Project Report: Exploring Performance Modeling in AutoPas

Tobias Humig

August 2023

Abstract

The particle simulation library AutoPas implements many algorithms with vastly different performance characteristics to solve the pairwise short-range particle interactions in molecular dynamics simulations. During the simulation, it uses black-box optimization techniques to automatically select the fastest algorithm for the current state. While they are able to find good algorithms eventually, they often try highly unsuitable ones at the start due to lack of initial performance information. As some algorithms perform orders of magnitude worse than the optimum for a given simulation state, this has a significant negative impact on the time to solution.

In this project, we gather knowledge about the performance characteristics of the algorithms through theoretical modeling, profiling, and benchmarking. We make the results available through a new white-box optimization strategy that is able to apply any domain-specific knowledge during optimization. It removes those algorithms from the candidate list that likely perform worst in the current simulation state. In our tests, removing the five percent slowest algorithms reduced the tuning time by up to 80 percent while still finding the best algorithm. Furthermore, we give insights and recommendations what can be done to potentially improve the performance further.

Contents

1 Introduction

1.1 Motivation

Implementing a fast simulator for particle simulations is challenging. Varying parameters of a simulation like particle density and distribution lead to very different compute requirements and memory access patterns. Thus, fully utilizing the available hardware across a range of these parameters is difficult. AutoPas tackles this problem by implementing (at the time of writing) 186 different algorithms and picking the most suitable one for the current simulation state, automatically.

For a given simulation, the algorithms perform orders of magnitude different compared to each other. AutoPas implements several black-box optimization strategies to choose the best algorithm for the current simulation. These strategies need to execute each algorithm at least once to collect data about its performance. This is problematic, because testing a single algorithm that takes three orders of magnitude longer than the optimal algorithm (something we did observe) negatively impacts the end-to-end simulation time. In practice, this means that sometimes, while AutoPas has implemented a fast algorithm for a simulation state, the time of finding it supersedes the improved simulation performance of the specialized algorithm.

1.2 Contribution

In this thesis, we explore how this problem can be solved by extending the scope of optimization strategies for selecting the best algorithm to white-box optimization strategies. These have pre-defined information about the specific optimization space they are working in. In our case, we give them domain-specific knowledge of the AutoPas developers about the available algorithms. This allows them to skip the worst performing algorithms without testing them even once while searching for the best one. Since tuning time is often dominated by a few very long-running algorithms, this has a great effect on end-to-end simulation time.

We built the new white-box optimization in such a way that it can be used for additional improvements in the future. Our ideas for that include the following: At first, we might be able to predict the potential of a new tuning phase by observing parameters that influence the performance of the algorithms. This way, we can trigger a new tuning phase early by scheduling it once the parameters changed more than a certain threshold, or we can postpone the next tuning phase if we see that the influential parameters did not change much. Second, in the future we might be able to finish a tuning phase early by modeling the expected optimal runtime and stopping a tuning phase once we found an algorithm that is close enough to the optimum. This would also open up another dimension of optimization, that is the order in which the algorithms are tested during tuning.

1.3 Approach

We divided our work in four steps, Profiling AutoPas, Theoretical Modeling, Rule-Based Tuning, and High-Variety Benchmarks. Each of these four steps is described in detail in its own section.

In Section 3, we describe the first step, Profiling AutoPas. Its goal was to understand the behavior of different algorithms and collect influential factors on the performance of them. This knowledge is then used in the second step, Theoretical Modeling. This is described in Section 4. Here, we derive closed formulas for basic properties of these algorithms like compute requirements, depending on parameters of the given simulation. In Section 5, we describe our extensible tuning strategy Rule-Based Tuning that is able to make use of the acquired domain-specific knowledge. Finally, we describe our high-variety benchmarks to collect information for writing rules in our Rule-Based Tuning in Section 6.

After our main contribution, we describe our work on build time improvements in Section 7, and some ideas for future work on AutoPas in Section 8 before we conclude in Section 9.

Before explaining our contribution, we give a short background on the Linked Cells and the Verlet Lists algorithm in Section 2.

```
def linked_cells_iteration():
  for cell in cells:
    for particle in cell:
      for neighbor in (neighbor_cells(cell) + cell):
        # Check that the neighbor particle is not the
        # particle itself, and that it is within the
        #cutoff radius
        if 0 < distance(particle, neighbor) <= cutoff:
          interact(particle, neighbor)
```

Figure 1: Pseudocode for the Linked Cells algorithm archetype.

2 Background

The two major algorithm archetypes for solving short-range particle simulations in AutoPas are Linked Cells and Verlet Lists. We present these shortly as a basis for the following topics.

Linked Cells partitions the domain into equally sized cells where a list of particles within a given cell can be accessed in $O(1)$. To find all neighbors for a particle, only particles in cells within the cutoff radius have to be checked. If the cell side-length is chosen to be greater or equal the cutoff radius of the short-range particle interaction, only the immediate 26 neighboring cells of the cell the given particle is in need to be checked. Assuming a constant maximum of particles per cells, this results in $O(n)$ runtime of this algorithm. Figure 1 shows pseudocode for the Linked Cells algorithm archetype.

When using the Verlet Lists algorithm, for each particle, a list of neighbor particles in a given distance is maintained. This distance, called interaction distance, has to be greater or equal to the cutoff radius. In this thesis, we call the factor by which the interaction distance is higher than the cutoff radius the skin factor, and skin is the difference of the interaction distance and the cutoff. If the skin factor is chosen larger, the neighbor lists have to be updated less often, assuming the particles move with the same velocity. For building and updating the neighbor lists in $O(n)$, the Linked Cells algorithm is typically used. Iterating through the neighbor list and updating them using Linked Cells has $O(n)$ total runtime. Figure 2 shows pseudocode for the Verlet Lists algorithm archetype.

```
def verlet_lists_iteration():
  if rebuild necessary:
    build Linked Cells with cell size = interaction distance
    use LinkedCells to build neighbor lists
  for particle in particles:
    for neighbor in particle.neighbor_list:
      # Check that the neighbor is within the cutoff radius
      if distance(particle, neighbor) <= cutoff:
        interact(particle, neighbor)
```

Figure 2: Pseudocode for the Verlet Lists algorithm archetype.

3 Profiling AutoPas

This section describes how we profiled AutoPas to understand the runtime behavior of various algorithms and get an idea of the major factors that influence their performance.

3.1 Method

We profiled AutoPas on our own desktop machine that runs Arch Linux with a Intel (R) Core (TM) i7-8700K CPU @3.70Ghz, and 32GB DDR4 RAM at 2133MHz. While profiling, we set the frequency governor to performance using `$ sudo cpupower frequency-set --governor performance` to minimize the impact of power-saving or turbo boost modes of the CPU, and allow all users full access to the Linux `perf` subsystem to collect performance data using `$ echo -1 | sudo tee /proc/sys/kernel/perf_event_paranoid`. The simulations where run using `md-flexible`. It was compiled using GCC 10.2.

In all cases, we made sure that as few other processes as possible were running on the system. This is especially important for multithreaded profiling where we expect that AutoPas uses 100 percent of the CPU. In these cases, it can make sense to configure AutoPas to use one or two cores less than available to stabilize the performance. Furthermore, we configured the simulation to have a runtime between a few seconds and a minute using the number of iterations configuration parameter. A shorter runtime leads to high instability of the results, a longer runtime to a high volume of data to analyze which takes longer, and potentially to CPU throttling due to high temperature. We used the tool `sensors` while running longer multithreaded simulations to observe the maximum CPU temperature. The `deltaT` configuration parameter was set to zero to only measure pair-wise force interaction time and have the simulation state static.

We used several profiling tools available for Linux, Intel VTune Profiler, `perf stat`, `perf record`, and Intel Advisor. These are described below in more detail.

3.1.1 Intel VTune Profiler

The Intel VTune Profiler[1] is an easy-to-use profiling tool that gives an accessible overview of various performance metrics of a program in a graphical user interface. Starting from the Performance Snapshot Analysis, it guides the user through various analysis types that are especially interesting for the current program. We found especially useful that VTune measures the maximum achievable memory bandwidth in the Memory Access Analysis, and displays how well the available memory bandwidth is utilized by the program.

3.1.2 `perf stat`

We used `perf stat`[2] to count specific interesting CPU hardware events that show information like instructions per clock (IPC), percentage of branch misses, last level cache misses, average load latency, and more. It is fast and easy to use on the command line. Looking at the right events gives insights into what is the bottleneck of the program execution, for example whether memory bandwidth or memory latency is the limiting factor. Which events are available depends, on the concrete CPU the program is executed on. A list of events can be retrieved using `perf list`. We found that perf Metric Groups are especially helpful to find interesting events and interpret them. These are listed in `perf list` under `Metric Groups`, as well.

3.1.3 `perf record`

To see where time is spent in case of surprising runtime behavior, for example when the majority of time is not spent in pairwise force calculation, we used `perf record`[3]. This tool regularly samples stack traces during program execution to give an overview in which parts of the code time is spent. While `perf` has a built-in profile viewer, it is hard to use. Alternative viewers that provide a good user experience are the Firefox Profiler[4], `pprof`[5] by Google, and the CLion profiler[6] which can be used to collect the profile as well, directly.

3.1.4 Intel Advisor

Intel Advisor[7] is another profiler from Intel that shows a Roofline analysis of the important loops in your program. This is very useful to determine whether the program is more memory or CPU bound, and how much potential performance

[1] https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html

[2] https://man7.org/linux/man-pages/man1/perf-stat.1.html

[3] https://man7.org/linux/man-pages/man1/perf-record.1.html

[4] https://github.com/firefox-devtools/profiler/blob/main/docs-user/guide-perf-profiling.md

[5] https://github.com/google/perf_data_converter

[6] https://www.jetbrains.com/help/clion/cpu-profiler.html#InterpretingTheResults_FlameChart

[7] https://www.intel.com/content/www/us/en/developer/tools/oneapi/advisor.html

improvement there is left solely from optimizing the implementation without changing the algorithm.

3.2 Selection of Scenarios

Profiling all 186 algorithms on a wide range of particle simulation scenarios manually would be an enormous undertaking, so we had to limit ourselves to a small subset of interesting algorithms and scenarios. The goal of this profiling is to get a feeling for how various parameters of the simulation influence the performance of the algorithms. For the majority of profiles, we selected the Linked Cells C18 algorithm with Newton 3 enabled, both with AoS and SoA data layout, and the Verlet Lists Cells C18 algorithm with Newton 3 enabled and data layout AoS. We chose these, because they represent both Verlet Lists and Linked Cells as the two major algorithms for particle simulations, and are comparable, because all use the C18 traversal. Spot-wise, we also tested different algorithms, for example the Pairwise Verlet Lists or a Linked Cells algorithm with Newton 3 disabled.

For the scenarios, we varied the following parameters. We varied between very low, medium, and very high density, between small, medium, and very large domains, and between uniform and Gaussian particle distribution. This resulted in a manageable number of 18 different configurations. We set the size of the domain together with the particle density to values such that the total number of particles varies from 10^5 to 10^8, while the average number of particles per cell varied from 0.1 to 25. In the Gaussian particle distribution, the particles were clustered heavily such that the majority of cells contained a single or even no particle. The center of the particle cluster was put at two thirds of the domain in all dimensions instead of in the center.

3.3 Observations

The work on profiling AutoPas made clear that modeling the performance of a particle simulation algorithm is difficult. Both the Linked Cells and the Verlet Lists algorithm can be memory or CPU bound, depending on the simulation scenario.

In high-density scenarios with uniform distribution where the average number of particles is higher than 10, the arithmetic intensity is high, because the number of particle interactions grows quadratically with the number of particles in a cell. This leads to both algorithms being CPU bound. Here, the Linked Cells algorithm with the SoA data layout performs best, as many calculations can be performed using SIMD instructions. With good vectorization, there are densities where the used memory bandwidth is considerably higher than the main memory bandwidth, so caches are used effectively.

In low-density scenarios with uniform distribution where the average number of particles is at most one, the arithmetic intensity is low, so we are not CPU bound. The memory access pattern is very irregular. Scanning all particles of a cell in order, as Linked Cells does, is not an advantage here, because each cell contains less than one particle on average. Thus, the performance is limited by memory latency, and not bandwidth. In this case, the Verlet Lists algorithm performs better, because it can skip empty cells, while the Linked Cells algorithm iterates over all cells and their neighbors.

In medium-density scenarios with uniform distribution, there was no clear explanation which algorithm performed best in which scenario.

For the Gaussian distribution, we see the two effects described above play against each other. The majority of the cells is almost empty or empty, so the constant overhead of Linked Cells per cell impacts the performance negatively. On the other hand, its more regular memory access pattern compared to Verlet Lists improves performance while working on the densely filled cells in the center of the particle cluster. Which effect dominates the other is difficult to predict, beforehand.

The impact of the size of the domain under equal particle density and distribution was not important in many cases. Only for very small domains, all algorithms benefited from the fact that the whole memory working set fitted into the L3 cache, so the irregular memory accesses did not have such a strong performance impact.

In general, we saw that the number of last level cache misses correlated with the performance. In most of the cases where we were not able to clearly explain why an algorithm performs better than the others, we saw that the number of last level cache misses was smaller for the best performing algorithm.

These insights are now used to build approximate theoretical models of the algorithms to predict which is better, at least in the cases where we see a clear performance difference.

4 Theoretical Modeling of Linked Cells vs. Verlet Lists

In this section, we present theoretical models we developed using the knowledge from profiling AutoPas. These models describe runtime properties of the Linked Cells and the Verlet Lists algorithm. Using these models, we compare both approaches and optimize the meta parameters of the algorithms. For Linked Cells, we optimize the `cellSizeFactor` that determines how large the individual cells are. For Verlet Lists, we optimize the `skin` and the `rebuildFrequency`.

While each implementation of the same algorithm behaves differently, there are intrinsic similarities between all of them, and fundamental differences between Verlet Lists and Linked Cells that potentially make it possible to determine which algorithm archetype is theoretically better suited for a given scenario using a simplified model of the algorithms.

4.1 Compute Model

In both algorithm archetypes, there are two kinds of calculations that need to be done. First, potential particle neighbors have to be checked if their distance is within the cutoff radius. Then, for all particle pairs, the real particle interaction has to be evaluated. The second calculation needs to be done in both algorithms. Only during SIMD execution, if particle pairs are masked out instead of being replaced with others, the work differs. The amount of work for the first calculation depends on the number of potential neighbors for each particle. The main motivation for using Verlet Lists is to reduce this number. Instead of checking the particles of all neighbor cells each iteration, this is only done when particles moved more than skin/2 when the particle neighbor lists are potentially invalid. However, these cells need to be larger compared to Linked Cells to capture all particles that need to be added to the lists.

The number of neighbor calculations in a scenario with uniform particle distribution can be estimated using Equation 1 for avg. particles per cell $> \frac{1}{27}$. For each particle, the neighbor calculation happens for all particles in the 27 surrounding cells, but not with itself. If Newton's third rule is used, the number of neighbor calculations is halved. This rule, from now on called Newton 3, states that for two particles, the force that the first particle exerts on the second particle is equal to the force that the second particle exerts on the first particle. Thus, this force only needs to be computed once. In practical implementations, this can only be leveraged when the parallelization scheme allows to update the force value of both particles at once without introducing data races.

$$\text{LC neighbor calculations} = \text{particles} \cdot (\text{avg. particles per cell} \cdot 27 - 1) \qquad (1)$$

For Verlet Lists, the neighbor calculation happens for each particle with all neighbors in its neighbor list. The neighbor list contains all particles within its skin radius. Its size can be estimated from the number of neighbor calculations from Linked Cells by multiplying with the fraction of the volume of the skin

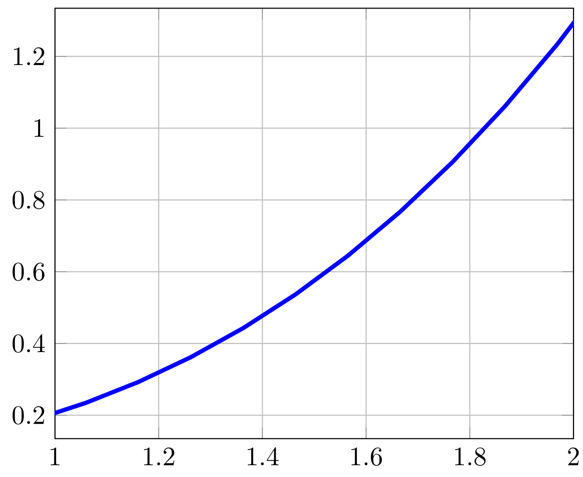

Figure 3: Factor by which Verlet Lists need less neighbor calculations than Linked Cells for a rebuild frequency of $\frac{1}{20}$.

sphere divided by the volume of the 27 surrounding cells. In addition to the neighbor calculations happening each iteration, there are additional neighbor calculations for building the neighbor lists. Each rebuild requires a full Linked Cells neighbor calculation. Equation 2 shows the formula to compute the average number of neighbor calculations per iteration including those of rebuilding the neighbor lists.

$$\begin{aligned}\text{VL neighbor calculations} = \text{LC neighbor calculations} \cdot \\ \cdot \left(\frac{\frac{4}{3} \cdot \pi \cdot \text{skin factor}^3}{3^3} + \text{rebuild frequency}\right)\end{aligned} \tag{2}$$

This means that the term $\frac{3^3}{\frac{4}{3} \cdot \pi \cdot \text{skin factor}^3}$ describes how many neighbor calculations Linked Cells needs more than Verlet Lists in each iteration, assuming we do not need to frequently rebuild. It only depends on the skin factor variable. As shown in Figure 3, with a rebuild frequency of $\frac{1}{20}$, Verlet Lists only need $\approx 31.8\%$ for a skin factor of 1.2. A skin factor larger than ≈ 1.83 increases the number of neighbor calculations over the number that Linked Cells needs and is hence considered as the upper limit of reasonable values.

4.2 Memory Access Model

Each neighbor distance and force calculation needs to load data from memory, and potentially store data back. For the force calculation, both Verlet Lists and Linked Cells need to load all necessary data of the particles, so the amount of data fetched is equal. For neighbor calculations, the amount of data fetched is linear in the number of neighbor calculations. If the Verlet Lists algorithm does less neighbor calculations by a factor x, the amount of data fetched during neighbor calculations is lowered by factor x, as well.

The most important difference is the pattern in which the algorithms access the memory. In Linked Cells, the particles are typically stored continuously within a cell, and the neighbor and force calculations iterate over these particles sequentially. In Verlet Lists, each particle stores a list of pointers to neighbors that do not lie continuously in memory. When iterating over the neighbor list, the hardware prefetcher is not able to prefetch the next particles before they are requested. Furthermore, Linked Cells needs to access all cells, even if they are empty, while Verlet Lists only needs to process all lists.

From Section 3, we know that the performance of the algorithms correlates with the number of last level cache misses in many cases. Thus, we construct a simplified model to predict this number.

In our model, we approximate the last level cache misses using the number of irregular memory accesses. For Verlet Lists, we can approximate the number of irregular memory accesses with the number of pairwise force interactions, because the particles in the neighbor lists are not stored sequentially in memory.

$$\text{VL irregular memory accesses} = \text{VL neighbor calculations} \tag{3}$$

For Linked Cells, we scan the particles in a cell in order, but we have an irregular memory access whenever we access a cell. The number of cell accesses is number of cells $\cdot$ avg. particles per cell $\cdot\, 27$, where 27 is the number of neighbor cells each cell has including itself (the three loops in the Linked Cells algorithm from Figure 1).

$$\text{LC irregular memory accesses} = \text{number of cells} \cdot \text{avg. particles per cell} \cdot 27 \tag{4}$$

Figure 4 shows how the number of irregular memory accesses develop with varying particles per cell in a domain of 50 cells in each dimension and skin factor of 1.2 and rebuild frequency of $\frac{1}{20}$. We see that at ≈ 3.18 avg. particles per cell, the number of irregular memory accesses is the same. Below, the Verlet Lists algorithm has less irregular memory accesses. Above, Linked Cells has an advantage.

4.3 Optimize `cellSizeFactor`

In the practical implementations of the Linked Cells algorithms in AutoPas, there is a constant overhead per cell in each iteration. This is not included in the compute model from Section 4.1. In very sparse scenarios, this overhead can surpass the neighbor and pairwise force calculations in each iteration by far. In this case, the performance can be improved by increasing the `cellSizeFactor` to a value where the number of cells is not larger than the number of particles, anymore. However, care must be taken for skewed particle distributions. If a

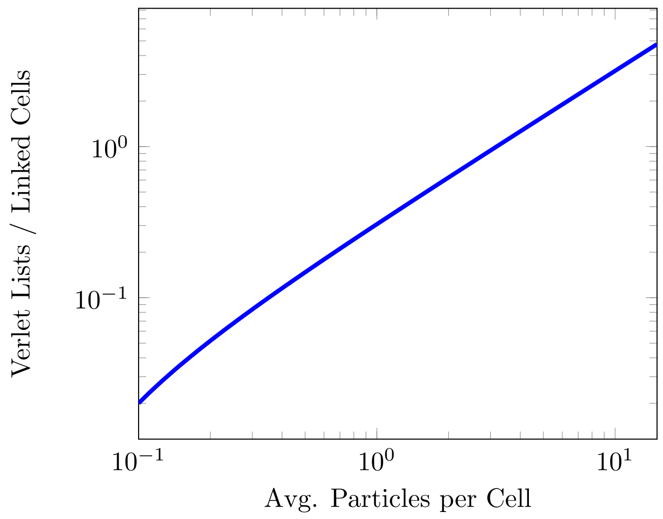

Figure 4: The factor of irregular memory accesses of Verlet Lists over Linked Cells.

large number of particles is located near to each other in s small part of the whole domain, increasing the `cellSizeFactor` can lead to quadratic runtime, because all particles might be located in a single cell, afterwards. This can be mitigated by estimating the maximum number of pair interactions that may happen after increasing the `cellSizeFactor`. Given the maximum number of particles contained in a single cell n, the maximum number after resize can be estimated by $n * \texttt{cellSizeFactor}^3$. The `cellSizeFactor` can now be increased only to a level where the number of pairwise interactions in a single cell is considerably smaller than the number of cells in the domain.

4.4 Optimize `rebuildFrequency` and `skin`

As we saw in Section 4.1, the neighbor calculation savings of Verlet Lists depend on the two variables rebuild frequency and skin factor. Decreasing the skin factor and increasing the rebuild frequency lowers the number of neighbor calculations necessary for Verlet Lists. Thus, choosing these values optimally is important. Valid are all variable combinations that still guarantee correctness of the algorithm. For this, no particle is allowed to move more than skin/2 in $\frac{1}{\text{rebuild frequency}}$ iterations. This requirement shows that the variables are not independent. Given a fixed skin factor, the optimal rebuild frequency can be computed from the number of iterations the fastest particle needs to travel cutoff distance as in (Equation 5), and the other way around.

$$\text{rebuild frequency} = \frac{1}{1 + \text{iterations for cutoff movement} \cdot \frac{\text{skin factor}-1}{2}} \tag{5}$$

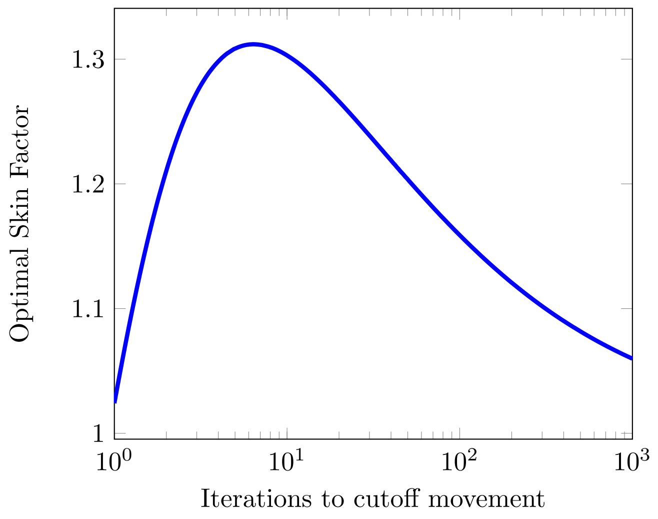

Figure 5: Optimal skin factor depending on the number of iterations the fastest particle needs to travel the cutoff radius.

Using Equation 2, we can choose both rebuild frequency and skin optimally for a given maximum particle movement per iteration, expressed as the number of iterations it takes a particle with maximum movement to travel the cutoff radius. Equation 6 shows how to compute the optimal skin factor from the maximum movement. Given the optimal skin factor, the optimal rebuild frequency can then be computed according to Equation 5.

$$\begin{aligned} &\text{optimal skin factor}(\text{m} := \text{iterations for cutoff movement}) = \\ &= \underset{\text{skin factor}>=1}{\arg\min} \frac{\frac{4}{3}\cdot\pi\cdot\text{skin factor}^3}{3^3} + \frac{1}{1+\text{m}\cdot\frac{\text{skin factor}-1}{2}} \\ &= \frac{m-2+\sqrt{4-4\cdot m+6\sqrt{\frac{6}{\pi}}m^{3/2}+m^2}}{2m} \end{aligned} \tag{6}$$

This function is plotted in Figure 5. As we can see, the optimal values range from ≈1–1.3. Larger skin factors to minimize the number of rebuilds further are not worth it in any case, according to our model. In practice, the number of neighbor calculations is a bit higher than predicted using this optimization in many cases, because the rebuild frequency can not be set continuously, but must be rounded down.

5 Rule-Based Tuning

5.1 Motivation

The overall performance of AutoPas is determined by its ability to quickly select a fast algorithm for each iteration of the pairwise force calculation. An algorithm in AutoPas consists out of five components, the container, the traversal, the Newton 3 option, the data layout, and the load estimator. The cell size factor can be seen as a sixth component. However, we don't consider it here, because we showed in Section 4.3 that good values for it can be chosen without tuning over several candidates. We call this a configuration. Right now, AutoPas periodically starts a tuning phase in which several configurations are tested using some pre-defined tuning strategy, and the best configuration is then employed until the next tuning phase starts. The length of this tuning phase and the choice of tested configurations have a high impact on the overall performance, because they determine the tuning overhead. This is the overhead over the runtime where the optimal configuration is correctly guessed in the beginning of each tuning phase and no other configurations that are slower are tested. Since the performance of different configurations in a given scenario spans multiple orders of magnitude, this overhead can become very high. All existing tuning strategies use black-box optimization algorithms that are not able to make use of existing domain specific about the configurations or the concrete running simulation the developers or users of AutoPas might have. This is a gap that Rule-Based Tuning fills. It enables developers and users of AutoPas to provide their domain specific knowledge as so called rules to help AutoPas in finding a fast configuration during tuning more quickly.

5.2 Description

Rule-Based Tuning is a tuning strategy based on Full Search. Full Search tests all possible configurations and thus always finds the optimal configuration, but also has the highest possible overhead. Rule-Based Tuning works the same way, but first filters out some configurations each tuning phase using user-provided rules. These rules can use metrics, called Live Info, collected from the current simulation state to define a partial order over configurations. Whenever a configuration is ordered after another configuration in this partial order, it is filtered out from the set of configurations to test this tuning phase, because another configuration is in the test set that provides better performance, given the rule is correct. These rules are defined in their own language, the Rule Language, described in Section 5.3. If the given rules are precise enough to filter out a significant part of the worst performing configurations, the tuning overhead compared to Full Search is drastically reduced, but the strategy still finds the optimal configurations.

Rule-Based Tuning can also be used to provide the subset of configurations as output that is determined worthwhile to test. On this subset, arbitrary other black-box tuning strategies can be executed. This combines the domain-specific knowledge of the Rule-Based Tuning with the smartness of existing black-box optimization algorithms.

5.3 Rule Language

The Rule Language is the domain specific language used to define rules. When using the Rule-Based Tuning, AutoPas expects a file containing all rules to use in this language. For reference, the full grammar of the language can be found in the AutoPas repository in `RuleLanguage.g4`.

A file in the Rule Language consists of a list of statements. The most important statement is a *configuration order*. Here is an example:

```
[container="VerletListsCells", dataLayout="AoS"] >=
        [container="VerletListsCells", dataLayout="SoA"]
        with same newton3, traversal, loadEstimator;
```

This configuration order defines the following partial order between configurations a, b:

$\forall a, b \in$ Configurations. container(a) = container(b) = VerletListsCells $\wedge$ dataLayout(a) = AoS $\wedge$ dataLayout(b) = SoA $\wedge$ newton3(a) = newton3(b) $\wedge$ traversal(a) = traversal(b) $\wedge$ loadEstimator(a) = loadEstimator(b) $\implies a \to b$

The configuration orders are not allowed to introduce circles of configurations to fulfill partial order requirements (no diagnostic required[8]). In general, a configuration order consists of two *configuration patterns* separated by `>=` and an optional `with same` clause.

Configuration orders can be applied conditionally using `if` statements:

```
if avgParticles < 6:
        [...] >= [...];
endif
```

In this case, the configuration order is only applied if the average number of particles in a cell is below 6. `avgParticles` is a Live Info, one of the metrics collected live from the simulation state in the beginning of the tuning phase. All available metrics are listed in the doxygen documentation of `LiveInfo::gather()`.

Finally, for convenience, there are two other types of statements, `define` and `define_list`. These allow to define constants and lists of constants in the rule file. They have global scope, starting from where they are defined.

```
define threshold = 0.5;
define_list linkedCellsContainer = LinkedCells, LinkedCellsReference;
if avgParticles < define_threshold:
        [container = "VerletClusterLists"] >=
                [container = linkedCellsContainer]
```

[8] https://en.cppreference.com/w/cpp/language/ndr

```
                    with same dataLayout;
endif
```

This example defines a threshold constant and a list of containers, and then orders all configurations with the `VerletClusterLists` container before all configurations where the container is in the defined list, as long as they have the same data layout, if the average particles per cell are below the given threshold.

5.4 Implementation

The rule language is parsed using ANTLR[9]. From the resulting AST, code is generated for a very simple stack-based virtual machine that is able to execute the given rule program dynamically at runtime and return the list of configuration orders that should be applied. Then, each available configuration is checked against the right-side pattern of all configuration orders. If it matches one of them, it is removed from the list of configurations to test.

5.5 Tooling

Since finding a correct set of rules that decreases the search space size significantly in many scenarios is challenging, multiple tools are provided to collect runtimes of configurations in various scenarios and quickly verify rules against these records.

To collect the necessary data to test new rules, a logger is introduced which sits between the AutoPas `AutoTuner` and the used tuning strategy as a proxy. This `TuningStrategyLoggerWrapper` logs all calls into the tuning strategy into a file that can later be replayed using a different tuning strategy. Thus, collecting data using Full Search makes it possible to quickly run and compare all other tuning strategies. To verify if rules are correct in a given scenario, the Rule-Based Tuning strategy has a verification mode where it tests all possible configurations similar to Full Search, but verifies for each incoming data point if it violates the partial order provided by the rule file. This combination makes it possible to once log data for hundreds of scenarios, and then verify a new rule against this data in seconds.

[9]https://www.antlr.org/

6 High-Variety Benchmarks

6.1 Motivation

With the Rule-Based Tuning, we have built a way to make domain-specific knowledge available to the tuning strategies of AutoPas. However, finding helpful and correct tuning rules is difficult. We derived rules from our theoretical models, but these are limited and only differentiate between the Linked Cells and Verlet Lists algorithm archetypes, not the many concrete algorithm implementations we have in AutoPas.

Thus, we needed a way to derive such rules at scale. For this, we executed all algorithms in a great variety of scenarios while collecting data about their performance. Then, we analyzed the data to find correlations of when a particular algorithm performs worse than another one, and used that information to write the corresponding rule. This not only allowed us to find correct rules and judge how large their impact on real simulations is, but this data also allows us to verify arbitrary new rules that developers or users come up with, in the future.

6.2 Scenario Generation

To get meaningful results, we need to cover a large variety of particle simulation scenarios. However, each scenario must be simulated by every algorithm multiple times to get reliable results. This takes a lot of time, so the number of scenarios we can use is limited. For the scenarios, we varied multiple parameters like domain size, number of particles, particle distribution, cutoff, and skin. We arrived at 241 scenarios. The exact parameters can be found in the generated SQLite database described in Section 6.3. While there are certainly important scenarios that we have missed, we believe we have covered a wide range for the following reason. Analyzing the data showed that almost all container and traversals perform best in one of the generated scenarios. Thus, it is unlikely that our scenarios only cover a small part of the wide range of possible scenarios. If it were, we would expect that a few algorithms dominate almost all scenarios.

6.3 Data Collection and Format

We execute all generated scenarios using the Full Search tuning strategy and log all data that relates to tuning using the `TuningStrategyLoggerWrapper` as explained in Section 5.5. While this format contains all the necessary data, it is difficult to analyze. For this reason, we developed a program that converts the tuning logs from all generated scenarios into a single SQLite3 database. This allows us to use SQL to analyze the data.

This SQLite database contains a table `Scenario` with information about the scenarios that were run, as well as a table `Measurement`. This table has a row for each algorithm in each scenario and stores the properties of the algorithm like container, traversal, data layout, and more, as well as the recorded runtime. The scenario is stored as a foreign key on the `Scenario` table. Along with these tables, the SQLite database is already created with several useful views on these tables that show the number of different algorithms that were executed, the best algorithm of each scenario, the algorithms that performed the best most often, the algorithms that never performed best, a view `configRanks` that contains for each Measurement information how much worse it performed compared to the best algorithms in all scenarios, how much time was spent tuning in each scenario, one that shows how much time was spent for an algorithm in all scenarios, a view that shows all configurations that are strictly worse than another configuration (that means in all scenarios, it performed worse than the superseding configuration), and more. To make the results well accessible, we automatically generate an HTML report for these and more insights.

6.4 Report generation

While analyzing the data using SQL is straightforward, viewing the results in the SQLite commandline is not. For this reason, we built an extensible python script that visualizes many interesting insights that are created using SQL queries in an HTML report. For example, a section of the report that uses a bar diagram to visualize the data can be created by writing `visualize_bar(dbcursor, 'select * from traversalWinners', 'Wins Per Traversal', True)` where `traversalWinners` is one of the predefined views. The third parameter indicates that the bar diagram should be horizontal.

6.5 Summary of Insights

In this section, we summarize the most interesting insights we gained from the large scale benchmarks. All these results and more are part of the generated report and can be viewed in more detail there. The queries that produce these results are part of the report generation script.

1. Across all scenarios, most algorithms perform between a factor of 2-50 worse compared to the best algorithm of the scenario. It is approximately Gaussian distributed. There are algorithms that perform 1000 times worse than the best.

2. Up to 80 percent of the tuning time is spent in the 10 worst performing algorithms in a scenario. On average, it is 40 percent.

3. All containers performed best at least one scenario, except the Pairwise Verlet Lists.

4. 31 configurations performed best in some scenario.

5. The Verlet Cluster Lists algorithm with AoS data layout, disabled Newton 3 and the `vcl_c01_balanced` traversal performed the most stable across all scenarios, performing a factor of two worse than the best algorithm

on average, but a factor of five at most. All other algorithms performed worse than a factor of five over the best algorithm in at least one scenario.

6. For 43 algorithms, there each is a single configuration that performed better in all scenarios. These configurations never need to be taken into account for tuning.

7. On average the slowest configuration is 80x slower than the fastest.

8. The majority of iterations we executed took less than five seconds, but some took up to 120 seconds.

6.6 Algorithm Subset Ranking

In addition to the analysis described above, we conducted one more complex analysis that could not be formulated as a single SQLite query to answer the following question: Which subset of n algorithms gives the best performance across all scenarios? The exact metric of what is best can for example be minimal overall runtime, or minimal average regression compared to the best algorithm in each scenario. While we have all the data to answer this question, we can only solve this naively by evaluating all algorithm combinations for very small n, like $n = 5$. For $n = 10$, there are already $\binom{186}{10} = 10^{16}$ combinations.

To answer this question for larger n, we implemented a genetic algorithm that finds a solution within up to a couple of minutes also for larger n. The genetic algorithm works as follows. Each individual has n genes. The value of a gene is a specific algorithm. As fitness function, we generate a SQL query that evaluates the performance of an individual according to the defined metric. For crossover, we randomly select parent genes until the child has n genes. Only if two parents are exactly equal, we generate a random child. For mutation, we change a random gene to a random algorithm. Our starting individuals are randomly generated. We found that a crossover rate of 60 percent, a mutation rate of 5 percent, and a population size of 50 work well. In each generation, we remove duplicate individuals. We compute up to 10'000 generations to make sure we find a good result. Finding the optimum takes between a few seconds and a few minutes, depending on the subset size.

We applied the script on a data collection of 144 algorithms in 133 different scenarios. While we cannot guarantee that the solution is optimal for larger n (we verified it up to $n = 6$ for a smaller subset of the algorithms), the performance of the subset improves consistently with increasing n until $n = 27$ (Figure 6). This is expected, as 27 algorithms are enough to perform optimally in all scenarios. For $n = 10$, we found an algorithm subset that performs at most 20 percent worse than the best algorithm in all scenarios. This subset is shown in Table 1. Rerunning the script returns approximately the same results.

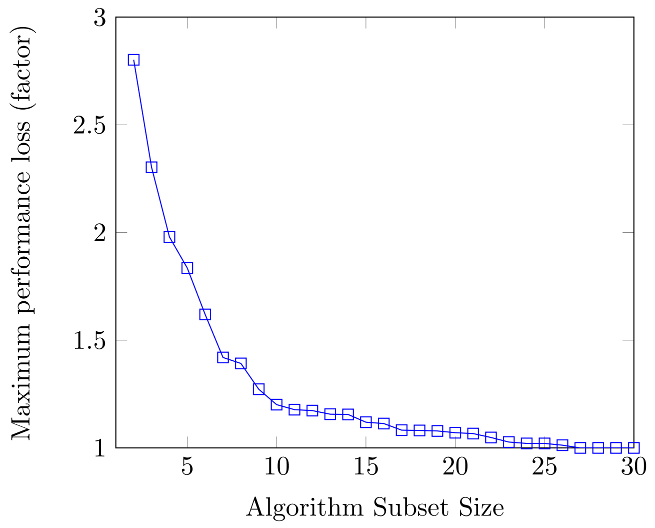

Figure 6: How increasing the size of the optimal subset steadily improves performance. The y-axis shows the worst performance of the subset in any scenario as factor over optimum performance for this scenario.

Container	Traversal	Data Layout	Newton3
LinkedCells	lc_c04_HCP	SoA	enabled
LinkedCellsReferences	lc_c08	SoA	enabled
LinkedCellsReferences	lc_c01	SoA	disabled
VerletClusterLists	vcl_c06	AoS	enabled
VerletClusterLists	vcl_cluster_iteration	AoS	disabled
LinkedCellsReferences	lc_c04_HCP	AoS	enabled
VerletListsCells	vlc_c18	AoS	enabled
VarVerletListsAsBuild	vvl_as_built	AoS	enabled
VerletClusterLists	vcl_c01_balanced	AoS	disabled
VerletClusterLists	vcl_c01_balanced	SoA	disabled

Table 1: The best algorithm subset of size 10 we found using our algorithm subset ranking optimization.

7 Other Work: Build-time improvements

While working on exploring performance modeling in AutoPas, we stumbled over several things that we thought could be improved. Of these, our build time improvements deserve special attention. Before our improvements, any change in the `md-flexible` compilation unit that calls AutoPas resulted in a full recompilation of all AutoPas code. Now, this is no longer the case.

Users of the AutoPas library provide their custom particle and functor class types to the library. Since these types are fundamental, almost all code in AutoPas is templated on at least one of them. This means that almost all code of AutoPas is templated and written in header files. If the user of the library now changes any code in the compilation unit that includes AutoPas, the whole AutoPas library needs to be recompiled. In the case of `md-flexible`, this was especially problematic as it uses four different functor types. Recompilation times on a laptop for a single compilation unit were up to 20 minutes for any change on an important source file in the `md-flexible` executable. However, recompilation was actually only necessary when the particle or functor class is changed. This is what we implemented.

To only recompile AutoPas when necessary, we split the declaration and the definition of the AutoPas library interface into two header files, similar to how the problem is solved for non-templated libraries. Then, we only included the AutoPas declaration in the compilation units of `md-flexible` that call into AutoPas. To make this compile with the templated interface, we externally declared the AutoPas library interface with the concrete template argument types in the files where they are used. For each externally declared unit, we then created a new compilation unit that includes the AutoPas definitions and instantiates the templated with the concrete template argument types. We did this both for the particle type that is passed to the `AutoPas<ParticleType>` class template and for the functor type that is passed to the template function `AutoPas<ParticleType>::iteratePairwise(FunctorType*)`.

By doing this, the changes in the `md-flexible` compilation unit no longer force the compiler to recompile all code in AutoPas. Only changes to the particle and the functor type do. Furthermore, the `iteratePairwise()` function is now compiled in a separate compilation unit for each functor type used in `md-flexible`, so they can be compiled in parallel instead of sequentially in a single compilation unit.

8 Future Work

In this section, we shortly describe some of the ideas we came up with to improve AutoPas while working on this project.

8.1 Spatial Hashing for Linked Cells

While analyzing the performance of Linked Cells, we observed that the overhead of storing and accessing empty cells dominates the memory usage and execution time in very sparse scenarios with large domains. Right now, the cells are stored in a `std::vector`. This could be changed to a hash table that has the position of this cell as key, and the cell itself as value. In this hash table, empty cells would not need to be stored at all. Since the cells are arranged on a 3D grid, using a spatial hash function sounds promising. If a scenario is really sparse, using a bloom filter that stores which coordinates are contained in the hash table could speed up checking if a cell is empty even more.

8.2 Early timeout for very long-running tuning iterations

As we have seen in this thesis, while trying different algorithms during a tuning phase, algorithms that take orders of magnitude longer than others dominate the tuning time and have considerable impact on the end-to-end simulation time. Since there is such a large difference in runtime between algorithms, aborting an iteration using a new algorithm if it already took a factor x longer then another algorithm that was already tested could help against this. If such a feature is implemented, one of the algorithms that showed the most stable performance across all tested scenarios in Section 6 should be tested first to get a reasonable baseline.

Cancellation could be implemented by passing an atomic bool or `std::stop_token` to the `traverseParticlePairs()` function of each traversal that is regularly checked. For Linked Cells, it could be checked after each interaction of a cell with all its neighbors. This has low overhead as reading it as long as it is unchanged is a simple load from memory that is likely cached.

8.3 Improve the Use of SIMD Instructions in the LJFunctor for Verlet Lists

While profiling the SoA Verlet Lists implementations, we noticed that the LJ Functor does not make good use of SIMD instructions in its neighbor list interaction functions. Being able to use SIMD instructions is one of the main benefits that Linked Cells SoA has over Linked Cells AoS, as our profiling showed.

8.4 Pack particles within cutoff in SIMD implementation of the LJFunctor

In the hand-crafted SIMD implementation of for the interaction of two cells in the LJFunctor, the force between two particles of the cells is always calculated, and then masked away if the particles are not within the cutoff. Since only about a sixth of the particles are actually in the cutoff range, this is a lot of wasted work. It could be more efficient to pack a SIMD register with particles that are all within the cutoff and then only calculate the force for those. A StackOverflow post describing a possible technique is the following: `https://stackoverflow.com/questions/36932240/avx2-what-is-the-most-efficient-way-to-pack-left-based-on-a-mask`

8.5 Collect data from ARM cluster and compare

In Section 6, we only collected data from a cluster with Intel CPUs. It would be interesting to compare how the algorithms perform differently on ARM CPUs, so it would be valuable to use the existing infrastructure to collect the same performance data on an ARM cluster and use SQL to analyze the difference.

9 Conclusion

In this thesis, we described our work on exploring performance modeling in AutoPas. In Section 3, we started with profiling a few selected algorithms in various scenarios to get a feeling for which parameters influence the performance of the algorithms. Then, we used this knowledge in Section 4 to construct theoretical models of various runtime properties of the Linked Cells and the Verlet Lists algorithm and provided closed formulas to optimize their meta parameters. In Section 5, we then built a new tuning strategy called Rule-Based Tuning that uses a declarative domain-specific language to apply the knowledge we gained so far on tuning. To get more insights in how AutoPas performs and improve the Rule-Based Tuning further, we conducted high-variety benchmarks in Section 6. Using the collected data, we gave valuable insights into how the algorithms in AutoPas behave and found a subset of 10 algorithms that performs optimal in many scenarios, and only up to 27 percent worse than the optimal in few scenarios using a genetic algorithm. Finally, we presented our work on build time improvements in Section 7 and presented some ideas for future work on AutoPas in Section 8.