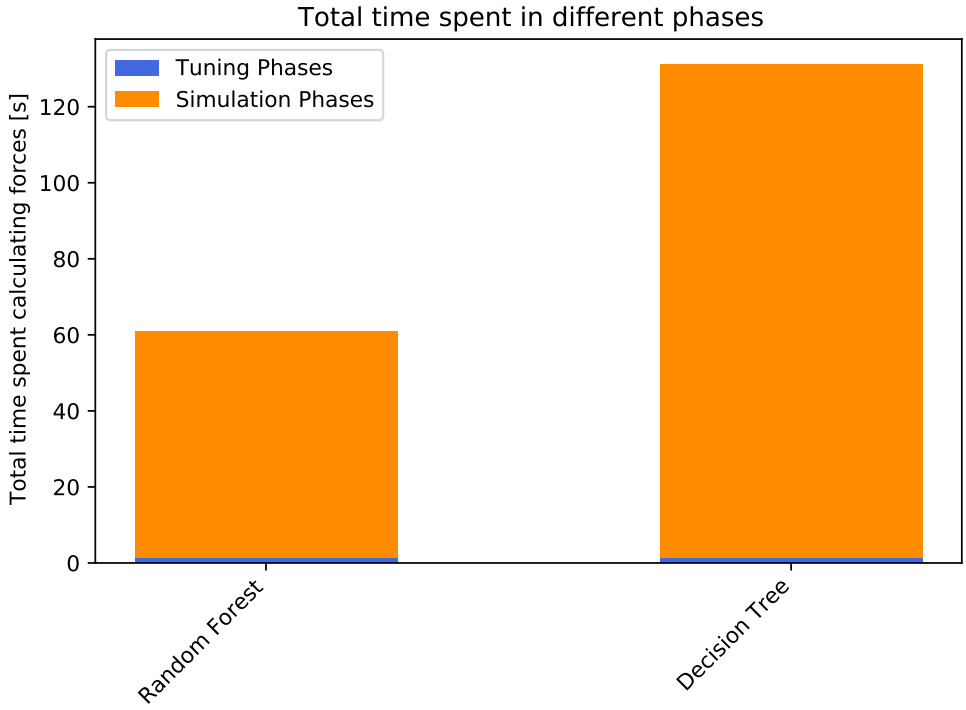


Figure 6.5.: Random Forest vs Decision Tree Comparison

7. Future Work

In this section, we discuss a set of possible improvements that could be made to the currently developed tuning strategy to improve its performance and usability.

7.1. "Live Training"

The model predicts and modifies the configuration queue by selecting an algorithm configuration based on its learned patterns. While this approach enables dynamic decision-making, it may result in selecting poorly performing configurations, particularly in cases where the model encounters an unfamiliar simulation state. Since the model does not currently assess the confidence of its predictions, it lacks a mechanism to handle such scenarios proactively.

As future work, introducing a confidence-based mechanism could address this limitation. If the model identifies a prediction with low confidence, it could switch to alternative tuning strategies defined in the runtime configuration file. This would provide a fallback mechanism to mitigate the impact of incorrect predictions. Additionally, implementing live training functionality could allow the model to update itself dynamically upon encountering unfamiliar states. By retraining or fine-tuning during runtime, the system could adapt to evolving simulation characteristics and improve its predictions in subsequent tuning phases. Combining these enhancements would reduce reliance on predefined strategies, minimize the risk of poor performance, and make the system more robust and autonomous.

7.2. Exploration of Different Features

The features we've selected to train the ML model were chosen mainly based on the intuition that density and the homogeneity of the particles during the simulation have an effect on which algorithm configuration performs best. However, as AutoPas continues to evolve, additional live simulation data may become available, offering more robust correlations for configuration selection. Exploring these new features could improve the accuracy and generalizability of the ML model. For example, incorporating features that reflect temporal changes in the simulation or higher-order spatial statistics

could provide richer information for the model. Future work could also investigate an automated selection of these features using advanced feature engineering techniques or exploratory data analysis tools.

7.3. Exploration of Different Classification Methods

The performance of other ML classifiers, such as support vector machines (SVMs), K-nearest neighbors, or artificial neural networks, could be explored. SVMs, for example, might offer better performance in cases with non-linear or complex decision boundaries, while ANNs could excel in capturing non-linear patterns in the data. Since the current implementation uses a decision tree-based classifier, which provides simplicity and interpretability, the existing infrastructure already interfaces with Python libraries like `scikit-learn`, making it suitable for testing other classifiers. Future work could evaluate these methods to determine if they offer improved accuracy, speed, or scalability for tuning in more complex or diverse simulation scenarios.

8. Conclusion

This thesis used a decision tree-based strategy to introduce a new tuning approach for AutoPas. In addition to a tuning strategy for the AutoPas framework, it also integrated smooth interaction functionality with Python scripts through Python development headers, offering a flexible foundation for future research projects.

The main achievement here was creating a tuning method that allows AutoPas, largely written in C++, to take advantage of machine learning libraries in Python. This setup paves the way for adding more ML-based tuning options, like different classification techniques, through a communication bridge between C++ and Python. The results showed that this decision tree-based tuning strategy significantly outperformed current methods in certain benchmarks, cutting overall simulation time by drastically reducing the tuning time while still finding optimal or near-optimal configurations.

However, while this approach with decision trees has clear potential, it requires substantial initial effort to gather data for building the model, which may be a hurdle for typical AutoPas users. Future research should aim to make the data collection and model training process more streamlined to make this decision tree tuning method more accessible to a broader audience of researchers.

In conclusion, the decision tree tuning strategy and data-driven model training represent a meaningful step forward in tuning AutoPas simulations and provide a strong basis for future work. Though challenges remain in making it more broadly accessible, the potential for major performance speedup makes this a promising area for continued development and refinement.

A. Appendix

This section serves as a collection of figures and listings to clarify the implementation details explained in Chapter 5.

A.1. UML Diagram for `DecisionTreeTuning`

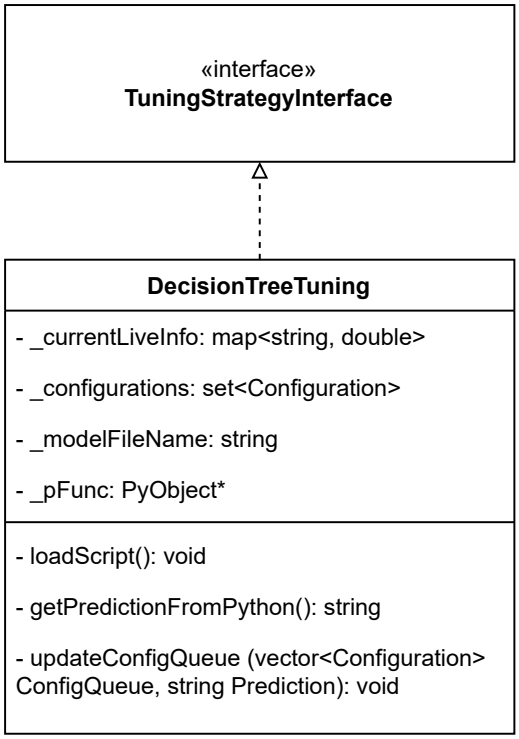


Figure A.1.: UML Diagram for `DecisionTreeTuning`

In A.1, we can see the UML diagram for `DecisionTreeTuning`. For brevity, we have only included the variables and methods not included in the implemented

`TuningStrategyInterface` interface. Below is the explanation for the class variables, and the explanation of the class methods can be found in A.2.2.

- `_currentLiveInfo`: A map that stores live information about the current simulation state, with each entry representing a key-value pair where the key is the metric name and the value is its current value. This data is updated in real-time and passed to the Python script for predictions.
- `_configurations`: A set containing possible configurations that the tuning strategy may consider. Each configuration defines a specific setup of algorithmic parameters, such as the container, traversal, and data layout options, which are evaluated based on the predicted performance.
- `_modelFileName`: A string that stores the file name containing the trained decision tree model. This model file is loaded by the Python script `predict.py` to predict the best configuration based on live simulation data. This can be changed in the `.yaml` files to pass to the program.
- `_pFunc`: A pointer to the `main` function in the `predict.py` Python script. This function uses the loaded model to make predictions based on the live information provided by `_currentLiveInfo`. The pointer ensures that the function is accessible across different methods in the `DecisionTreeTuning` class.

A.2. Code Listings

This section is a collection of code listings of crucial functions used for the implementation of the decision tree-based tuning strategy.

A.2.1. Python Script Functions for Training and Prediction

Here are functions developed in Python for the training and prediction scripts will be listed.

Training Script

Here we will display listings of functions used in the training script. Only the crucial functions, namely for preprocessing and training, will be included for brevity.

```
def preprocess_data(merged_df: pd.DataFrame) -> tuple:
    # Select the features for training
```

```
    features = ['avgParticlesPerCell', 'maxParticlesPerCell', 'homogeneity
        ', 'maxDensity', 'particlesPerCellStdDev', 'threadCount']

    # Select the target
    targets = ['Container', 'Traversal', 'Data␣Layout', 'Newton␣3']

    # Encode using LabelEncoder
    label_encoders = {target: LabelEncoder() for target in targets}

    for target in targets:
        merged_df[target] = label_encoders[target].fit_transform(merged_df
            [target])

    X = merged_df[features]
    y = merged_df[targets]

    return X, y, label_encoders
```

Listing A.1: `preprocess_data` Function

As we can see in A.1, the function uses a set of features described in Chapter 5 to create a merged DataFrame [pan24] of features and targets, and this DataFrame is later used to train a random forest model.

```
def train_model(X: pd.DataFrame, y: pd.DataFrame) -> dict:
    # Split data into training and test sets
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
        =0.2, random_state=42)

    # Train a RandomForestClassifier for each target
    models = {}
    for i, column in enumerate(y.columns):
        model = RandomForestClassifier(n_estimators=100, random_state=42)
        model.fit(X_train, y_train.iloc[:, i])
        models[column] = model

        # Test accuracy on test set for this model
        y_pred = model.predict(X_test)
        accuracy = accuracy_score(y_test.iloc[:, i], y_pred)
        print(f'Accuracy␣for␣{column}:␣{accuracy:.2f}')
```

```
    return models
```

Listing A.2: `train_model` Function

The `train_model` function trains a separate `RandomForestClassifier` for each target column in the provided labels, `y`. It first splits the data into training and test sets, reserving 20% of the data for testing. For each target column, a random forest model with 100 estimators is created, trained on the training set (`X_train`, `y_train`), and stored in the `models` dictionary under the column name as the key. After training each model, it predicts the values on the test set, and the function calculates and prints the accuracy score for each target, providing insight into the model's performance on unseen data. The function finally returns the dictionary containing all trained models.

Prediction Script

Here we will display listings of functions used in the prediction script. Similar with the training script, only the crucial functions, namely for prediction and the main entry point from C++, will be included for brevity.

```
def predict(live_info: dict, models: dict, label_encoders: dict) -> dict:
    live_info_df = preprocess_live_info(live_info)

    predictions = {}

    # Perform prediction for each target
    for target, model in models.items():
        prediction_encoded = model.predict(live_info_df)
        prediction = label_encoders[target].inverse_transform(
            prediction_encoded)
        predictions[target] = prediction[0]

    return predictions
```

Listing A.3: `predict` Function

The `predict` function takes live information from the main function, which is passed as a dictionary `live_info`, along with a dictionary of trained models `models`, and a dictionary of label encoders `label_encoders`. It first preprocesses `live_info` by converting it into a DataFrame through the `preprocess_live_info` function. Then, for each target, the function retrieves the corresponding model from `models` and performs a prediction on the processed `live_info_df`. The encoded prediction is transformed

back to the original label using the relevant label encoder, and the result is stored in the `predictions` dictionary under the target name. Finally, the function returns `predictions`, a dictionary containing predictions for each target variable.

```
def main(model_file: str, live_info_json: str) -> str:
    # Load models and encoders
    models, label_encoders = load_models_and_encoders(model_file)

    # Parse the live info
    live_info = json.loads(live_info_json)

    # Make predictions
    predictions = predict(live_info, models, label_encoders)

    return json.dumps(predictions)
```

Listing A.4: `main` Function

The `main` function manages the overall prediction workflow. It begins by loading the pre-trained models and label encoders from a specified file, `model_file`, for which the model filename is passed from C++. Then, it parses the `live_info_json` string, which contains the current simulation state from the C++ `DecisionTreeTuning` class, into a Python dictionary. Using the `predict` function, it generates predictions based on this live information and the loaded models. Finally, the function converts the `predictions` dictionary to JSON format and returns it as a string, ready to be passed back to the C++ environment.

A.2.2. C++ Functions for Interfacing with Python

Here we include listings of a collection of functions for interfacing with Python and the random forest model. For brevity, we again include the most important functions, namely `loadScript` and `getPredictionFromPython`. For a more detailed look at the implementation, see the GitHub repository.

```
void DecisionTreeTuning::loadScript() {
  PyObject *sysPath = PySys_GetObject((char *)"path");
  PyObject *cwd = PyUnicode_FromString(".");
  PyList_Append(sysPath, cwd);
  Py_DECREF(cwd);

  PyObject *pName = PyUnicode_DecodeFSDefault("predict");
```

```
  if (!pName) {
    utils::ExceptionHandler::exception("Failed␣to␣convert␣script␣name␣to␣
        Python␣string.");
  }

  PyObject *pModule = PyImport_Import(pName);
  Py_DECREF(pName);

  if (!pModule) {
    PyErr_Print();
    utils::ExceptionHandler::exception("Failed␣to␣load␣Python␣module:␣
        predict.py");
  }

  PyObject *pDict = PyModule_GetDict(pModule);
  Py_DECREF(pModule);

  if (!pDict) {
    PyErr_Print();
    utils::ExceptionHandler::exception("Failed␣to␣get␣dictionary␣from␣
        Python␣module:␣predict.py");
  }

  PyObject *pFunc = PyDict_GetItemString(pDict, "main");
  if (!pFunc || !PyCallable_Check(pFunc)) {
    PyErr_Print();
    utils::ExceptionHandler::exception("Failed␣to␣get␣Python␣function:␣
        main");
  }

  _pFunc = pFunc;
  Py_INCREF(_pFunc);
}
```

Listing A.5: `loadScript` Function

The `loadScript` function loads the `predict.py` Python script into the C++ environment. First, it adds the current directory to Python's search path, allowing Python to locate the script. Then, the function attempts to load `predict.py` by importing it as a module. If loading fails, it triggers an exception to indicate the issue. Once the

module is loaded successfully, the function retrieves the module's dictionary, `pDict`, which contains all defined variables, functions, and classes. Using this dictionary, it accesses the `main` function within the `predict.py` script. If `main` is found and is callable, it increments the function's reference count to prevent garbage collection, as `_pFunc` is stored as a class member for later use.

```
std::string DecisionTreeTuning::getPredictionFromPython() {
  std::string liveInfoJson = "{";
  for (const auto &[key, value] : _currentLiveInfo) {
    liveInfoJson += "\"" + key + "\":" + std::to_string(value) + ",";
  }
  liveInfoJson.back() = '}';

  PyObject *firstArg = PyUnicode_FromString(_modelFileName.c_str());
  PyObject *secondArg = PyUnicode_FromString(liveInfoJson.c_str());
  PyObject *pArgs = PyTuple_Pack(2, firstArg, secondArg);

  PyObject *pResult = PyObject_CallObject(_pFunc, pArgs);

  Py_DECREF(firstArg);
  Py_DECREF(secondArg);
  Py_DECREF(pArgs);

  if (!pResult) {
    PyErr_Print();
    utils::ExceptionHandler::exception("Error␣occurred␣during␣Python␣
        function␣call");
  }

  const char *prediction = PyUnicode_AsUTF8(pResult);
  std::string configPrediction(prediction);
  Py_DECREF(pResult);

  return configPrediction;
}
```

Listing A.6: `getPredictionFromPython` Function

The `getPredictionFromPython` function constructs a JSON string representation of the current live information, `_currentLiveInfo`, to send it as an argument to the Python

function. It iterates over the `_currentLiveInfo` map and formats each key-value pair as a JSON entry. The function then prepares two arguments for the Python `main` function in `predict.py`: the model file name and the JSON string of live information. It creates a Python tuple `pArgs` containing these arguments and calls `_pFunc` with them. If the function call succeeds, it extracts the result, which is a JSON string of predictions, converts it to a C++ string, and returns it. Any errors during the function call are printed, and an exception is raised if necessary.

A.3. `.yaml` Config Files

In this section, we will go over the `.yaml` files used during the data collection process, and which of the parameters in these files were varied to cover the target space as much as possible to train a generalizable model. For the example scenarios included within `md-flexible`, `fallingDrop.yaml` and `explodingLiquid.yaml`, we will only discuss which parameters were varied to create a dataset covering the target space.

A.3.1. Varied Parameters

Here, we will discuss the parameters adjusted in order to build the dataset and train the model. `md-flexible`, the simulation included in AutoPas, can be adjusted to run different scenarios using `.yaml` files. These configuration files have a set of parameters to adjust simulation parameters and define particles to run the simulation. Below is a list of these parameters adjusted to create different scenarios that auto-tune to different configurations to cover the target space:

- `cutoff`: Lennard-Jones force cutoff.
- `particle-spacing`: Space between two particles for the grid generator. The `particle-spacing` parameter is changed in the range of $[1/5 * \text{cutoff}, \text{cutoff}]$.
- `deltaT`: Length of a timestep. Set to zero to deactivate time integration. This value is kept bigger than zero to enable time integration during the data collection phase.

A.3.2. `cubeGauss.yaml`

In this configuration, we additionally use different mean and standard deviation parameters of the `cubeGauss` object to run simulations with different homogeneity to build our dataset.

```
container                          :  # See AllOptions.yaml
verlet-rebuild-frequency           :  10
verlet-skin-radius-per-timestep    :  0.05
verlet-cluster-size                :  4
data-layout                        :  [AoS, SoA]
traversal                          :  # See AllOptions.yaml
tuning-strategies                  :  []
tuning-interval                    :  2500
tuning-samples                     :  3
tuning-max-evidence                :  10
functor                            :  Lennard-Jones AVX
newton3                            :  [disabled, enabled]
cutoff                             :  3
box-min                            :  [0, 0, 0]
box-max                            :  [7.25, 7.25, 7.25]
cell-size                          :  [1]
deltaT                             :  0.000001
pause-simulation-during-tuning     :  true
iterations                         :  15000
boundary-type                      :  [none, none, none]
globalForce                        :  [0, 0, 0]
Objects:
  CubeGauss:
    0:
      distribution-mean            : [ 17, 17, 17 ]
      distribution-stddeviation    : [ 2, 2, 2 ]
      numberOfParticles            : 100
      box-length                   : [ 8, 8, 8 ]
      bottomLeftCorner             : [ 15, 15, 15 ]
      velocity                     : [ 0, 0, 0 ]
      particle-type-id             : 0
```

Listing A.7: A minimized version of the `cubeGauss.yaml` file

A.3.3. `cubeUniform.yaml`

In `cubeUniform.yaml`, a similar procedure as in `cubeGauss` is used to vary the parameters to cover the target space, but we change the `numberOfParticles` parameter to

change the density of the simulation instead:

```
Objects:
  CubeUniform:
    0:
      numberOfParticles          : 100
      box-length                 : [ 8, 8, 8 ]
      bottomLeftCorner           : [ 15, 15, 15 ]
      velocity                   : [ 0, 0, 0 ]
      particle-type-id           : 0
```

Listing A.8: `cubeUniform` Object

A.4. Logging

A.4.1. `liveInfoLogger` Fields

The `LiveInfoData` file currently includes fields that provide summary statistics about the simulation state at each iteration. This file, generated by the `LiveInfoLogger` class in the AutoPas library, logs key data for the decision tree-based tuning strategy, serving as its main source of information to make predictions to adjust the algorithm configuration throughout the simulation. Refer to AutoPas Documentation for more details. It contains the following fields:

- **Date**: The timestamp of the time the data was collected.
- **Iteration**: The current iteration number of the simulation.
- **avgParticlesPerCell**: The average number of particles per cell in the simulation.
- **cutoff**: The cutoff radius for force potential calculation.
- **domainSizeX**: The domain size in the X dimension.
- **domainSizeY**: The domain size in the Y dimension.
- **domainSizeZ**: The domain size in the Z dimension.
- **estimatedNumNeighborInteractions**: Rough estimation of the number of neighbor interactions assuming that neighboring cells contain roughly the same number of particles.
- **homogeneity**: A measure of the homogeneity of particles across the cells.

- **maxDensity**: The maximum density of particles in any cell.
- **maxParticlesPerCell**: The maximum number of particles in a cell.
- **minParticlesPerCell**: The minimum number of particles in a cell.
- **numCells**: The total number of cells in the simulation domain.
- **numEmptyCells**: The number of cells with no particles.
- **numHaloParticles**: The number of particles outside of the simulation domain.
- **numParticles**: The total number of particles in the container.
- **particleSize**: The number of bytes a single particle in memory occupies in AoS layout.
- **particleSizeNeededByFunctor**: The number of bytes the information needed by the functor from each particle occupies. Important for the SoA data layout.
- **particlesPerBlurredCellStdDev**: The standard deviation of the number of particles in each blurred cell (1/27th of the domain)
- **particlesPerCellStdDev**: The standard deviation of the number of particles in each cell.
- **rebuildFrequency**: The current verlet-rebuild-frequency of the simulation.
- **skin**: The configured skin radius that is added to the cutoff radius to create a buffer for neighbor lists.
- **threadCount**: The number of threads used for parallel processing.

A.4.2. `tuningResults` Fields

This logging option includes fields that provide information about the tuning results produced throughout the simulation runtime. The columns in this CSV file are mainly used as the target for the model's training. It contains the following fields:

- **Date**: The timestamp of the time the data was collected.
- **Iteration**: The current iteration number of the simulation.
- **Container**: The container used to store the particles during the simulation (e.g., LinkedCells, VerletLists)

- **CellSizeFactor**: Factor to determine the size of the cells relative to the cutoff radius.
- **Traversal**: Traversal method used to calculate interactions (e.g., C01, C18, C08) between particles
- **Load Estimator**: Strategy used to balance the load across different parts (e.g. `squaredParticlesPerCell`, `neighborListLength`
- **Data Layout**: Indicates how the particles are arranged in memory (e.g. AoS, SoA)
- **Newton 3**: Whether the optimization using Newton's 3rd Law is used or not for force calculation.

A.5. Numerical Values for the Plots in 6

A.5.1. Numerical Values for the Falling Drop Benchmark

Total Time Plot

	Decision Tree	Fuzzy Tuning	Predictive	Full Search
Tuning Phase (s)	1.61	2.25	22.37	40.88
Simulation Phase (s)	230.11	230.16	241.85	230.54

Table A.1.: Execution times for 6.1

Percentage Time Plot

	Decision Tree	Fuzzy Tuning	Predictive	Full Search
Tuning Phase (s)	0.07	0.09	8.46	15.06
Simulation Phase (s)	99.3	99.1	91.54	84.94

Table A.2.: Percentage of execution times for 6.2

A.5.2. Numerical Values for the Exploding Liquid Benchmark

Total Time Plot

	Decision Tree (in dataset)	Fuzzy Tuning	Decision Tree (not in dataset)	Predictive	Full Search
Tuning Phase (s)	1.26	2.25	1.39	15.38	35.89
Simulation Phase (s)	51.69	53.42	59.84	56.88	52.59

Table A.3.: Execution times for 6.3

Percentage Time Plot

	Decision Tree (in dataset)	Fuzzy Tuning	Decision Tree (not in dataset)	Predictive	Full Search
Tuning Phase (s)	2.38	4.05	2.27	21.28	40.56
Simulation Phase (s)	97.62	95.95	97.73	78.72	59.44

Table A.4.: Percentage of execution times for 6.4

List of Figures

List of Tables

Listings

Bibliography

[Aba+15] M. Abadi, A. Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Y. Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015.

[Abr+24] M. Abraham, A. Alekseenko, V. Basov, C. Bergh, E. Briand, A. Brown, M. Doijade, G. Fiorin, S. Fleischmann, S. Gorelov, G. Gouaillardet, A. Grey, M. E. Irrgang, F. Jalalypour, J. Jordan, C. Kutzner, J. A. Lemkul, M. Lundborg, P. Merz, V. Miletic, D. Morozov, J. Nabet, S. Pall, A. Pasquadibisceglie, M. Pellegrino, H. Santuz, R. Schulz, T. Shugaeva, A. Shvetsov, A. Villa, S. Wingbermuehle, B. Hess, and E. Lindahl. *GROMACS 2024.3 Manual*. Version 2024.3. Zenodo, Aug. 2024. DOI: 10.5281/zenodo.13457083.

[Bre+84] L. Breiman, J. Friedman, C. Stone, and R. Olshen. *Classification and Regression Trees*. Taylor & Francis, 1984. ISBN: 9780412048418.

[Bre00] D. Brenner. "The Art and Science of an Analytic Potential." In: *physica status solidi (b)* 217.1 (2000), pp. 23–40. DOI: https://doi.org/10.1002/(SICI)1521-3951(200001)217:1<23::AID-PSSB23>3.0.CO;2-N. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/%28SICI%291521-3951%28200001%29217%3A1%3C23%3A%3AAID-PSSB23%3E3.0.CO%3B2-N.

[Bun+13] H. Bungartz, S. Zimmer, M. Buchholz, D. Pflüger, S. Borne, and R. Borne. *Modeling and Simulation: An Application-Oriented Introduction*. Springer Undergraduate Texts in Mathematics and Technology. Springer Berlin Heidelberg, 2013. ISBN: 9783642395246.

[Can19] D. Candas. "Auto-Tuning via Machine Learning in AutoPas." en. MA thesis. Technical University of Munich, Sept. 2019.

[Che+18] S. Chen, J. Fang, C. Donglin, C. Xu, and Z. Wang. "Optimizing Sparse Matrix-Vector Multiplication on Emerging Many-Core Architectures." In: (May 2018). DOI: 10.48550/arXiv.1805.11938.

[Com07] W. Commons. *12-6 Lennard-Jones Potential*. 2007. URL: https://upload.wikimedia.org/wikipedia/commons/5/51/12-6-Lennard-Jones-Potential.svg.

[Fis20] V. Fischer. "Implementation and Analysis of Load Balancing Options for AutoPas' Sliced Traversal." en. MA thesis. Technical University of Munich, Sept. 2020.

[Fou20] P. S. Foundation. *Python/C API Reference Manual*. Python 2.7.18 Documentation. 2020.

[Gel+14] A. Gelzinis, A. Verikas, E. Vaiciukynas, M. Bacauskiene, J. Minelga, M. Hallander, V. Uloza, and E. Padervinskis. "Exploring sustained phonation recorded with acoustic and contact microphones to screen for laryngeal disorders." In: Dec. 2014, pp. 125–132. DOI: 10.1109/CICARE.2014.7007844.

[Gra+22] F. A. Gratl, S. Seckler, H.-J. Bungartz, and P. Neumann. "N ways to simulate short-range particle systems: Automated algorithm selection with the node-level library AutoPas." In: *Computer Physics Communications* 273 (2022), p. 108262. ISSN: 0010-4655. DOI: https://doi.org/10.1016/j.cpc.2021.108262.

[GSB16] D. Grzonka, G. Suchacka, and B. Borowik. "Application of Selected Supervised Classification Methods to Bank Marketing Campaign." In: *Information Systems in Management* 5 (June 2016), pp. 36–48.

[Her] T. Hermann. *frugally-deep*.

[Ho95] T. K. Ho. "Random decision forests." In: *Proceedings of 3rd International Conference on Document Analysis and Recognition*. Vol. 1. 1995, 278–282 vol.1. DOI: 10.1109/ICDAR.1995.598994.

[Ho98] T. K. Ho. "The random subspace method for constructing decision forests." In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 20.8 (1998), pp. 832–844. DOI: 10.1109/34.709601.

[HS12] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Morgan Kaufmann, 2012. ISBN: 9780123977953.

[HTF09] T. Hastie, R. Tibshirani, and J. Friedman. *The elements of statistical learning: data mining, inference and prediction*. 2nd ed. Springer, 2009.

[Hum23] T. Humig. "Project Report: Exploring Performance Modeling in AutoPas." en. MA thesis. Technical University of Munich, Oct. 2023.

[Kot07] S. B. Kotsiantis. "Supervised Machine Learning: A Review of Classification Techniques." In: *Proceedings of the 2007 Conference on Emerging Artificial Intelligence Applications in Computer Engineering: Real Word AI Systems with Applications in EHealth, HCI, Information Retrieval and Pervasive Technologies*. NLD: IOS Press, 2007, pp. 3–24. ISBN: 9781586037802.

[Kur21] L. Kury. "Automated Selection of Scheduling Techniques in OpenMP using Reinforcement Learning." MA thesis. University of Basel, Sept. 2021.

[Lau22] L. Laumeyer. "Can Reinforcement Learning be used to improve the auto-tuning process within AutoPas?" en. MA thesis. Technical University of Munich, Sept. 2022.

[Len31] J. E. Lennard-Jones. "Cohesion." In: *Proceedings of the Physical Society* 43.5 (Oct. 1931), p. 461. DOI: 10.1088/0959-5309/43/5/301.

[Ler24] M. Lerchner. "Exploring Fuzzy Tuning Technique for Molecular Dynamics Simulations in AutoPas." en. MA thesis. Technical University of Munich, Aug. 2024.

[Mur13] K. P. Murphy. *Machine learning : a probabilistic perspective*. Cambridge, Mass. [u.a.]: MIT Press, 2013. ISBN: 9780262018029 0262018020.

[New+23] S. J. Newcome, F. A. Gratl, P. Neumann, and H.-J. Bungartz. "Towards auto-tuning Multi-Site Molecular Dynamics simulations with AutoPas." In: *Journal of Computational and Applied Mathematics* 433 (2023), p. 115278. ISSN: 0377-0427. DOI: https://doi.org/10.1016/j.cam.2023.115278.

[Ngu20] J. Nguyen. "Mixed discrete-continuous Bayesian Optimization for Auto-Tuning." en. MA thesis. Technical University of Munich, Oct. 2020.

[NPM99] V. Novák, I. Perfilieva, and J. Mockor. *Mathematical Principles of Fuzzy Logic*. The Springer International Series in Engineering and Computer Science. Springer US, 1999. ISBN: 9780792385950.

[pan24] pandas documentation contributors. *pandas.DataFrame*. Accessed: 2024-11-09. pandas Development Team. 2024.

[Pel20] J. M. Pelloth. "Implementing a predictive tuning strategy in AutoPas using extrapolation." en. MA thesis. Technical University of Munich, Sept. 2020.

[PH13] S. Páll and B. Hess. "A flexible algorithm for calculating pair interactions on SIMD architectures." In: *Computer Physics Communications* 184.12 (2013), pp. 2641–2650. ISSN: 0010-4655. DOI: https://doi.org/10.1016/j.cpc.2013.06.003.

[Sec+21] S. Seckler, F. Gratl, M. Heinen, J. Vrabec, H.-J. Bungartz, and P. Neumann. "AutoPas in ls1 mardyn: Massively parallel particle simulations with node-level auto-tuning." In: *Journal of Computational Science* 50 (2021), p. 101296. ISSN: 1877-7503. DOI: https://doi.org/10.1016/j.jocs.2020.101296.

[SR20] O. Sagi and L. Rokach. "Explainable Decision Forest: Transforming a decision forest into an interpretable tree." In: *Information Fusion* 61 (Mar. 2020). DOI: 10.1016/j.inffus.2020.03.013.

[SW23] C. Stylianou and M. Weiland. *Optimizing Sparse Linear Algebra Through Automatic Format Selection and Machine Learning*. 2023. arXiv: 2303.05098 [cs.LG].

[Tho+22] A. P. Thompson, H. M. Aktulga, R. Berger, D. S. Bolintineanu, W. M. Brown, P. S. Crozier, P. J. in 't Veld, A. Kohlmeyer, S. G. Moore, T. D. Nguyen, R. Shan, M. J. Stevens, J. Tranchida, C. Trott, and S. J. Plimpton. "LAMMPS - a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales." In: *Comp. Phys. Comm.* 271 (2022), p. 108171. DOI: 10.1016/j.cpc.2021.108171.

[TKV08] G. Tsoumakas, I. Katakis, and I. Vlahavas. "Effective and efficient multilabel classification in domains with large number of labels." In: (Jan. 2008).

[VBC08] G. Viccione, V. Bovolin, and E. P. Carratelli. "Defining and optimizing algorithms for neighbouring particle identification in SPH fluid simulations." In: *International Journal for Numerical Methods in Fluids* 58.6 (2008), pp. 625–638. DOI: https://doi.org/10.1002/fld.1761. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/fld.1761.

[Ver67] L. Verlet. "Computer "experiments" on classical fluids. I. Thermodynamical properties of Lennard-Jones molecules." In: *Physical Review* 159.1 (1967). Cited by: 7750; All Open Access, Bronze Open Access, pp. 98–103. DOI: 10.1103/PhysRev.159.98.

[Wan+24] H. Wang, S. Yu, C. Chen, B. Turhan, and X. Zhu. "Beyond Accuracy: An Empirical Study on Unit Testing in Open-source Deep Learning Projects." In: *ACM Trans. Softw. Eng. Methodol.* 33.4 (Apr. 2024). ISSN: 1049-331X. DOI: 10.1145/3638245.

[Wil+17] T. Wilde, M. Ott, A. Auweter, I. Meijer, P. Ruch, M. Hilger, S. Kuhnert, and H. Huber. "CooLMUC-2: A supercomputing cluster with heat recovery for adsorption cooling." In: Jan. 2017, pp. 115–121. DOI: 10.1109/SEMI-THERM.2017.7896917.

[YR10] S. Yip and T. Rubia. *Scientific Modeling and Simulations*. Lecture Notes in Computational Science and Engineering. Springer Netherlands, 2010. ISBN: 9781402097416.