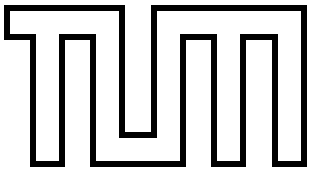


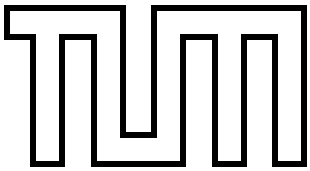
SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Exploring Fuzzy Tuning Technique for Molecular Dynamics Simulations in AutoPas

Manuel Lerchner



SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Exploring Fuzzy Tuning Technique for Molecular Dynamics Simulations in AutoPas

Remove all TODOS

Untersuchung von Fuzzy Tuning Verfahren für Molekulardynamik-Simulationen in AutoPas

Author:	Manuel Lerchner
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisors:	Manish Kumar Mishra, M.Sc. &
	Samuel Newcome, M.Sc.
Date:	10.08.2024

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

Munich, 10.08.2024 Manuel Lerchner

Acknowledgements

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Abstract

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Zusammenfassung

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Contents

1. Introduction

Write some useful intro. Here are tips along the way:

1.1. A

2. Theoretical Background

2.1. Molecular Dynamics

Molecular Dynamics (MD) is a computational method used to simulate the behavior of atoms and molecules over time. In recent years, MD simulations have become essential in many scientific fields, including chemistry, physics, biology, and materials science. Such simulations are used to study various systems, ranging from simple gases and liquids to complex biological molecules and new materials.

MD simulations act on an atomic level and attempt to explain macroscopic properties of a system from the interactions between the individual atoms and molecules. The recent advances in computational power have made it possible to simulate systems with millions of particles over long time scales, allowing researchers to study complex systems in unprecedented detail. Contrary to experimental methods, MD simulations can provide detailed information about the behavior of atoms and molecules, sometimes inaccessible to experimental methods [PS17].

Two illustrations of such simulations are shown in Figure 2.1 and Figure 2.2. The first image shows a simulation of the HIV-1 capsid, a protein shell that surrounds the genetic material of the human immunodeficiency virus (HIV). Using a simulation-based approach, researchers could study critical properties of the HIV-1 capsid, which would be difficult to access using other methods [PS17]. The second image shows a simulation of shear band formation around a precipitate in metallic glass. This simulation found evidence that depending on the precipitate size, shear bands can either dissolve, wrap around, or be blocked by the precipitate [BPR$^+$16]. This information is crucial for understanding the mechanical properties of metallic glasses and can be used to design new materials with improved properties.

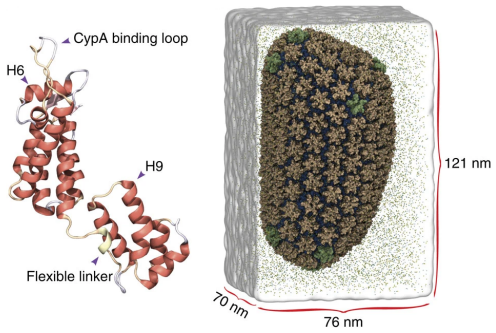


Figure 2.1.: MD simulation with 64,423,983 atoms of the HIV-1 capsid. Perilla et al. [PS17] investigated properties of the HIV-1 capsid at an atomic resolution.

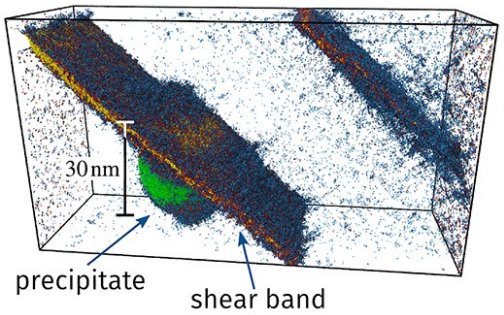


Figure 2.2.: MD simulations of shear band formation around a precipitate in metallic glass, as demonstrated by Brink et al. [BPR$^+$16].

2.1.1. Quantum Mechanial Background

Our current knowledge of physics suggests that the behavior of atoms and molecules is governed by the laws of quantum mechanics, where particles are described by probabilistic wave functions evolving over time. In 1926, Austrian physicist Erwin Schrödinger formulated a mathematical model describing this concept, which has since gained widespread acceptance and is now generally known as the Schrödinger equation. The Schrödinger equation is a partial differential equation describing the time evolution of a quantum system and is given by:

$$i\hbar\frac{\partial\Psi(\vec{r},t)}{\partial t} = \hat{H}\Psi(\vec{r},t) \tag{2.1}$$

Where $\Psi(\vec{r},t)$ is the system's wave function, evolving over time t and space $\vec{r}$. $\hat{H}$ is the Hamiltonian operator describing the system's energy, t is the time, and $\hbar$ is the reduced Planck constant.

The Schrödinger equation provides a way to calculate the future states of a quantum system given the system's current state. However, the computational complexity of solving this equation increases dramatically with the number of particles involved and quickly becomes infeasible for systems with more than a few particles [LM15]. To illustrate this complexity, consider simulating a single water molecule. This molecule consists of three nuclei (two hydrogen atoms and one oxygen atom) and 10 electrons. Each of these 13 objects requires three spatial coordinates to describe its position, resulting in a total of $(2+1+10)\times 3 = 39$ variables. The Schrodinger equation for this single water molecule can therefore be written as:

$$i\hbar\frac{\partial\Psi}{\partial t} = -\hbar^2\sum_{i=1}^{13}\frac{1}{2m_i}\left(\frac{\partial^2\Psi}{\partial x_i^2}+\frac{\partial^2\Psi}{\partial y_i^2}+\frac{\partial^2\Psi}{\partial z_i^2}\right)+U_p(x_1,y_1,z_1,\ldots,x_{13},y_{13},z_{13})\Psi \tag{2.2}$$

In this equation, m_i is the mass of the i-th object, x_i, y_i, and z_i are the spatial coordinates of the i-th object, and U_p is the potential energy function of the system.

As the Schrödinger equation is a partial differential equation, it is computationally expensive to solve for systems with many particles, as one quickly runs into the curse of dimensionality. Larger systems, such as the HIV-1 capsid shown in Figure 2.1 consisting of millions of atoms, are practically impossible to simulate using the Schrödinger equation directly.

Luckily, the Born-Oppenheimer approximation simplifies the Schrödinger equation so that it becomes computationally feasible to simulate even large systems of particles. The approximation exploits the significant mass difference between electrons and nuclei[1], making it possible to solve both motions independently [ZBB+13]. As the forces acting on the heavy nuclei cause way slower movements compared to the same force acting on the electrons, it is possible to approximate the position of the nucleus as entirely stationary. This simplification yields a new potential energy function U combining all electronic and nuclear energies, which depends only on the nuclei's positions. The potential energy function U fundamentally

[1]The mass ratio of a single proton to an electron is approximately 1836:1, illustrating the vast difference in mass between nuclei and electrons.

controls the motions of the nuclei [ZBB+13] and is typically obtained through quantum mechanical calculations or fitted to empirical data [Iri21].

As the Born-Oppenheimer approximation is based on simplifications of the full model, it is not always accurate. Depending on the system under investigation and the chosen potential energy function U, the Born-Oppenheimer approximation may neglect specific quantum mechanical effects, resulting in inaccuracies in the simulation.

Despite these limitations, the Born-Oppenheimer approximation is widely used in molecular dynamics simulations and is the best-known method to simulate systems with many particles.

2.1.2. Classical Molecular Dynamics

After applying the Born-Oppenheimer approximation and using Newton's second law of motion, the Schrödinger equation can be transformed into a system of ordinary differential equations of the form:

$$m_i \frac{d^2 \vec{r}_i}{dt^2} = -\nabla_i U \tag{2.3}$$

Where m_i is the mass of the i-th particle, $\vec{r}_i$ is the position of the i-th particle, and U is the potential energy function of the system. These equations precisely describe a classical particle system, where particles are treated as point masses moving through space under the influence of forces. The forces are derived from the potential energy function U and are calculated using the negative gradient of the potential energy function $\nabla_i U$.

2.1.3. Potential Energy Function

As stated above, the potential energy function U is a critical component of molecular dynamics simulations as it fundamentally defines the properties of the system. MD simulations use many different potential energy functions, all of which are tailored to describe specific aspects of the system. Those potentials typically use a mixture of 2-body, 3-body, and 4-body interactions between the particles, each used to describe different aspects of particle interactions. The 2-body interactions typically express the effect of Pauli repulsion, atomic bonds, and coulomb interactions, while higher-order interactions allow for asymmetric wave functions for atoms in bound-groups [LM15].

A common choice for the potential energy function is the Lennard-Jones potential. This potential can reproduce the potential energy surfaces of many biological systems [Phy] while still being very simple and efficient to compute. It mainly emulates the attractive Van-der-Waals forces and the repulsive Pauli repulsion forces between the particles [Che].

The Lennard-Jones potential is given by:

$$U_{LJ}(r) = 4\epsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right] \tag{2.4}$$

Where r is the distance between the particles, ϵ is the depth of the potential well, and σ is the distance at which the potential is zero. The parameters ϵ and σ can differ for each type of particle interaction and are either determined from theoretical considerations of the material or chosen to match experimental data [MAMM20].

2.1.4. Numerical Integration

Since the simulation domain potentially consists of a vast number of particles all interacting with each other, it is generally not possible to solve the equations of motion analytically. This problem is known under the N-body problem, and it can be shown that there are no general solutions for systems with more than two particles. It is, however, possible to approximate solutions of these equations of motion using numerical integration methods. A standard method for solving such systems is the Verlet algorithm. This integration scheme is derived from the Taylor expansion of the position of the i-th object $\vec{r}_i$ at time $t - \Delta t$ and $t + \Delta t$ and is given by:

$$\vec{r}_i(t + \Delta t) = 2\vec{r}_i(t) - \vec{r}_i(t - \Delta t) + \vec{a}_i(t)\Delta t^2 \tag{2.5}$$

Where $\vec{a}_i(t)$ is the acceleration of the i-th object at time t, this acceleration can be calculated from the particle mass and the acting forces using Newton's second law of motion $\vec{a}_i(t) = \frac{F_i}{m_i} = \frac{-\nabla_i U}{m_i}$.

2.1.5. Simulation Loop

Using the methods described above, it is possible to simulate the behavior of a system of particles over time. The general simulation loop for a molecular dynamics simulation can be described as follows:

1. **Initialization**
   The simulation starts by initializing the positions and velocities of the particles. The initial positions and velocities can be chosen randomly or based on experimental data, depending on the system under investigation.
2. **Position Updates**
   In this step, the positions of the particles are updated using a numerical integration scheme such as the Verlet algorithm.
3. **Force Calculation**
   The forces acting on the particles are calculated based on the current positions of the particles and the chosen potential energy function U.
4. **Acceleration and Velocity Updates**
   The acceleration of each particle is calculated using Newton's second law of motion and the forces acting on the particles. The velocities of the particles are then updated accordingly.
5. **External Forces and Constraints**
   In this step, the simulation can be modified by applying external forces or constraints to the particles. For example, it is possible to introduce boundary conditions, temperature control, or other outside influences to the simulation.
6. **Update Time and Repeat**
   The simulation time is updated, and the simulation loop (steps 2-5) are repeated until the desired simulation time is reached. The collected data can then be analyzed to study properties of the system.

Many different software packages exist to perform such simulations. Some widely used examples of such systems are LAAMPS[2] and GROMACS[3]. Both attempt to efficiently solve the underlying N-body problem and provide the user with a high-level interface to specify the parameters and properties of the simulation.

Many different approaches exist to efficiently solve the N-body problem, and no single best approach works well for all systems as the optimal implementation heavily depends on the simulation state of the hardware used to perform the simulation. However, LAAMPS and GROMACS use a single implementation and cannot adapt their algorithms to the current simulation state.

In the following section, we will introduce AutoPas, a library designed to efficiently deal with evolving particle simulations, leveraging the idea of automatically switching between different implementations to achieve the best performance for the current simulation state.

2.2. AutoPas

AutoPas is an open-source library designed to achieve optimal node-level performance for short-range particle simulations. On a high level, AutoPas can be seen as a black box performing arbitrary N-body simulations with short-range particle interactions. However, AutoPas differentiates itself from other libraries by providing many algorithmic implementations for the N-body problem, each with different performance and memory usage trade-offs. No implementation is optimal for all simulation scenarios as the optimum can change over time **??**. Consequently, AutoPas is designed to be adaptive and can periodically switch between different implementations to remain reasonably close to the current optimal configuration.

Since AutoPas provides a high-level interface for short-range N-body simulations, the user must specify the desired model and simulation parameters. Fortunately, AutoPas also provides some example implementations, such as `md_flexible`. `md_flexible` is a simple molecular dynamics framework built on top of AutoPas that allows users to specify the desired simulation parameters and run simulations. This work will primarily focus on `md_flexible`, but the concepts can be easily transferred to other simulation frameworks.

2.2.1. Autotuning in AutoPas

AutoPas internally alternates between two phases of operation. The first phase is the *tuning phase*, where AutoPas tries to find the best configuration of parameters that minimize a chosen performance metric (e.g., time, energy usage) for the current simulation state. This is achieved by trying out different configurations of parameters and measuring their performance. The configuration that optimizes the chosen performance metric is then used in the following *simulation phase*, assuming that the optimal configuration found in the tuning phase still performs reasonably well during this phase. As the simulation progresses and the characteristics of the system change, the previously chosen configuration can drift arbitrarily far from the actual optimal configuration. To counteract this, AutoPas periodically alternates

[2] `https://lammps.sandia.gov/`
[3] `https://www.gromacs.org/`

between tuning and simulation phases to ensure that the used configuration remains close to optimal.

The power of AutoPas comes from its vast amount of tunable parameters and the enormous search space associated with them. Other software packages, such as LAAMPS and GROMACS, are limited to just one implementation and can operate outside the theoretically achievable performance regime.

In the following section, we will discuss the tunable parameters in AutoPas and the different tuning strategies available to find the best configuration of parameters for the current simulation state.

2.2.2. Tunable Parameters

AutoPas currently provides six tunable parameters, which can mostly[4] be combined freely. A collection of parameters is called a *Configuration*, and the set of all possible configurations is called the *Search Space*. Each configuration consists of the following parameters:

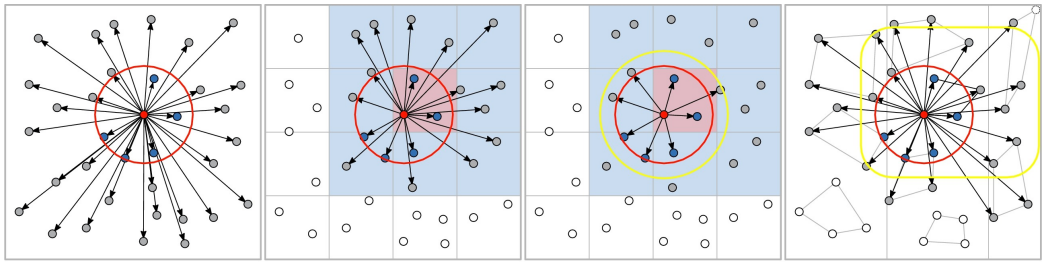
1. **Container Options:**
   The container options are related to the data structure used to store the particles. The most important categories of data structures in this section are:
   a) **DirectSum**
   DirectSum does not use any additional data structures to store the particles. Instead, it simply holds a list of all particles. Consequently, it needs to rely on brute-force calculations of the forces between all pairs of particles. This results in a complexity of $O(N^2)$ distance checks in each iteration. This inferior complexity renders it completely useless for larger simulations.
   Generally should not be used except for tiny systems or demonstration purposes. [VBC08]
   b) **LinkedCells**
   LinkedCells segments the domain into a regular cell grid and only considers interactions between particles from neighboring cells. This results in the trade-off that particles further away are not considered for the force calculation. In practice, this is not a big issue, as all short-range forces drop off quickly with distance anyway. LinkedCells also provides a high cache hit rate as particles inside the same cell can be stored contiguously in memory. Typically, the cell size is chosen to equal the force cutoff radius r_c, meaning each particle only needs to check interactions between particles inside the $3 \times 3 \times 3$ cell grid around the current cell. All other particles are guaranteed to be further away than the cutoff radius and, therefore, cannot contribute to the acting forces. This reduction in possible interactions can result in a complexity of just $O(N)$ distance checks in each iteration if the particles are spread evenly. However, there is room for improvement as the constant overhead factor can be pretty high, as most distance checks performed by LinkedCells still do not contribute to the force calculation. Due to the uneven scaling of sphere and cube volumes, only about 15.5% of all particles present in the $3 \times 3 \times 3$ cell grid around a particle are within the cutoff

[4]There are some exceptions as some choices of parameters are incompatible with each other.

radius [GST+19].
However, still generally good for large, homogeneous[5] systems.

c) **VerletLists**
VerletLists are another approach to creating neighbor lists for the particles. Contrary to LinkedCells, VerletLists does not rely on a regular grid but instead uses a spherical region around each particle to determine its relevant neighbors. The algorithm creates and maintains a list of all particles present in a sphere within radius $r_c \cdot s$ around each particle, where r_c is the cutoff radius and $s > 1$ is the skin factor allowing for a buffer zone around the cutoff radius. By choosing a suitable buffer zone, such that no fast-moving particle can enter the cutoff radius unnoticed, it is possible to only recalculate the neighbor list every few iterations. This method also results in a complexity of $O(N)$ distance checks in each iteration but provides a higher interaction rate between particles of $1/s^3$. Ideally, the skin factor should be chosen so the ratio is close to 1, resulting in no unnecessary distance checks. This, however, reduces the buffer zone around the cutoff radius, meaning that the neighbor list needs to be updated more frequently, or the simulation needs to be run at higher temporal precision to remain accurate. Finding a good balance between the two is therefore of great importance.
Generally good for large systems with high particle density.

d) **VerletClusterLists**
VerletClusterLists differ from regular VerletLists in the way the neighbor lists are stored. Instead of storing the neighbor list for each particle separately, $n_{cluster}$ particles are grouped into a so-called *cluster*, and a single neighbor list is created for each cluster. This reduces memory overhead as the neighbor list only needs to be stored once for each cluster. Whenever two clusters are close, all interactions between the particles in the two clusters are calculated. This also results in a complexity of $O(N)$ distance checks in each iteration but provides the advantage of greatly reduced memory usage compared to regular VerletLists.
Generally suitable for large systems with high particle density



(a) Direct Sum (b) Linked Cells (c) Verlet Lists (d) Verlet Cluster Lists

Figure 2.3.: Visualization of different container options. Source: Gratl et al. [GSBN21]

2. Load Estimator Options:
The Load Estimator Options relate to how the simulation behaves in a parallelized setting. The load estimator estimates each MPI rank's computational load and guides

[5]Homogeneous in this context, the particles are distributed evenly across the domain.

the simulation's load balancing. In this thesis, however, we will not further describe the Load Estimator Options as we primarily focus on the tuning aspect of single-node simulations.

3. **Traversal Options:**
These options are related to the traversal algorithm used to calculate the forces between the particles. The traversal determines the order in which the particles are visited and how the forces are applied to the particles. Furthermore, traversal options should prevent race conditions when using multiple threads and can automatically provide load balancing at the node level [SGH+21].

Some avaialble traversal options are:

a) **Sliced Traversal**
Sliced Traversal is a way to parallelize the force calculation by dividing the domain into different slices and assigning each to a different thread. When the slices are chosen correctly, no two threads can work on cells with common neighbors, allowing for the force calculation to be parallelized easily, even when using Newton's third law. However, on boundaries, the threads need to be synchronized using locks [GSBN21] to prevent data races.

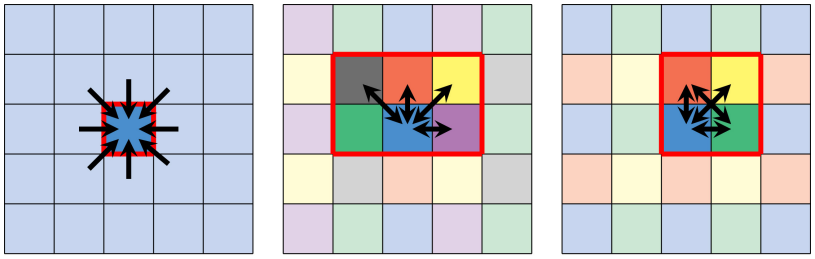
b) **Colored Traversal**
Since both LinkedCells and VerletLists only consider interactions with particles from neighboring cells/particles, it is possible to parallelize the force calculation by calculating forces for particles in different cells in parallel. However, when using Newton's third law, this is only possible if all simultaneously calculated particles do not share familiar neighbors, as updating familiar neighbors could introduce data races when multiple threads act simultaneously. This is where the concept of coloring comes into play. Coloring is a way to assign a color to each cell so that cells with the same color do not share familiar neighbors. This allows for the force calculation of particles in cells with the same color to be parallelized trivially, as data races are impossible. Some ways to color the domain are:

- **C01**
The C01 traversal uses no coloring and hands all cells to currently available threads. This method is embarrassingly parallel but comes at the cost of being incompatible with the Newton 3 optimization, as there is no way of preventing data races. As a result, the forces between all pairs of particles are calculated twice, once for each particle. This method results in a constant overhead of factor 2.

- **C18**
The C18 traversal is a more sophisticated way of coloring the domain. The domain is divided into 18 colors, so no two neighboring cells share the same color. This method also utilizes the Newton 3 law to reduce the number of force calculations. This is achieved by only computing the forces with forward neighbors (neighbors with greater index.) [GSBN21]

- **C08**
The C08 traversal is closely related to the C18 traversal but only uses eight

colors. Due to the fewer colors, there is less synchronization overhead, and a higher degree of parallelism can be achieved. Newton 3 can be used to reduce the number of force calculations



(a) C01 (b) C18 (c) C08

Figure 2.4.: Visualization of different color-based traversal options. Source: Newcome et al. [NGNB23]

4. **Data Layout Options:**
   The Data Layout Options determine how the particles are stored in memory. The two possible data layouts are:

   a) **SoA**
   The SoA (Structure of Arrays) data layout stores the particles' properties in separate arrays. For example, all particles' x-,y- and z-coordinates are stored in separate arrays. This data layout is beneficial for vectorization as the properties of the particles are stored contiguously in memory. This allows for efficient vectorization of the force calculations as the properties of the particles can be loaded into vector registers in a single instruction.

   b) **AoS**
   The AoS (Array of Structures) data layout stores all particle properties in different structures. This allows for efficient cache utilization when working on particles, as all properties are close to each other in memory. However, this data layout is not beneficial for vectorization as the same properties of different particles are not stored contiguously in memory. Consequently, they need to be loaded into vector registers individually, which can result in inefficient vectorization of the force calculations.

5. **Newton 3 Options:**
   The Newton 3 Options relate to how the forces between the particles are calculated. Newton's third law states that for every action, there is an equal and opposite reaction, which means that the magnitude of the force between two particles is the same, regardless of which particle is the source and which is the target. In Molecular Dynamics simulations, this rule can be exploited to reduce the number of force calculations by a factor of 2. The two possible Newton 3 options are:

   a) **Newton3 Off**
   If Newton 3 is turned off, the forces between all pairs of particles are calculated twice, once for each particle. This results in a constant overhead of factor 2.

b) **Newton3 On**
If Newton 3 is turned on, the forces between all pairs of particles are calculated only once. There is no more overhead due to recalculating the forces twice, but turning on Newton 3 requires additional bookkeeping, especially in multi-threaded environments. This results in more complicated traversal algorithms and can result in a performance overhead.
Generally should be turned on whenever available.

6. **Cell Size Factor:**
The Cell Size Factor is a parameter that is used to determine the size of the cells in the LinkedCells-Container[6]. The cell size factor is typically chosen to be equal to the cutoff radius r_c, meaning that each particle only needs to check the forces with particles inside the $3 \times 3 \times 3$ cell grid around it as all other particles are guaranteed to be further away than the cutoff radius. We saw in the previous section that this could result in many unnecessary distance checks. The amount of spurious distance checks can be reduced by choosing smaller cell sizes. However, the increased overhead of managing more cells can quickly offset the performance gain. A trade-off between the two needs to be found.

2.2.3. Tuning Strategies

Tuning strategies are the heart of AutoPas and attempt to efficiently find the best parameters for the current simulation state.

The default tuning strategy in AutoPas uses a brute-force approach to find the best parameters for the current simulation state by trying out all possible combinations of parameters and choosing the one that performed best. This approach is called *Full Search* and is guaranteed to find the best parameters for the current simulation state. However, it is typically very costly in terms of time and resources as it has to spend a lot of time measuring bad parameter combinations. This is a big issue as the number of possible parameter combinations grows exponentially with the number of parameters, and many potentially perform very poorly. This makes the full search approach infeasible, especially if more tunable options are added to AutoPas.

To overcome this issue, AutoPas provides a couple of different tuning strategies that can be used to reduce the number of parameter combinations that need to be tested. This is generally achieved by allowing the tuning strategy to modify the queue of parameters that need to be tested, allowing them to exclude unlikely configurations from the testing process.

Developing tuning strategies that can effectively prune the search space is of utmost importance, as bad parameter choices can result in a significant slowdown of the simulation. The following section will introduce the currently available tuning strategies in AutoPas.

1. **Full Search**
The Full Search strategy is the default tuning strategy in AutoPas. It tries out all possible combinations of parameters and chooses the one that optimizes the chosen performance metric. It is guaranteed to find the best parameters for the current simulation state, but as mentioned before, it is very costly.

[6]The option is also relevant for other containers such as VerletLists as those configurations internally also build their neighbor lists using a Cell Grid

2. **Random Search**
   The Random Search strategy is a simple tuning strategy that randomly samples a given number of configurations from the search space and chooses the one that optimizes the chosen performance metric. This approach is faster than the Full Search strategy as it does not need to test all possible combinations of parameters. However, it does not guarantee to find the best parameters for the current simulation state.
3. **Predictive Tuning**
   The Predictive Tuning strategy attempts to extrapolate previous measurements to predict how the configuration would perform in the current simulation state. It filters the search space and only keeps configurations predicted to perform reasonably well. The extrapolations are accomplished using methods such as linear regression or constructing polynomial functions through the data points.
4. **Bayesian Search**
   Two implementations of Bayesian tuning exist in AutoPas. Those methods apply Bayesian optimization techniques to predict suitable configurations using performance evidence from previous measurements.
5. **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 built on expert knowledge and could look like this:

   ```
   if numParticles < lowNumParticlesThreshold:
         [dataLayout="AoS"] >= [dataLayout="SoA"] with same
               container, newton3, traversal, loadEstimator;
   endif
   ```

   The rule states that the data layout "AoS" is generally better than "SoA" if the number of particles is below a certain threshold. The rule-based method can be very effective if the rules are well-designed.

This thesis aims to extend these tuning strategies with a new approach based on Fuzzy Logic. Conceptually, this new fuzzy logic-based tuning strategy is very similar to the rule-based tuning strategy as it uses expert knowledge of fuzzy rules to prune the search space. However, contrary to classical rules, fuzzy logic can deal with imprecise and uncertain information, which allows it to only partially activate rules depending on the *degree of truth* of the condition. All the suggestions can then be combined based on their degree of activation rather than just following the binary true/false logic. This allows for a more nuanced approach and allows the tuning strategy to interpolate the effect of many different rules to choose the best possible configuration, even if there is no direct rule for this specific case.

In the following section, we will introduce the basic fuzzy logic concepts.

2.3. Fuzzy Logic

Fuzzy Logic is a mathematical framework that allows for reasoning under uncertainty. It is an extension of classical logic and extends the concept of binary truth values ($true$ and $false$)

to a continuous range of truth values in the interval $[0, 1]$. This allows for a more nuanced representation of the truth values of statements, which can be beneficial when dealing with imprecise or uncertain information. Instead of just having true or false statements, it is now possible for statements to be, for example, 40% true. This concept is beneficial when modeling human language, as the words tend to be imprecise. For example, *hot* can mean different things to different people. For some people, a temperature of 30 degrees Celsius might be considered *hot*, while for others, a temperature of 40 degrees Celsius might be considered *hot*. There is no clear boundary between what is considered hot and what is not, but rather a gradual transition between the two. Fuzzy Logic allows modeling such gradual transitions by assigning a degree of truth to each statement.

2.3.1. Fuzzy Sets

Mathematically, the concept of Fuzzy Logic is based on Fuzzy Sets. A Fuzzy Set is a generalization of a classical set where an element can lie somewhere between being a set member and not being a member. Instead of having a binary membership function that assigns a value of 1 to elements that are members of the set and 0 to elements that are not, elements in a fuzzy set have a certain degree of membership in the set. This degree of membership takes values in the interval $[0, 1]$ where 0 means that the element is not a member of the set, and 1 means that the element is a full member of the set.

Formally a fuzzy set $\tilde{A}$ over a crisp/classical set X is defined by a membership function

$$\mu_{\tilde{A}} : X \to [0, 1] \tag{2.6}$$

which assigns each element $x \in X$ a degree of membership in the interval $[0, 1]$. The classical counterpart of the element operator could be written as $\in_A : X \to \{true, false\}$.

The shape of the function can be chosen freely and depends on the specific application. However, typical choices involve triangular, gaussian, or sigmoid-shaped functions, depending on whether the value represents one- or two-sided properties.

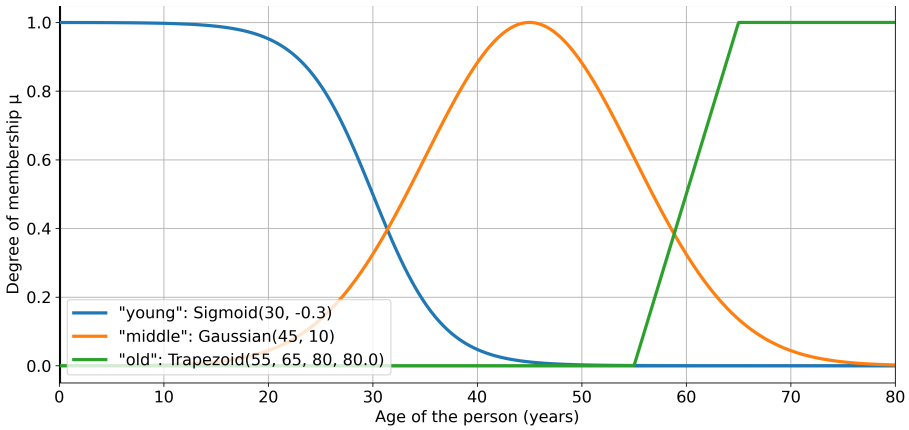


Figure 2.5.: Example of fuzzy sets for the age of a person. Fuzzy sets can be used to model the gradual transition between age groups. The distributions could be derived from survey data on how people perceive age groups. In this example, most people would consider a person middle-aged if they are between 35 and 55; there are, however, outliers ranging as low as 20 and as high as 70.

2.3.2. Fuzzy Logic Operations

Fuzzy Sets are a generalization of classical sets, and as such, they also support the classical set operations of union, intersection, and complement. Those operations need to be extended to work with fuzzy sets and combine both the semantics of classical sets and the concept of vague membership degrees of fuzzy sets.

The extension of classical operators to fuzzy sets uses so-called De Morgan Triplets. Such a triplet $(\top, \bot, \neg)$ consists of a t-norm $\top : [0,1] \times [0,1] \to [0,1]$, a t-conorm $\bot : [0,1] \times [0,1] \to [0,1]$ and a strong complement operator $\neg : [0,1] \to [0,1]$. Those operators generalize the classical logical operators, which are only defined on the binary truth values $\{true, false\}$ to continuous values from the continuous interval $[0,1]$. $\top$ generalizes the logical AND operator, $\bot$ generalizes the logical OR operator, and $\neg$ generalize the logical NOT operator. Instead of the binary functions used in classical logic, those new operators are continuous functions implementing mappings between degrees of truth.

The binary operators $\top$ and $\bot$ are often written in infix notation as $a \top b$ and $a \bot b$, similar to how classical logical operators are written.

For the t-norm $\top$ to be valid, it needs to satisfy the following properties:

$$
\begin{aligned}
a \top b &= b \top a && \text{(Commutativity)} \\
a \top b &\leq c \top d \quad \text{if } a \leq c \text{ and } b \leq d && \text{(Monotonicity)} \\
a \top (b \top c) &= (a \top b) \top c && \text{(Associativity)} \\
a \top 1 &= a && \text{(Identity Element)}
\end{aligned}
$$

A strong complement operator $\neg$ needs to satisfy the following properties:

$$
\begin{aligned}
\neg 0 &= 1 && \text{(Boundary Conditions)} \\
\neg 1 &= 0 && \text{(Boundary Conditions)} \\
\neg y &\leq \neg x \quad \text{if } x \leq y && \text{(Monotonicity)} \\
\neg \neg x &= x && \text{(Involution)}
\end{aligned}
$$

The default negation operator in fuzzy logic is $\neg x = 1 - x$. This negation operator satisfies all the abovementioned properties and is the most common choice in practice. In the following sections, we will only consider this standard negation operator.

As in classical logic, the t-conorm $\bot$ can be expressed using $\top$ when applying the generalized De Morgan's laws. De Morgan's laws state that $a \lor b = \neg(\neg a \land \neg b)$ for classical logic, which results in $\bot(a,b) = 1 - \top(1-a, 1-b)$ for fuzzy logic. Consequently, the properties of the t-conorm can be expressed using the t-norm's properties and omitted here for brevity.

Some common choices for t-norms and t-conorms used in practice are shown in Table 2.1.

Name	t-norm $a \top b$	Corresponding t-conorm $a \bot b$
Min/Max	$\min(a, b)$	$\max(a, b)$
Algebraic	$a \cdot b$	$a + b - a \cdot b$
Einstein	$\frac{a \cdot b}{2-(a+b-a \cdot b)}$	$\frac{a+b}{1+a \cdot b}$
Lukasiewicz	$\max(0, a + b - 1)$	$\min(1, a + b)$

Table 2.1.: Common t-Norms and corresponding t-Conorms concerning the standard negation operator $\neg x = 1 - x$

With these choices of t-norms, t-conorms, and negation operators, it is possible to define the classical set operations of union, intersection, and complement for fuzzy sets. We will only consider the minimum t-norm and maximum t-conorm in the following sections as they are the most common choices in practice. However, we included a comparison of different t-norms and their effect on the intersection operation in Figure 2.6.

- **Intersection**
  By expanding the definition of the classical set operation $\cap$ using its boolean form $x \in A \cap B \iff x \in A \ \wedge \ x \in B$, we can directly translate this to the fuzzy set intersection operation using the t-norm $\top$. The resulting membership function is given by $\mu_{\tilde{A} \cap \tilde{B}}(x) = \mu_{\tilde{A}}(x) \top \mu_{\tilde{B}}(x)$. Using the minimum t-norm, the intersection of two fuzzy sets $\tilde{A}$ and $\tilde{B}$ is described by the following membership function:

$$\mu_{\tilde{A} \cap \tilde{B}}(x) = \min(\mu_{\tilde{A}}(x), \mu_{\tilde{B}}(x))$$

- **Union**
  By expanding the definition of the classical set operation $\cup$ using its boolean form $x \in A \cup B \iff x \in A \ \vee \ x \in B$, we can directly translate this to the fuzzy set union operation using the t-conorm $\bot$. The resulting membership function is given by $\mu_{\tilde{A} \cup \tilde{B}}(x) = \mu_{\tilde{A}}(x) \bot \mu_{\tilde{B}}(x)$. Using the maximum t-conorm, the union of two fuzzy sets $\tilde{A}$ and $\tilde{B}$ is described by the following membership function:

$$\mu_{\tilde{A} \cup \tilde{B}}(x) = \max(\mu_{\tilde{A}}(x), \mu_{\tilde{B}}(x))$$

- **Complement**
  By again expanding the definition of the classical set operation A^c using its boolean form $x \in A^c \iff \neg(x \in A)$, we can directly translate this to the fuzzy set complement operation using the negation operator $\neg$. The resulting membership function is given by $\mu_{\tilde{A}^c}(x) = \neg \mu_{\tilde{A}}(x)$. Using the standard negation operator, the complement of a fuzzy set $\tilde{A}$ is described by the following membership function:

$$\mu_{\tilde{A}^c}(x) = 1 - \mu_{\tilde{A}}(x)$$

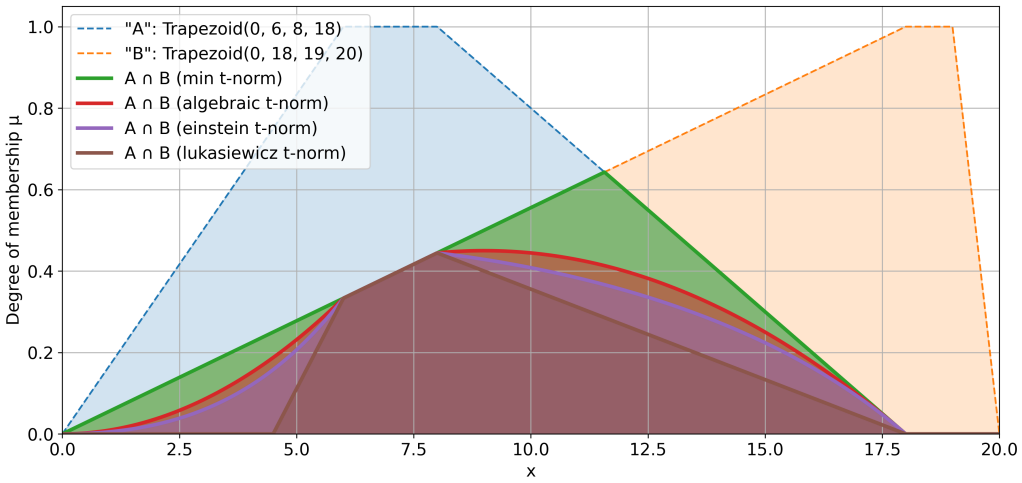


Figure 2.6.: Effect of different t-norms on the intersection of two fuzzy sets $\tilde{A}$ and $\tilde{B}$. We can see that the choice of t-norm significantly affects the shape of the resulting fuzzy set.

2.3.3. Linguistic Variables

Linguistic variables collect multiple fuzzy sets defined over the same crisp set X into a single object. This variable then allows us to reason about the possible states of the variable more naturally. Contrary to their classical counterparts, linguistic variables do not take a precise numerical value but rather a vaguely defined linguistic term. For example, all fuzzy sets depicted in Figure 2.5 form the linguistic variable "age". Instead of precisely stating the age of a person using a numerical value, we can now declare the person to be *young*, *middle-aged*, or *old*, where the underlying fuzzy sets capture the transition between those states.

2.3.4. Fuzzy Logic Rules

Fuzzy Logic Rules are a way to encode expert knowledge into a Fuzzy Logic system. The rules specify the relationship between input and output variables of the system and, therefore, are the backbone of fuzzy logic systems. The rules are typically encoded in a human-readable way and often have the form "IF antecedent THEN consequent" where both the antecedent and the consequent are fuzzy sets. The antecedent is a condition that must be satisfied for the rule to be applied, while the consequent is the action taken if the rule is applied. Since we are not dealing with binary truth values, it is possible that the antecedent is only partially satisfied. As a result, the rule's effect is also only partially considered.

The antecedent can be arbitrarily complicated and may consist of multiple fuzzy sets and logical operators. The consequent is typically a single fuzzy but could theoretically also be arbitrarily complicated. In the implementation of this thesis, we will consider rules generated by the following grammar:

FuzzyRule	::=	IF FuzzySet THEN FuzzySet	(Rule)
FuzzySet	::=	(FuzzySet)	(Parentheses)
		| FuzzySet AND FuzzySet	(Conjunction)
		| FuzzySet OR FuzzySet	(Disjunction)
		| NOT FuzzySet	(Negation)
		| $\tilde{A} = a$	(Selection)

The boolean operators AND, OR, and NOT represent the set operations of intersection, union, and complement, respectively. The selection operator $\tilde{A} = a$ states that the linguistic variable $\tilde{A}$ should have a high degree of membership in the fuzzy set a for the rule to be activated fully.

Using this grammar, a typical rule might look like this:

$$\text{IF } (\tilde{A} = a \text{ AND } \tilde{B} = b) \text{ THEN } \tilde{C} = c$$

This rule states that if the state of the linguistic variable A is a and the state of the linguistic variable B is b, then the state of the linguistic variable C should be c. However, contrary to classical logic, the rule does not have to activate fully but can have a degree of activation in the interval $[0, 1]$. If the antecedent is only partially true (for example, if $\mu_{\tilde{A}}(a) = 0.8$ and $\mu_{\tilde{B}}(b) = 0.6$), the rule is only partially applied and the effect of adapting the consequent is reduced accordingly.

The inference step can be seen as an extension of the boolean implication operator

$$\text{IF antecedent THEN consequent} \iff (\text{antecedent} \implies \text{consequent})$$

Instead of deriving the membership function of the implication operator, the Mamdani implication is typically used. This particular implication is defined as the AND operation $\min(a, b)$. This choice is counterintuitive as it violates its equivalent in classical logic. However, in the context of fuzzy systems, it is a preferred choice, as instead of evaluating the truthiness of the implication, it computes the degree of activation for a rule [BMK96].

Evaluation of Fuzzy Logic Rules

Consider the rule IF $\tilde{A} = a$ THEN $\tilde{C} = c$. To calculate the resulting fuzzy set, we perform the following steps:

1. Obtain the input values $(x_1, x_2, \ldots, x_n) \in X_A$ occuring in the crisp set of the antecedent.
2. Evaluate the degree of membership μ of those input values in the antecedent. This is the degree to which the antecedent is satisfied, and the rule is activated.
3. Define a new fuzzy set $R = \tilde{C} \uparrow \mu$ as the result of the rule activation. $\uparrow$ is the cut operator and is defined as $\mu_{\tilde{C}\uparrow\mu}(x) = \min(\mu_{\tilde{C}}(x), \mu)$ following the Mamdani implication.

The same steps are performed for every other rule, and the resulting fuzzy sets are combined using the fuzzy union operation specified in the previous section. This final fuzzy set represents the combined effect of all the rules on the output variable.

2.3.5. Defuzzification

The final step in a Fuzzy Logic system is the defuzzification step. In this step, the resulting fuzzy set of the previous section is converted back into a crisp, numeric value that can be used to control natural-world systems. There are many ways to defuzzify a fuzzy set, but a common theme is finding a most likely value that maintains certain aspects of the fuzzy set. In the following sections we will make use of the following defuzzification methods:

- **Centroid**
  The Centroid method calculates the center of mass of the fuzzy set and returns this value as the crisp output. This method tries to find a weighted interpolation of all the activated fuzzy sets and tries to find an optimal compromise between all the possible values. The Centroid method is the most common defuzzification method and is often used in practice due to its simplicity and robustness. It is defined as:

$$\text{Centroid} = \frac{\int_X x \cdot \mu_{\tilde{C}}(x)\, dx}{\int_X \mu_{\tilde{C}}(x)\, dx} \tag{2.7}$$

- **Mean of Maximum**
  The Mean of Maximum method is simpler than the Centroid method and only considers values, resulting in the highest possible membership value. If multiple such values exist, the arithmetic mean of those values is returned. Contrary to the Centroid method, there is usually no interpolation between the different fuzzy sets, as they usually have different degrees of activation. It is defined as follows:

$$\text{Mean of Maximum} = \frac{\int_{X'} x\, dx}{\int_{X'} dx} \tag{2.8}$$

  where X' is the set of all input values resulting in the maximum membership value of the fuzzy set.

3. Implementation

This chapter describes the implementation of the Fuzzy Tuning technique in AutoPas. The implementation is divided into three main parts: the generic fuzzy logic framework, the rule parser, and the fuzzy tuning Strategy. The fuzzy logic framework is the core of this implementation and implements the mathematical foundation of this technique. The rule parser loads the supplied knowledge base from a rule file. Finally, the Fuzzy Tuning Strategy implements the interface between the fuzzy logic framework and the AutoPas simulation. It is responsible for updating the configuration queue to select configurations to be tested next. A simplified class diagram of the implementation can be seen in Figure 3.4.

3.1. Fuzzy Tuning Framework

The Fuzzy Tuning framework implements the mathematical foundation of the Fuzzy Tuning technique. It consists of several components that fully implement the mathematical concepts of fuzzy logic. It consists of the following components:

- **Crisp Set**
  The Crisp Set class models classical sets using k-cells[1] in order to represent the universe of discourse for the fuzzy sets. Therefore, it keeps track of the ranges of the input variables, which are later used in the defuzzification step. Using k-cells, we can only model continuous variables with a finite range of values. This is an acceptable limitation for the current use case in AutoPas, as all relevant parameters either fulfill this requirement or can be encoded as such (See Subsection 3.3.1). However, there exist methods to directly use nominal values as described in [RdCC12] or [JPRS06], but those are not implemented in this work.

- **Fuzzy Set**
  As mentioned previously, fuzzy sets consist of a membership function $\mu : X \to [0, 1]$, assigning a degree of membership to each element of the associated Crisp Set C. For the implementation in C++, we distinguish between two types of membership functions: The `BaseMembershipFunction` and the `CompositeMembershipFunction`. The `BaseMembershipFunction` implements membership functions over 1-dimensional k-cells (1-cells), typically the real numbers $\mathbb{R}$. It implements the *conventional* type of membership function and is implemented as a lambda function $f : \mathbb{R} \to [0, 1]$ that directly assigns the degree of membership to each input value. Generic examples of triangular, trapezoidal, gaussian, and sigmoid-shaped membership functions are implemented this way and can be selected by the user via the rule file.

[1] A k-cell is a hyperrectangle in the k-dimensional space constructed from the Cartesian product of k intervals $C = I_1 \times I_2 \times \ldots \times I_k$ where $I_i = [x_{low}, x_{high}] \subset \mathbb{R}$ is an interval in the real numbers.

The `CompositeMembershipFunctions` implement membership functions over k-cells. This distinction is necessary, as we will later use a recursive approach to construct complex fuzzy sets from simpler ones, and those newly constructed fuzzy sets should compose their children's membership functions to calculate their own membership value. Thus requiring a different interface than the `BaseMembershipFunction`. The `CompositeMembershipFunctions` are primarily meant to define fuzzy sets resulting from applying logical operations. To demonstrate the concept, let us consider the fuzzy set $\tilde{C} = \tilde{A} \cap \tilde{B}$. This new fuzzy set $\tilde{C}$ is defined over the Crisp Set $C = A \times B$, where A and B are the Crisp Sets of the fuzzy sets $\tilde{A}$ and $\tilde{B}$, respectively. As explained in previous chapters, the membership function $\mu_{\tilde{C}:C\to[0,1]}$ can be calculated as $\mu_{\tilde{C}}(x, y) = \min(\mu_{\tilde{A}}(x), \mu_{\tilde{B}}(y))$, by recursively making use of the membership functions of the *child* fuzzy sets $\tilde{A}$ and $\tilde{B}$. The only new information the `CompositeMembershipFunction` needs to store is the function that should be used to combine the membership values of the children. As these membership functions operate in a higher-dimensional space, they are implemented as lambda functions $f : \mathbb{R}^k \to [0, 1]$ combining the membership values of their *children* using some logical operation to produce their own membership values. The logical operations min, max, and $\neg$ are implemented this way as they combine existing fuzzy sets to form new ones. Internally, all fuzzy sets are represented using a tree-like data structure. The tree's root node represents the fuzzy set itself, and every internal node represents an intermediate fuzzy set defining the smaller fuzzy sets, which can be combined to form the root fuzzy set. In this tree structure, the `CompositeMembershipFunctions` act as a link between existing fuzzy sets (the *children*) and lead to the definition of a more complex fuzzy set (the *parent*). The Leaf nodes of a fuzzy set can no longer be decomposed into simpler fuzzy sets and are consequently defined using the `BaseMembershipFunctions`. Figure 3.1 shows a larger example of how complex fuzzy sets can be constructed from simpler fuzzy sets using the `CompositeMembershipFunctions` and `BaseMembershipFunctions`.
Furthermore, the Fuzzy Set class provides methods for defuzzification and combining fuzzy sets using logical operations.

- **Linguistic Variable**
  Linguistic variables act as simple containers for fuzzy sets. Each Linguistic Variable has a name (e.g., `Temperature`) and stores linguistic terms consisting of a name (e.g., `hot`) and a corresponding fuzzy set $\tilde{H}$ describing the distribution of the term.
- **Fuzzy Rule**
  The Fuzzy Rule class stores an antecedent and a consequent fuzzy set ($\tilde{A}$ and $\tilde{C}$). Additionally, the class provides a method to apply the rule, producing a new fuzzy set $R = \tilde{C} \uparrow \mu$ consisting of the partially activated fuzzy set $\tilde{C}$ where μ is the degree of membership of the supplied input values in the antecedent fuzzy set $\tilde{A}$.
- **Fuzzy Control System:** The Fuzzy Control System combines all the concepts described above to create a system that can evaluate a set of fuzzy rules and generate a defuzzified output value based on the supplied input values. Such a control system acts like a black box $f : \mathbb{R}^n \to \mathbb{R}$ that maps crisp input values to a crisp output value. Multiple such systems are used in later sections to implement the tuning strategy.

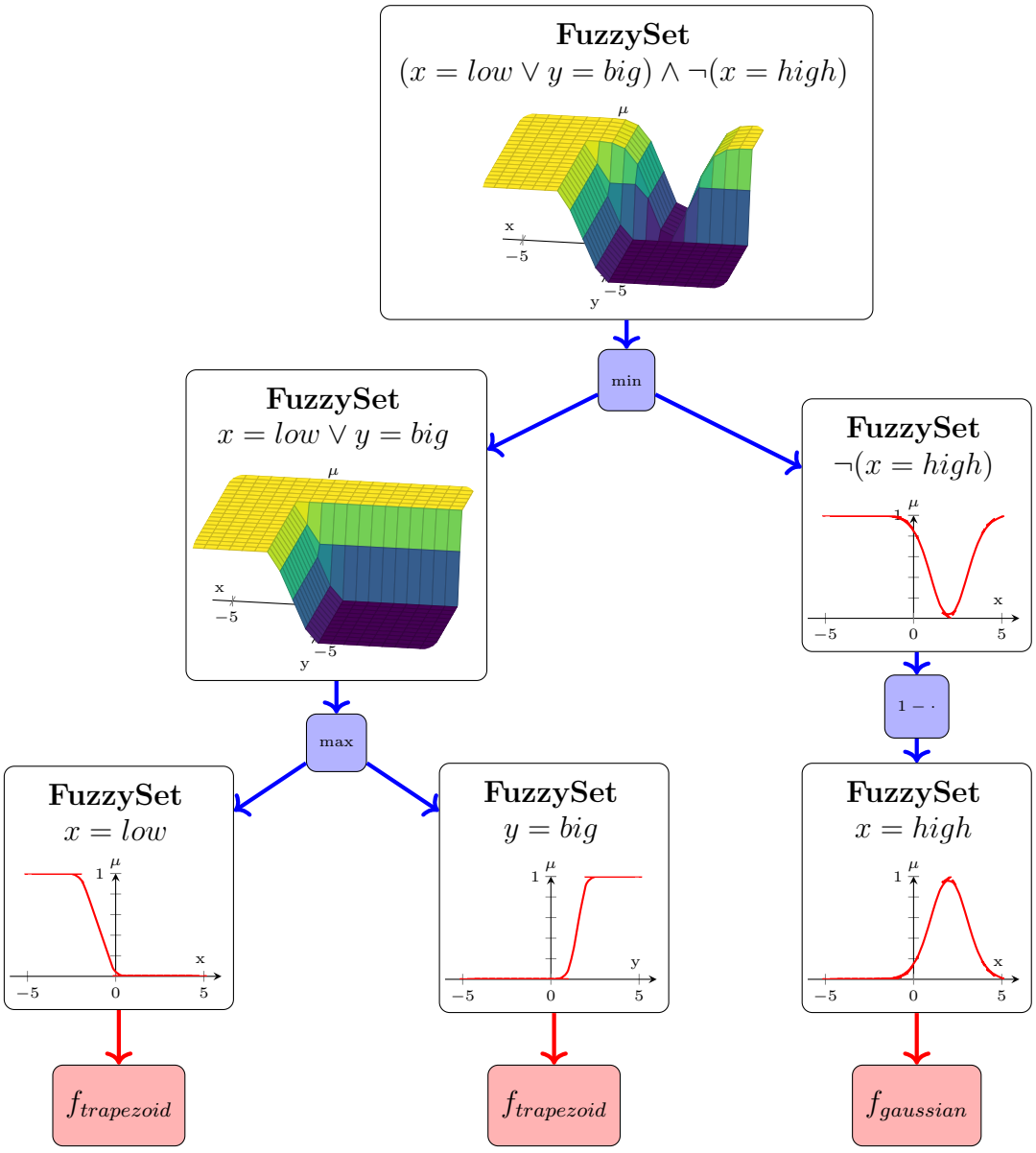


Figure 3.1.: Recursive construction of a complex fuzzy set from simpler fuzzy sets. Using the linguistic variables x with the terms $\{low, high\}$ and y with the terms $\{big, small\}$ we can construct the fuzzy set $(x = low \vee y = big) \wedge \neg(x = high)$ by combining the fuzzy sets $x = low \vee y = big$ and $\neg(x = high)$. Those fuzzy sets are again constructed from the simpler fuzzy sets $x = low$, $y = big$ and $x = high$.
The fuzzy sets at the leaf level can be directly constructed using predefined `BaseMembershipFunctions` (e.g., trapezoid, sigmoid, gaussian ...) and provide the foundation for the more complex fuzzy sets All other fuzzy sets are created by combining other fuzzy sets using `CompositeMembershipFunctions`. The logical operators min, max, and $1 - \cdot$ are implemented this way, as they directly act on top of other fuzzy sets.

3.2. Rule Parser

The Rule Parser is responsible for parsing the knowledge base supplied by the user and converting it into the internal representation used by the Fuzzy Tuning framework. It is based on the ANTLR4[2] parser generator and makes use of a domain-specific language tailored to the needs of the Fuzzy Tuning. The language is inspired by common standards such as the Fuzzy Control Language (FCL)[3] but is designed to be more lightweight and directly incorporates aspects of AutoPas, such as configurations, into the language. All supplied rules predicting values for the same output variable are grouped together, forming a single Fuzzy Control System.
The conversion between the generated parse tree and the internal representation is done by a visitor pattern that traverses the parse tree generated by ANTLR4 and internally builds the corresponding object hierarchy. A minimal example demonstrating the syntax of the rule file can be seen in Listing 3.1.

```
# Define the settings of the fuzzy control systems
FuzzySystemSettings:
    defuzzificationMethod:  "meanOfMaximum"
    interpretOutputAs:      "IndividualSystems"

# Define linguistic variables and their linguistic terms
FuzzyVariable: domain: "homogeneity" range: (-0.009, 0.1486)
    "lower than 0.041":     SigmoidFinite(0.0834, 0.041, -0.001)
    "higher than 0.041":    SigmoidFinite(-0.001, 0.041, 0.0834)

FuzzyVariable: domain: "threadCount" range: (-19.938, 48.938)
    "lower than 18.0":      SigmoidFinite(38.938, 18.0,  -2.938)
    "lower than 26.0":      SigmoidFinite(46.938, 26.0,   5.061)
    "lower than 8.0":       SigmoidFinite(28.938,  8.0, -12.938)
    "higher than 18.0":     SigmoidFinite(-2.938, 18.0,  38.938)
    "higher than 26.0":     SigmoidFinite(5.0617, 26.0,  46.938)
    "higher than 8.0":      SigmoidFinite(-12.93,  8.0,  28.938)

FuzzyVariable: domain: "particlesPerCellStdDev" range: (-0.017, 0.072)
    "lower than 0.013":     SigmoidFinite(0.0639, 0.038,  0.012)
    "higher than 0.013":    SigmoidFinite(0.012,  0.013,  0.0639)

FuzzyVariable: domain: "Newton 3" range: (0, 1)
    "disabled, enabled":   Gaussian(0.3333, 0.1667)
    "enabled":             Gaussian(0.6667, 0.1667)

# Define the interpretation of the output variables in the context of AutoPas
OutputMapping:
 "Newton 3":
     0.333 => [newton3 = "disabled"], [newton3 = "enabled"]
     0.666 => [newton3 = "enabled"]

# Define rules connecting the input variables to the output variables
if ("threadCount" == "lower than 18.0") && ("threadCount" == "higher than 8.0")
     && ("homogeneity" == "lower than 0.041")
   then ("Newton 3" == "enabled")
if ("threadCount" == "higher than 26.0") && ("particlesPerCellStdDev" == "lower than
     0.013")
   then ("Newton 3" == "disabled, enabled")
```

Listing 3.1: Demonstration of the domain-specific language used for Fuzzy Tuning

[2]https://www.antlr.org/
[3]https://www.fuzzylite.com/

3.3. Tuning Strategy

The Tuning Strategy implements the interface between the Fuzzy Tuning framework and the AutoPas simulation and is responsible for updating the configuration queue of configurations to be tested next. To achieve this, the strategy evaluates all fuzzy systems present in the rule file using the *LiveInfoData* (See A.4) collected by AutoPas. These data points contain summary statistics about various aspects of the current simulation state, such as the total number of particles, the average particle density, or the average homogeneity of the particle distribution. Each evaluation of a Fuzzy Control System (FCS) yields a single numeric value, which is then passed on to the `OutputMapper` object. The `OutputMapper` is responsible for mapping the continuous output value of the Fuzzy Control System to the discrete configuration space of AutoPas.

Internally, the `OutputMapper` stores an ideal numerical location for each configuration-pattern[4] and always selects the option closest to the predicted value. This method of assigning discrete values to the output of fuzzy systems is inspired by Mohammed et al. [MKEC22]'s work on scheduling algorithms, where the authors used a similar approach.

All the configuration patterns predicted by the Fuzzy Control Systems are then collected and used to update AutoPas's configuration queue. In the following iterations, AutoPas will benchmark every selected configuration for a few iterations and evaluate its performance based on the chosen metric. The best configuration is then declared the winner of this tuning phase and is used for the following simulation phase.

Currently, two different approaches using Fuzzy Tuning to predict *optimal* configurations are implemented: The *Component Tuning Approach* and the *Suitability Tuning Approach*. Both approaches are described in detail in the following sections.

3.3.1. Component Tuning Approach

The Component Tuning Approach assumes that each tunable parameter can be tuned independently of the others, making it possible to define a separate Fuzzy Control System for each tunable parameter.

All those Fuzzy Control Systems should then attempt to predict the best value of their parameter independent of the other parameters. This approach requires the rule file to only define $\#Parameters$ different Fuzzy Control Systems and a corresponding `OutputMapper` for each parameter. Creating such rule files is straightforward and could be reasonably created manually by a domain expert. An obvious drawback of this method is the independence assumption between the parameters, which might not hold in practice. However, the practical Experiments carried out in Chapter 5 will show quite good results, even with this simplification.

Another problem of this approach lies in the defuzzification step. As this method relies on defining a single system for all values of a tunable parameter, we must define a numerical *ranking* of all those values. Such a ranking is problematic, as most tunable variables are

[4] A configuration-pattern is a tuple of all tunable parameters, where each component of the tuple describes a set of possible values for this parameter. The wildcard value `*` allows any possible value. For example, the configuration-pattern `(Container=LinkedCells, Traversal=*, DataLayout=SoA, Newton3=enabled)` matches the specified configuration, regardless of the value of the `Traversal` parameter.

nominal and thus do not have a natural order. To circumvent this problem, we can choose a defuzzification method that does not perform interpolation between the values and use an arbitrary order for the values. One such method is MOM. It selects the mean of all x-values for which the membership function is maximal. When using Gaussian-shaped membership functions for the output values, this method will always return the mean of the Gaussian with the highest activation and will entirely regard the numerical placements of the values[5].

The OutputMapper is set up such that the mean values of the Gaussian membership functions are directly mapped to the corresponding values of the tunable parameters. After evaluating this ruleset, one ends up with a list of configuration patterns, each demanding a specific value for their parameter. All those patterns are then used to purge the configuration queue, excluding every configuration that does not match all the predicted patterns. Figure 3.2 shows a schematic of the prediction process for the Component Tuning Approach.

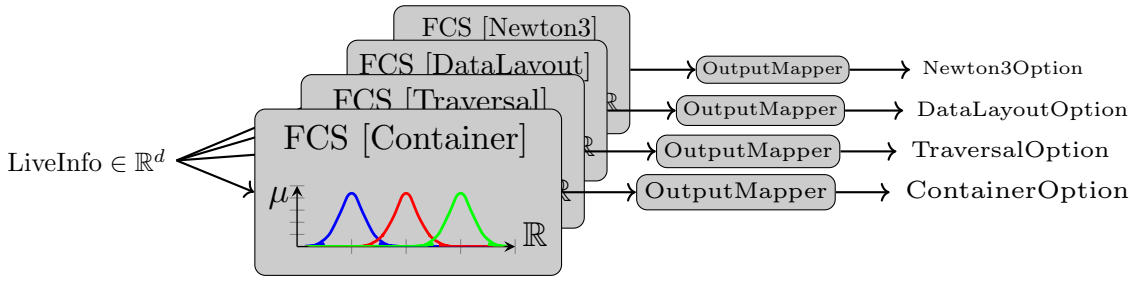


Figure 3.2.: Example Visualization of the fuzzy control systems for the Component Tuning Approach. The parameters `Container`, `Traversal`, `DataLayout`, and `Newton3` are tuned independently. The OutputMapper maps the defuzzified output values to their respective configuration options.

3.3.2. Suitability Tuning Approach

The Suitability Approach mainly differs from the Component Tuning Approach in that it utilizes $\#Container\_options \cdot \#Traversal\_options \cdot \#DataLayout\_options \cdot \#Newton3\_options$ different Fuzzy Control Systems, one for each possible combination of those parameters. Each Fuzzy Control System is responsible for predicting the suitability of its configuration.

The advantage of this approach is that there is no need to rank the output values, and one can utilize the power of Fuzzy Control Systems to interpolate between different predictions. This method uses the default defuzzification method of COG as the possible suitability predictions have a natural order (higher suitability is better). Furthermore, dependencies and incompatibilities between the parameters can be modeled accurately, as each way of combining the parameters is handled with a separate Fuzzy Control System. The downside of this method is the enormous complexity of the rule file, which quickly becomes infeasible to maintain by hand. Surprisingly, the cost of evaluating all those Fuzzy Control Systems is negligible compared to the overhead of other tuning strategies, as later experiments in Chapter 5 will show.

After evaluating all Fuzzy Control Systems and using a trivial OutputMapping, the

[5]There are exceptions when two Gaussians have the same activation. Such cases rarely happen in practice and could be resolved with other defuzzification methods such as SOM

method yields a list of `(Configuration, Suitability)` pairs, which can then be used to update the configuration queue. The current implementation selects the highest possible suitability value and then chooses every configuration performing within a certain threshold of the best configuration. Those configurations are then used to overwrite the configuration queue. Figure 3.3 shows a schematic of the prediction process for the Suitability Tuning Approach.

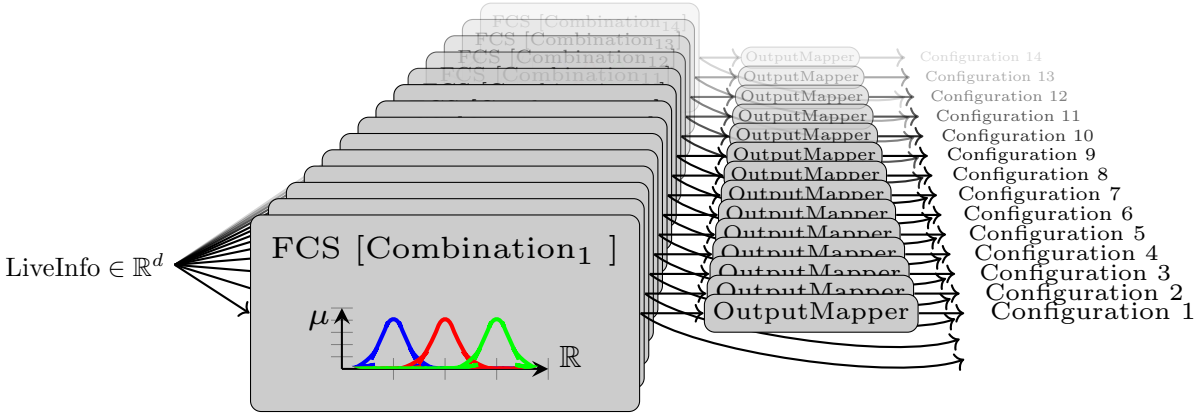


Figure 3.3.: Example Visualization of the fuzzy control systems for the Suitability Tuning Approach. Each fuzzy control system is responsible for predicting the suitability of a specific combination of tunable values, resulting in an enormous amount of fuzzy control systems. The `(Configuration, Suitability)` pairs are passed to the Fuzzy Tuning Strategy, which then updates the configuration queue

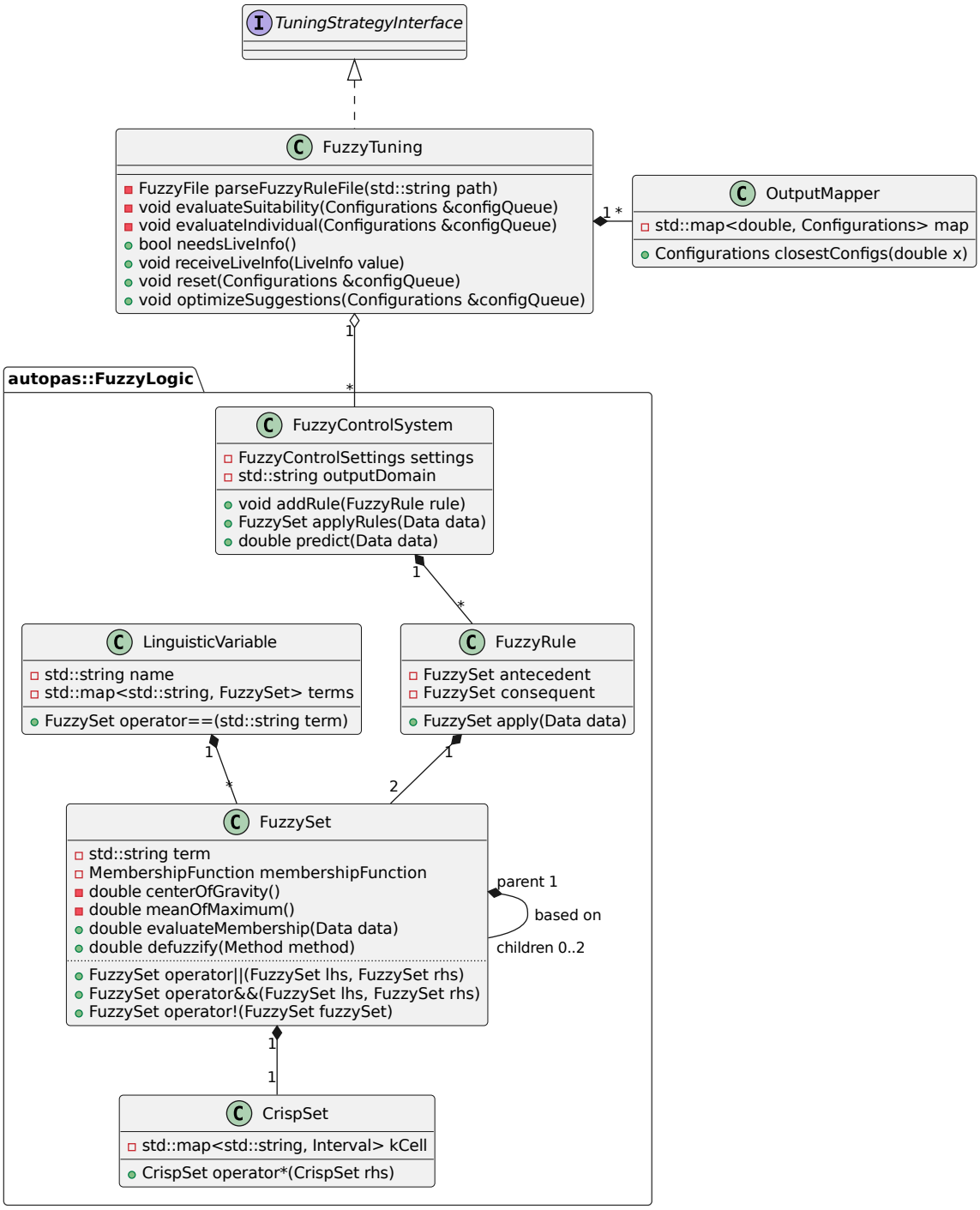


Figure 3.4.: Simplified class diagram of the Fuzzy Tuning strategy. There is a clear separation between implementing the Fuzzy Logic framework and the tuning strategy. This allows for an easy reuse of the Fuzzy Logic framework in other parts of AutoPas if desired.

4. Proof of Concept

This chapter presents a proof of concept for the fuzzy tuning technique. We will develop two different knowledge bases to predict the optimal configurations for `md_flexible` simulations.

As for all fuzzy systems, creating the knowledge base is one of the most complex parts of developing a fuzzy system, as it typically requires a profound understanding of the system to create meaningful rules. Luckily, methods such as data-driven approaches exist to semi-automate this process. This is extremely useful as those methods do not require prior expert knowledge about the system. For a problem as complex as tuning molecular dynamics simulations, it would be extremely hard to create a meaningful knowledge base from scratch. Such data-driven methods provide a good initial set of rules that experts can manually evaluate and adjust if desired. In this work, we will use a decision tree approach proposed by Crockett et al. [CBMO06]. This proposed method uses machine learning to first train decision trees on the dataset to create an initial crisp rule base. In the second step, the decision trees are converted into so-called fuzzy decision trees, which can then be used to extract the linguistic variables and fuzzy rules.

4.1. Data Driven Rule Extraction

4.1.1. Decision Trees

Decision trees are prevalent machine learning algorithms used for classification and regression tasks. They work by recursively partitioning the input using axis-parallel splits so that the resulting subsets are as pure as possible. Concretely, they try to minimize a given impurity metric, such as the Gini impurity $I_G = \sum_{i=1}^{n} p_i(1-p_i)$ or the entropy $H = -\sum_{i=1}^{n} p_i \log_2(p_i)$ of the subsets [Mur12]. The probability p_i is the fraction of samples in the subset that belong to class i, and n is the total number of classes.

Since decision trees directly partition the input space into regions with different classes, they can also be depicted using their decision surface if the dimensionality allows it. The decision surface of a decision tree is a piecewise constant function that assigns the predicted class label to each point in the input space of the decision tree. An example decision tree and its decision surface are shown in Figure 4.1 and Figure 4.2.

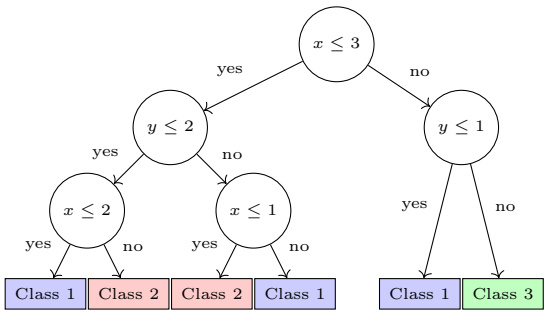


Figure 4.1.: An example decision tree for a dataset with two features x and y. There are three distinct classes in the dataset

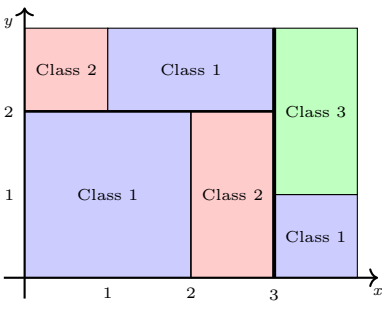


Figure 4.2.: The decision surface of the decision tree from Figure 4.1 on $\mathcal{D} = [0, 4] \times [0, 3]$.

4.1.2. Conversion of Decision Trees to Fuzzy Control Systems

This section will demonstrate how to convert a classical decision tree into a fuzzy decision system using the fictional decision tree from Figure 4.1 as an example.

Fuzzy Decision Trees

As the conversion makes use of so-called fuzzy decision trees, we will briefly introduce them here. Fuzzy decision trees are a generalization of classical decision trees and allow for fuzzy logic to be used in the decision-making process. Instead of following the classical `if then else` logic to descend into the decision tree, it uses fuzzy sets at each node of the tree to fuzzily calculate the contribution of each branch to the final decision based on the degree of truth of both possible paths. Contrary to classical decision trees, which follow a single path from the root to a leaf node, fuzzy decision trees explore all possible paths simultaneously and make a final decision by aggregating the results of the paths using fuzzy logic operations.

Conversion

A classical decision tree is converted into a fuzzy decision tree by replacing the crisp decision (e.g., $x \leq 3$) at each internal node of the decision tree with a fuzzy set. Those fuzzy sets should maintain the same semantics as the crisp decision but should provide a continuous value in the range $[0, 1]$ specifying the degree of how much each branch should be considered. Classical decision trees can only make binary decisions $(0, 1)$, which causes them to only consider a single path of the decision tree. Allowing them to consider multiple paths simultaneously can drastically increase the decision-making capabilities of the decision tree, especially in boundary cases where the decision can be ambiguous.

The shape of the membership functions of the fuzzy sets can be chosen arbitrarily, but since trees work with one-sided boundaries, typical choices include complementary `sigmoid`-shaped functions that are centered around the crisp decision boundary (See Figure 4.3), as those function shapes maintain the semantics of the branching idea. Crockett et al. [CBMO06] suggested creating those sigmoid shapes with a *width* proportional to the standard deviation of the attribute. In Particular, the authors suggested an interval $[t - n \cdot \sigma, t + n \cdot \sigma]$ for the membership function, where t is the value of the decision boundary, σ is the standard deviation of the attribute and n is a parameter that can be adjusted to control the *width* of the membership function. This interval specifies the region of the membership function where most of the change in membership occurs. The value of n is typically chosen from the interval $n \in [0, 5]$ as the membership function can become too broad otherwise and could even weaken the decision-making process [CBMO06]. In this work, we will use $n = 2$ as a default value. Using this method, it is possible to fully automate the conversion of a decision tree into a fuzzy decision tree, which we will utilize in this work.

Once the internal nodes of the decision tree have been converted, the next step is to convert the leaf nodes of the decision tree to fuzzy leaf nodes. As the outputs of decision trees are specific class labels, we can define a single linguistic variable consisting of all possible class labels together with a corresponding fuzzy set. The shapes of the membership functions for the fuzzy sets can again be chosen mostly arbitrarily, but we will use `gaussian` functions with a different mean as they are a good choice for representing class labels in a continuous domain. This way of placing the fuzzy sets is not directly obvious, and the placement can

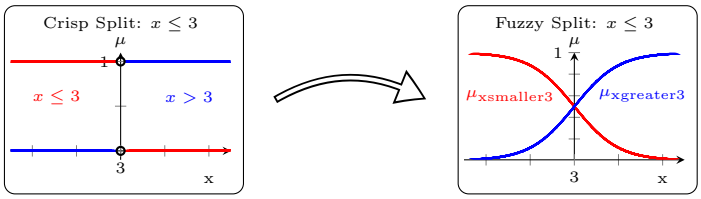


Figure 4.3.: Conversion of crisp set membership functions to fuzzy set membership functions. The classical membership functions $x \leq 3$ and $x > 3$ of a decision tree node are replaced by two `sigmoid`-shaped membership functions $\mu_{\text{xsmaller3}}$ and $\mu_{\text{xgreater3}}$ that specify to which degree one should traverse left or right. The *width* of the membership functions is data dependent and is determined by $n \cdot \sigma = 2 \cdot \sigma$.

significantly influence the defuzzification process depending on the chosen defuzzification function, as we will see later. However, we can minimize this issue by carefully choosing a suitable defuzzification method. The resulting conversion of the decision tree in Figure 4.1 into a fuzzy decision tree is shown in Figure 4.4.

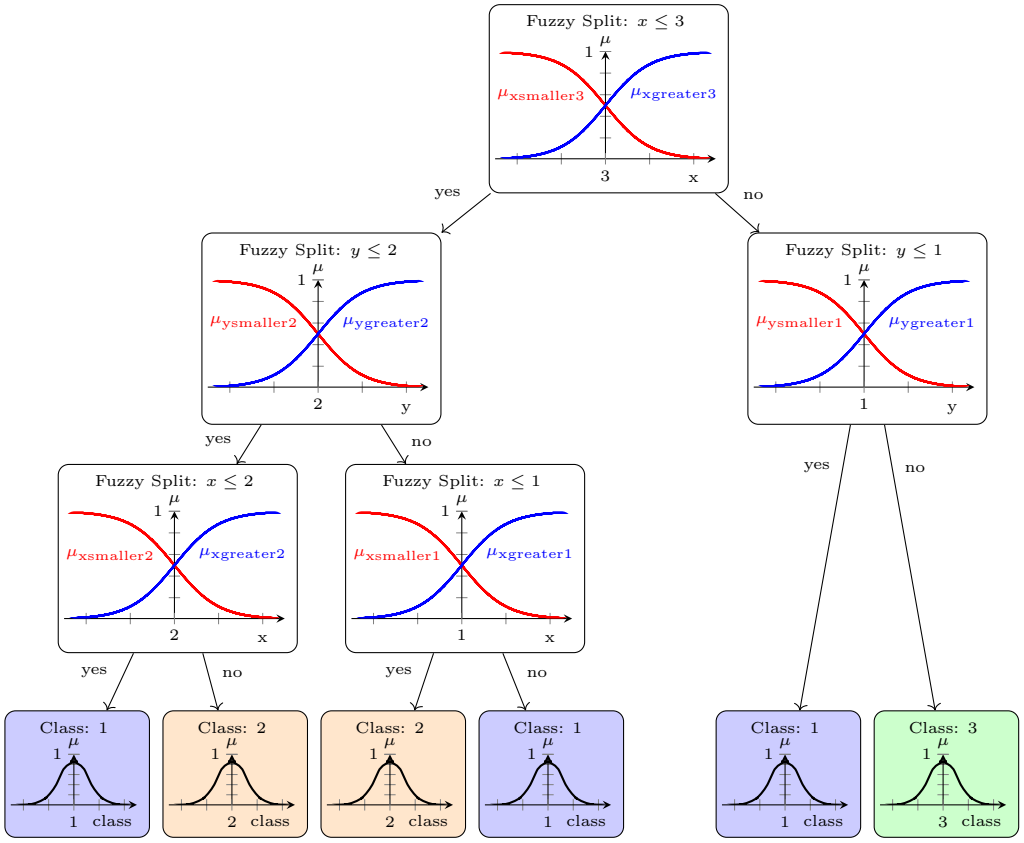


Figure 4.4.: The fuzzy decision tree corresponding to the decision tree in Figure 4.1. Internal nodes use two `sigmoid` membership functions (μ_{smaller} and μ_{greater}) instead of a crisp decision. The leaf nodes use different `gaussian` membership functions centered around a unique mean.

It is now possible to fully extract all linguistic variables from the fuzzy decision tree. Every fuzzy set defined over the same variable can be collected into a single linguistic variable. This results in linguistic variables consisting of a bunch of different `sigmoid` membership functions for input variables (internal nodes) and a single linguistic variable with different `gaussian` membership functions for the output variable (leaf nodes). The resulting linguistic variables are shown in Figure 4.5.

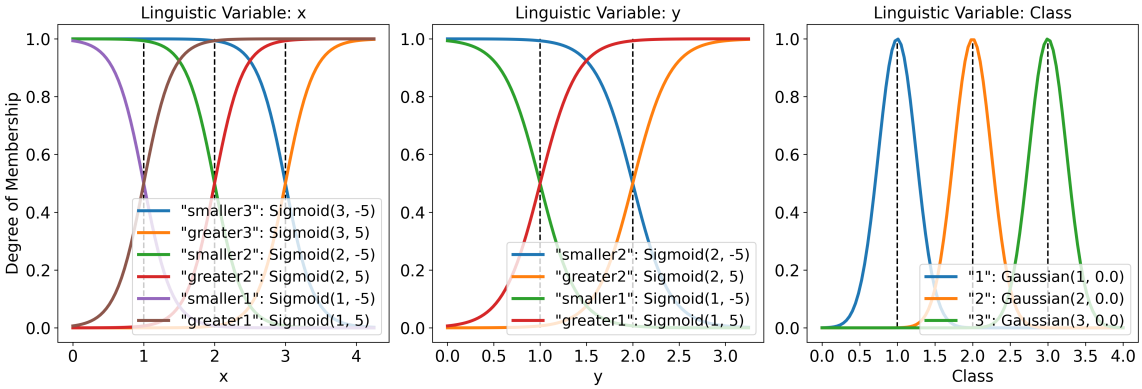


Figure 4.5.: Linguistic variables used in the fuzzy decision tree of Figure 4.4. The standard deviation of the attributes is assumed to be $\sigma \approx 0.5$ such that the *width* of the sigmoid membership functions is $n \cdot \sigma \approx 1$. The standard deviation of the class values is chosen so that they do not overlap too much.

Rule Extraction

The final step is to extract the fuzzy rules from the tree. This can be done by traversing the tree in a depth-first manner and collecting the correct membership functions for each path ending in a leaf node along the way. As all conditions of this path have to hold simultaneously, all fuzzy sets are connected using the `AND` operation. This newly created fuzzy set, which consists of the intersection of all fuzzy sets along the way, is the premise of selecting the fuzzy set of the leaf node. Consequently, both fuzzy sets are connected using the Mamdani `IMPLICATION` operator. This implication then forms a rule for the fuzzy system. Therefore, each unique path traversing the tree results in a unique rule. This process essentially mimics the decision surface seen in Figure 4.2, as we create precisely one rule for each region of the decision surface. The rules extracted from the fuzzy decision tree in Figure 4.4 using this method are shown in Table 4.1.

Rule	Antecedent	Consequent
1	x *is* smaller3 $\wedge$ y *is* smaller2 $\wedge$ x *is* smaller2	*class is* 1
2	x *is* smaller3 $\wedge$ y *is* smaller2 $\wedge$ x *is* greater2	*class is* 2
3	x *is* smaller3 $\wedge$ y *is* greater2 $\wedge$ x *is* smaller1	*class is* 2
4	x *is* smaller3 $\wedge$ y *is* greater2 $\wedge$ x *is* greater1	*class is* 1
5	x *is* greater3 $\wedge$ y *is* smaller1	*class is* 1
6	x *is* greater3 $\wedge$ y *is* greater1	*class is* 3

Table 4.1.: Extracted fuzzy rules from the fuzzy decision tree in Figure 4.4 in the format: **IF** Antecedent **THEN** Consequent

Fuzzy Control System

With the linguistic variables and fuzzy rules extracted from the decision tree, we can finally use them to create a fuzzy system that can predict the class of a new data point based on its features. Since the fuzzy system can be seen as a black box mapping continuous input features to continuous output classes, we represent them as shown in Figure 4.6 from now on. It is also possible to visualize the fuzzy system's decision surface by evaluating the system for each point in the input space, as shown in **??** for two different defuzzification methods. Such visualizations can help understand the decision-making process of the fuzzy system but, unfortunately, are not possible for higher-dimensional input spaces.

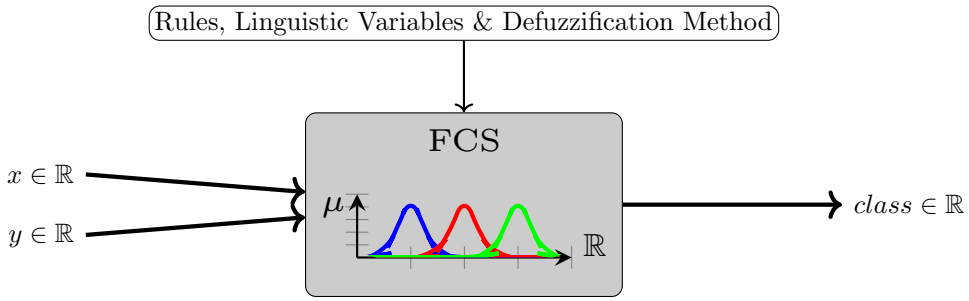


Figure 4.6.: The fuzzy control system created from the fuzzy decision tree in Figure 4.4 can be seen as a black box that maps continuous input features to continuous output classes.

Choice of Defuzzification Method

The exact shape of the decision surface depends on the specific defuzzification method used. The most common choice in literature is the COG method, which calculates the x-position of the center of gravity of the membership function of the final fuzzy set. However, using the COG method can lead to undesired results when using nominal values for the output classes, which we will see later. The main problem is that the values have no concept of order. Without such an ordering, the interpolation between the different classes performed by methods such as COG is not meaningful and leads to wrong predictions. Other methods, such as the MOM method, can be used instead. This method calculates the mean value of the maximum membership functions. In most cases, this method will return precisely

the center of the membership function with the highest value and is a good choice for such types of knowledge bases. A direct comparison of the two methods on a critical datapoint is shown in Figure 4.7 and Figure 4.8. Using the COG method results in an opposite class than originally predicted from the fuzzy system, while the MOM method correctly predicts the class.

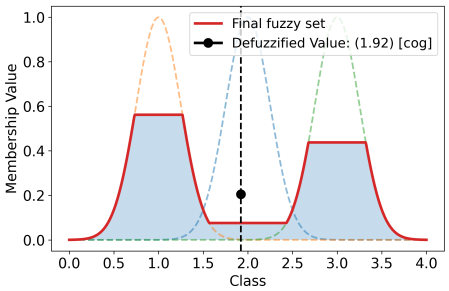


Figure 4.7.: Defuzzification on the fuzzy obtained by applying the rules from Table 4.1 on the data point ($x = 2.95, y = 2.5$). There are clear peaks at the class values 1 and 3. However, the COG method incorrectly suggests values close to class 2, thus turning the two good predictions into a bad one.

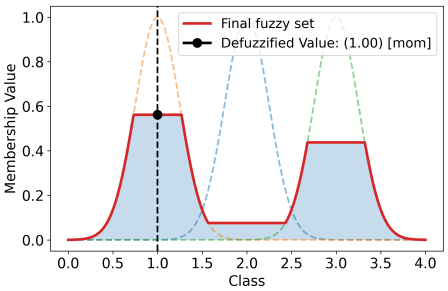


Figure 4.8.: Defuzzification on the fuzzy obtained by applying the rules from Table 4.1 on the data point ($x = 2.95, y = 2.5$). The MOM method correctly suggests the class value 1, as it is the class with the highest membership value.

As previously mentioned, we can recreate the process depicted in Figure 4.6 for every possible input data point to visualize the fuzzy systems' decision surface. Both resulting decision surfaces are shown in Figure 4.7 and Figure 4.8, respectively. The decision surface using the COG tries to smoothly interpolate between the different classes, which causes interpolation errors if there are other classes in between. The decision surface using the MOM method is valid and closely resembles the decision surface of the crisp decision tree in Figure 4.2.

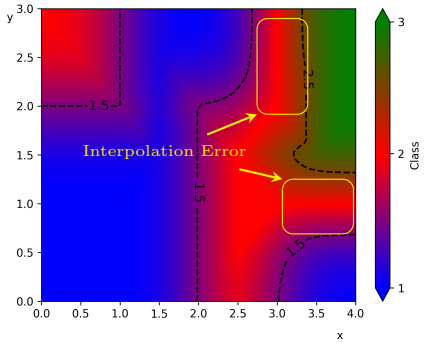


Figure 4.9.: Decision surface of the developed fuzzy-system using the COG defuzzification method. The highlighted areas show interpolation errors caused by the defuzzification.

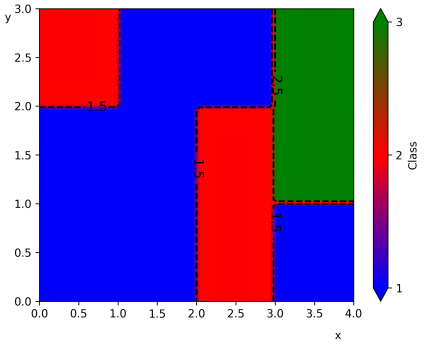


Figure 4.10.: Decision surface of the developed fuzzy-system using the MOM defuzzification method. There are no invalid regions in the decision surface.

4.2. Fuzzy Control Systems for md_flexible

This section will demonstrate how to generate fuzzy control systems for md_flexible simulations using the approach described in the previous section. The first section will describe the data collection process needed to train the classic decision trees, and the later sections will use the obtained data to generate two different styles of fuzzy control systems for making predictions about the optimal configuration of the current simulation.

4.2.1. Data Collection

Included Scenarios

We chose to include the prominent example scenarios provided by md_flexible such as `fallingDrop.yaml`, `explodingLiquid.yaml` and `SpinodalDecomposition.yaml` as the primary source of data. Additionally, we included some simulations of uniform cubes with different densities and particle counts to gather more data about the performance of the different configurations under lab-like conditions.

All simulations were run on the serial partition of the CoolMUC-2 cluster (see 1) and were repeated twice to account for fluctuations in performance. Furthermore, every simulation was repeated with 1, 4, 12, 24, and 28 threads to additionally gather data on how parallelization affects the ideal configuration.

All the values were collected with the newly created `PAUSE_SIMULATION_DURING_TUNING` CMake option enabled to ensure that the simulation state does not change during the tuning phases. This guarantees a fair comparison of the tested configurations, as all of them are evaluated under the exact same conditions. To gather the maximal amount of data, we used the `FullSeach` tuning strategy, which executes all possible configurations during the tuning phases.

Collected Parameters

The existing `TuningDataLogger` and the newly created `LiveInfoLogger` classes of the AutoPas framework allow us to collect a wide variety of parameters during the simulation, such as the used configuration, information about the state of the simulation, and the measured timing data for each iteration of the simulation.

The complete shape of the collected data can be found in Section A.3 and Section A.4, respectively. We will however only make use of a subset of the available LiveInfo data, as we are only interested in *relative* values that do not change when the simulation is scaled up or down and are therefore only primarily focus on: `avgParticlesPerCell`, `maxParticlesPerCell`, `homogeneity`, `maxDensity`, `particlesPerCellStdDev` and `threadCount`. This focus should help the fuzzy systems generalize better to unseen data, as they are less likely to overfit the training data.

Limitations

As the performance of machine learning models may degrade quickly when confronted with significantly different data than the data they were trained on, it is essential to collect a wide variety of scenarios to cover as many possible use cases as possible. As we only included a

limited number of scenarios, we have to keep in mind that the generated fuzzy systems will only be able to make confident predictions about scenarios similar to the included ones, and we should not expect them to generalize well to unseen data. To guarantee a fair evaluation of this tuning approach, we will only focus on slight variations of the included scenarios in the following evaluation section.

4.2.2. Data Preprocessing

In order to make predictions about the performance of different configurations, we first need to define an appropriate metric to compare them. As we *froze* the simulation during the tuning process with the `PAUSE_SIMULATION_DURING_TUNING` option, we can safely use the runtimes of the different configurations to compare them. Those runtimes are, however, absolute values and may differ significantly between tuning phases as the underlying simulation changes. To compare runtimes between different tuning phases, we introduce the concept of *speedup*, which measures how well a configuration performs compared to the best configuration in the same tuning phase, and augment the collected timing data with this metric. The speedup is calculated as

$$\text{speedup}^{(i)}_{\text{config}} = \frac{t^{(i)}_{\text{best}}}{t^{(i)}_{\text{config}}} \tag{4.1}$$

Where $t^{(i)}_{\text{best}}$ is the runtime of the best configuration during the i-th tuning phase and $t^{(i)}_{\text{config}}$ is the runtime of the configuration we are interested in.

This value will range from 0 (being infinitely worse than the best configuration) to 1 (being equally good as the best configuration) for each configuration. Additionally, we chose to combine the fields `Container` and `DataLayout` of the configuration into a single field `ContainerDataLayout` as they are closely related, and the performance of one is heavily dependent on the other.

We can now create the augmented dataset shown in Table 4.2, which will be used in the following sections to train the decision trees and generate the fuzzy systems.

ParticlesPerCell			Miscellaneous			Configuration			
avg	max	stddev	homo-genity	max-density	threads	Container DataLayout	Traversal	Newton3	speedup
0.905	23	0.0129	0.0354	0.531	1	LinkedCells_AoS	lc_sliced	enabled	0.450641
2.201	13	0.0144	0.0861	0.627	24	VerletListsCells_AoS	vlc_sliced	disabled	0.594117
0.905	18	0.0136	0.0431	0.319	4	LinkedCells_AoS	lc_sliced_c02	enabled	0.454632
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Table 4.2.: Augmented dataset used for creating the fuzzy systems. The dataset contains the average, maximum, and standard deviation of the particles per cell, the homogeneity, density, and thread count of the simulation, as well as the configuration options and the speedup of the configuration compared to the best configuration in the same tuning phase.

As described previously, we will create two different kinds of knowledge bases using this dataset. The first attempt is called *Component Tuning Approach* and tries to independently predict the best options for each tunable parameter. The second attempt is the *Suitability Tuning Approach*, which tries to predict the *speedup* values for all possible configurations and then selects the configurations expected to perform best. Both approaches will be described in the following sections.

4.2.3. Component Tuning Approach

This approach makes use of three different `FuzzyControlSystems`, one for each of the tunable parameters of the simulation (`ContainerDataLayout`, `Traversal`, and `Newton3`). All those systems should independently predict the best configuration for their respective parameter based on the current LiveInfoData. **??** shows the structure of this approach.

As we only want to create a fuzzy system predicting appropriate configurations, we naively remove all configurations performing worse than a certain suitability threshold (we chose 70%) as depicted in Figure 4.11. The remaining configurations, which are known to perform well, are then used to create the fuzzy systems.

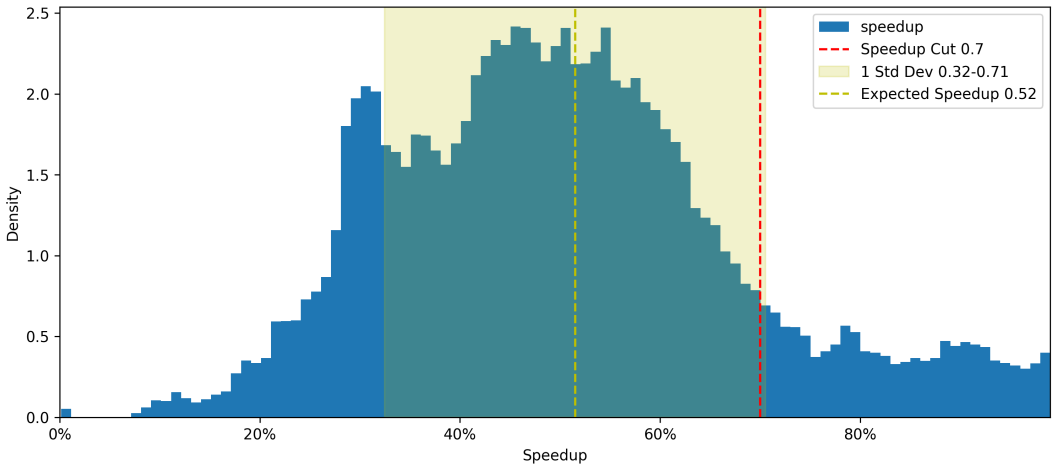


Figure 4.11.: Speedup distribution of the collected data. The suitability threshold is set to 70%, thus removing all configurations performing worse than this threshold. From the plot, we can also see that the average configuration performs just 52% as well as the best configuration, with some configurations also performing ten times worse than the best in certain tuning phases.

Afterward, we group all configurations evaluated in the same tuning phase and aggregate all the present values of tunable parameters into a single term. As we *froze* the simulation during the tuning phase, the LiveInfoData will be equal for such configurations, and the aggregated terms will therefore represent all *good* values for the parameters in this simulation state (as they occur in configurations with $\geq 70\%$ suitability). The aggregated training data is shown in Table 4.3 and is used to fit three decision trees, which are then converted into fuzzy control systems using the method described in the previous section. The extracted fuzzy rules are shown in Table 4.4. The linguistic variables are omitted for brevity.

ParticlesPerCell			Miscellaneous			Aggregated Configuration Terms		
avg	max	stddev	homo-genity	max-density	threads	Container DataLayout	Traversal	Newton3
0.906	15	0.015	0.055	0.297	4	"LinkedCells_SoA, VerletClusterLists_SoA, VerletListsCells_AoS"	"lc_sliced, lc_sliced_balanced, lc_sliced_c02"	"enabled"
0.945	25	0.041	0.084	0.673	24	"LinkedCells_SoA, VerletClusterLists_SoA, VerletListsCells_AoS"	"lc_c04, lc_c08, lc_sliced, lc_sliced_balanced"	"disabled, enabled"
0.906	20	0.014	0.041	0.336	24	VerletClusterLists_SoA, VerletListsCells_AoS	"vcl_c06, vlc_c01, vlc_c18"	"disabled, enabled"
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Table 4.3.: Aggregated training data for the Component Tuning Approach. Each row represents a different tuning phase. The numerical values stem from the LiveInfoData during that tuning phase, and the aggregated configuration terms represent the configuration options that are known to perform well under the given conditions.

Antecedent				Consequent
avgParticlesPC	homogeneity	particlesPCStdDev	threadCount	ContainerDataLayout
lower than 3.45	lower than 0.05		lower than 18.0	"VerletClusterLists_SoA, VerletListsCells_AoS"
lower than 3.45	higher than 0.05	lower than 0.024	higher than 18.0	"LinkedCells_SoA, VerletClusterLists_SoA, VerletListsCells_AoS"
⋮	⋮	⋮	⋮	⋮

Antecedent				Consequent
avgParticlesPC	homogeneity	particlesPCStdDev	threadCount	Traversal
lower than 1.553	higher than 0.047	lower than 0.023	higher than 2.5	"lc_sliced, vlc_c18, lc_sliced_c02"
	lower than 0.037	lower than 0.023	lower than 26.0	"vcl_c06, vlc_c18, vlc_sliced_c02"
⋮	⋮	⋮	⋮	⋮

Antecedent				Consequent
avgParticlesPC	homogeneity	particlesPCStdDev	threadCount	Newton 3
		higher than 0.03	higher than 18.0	"disabled, enabled"
		higher than 0.023 ∧ lower than 0.037	lower than 18.0 ∧ higher than 8.0	"enabled"
⋮	⋮	⋮	⋮	⋮

Table 4.4.: Extracted fuzzy rules from the decision trees for the Component Tuning Approach. The rules are grouped by the tunable parameter they predict. The first row is read as: IF (avgParticlesPC = lower than 3.454) ∧ (homogeneity = lower than 0.05) ∧ (threadCount = lower than 18.0) THEN (ContainerDataLayout = "VerletClusterLists_SoA, VerletListsCells_AoS")

As described previously, we use `gaussian` membership functions for each linguistic term of the consequent linguistic variables. The placement of the values is irrelevant as we will use the MOM defuzzification method. Figure 4.12 and Figure 4.13 show the resulting linguistic variables for the homogeneity linguistic variable (an input variable) and the Newton3 linguistic variable we want to predict. The visualization of the other variables follows a similar pattern but is more complex due to the higher number of terms and are therefore not shown here.

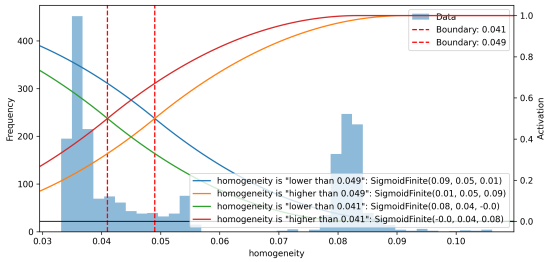


Figure 4.12.: This figure shows the linguistic variable for the homogeneity attribute. We see the different fuzzy sets created from the decision trees. The background shows the histogram of all homogeneity values present in the dataset.

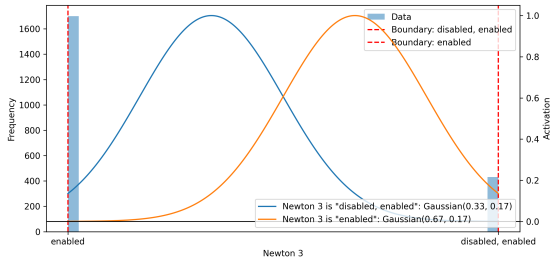


Figure 4.13.: This figure shows the linguistic variable for the Newton3 attribute. We see the two `gaussian` membership functions representing the two possible values for the Newton3 attribute.

These linguistic variables and fuzzy rules are then used to create the fuzzy control systems for the component tuning approach. In Chapter 5, we will evaluate the performance of these systems and compare them to the suitability approach.

add link to full rule file

4.2.4. Suitability Approach

The suitability approach differs from the component tuning approach in that it tries to predict a configuration's numerical *suitability* value under the current conditions. Therefore, each possible configuration is assigned a unique fuzzy system tailored to just evaluating this configuration. Figure 3.3 shows the structure of this approach.

To train the decision trees, we again use a classification-based approach with `terrible`, `bad`, `average`, `good`, and `excellent` as the possible linguistic terms corresponding to specific ranges of suitability values (see Figure 4.14 for the exact placement). Conveniently, we can use $suitability = speedup$, as the speedup value already measures how well a configuration performs.

We create the training data for the decision trees by adding a new column to the aggregated training data containing the suitability class to which the numeric speedup value mostly belongs. The final training data is shown in Table 4.5. After grouping the data by possible configurations, it is again possible to use these data points to train the decision trees and extract the fuzzy rules from them. Due to the grouping, each configuration receives its own fuzzy system with rules explicitly tailored to it. The resulting rules are shown in Table 4.6.

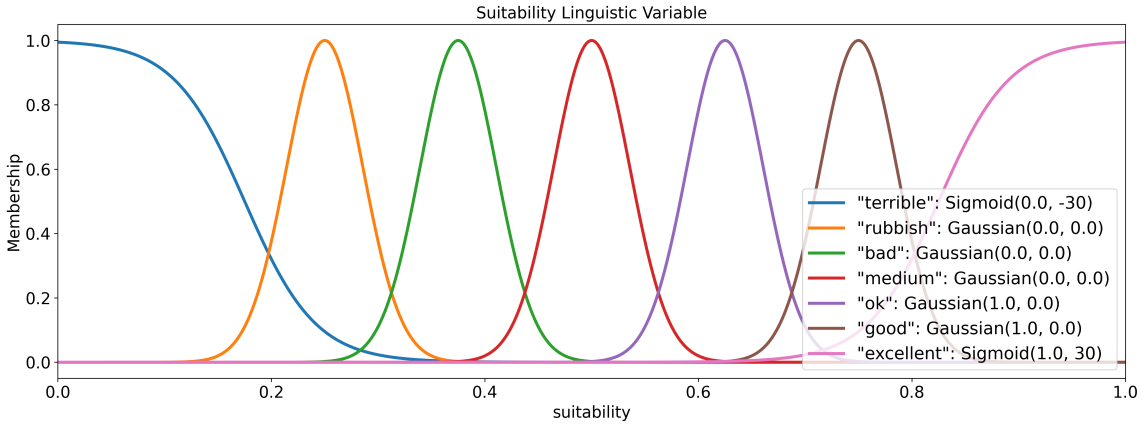


Figure 4.14.: This figure shows the linguistic variable for the Suitability attribute. The domain between 0% suitability and 100% suitability is divided into 7 classes: `terrible`, `rubbish`, `bad`, `medium`, `ok`, `good`, and `excellent`. The central membership functions have a `gaussian` shape, while the outer ones have a `sigmoid` shape to capture the one-sided nature of those boundaries. The choice to use seven classes was made somewhat arbitrarily, with the intention to densely cover the range of suitability values.

This approach makes it possible to use the COG method for defuzzification, as the suitability values are ordinal and have a clear order. In order for the different fuzzy systems to be able to make individual choices, the Suitability values seen in Figure 4.14 have to be duplicated for each configuration.

The resulting fuzzy systems can then be used to predict the suitability of all available configurations. The FuzzyTuningStrategy will combine all these predicted suitabilities and will only keep the configurations with the highest suitability values. In Chapter 5, we will evaluate the performance of this approach.

ParticlesPerCell			Miscellaneous			Configuration				
avg	max	stddev	homo-genity	max-density	threads	Container DataLayout	Traversal	Newton3	Speedup	Suitability
0.905	15	0.012	0.035	0.531	1	LinkedCells_AoS	lc sliced	enabled	0.450	"bad"
0.944	25	0.012	0.083	0.691	28	VerletClusterLists_AoS	vcl_c06	disabled	0.319	"rubbish"
0.944	20	0.012	0.079	0.041	12	LinkedCell_SoA	vlc_ sliced	enabled	0.989	"excellent"
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Table 4.5.: Training data for the Suitability Approach. The dataset contains the LiveInfoData of the simulation, the current configuration and the speedup and suitability values of the configuration. Each row represents a different configuration evaluated in a tuning phase.

Antecedent				Consequent
avgParticlesPC	homogeneity	particlesPCStdDev	threadCount	Suitability LinkedCells_AoS_lc_c01_disabled
	`lower than 0.084`	`higher than 0.029`	`higher than 26.0`	"medium"
	`higher than 0.084`	`higher than 0.029`	`higher than 26.0`	"bad"
		`higher than 0.02`	`lower than 2.5`	"rubbish"
⋮	⋮	⋮	⋮	⋮

Antecedent				Consequent
maxParticlesPerCell	homogeneity	particlesPCStdDev	threadCount	Suitability LinkedCells_AoS_lc_c04_disabled
`higher than 18.5`	`lower than 0.082`		`higher than 18.0` ∧ `lower than 26.0`	"medium"
`higher than 18.5`	`higher than 0.082`		`higher than 18.0` ∧ `lower than 26.0`	"bad"
⋮	⋮	⋮	⋮	⋮

⋮

Table 4.6.: Some extracted fuzzy rules from the decision trees for the Suitability Approach. The rules are grouped by the configuration they predict. The first row is read as: IF (homogeneity = lower than 0.084) ∧ (particlesPerCellStdDev = higher than 0.029) ∧ (threadCount = higher than 26.0) THEN (Suitability LinkedCells_AoS_lc_c01_disabled = "medium")

5. Comparison and Evaluation

In this section, we compare the fuzzy tuning technique with other tuning techniques present in AutoPas and evaluate its performance.

To measure the performance of the fuzzy tuning strategy, we also use the scenarios present in md_flexible and compare the results with the other tuning strategies present in AutoPas. The benchmarks are run on the CoolMUC-2[1] cluster and are repeated with 1, 12, 24, and 28 threads. We use the `timeSpentCalculatingForces` metric to evaluate the performance of the tuning strategies as it gives a good indication of the overall performance of the simulation.

5.0.1. Exploding Liquid Benchmark (Included in Training Data)

The exploding liquid benchmark simulates a high-density liquid that expands outwards as the simulation progresses. As the data of this scenario was included in the training data, we expect the fuzzy tuning technique to perform well. We only include the benchmark results with one thread for brevity, as the results for the other thread counts are very similar.

The plot in Figure 5.1a shows the time spent calculating the forces for each tuning strategy throughout the simulation. The fuzzy tuning strategies typically perform close to optimal and are very stable. All other tuning strategies show a much higher variance caused by testing many configurations during the tuning phases.

The low tuning overhead is the most significant contributor to the performance of the fuzzy tuning strategies. As the tuning phases of the fuzzy tuning strategies are very short and mainly consist of evaluating already known suitable configurations, there is no overhead caused by the tuning phases. This contrasts with the classical tuning strategies, which spend significant time in the tuning phases.

To show this in more detail, we also include a boxplot of the time spent calculating the forces for each tuning strategy based on the current phase in Figure 5.1b. All tuning strategies show similar timings during the simulation phases, as they eventually found a perfect configuration during the tuning phases but differ drastically in the tuning phases. The fuzzy tuning strategies have a much lower median time spent during tuning phases, with the individual tuning approach performing best. We see that the suitability approach performs worse than the other strategies during simulation phases because the suitability approach chose a suboptimal configuration for the first simulation phase, which was then corrected from the second tuning phase onwards. All other strategies eventually found a perfect configuration during the tuning phases, which caused them to perform better during the simulation phases.

[1] CoolMUC-2 is a supercomputer located at the Leibniz Supercomputing Centre in Garching, Germany. It consists of 812 Haswell-based nodes with 14 cores each. As a result of hyperthreading, each node supports up to 28 threads. More information can be found at `https://doku.lrz.de/coolmuc-2-11484376.html`

This plot also shows that the interquartile range of the classical tuning strategies is very similar, with all having nearly identical means. However, all of them are plagued by massive outliers, sometimes taking ten times longer than the median configuration and up to 100 times longer than the optimal configuration. Those extremely bad configurations are the main reason for the poor performance of the classical tuning strategies.

The last plot in Figure 5.1c shows the total time spent calculating the forces for each tuning strategy, again divided into simulation and tuning time. The fuzzy tuning strategies have the lowest total time, with practically no time spent in the tuning phases. Both fuzzy tuning approaches perform similarly and are by far the best-performing strategies. All other strategies typically spend more than 50% of their time in tuning phases where they potentially encounter very bad configurations, which causes them to perform much worse than the fuzzy tuning strategies.

5.0.2. Spinodal Decomposition Benchmark MPI (Related to Training Data)

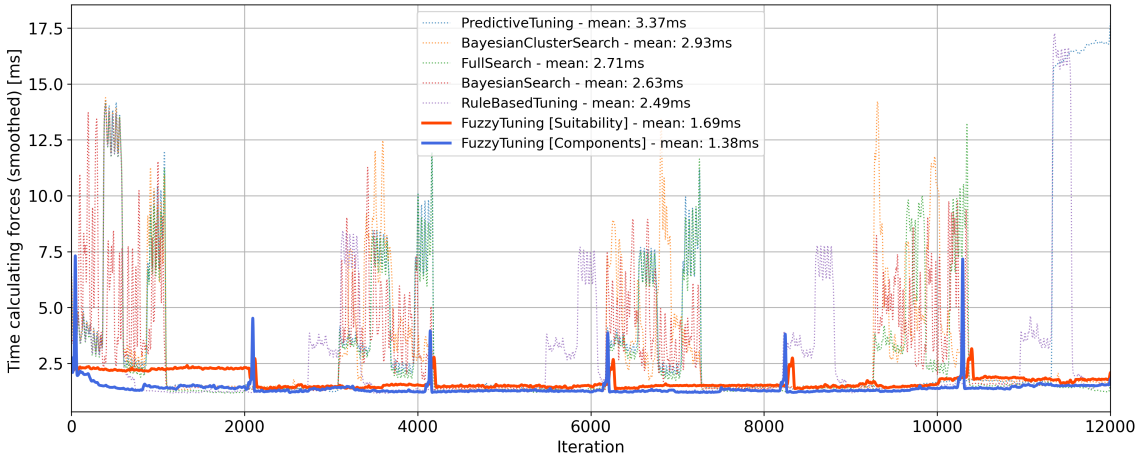
The spinodal decomposition benchmark simulates an unstable liquid that separates into two phases, each having different characteristics. To improve the performance of the simulation, we used four different MPI ranks, each running on 14 threads to simulate the scenario. As the complete spinodal decomposition benchmark was included in the training data, and the scenario is very homogeneous, we expect the fuzzy tuning strategies to also perform well in this scenario, even if there are no direct training data points for the division of the simulation into multiple MPI ranks.

For brevity, we only include the benchmark results for the 0th MPI rank, as the results for the other MPI ranks are nearly identical.

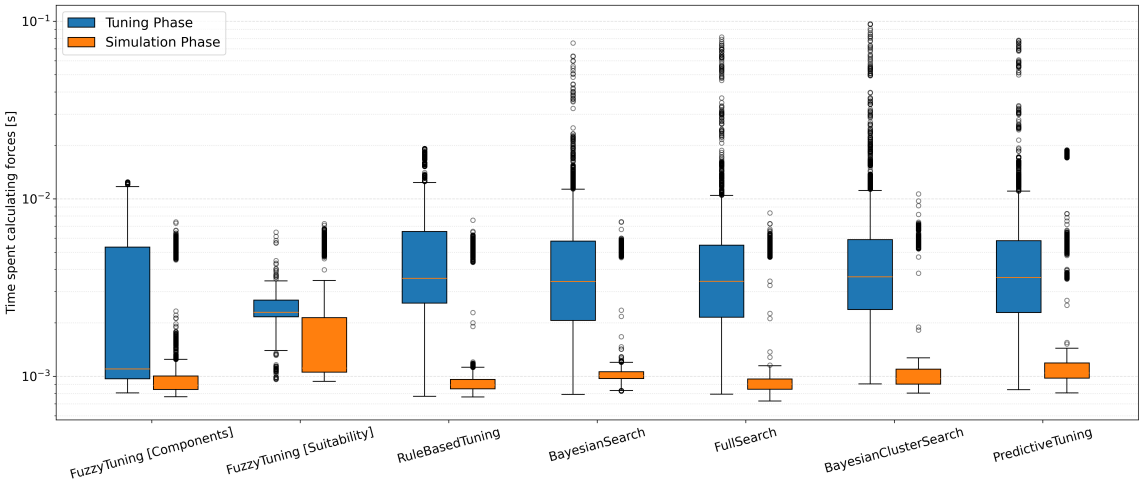
The plot in Figure 5.2a shows the time spent calculating the forces for each tuning strategy throughout the simulation. This time, we see a difference in both fuzzy tuning strategies, as the component tuning approach performs way better than the suitability approach for most of the simulation. By looking at the boxplots in Figure 5.2b, we see that the suitability approach has the lowest median time spent during the tuning phases. However, it struggles to find the optimal configurations.

Currently, the rule file for the suitability approach specifies that only the top 10% of configurations with the highest suitability should be selected. Increasing this threshold and spending more time in the tuning phase may be beneficial, as currently, the most considerable slowdown is caused by the inability to find the optimal configuration. From Figure 5.2c, we can confirm that the suitability approach spends the least time in the tuning phases, and it could be very worthwhile to increase this limit to improve the total performance.

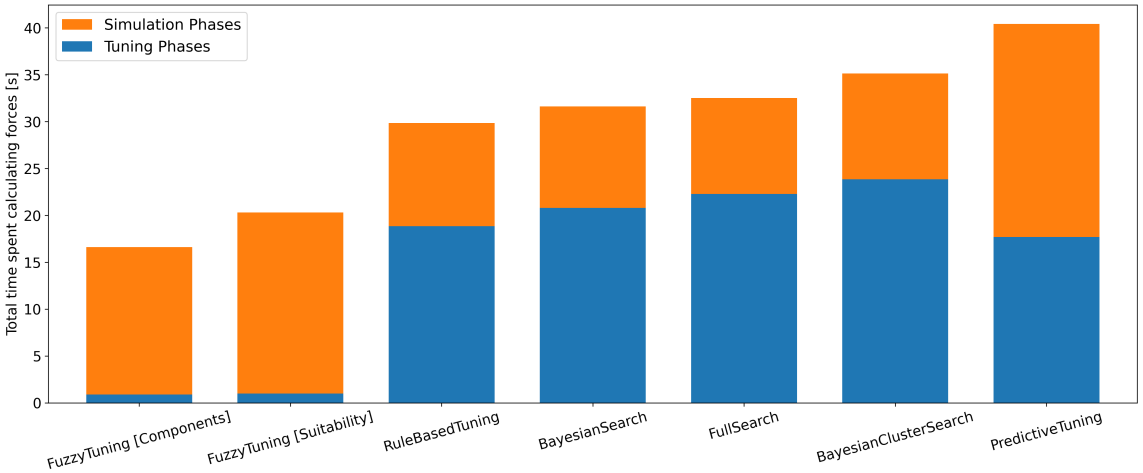
Figure 5.2 shows that the component tuning approach again performs best, with a decrease in simulation time of around 30% compared to the FullSeach- and BayesianOptimization-based tuning strategies. The suitability and predictive tuning approaches perform similarly and are in second place. Remarkably, the suitability approach performed reasonably well despite never finding the optimal configuration, mainly due to basically no time wasted during the tuning phases. This shows the importance of efficient tuning phases, as they can cause tremendous overhead if not done correctly.



(a)

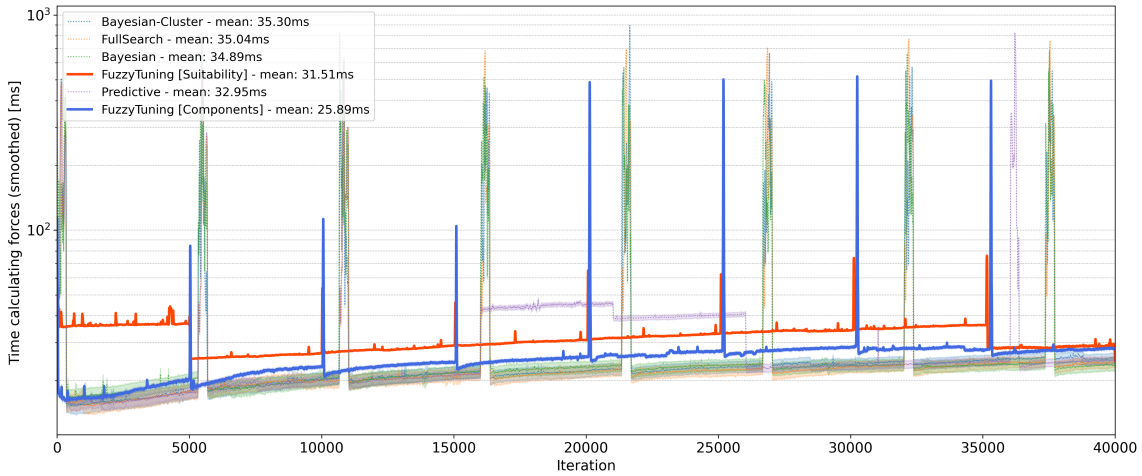


(b)

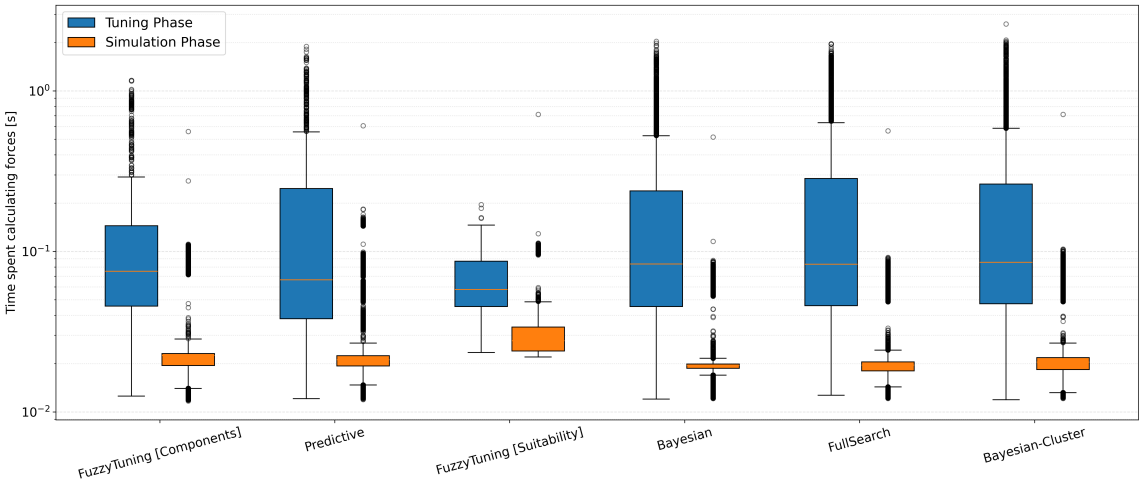


(c)

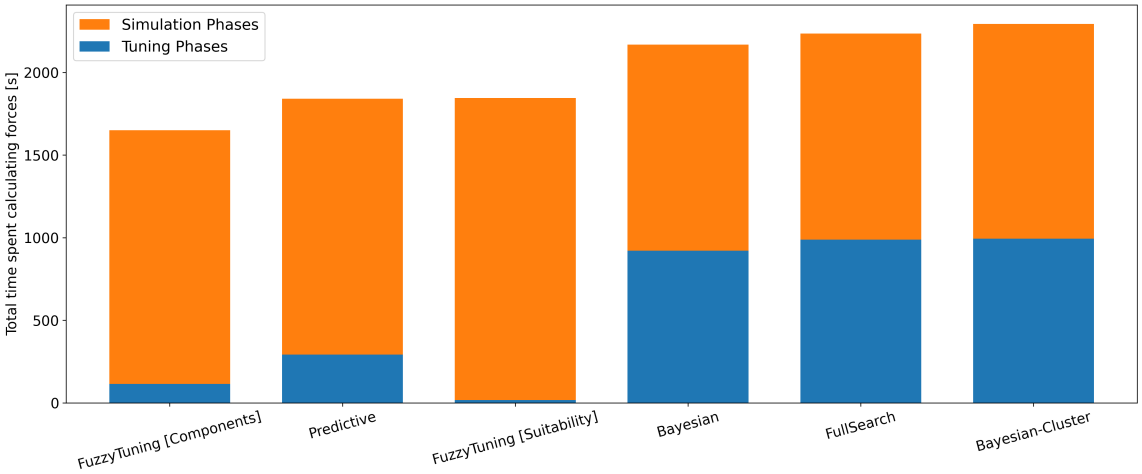
Figure 5.1.: Exploding liquid benchmark with 1 thread. (a) Time spent calculating forces. (b) Boxplots of time spent calculating forces divided into simulation and tuning phases. (c) Total time spent calculating forces



(a)



(b)



(c)

Figure 5.2.: 0th Rank of the Spinodal decomposition benchmark (Total: 4 MPI ranks, 14 threads each). (a) Time spent calculating forces. (b) Boxplots of time spent calculating forces divided into simulation and tuning phases. (c) Total time spent calculating forces

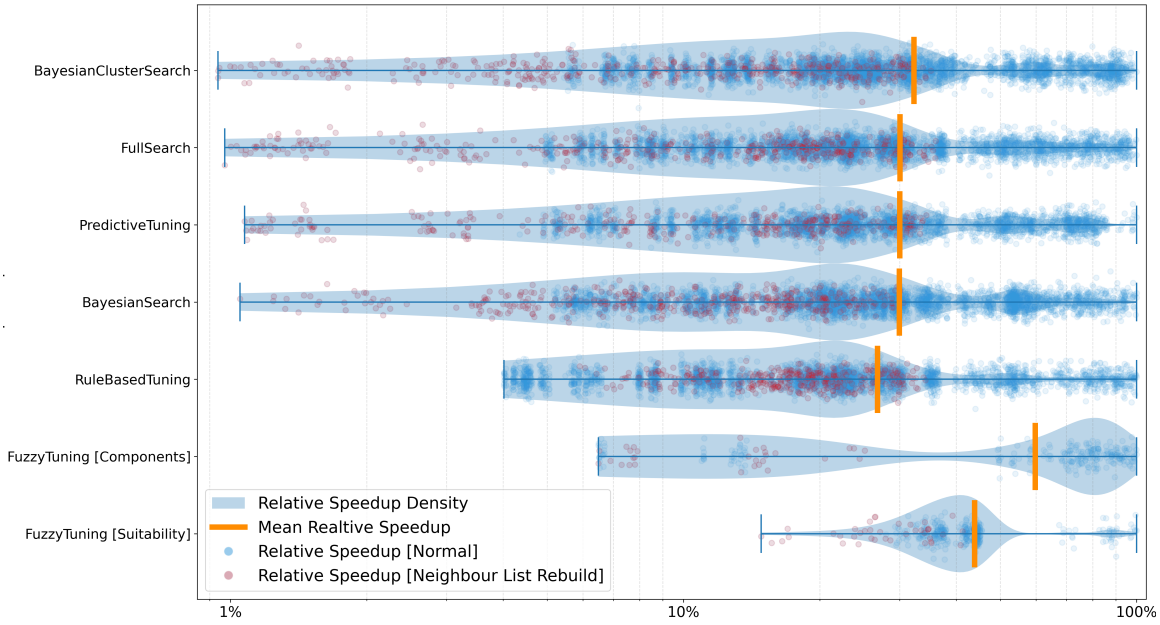
5.0.3. General Observations

As described above, a tremendous slowdown of the classical tuning strategies is caused by very bad configurations that are sometimes encountered during the tuning phases. To further illustrate this, we will investigate the speedup density distribution of all configurations evaluated during the tuning phases of the different strategies. In particular, we will look at the Exploding Liquid and Spinodal Decomposition MPI scenarios described above, as they represent *small* and *large* scenarios, respectively.

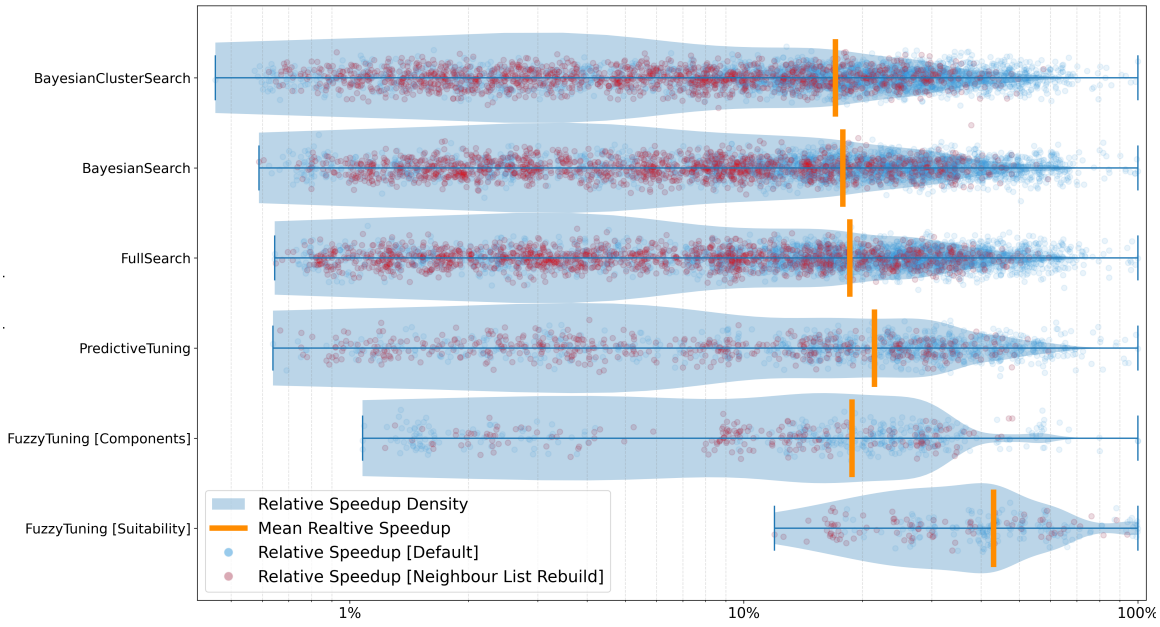
The plots in Figure 5.3 show these relative speedup distributions. All non-rule-based tuning strategies tend to encounter configurations with extremely low speedups during the tuning phases. Some of those bad configurations are ≈ 100 times slower than the best configurations of the tuning phase. As seen in the previous plots, those very slow configurations completely ruin the performance of the classical tuning strategies during the tuning phases.

It is also interesting to note that most of those bad performances are caused by very slow Neighbour List rebuilds, especially in the Spinodal Decomposition MPI scenario. This means that tuning strategies with very short tuning phases, like the fuzzy tuning strategies, have a significant advantage, as they only tend to evaluate configurations expected to suit the current state.

In both scenarios, the fuzzy tuning strategies have the best speedup distributions and, by far, the highest mean speedup during the tuning phases. However, this does not automatically mean they will perform their best in the following simulation phase, as they could fail to predict the optimal configuration.



(a) Exploding Liquid scenario with one thread.



(b) Rank 0 of the Spinodal Decomposition MPI scenario with 14 threads.

Figure 5.3.: The plot shows the speedup density distribution of all configurations evaluated during the tuning phases of both the Exploding Liquid and Spinodal Decomposition MPI scenarios. Most slow configurations are caused by the very slow Neighbour List rebuilds, especially in the Spinodal Decomposition MPI scenario, as there are many more particles.

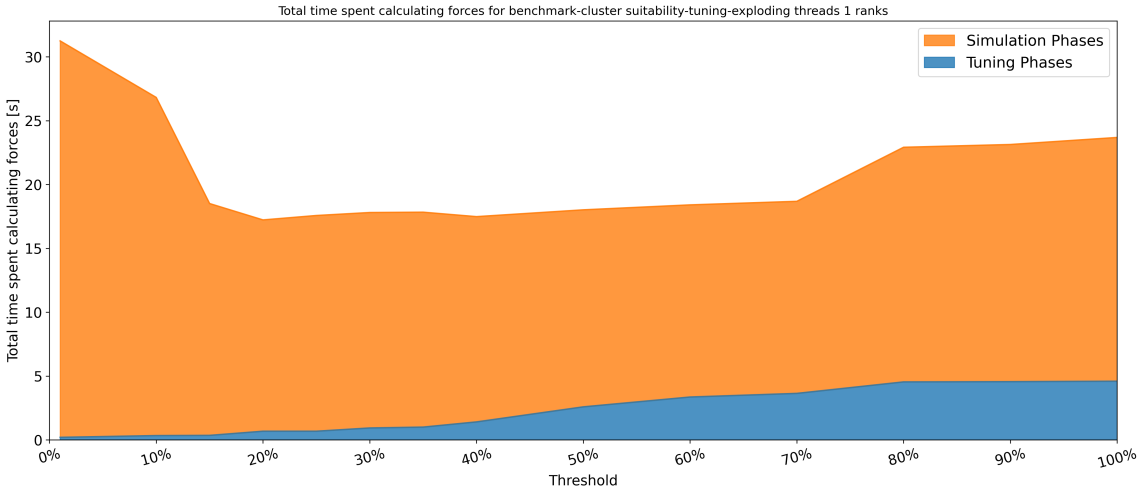


Figure 5.4.: Exploding liquid benchmark with different suitability thresholds. The suitability threshold specifies the width of the interval around the best configuration, which determines how many configurations are selected for the tuning phase. Very low thresholds perform poorly, as they leave no margin for error and fail to select the best configuration. Very high thresholds also perform poorly, as high suitability values cause the strategy to behave like FullSeach. The optimal threshold for this scenario is between 20% and 40%.

6. Future Work

We demonstrated that the expert knowledge extracted from the collected dataset can lead to a significant reduction of tuning time and can provide an overall improvement in the performance of the simulation. However, the expert knowledge is imperfect and can be improved in several ways. In this chapter, we discuss some of the possible improvements that can be made to the expert knowledge and the data collection process.

6.1. Better Data Collection

The data collection process can be improved in several ways. One of the main limitations of the current data collection process is that it is challenging and possibly impossible to collect a dataset representative of all possible scenarios. The current dataset is limited to a few scenarios, and the expert knowledge extracted from this dataset may not apply to other scenarios.

We showed that using the fuzzy-tuning strategy leads to a near-optimal prediction of configurations when the current scenario is represented in the dataset, as it is probably impossible to account for all possible scenarios. One could look into adaptively updating the expert knowledge as new scenarios are encountered. This could be done by spending extra time during the simulation to evaluate the performance data of recently executed configurations and update the expert knowledge accordingly.

6.2. Verification of Expert Knowledge

The currently used expert knowledge is directly extracted from the collected dataset without further validation. It would be interesting to investigate ways of validating the expert knowledge to rule out implausible rules and insert concepts not currently covered by the expert knowledge.

6.3. Future Work on Tuning Strategies

Another follow-up tuning strategy that could be investigated is using adaptive neuro-fuzzy inference systems (ANFIS). Those systems combine neural networks and fuzzy logic and the learning capabilities of neural networks with the uncertainty-handling capabilities of fuzzy logic. ANFIS systems could be used to predict the suitability values of the configurations with way more flexibility than currently allowed by expert knowledge.

7. Conclusion

7.1. A

A. Appendix

A.1. Glossary

AutoPas Node-level auto-tuned particle simulation library written in C++. See `https://github.com/AutoPas/AutoPas`. 33, 49–51

COG Center of Gravity Defuzzification Method. 24, 31, 32, 38

FCS Fuzzy Control System. 23

md_flexible A flexible molecular dynamics simulation framework built on top of AutoPas. 6, 27, 33, 40

MOM Mean of Maximum Defuzzification Method. 24, 31, 32, 37

SOM Start of Maximum Defuzzification Method. 24

A.2. IterationLogger Fields

The following fields are currently available in the IterationData file. The IterationData file is a CSV file containing the configuration and performance data for each iteration of the simulation. The data is collected and logged by the `IterationLogger` class of the AutoPas library.

Date	The date and time when the data was collected.
Iteration	The current iteration number of the simulation.
inTuningPhase	Indicates whether the simulation is in the tuning phase (true/false).
Container	The type of container used to store the particles in the simulation (e.g., LinkedCells, VerletLists).
CellSizeFactor	A factor that determines the size of the cells relative to the cutoff radius.
Traversal	The method used to traverse the cells and calculate interactions between particles.
Load Estimator	The strategy used to estimate and balance the computational load across different parts of the simulation domain.
Data Layout	The arrangement of particle data in memory (e.g., AoS for Array of Structures, SoA for Structure of Arrays).
Newton 3	Indicates whether Newton's third law optimization is used to reduce computation by only calculating forces once per particle pair (enabled/disabled).
iteratePairwise	The time taken to iterate over all particle pairs and calculate interactions.
remainderTraversal	The time taken to perform any remaining traversal operations.
rebuildNeighborLists	The time taken to rebuild the neighbor lists.
iteratePairwiseTotal	The total time taken to iterate over all particle pairs and calculate interactions.
tuning	The time taken to perform any tuning operations.
energyPsys	The energy consumed by the system (in Joules).
energyPkg	The energy consumed by the package (CPU) (in Joules).
energyRam	The energy consumed by the RAM (in Joules).

A.3. TuningDataLogger Fields

The following fields are currently available in the TuningData file. The TuningData file is a CSV file containing the performance information for all tested configurations. The data is collected and logged by the `TuningDataLogger` class of the AutoPas library.

Date	The date and time when the data was collected.
Iteration	The current iteration number of the simulation.
Container	The type of container used to store the particles in the simulation (e.g., LinkedCells, VerletLists).
CellSizeFactor	A factor that determines the size of the cells relative to the cutoff radius.
Traversal	The method used to traverse the cells and calculate interactions between particles.
Load Estimator	The strategy used to estimate and balance the computational load across different parts of the simulation domain.
Data Layout	The arrangement of particle data in memory (e.g., AoS for Array of Structures, SoA for Structure of Arrays).
Newton 3	Indicates whether Newton's third law optimization is used to reduce computation by only calculating forces once per particle pair (enabled/disabled).
Reduced	The reduced performance data for all sample points is calculated by aggregating the data across all sample points. The specific aggregation method can be configured via the `.yaml` configuration file.
Smoothed	A smoothed version of the reduced performance data.

A.4. LiveInfoLogger Fields

The following fields are currently available in the LiveInfoData file. The LiveInfoData file is a CSV file containing summary statistics about the simulation state at each iteration. The data is collected and logged by the `LiveInfoLogger` class of the AutoPas library.

Iteration	The current iteration number of the simulation.
avgParticlesPerCell	The average number of particles per cell in the simulation domain.
cutoff	The cutoff radius for the interaction of particles, beyond which particles do not interact.
domainSizeX	The size of the simulation domain in the X dimension.

domainSizeY	The size of the simulation domain in the Y dimension.
domainSizeZ	The size of the simulation domain in the Z dimension.
estimatedNumNeighborInteractions	The estimated number of neighbor interactions between all particles in the simulation domain.
homogeneity	A measure of the distribution uniformity of particles across the cells.
maxDensity	The maximum density of particles in any cell.
maxParticlesPerCell	The maximum number of particles found in any single cell.
minParticlesPerCell	The minimum number of particles found in any single cell.
numCells	The total number of cells in the simulation domain.
numEmptyCells	The number of cells that contain no particles.
numHaloParticles	The number of particles in the halo region (boundary region) of the simulation domain.
numParticles	The total number of particles in the simulation domain.
particleSize	The number of bytes used to store a single particle in memory.
particleSizeNeededByFunctor	The particle size required by the functor (the function used for calculating interactions).
particlesPerBlurredCellStdDev	The standard deviation of the number of particles per blurred cell provides a measure of particle distribution variability.
particlesPerCellStdDev	The standard deviation of the number of particles per cell, indicating the variability in particle distribution.
rebuildFrequency	The frequency at which the neighbor list is rebuilt.
skin	The skin width is added to the cutoff radius to create a buffer zone for neighbor lists, ensuring efficient interaction calculations.
threadCount	The number of threads used for parallel processing in the simulation.

A.5. Scenarios used for Data Generation

The scenarios used for data generation stem directly from the examples provided in the `AutoPas` library. Their state at the time of the data generation can be looked up at `https://github.com/AutoPas/AutoPas/tree/563528b4ef55d29b8ec148f6387f0be1adb400ed/examples/md-flexible/input`

The specific scenarios used are: `explodingLiquid`, `spinodalDecompositionEquilibration`, `spinodalDecomposition`, and `fallingDrop`.

A.6. Data Analysis

We performed some exploratory data analysis to gain insights into the collected dataset. The results are presented in the following figures.

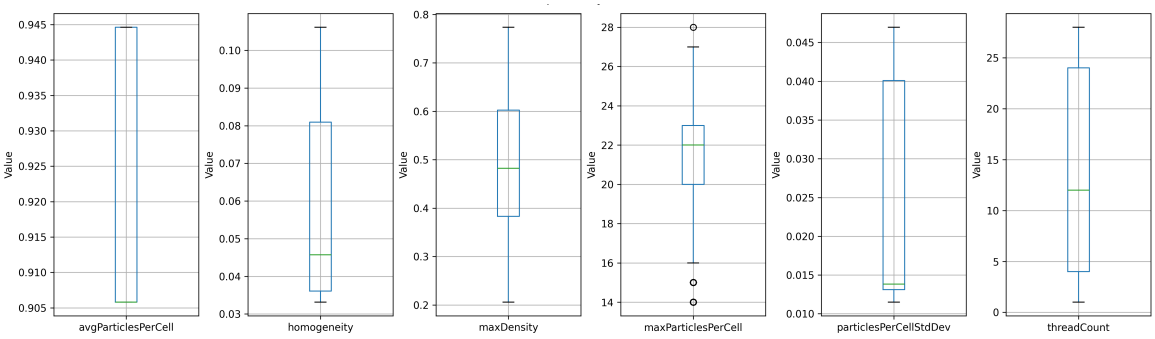


Figure A.1.: Boxplot showing the distribution of the collected data of the LiveInfoData files

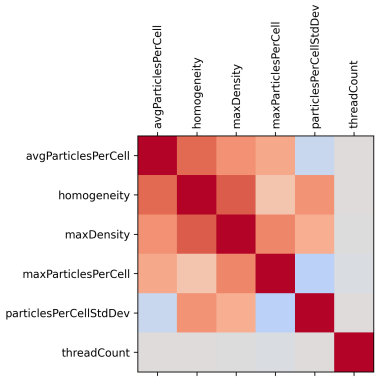


Figure A.2.: Correlation matrix showing the correlation between the collected parameters of the LiveInfoData files. We can see that many of the collected parameters are slightly positively correlated with each other.

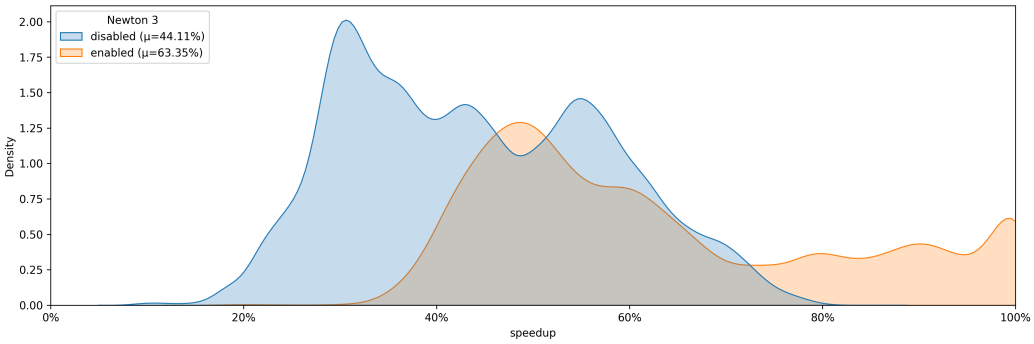


Figure A.3.: Density plot showing the distribution of the relative speedup based on the Newton 3 option. We can see that Newton3=enabled is generally the better option as it generally allows for a higher relative speedup. All performances with relative speedups of at least 80% use the Newton3 optimization. Therefore, we can confirm that Newton 3 is generally a good option.

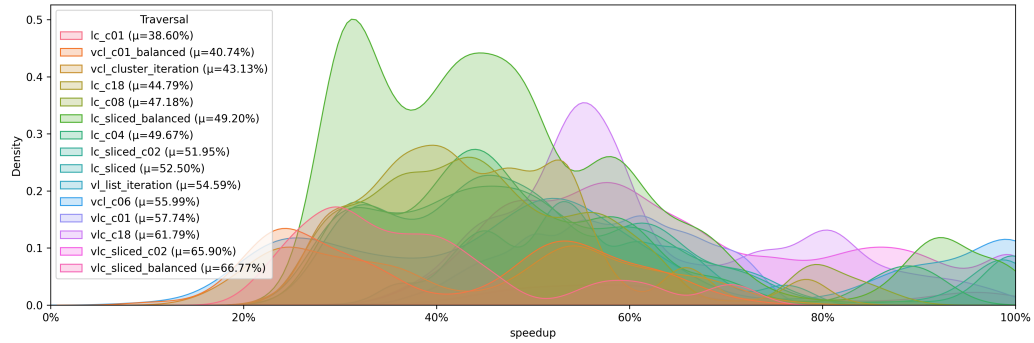


Figure A.4.: Density plot showing the distribution of the relative speedup based on the Traversal option. We can see that the vlc_sliced_balanced option generally performed better than the other options on this dataset with an expected relative speedup of 66%.

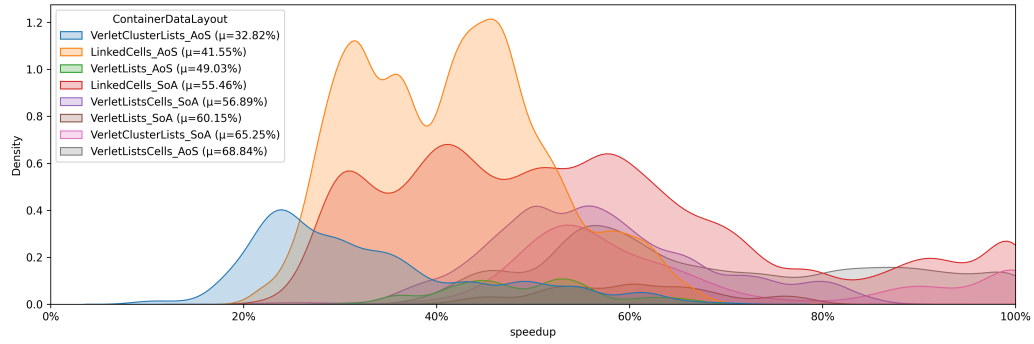


Figure A.5.: Density plot showing the distribution of the relative speedup based on the ContainerDatalayout option. The VerletListCells_AoS ContainerDatalayout performed best on this dataset with an expected relative speedup of 68.8%.

List of Figures

List of Tables

Bibliography

[BMK96] Bernadette Bouchon-Meunier and Vladik Kreinovich. Axiomatic description of implication leads to a classical formula with logical modifiers: (in particular, mamdani's choice of "and" as implication is not so weird after all). 1996.

[BPR+16] Tobias Brink, Martin Peterlechner, Harald Rösner, Karsten Albe, and Gerhard Wilde. Influence of crystalline nanoprecipitates on shear-band propagation in cu-zr-based metallic glasses. *Phys. Rev. Appl.*, 5:054005, May 2016.

[CBMO06] Keeley Crockett, Zuhair Bandar, David Mclean, and James O'Shea. On constructing a fuzzy inference framework using crisp decision trees. *Fuzzy Sets and Systems*, 157(21):2809–2832, 2006.

[Che] Chemie.de. Lennard-jones-potential. `https://www.chemie.de/lexikon/Lennard-Jones-Potential.html`. Accessed: 2024-07-07.

[GSBN21] Fabio Alexander Gratl, Steffen Seckler, Hans-Joachim Bungartz, and Philipp Neumann. N ways to simulate short-range particle systems: Automated algorithm selection with the node-level library autopas. *Computer Physics Communications*, 273:108262, 2021.

[GST+19] Fabio Alexander Gratl, Steffen Seckler, Nikola Tchipev, Hans-Joachim Bungartz, and Philipp Neumann. Autopas: Auto-tuning for particle simulations. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 748–757, 2019.

[Iri21] Karl Irikura. Physics-guided curve fitting for potential-energy functions of diatomic molecules. *Authorea*, April 2021.

[JPRS06] E. Jodoin, C.A. Pena Reyes, and E. Sanchez. A method for the fuzzification of categorical variables. In *2006 IEEE International Conference on Fuzzy Systems*, pages 831–838, 2006.

[LM15] Benedict Leimkuhler and Charles Matthews. *Molecular Dynamics: With Deterministic and Stochastic Numerical Methods*. Interdisciplinary Applied Mathematics. Springer, May 2015.

[MAMM20] C Y Maghfiroh, A Arkundato, Misto, and W Maulina. Parameters (sigma, epsilon) of lennard-jones for fe, ni, pb for potential and cr based on melting point values using the molecular dynamics method of the lammps program. *Journal of Physics: Conference Series*, 1491(1):012022, October 2020.

[MKEC22] Ali Mohammed, Jonas H. Müller Korndörfer, Ahmed Eleliemy, and Florina M. Ciorba. Automated scheduling algorithm selection and chunk parameter calculation in openmp. *IEEE Transactions on Parallel and Distributed Systems*, 33(12):4383–4394, 2022.

[Mur12] Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective*. The MIT Press, 2012.

[NGNB23] Samuel James Newcome, Fabio Alexander Gratl, Philipp Neumann, and Hans-Joachim Bungartz. Towards auto-tuning multi-site molecular dynamics simulations with autopas. *Journal of Computational and Applied Mathematics*, 433:115278, 2023.

[Phy] Nexus Physics. The lennard-jones potential. `https://www.compadre.org/nexusph/course/The_Lennard-Jones_Potential`. Accessed: 2024-07-07.

[PS17] Juan R. Perilla and Klaus Schulten. Physical properties of the hiv-1 capsid from all-atom molecular dynamics simulations. *Nature Communications*, 8(1):15959, 2017.

[RdCC12] Pilar Rey-del Castillo and Jesús Cardeñosa. Fuzzy min-max neural networks for categorical data: application to missing data imputation. *Neural Computing and Applications*, 21(7):1349–1362, 2012.

[SGH+21] Steffen Seckler, Fabio Gratl, Matthias Heinen, Jadran Vrabec, Hans-Joachim Bungartz, and Philipp Neumann. Autopas in ls1 mardyn: Massively parallel particle simulations with node-level auto-tuning. *Journal of Computational Science*, 50:101296, 2021.

[VBC08] G. Viccione, V. Bovolin, and E. Pugliese Carratelli. Defining and optimizing algorithms for neighbouring particle identification in sph fluid simulations. *International Journal for Numerical Methods in Fluids*, 58(6):625–638, 2008.

[ZBB+13] Franck Zielinski, Pierre Baudin, Gérard Baudin, Pierre Baudin, and Gérard Baudin. Quantum states of atoms and molecules. *Chemical Education Digital Library*, 1(1):1–10, 2013.