Algorithm Selection and Auto-Tuning in AutoPas

Manuel Lerchner
Technical University of Munich
Munich, Germany

***Abstract*—Simulating molecular dynamics (MD) presents a significant computational challenge due to the vast number of particles involved in modern experiments. Naturally, researchers have put much effort into developing algorithms and frameworks that can efficiently simulate these systems. This paper focuses on the AutoPas framework, a modern particle simulation library that uses dynamic optimization techniques to achieve high performance in complex simulation scenarios. We compare AutoPas with other prominent MD engines, such as GROMACS, LAMMPS and ls1 mardyn, and investigate a possible improvement to AutoPas' auto-tuning capabilities by introducing an early stopping mechanism aiming to reduce the overhead of parameter space exploration. Our evaluation shows that such a mechanism can reduce the total simulation time by up to 18.9% in certain scenarios, demonstrating the potential of this improvement.**

***Index Terms*—molecular dynamics, auto-tuning, autopas, early-stopping, gromacs, lammps, ls1 mardyn**

I. Introduction

Molecular dynamics simulations represent a computational cornerstone in various scientific fields. These simulations typically use complex and computationally intensive interaction models acting on enormous numbers of particles to ensure accurate results. Consequently, the computational requirements for these simulations can be substantial and require highly optimized algorithms and frameworks to achieve feasible performance.

Well established molecular dynamics engines, such as GROMACS, LAMMPS, and ls1 mardyn solve this challenge by providing a single, highly optimized implementation determined prior to the start of the simulation. As their implementation is detirmend statically, these engines must rely on static optimizations to improve their performance. Static optimizations are typically selected based on predefined performance models and must be fine-tuned for specific hardware architectures. Common static optimizations techniques include automatic optimizations performed by modern compilers (e.g., loop unrolling, inlining, auto-vectorization), conditional compilation based on the target hardware (e.g. SIMD), or manual selection of simulation parameters based on expert knowledge [1].

AutoPas, approaches the challenge of high-performance molecular dynamics simulations differently: Instead of choosing a single implementation prior to the simulation, AutoPas uses on dynamic optimizations to adjust its implementation based on the simulation state and the actual hardware performance. This approach is favourable, as the engine can adapt and optimize itself without external intervention. However, dynamic optimizations come at the cost of increased complexity and potential overhead due to the need for frequent re-evaluations of the simulation state.

This paper provides an overview of the AutoPas framework and the benefits and challenges of using dynamic auto-tuning in molecular dynamics simulations. We compare AutoPas with other prominent MD engines, such as GROMACS, LAMMPS, and ls1 mardyn, and investigate a possible improvement to AutoPas' auto-tuning capabilities by introducing an early stopping mechanism aiming to reduce the overhead of parameter space exploration.

II. AutoPas

AutoPas was developed on the basis of creating an efficient particle / N-Body simulation engine applicable to a wide range of scientific fields [2]. AutoPas employs a flexible architecture ensuring that algorithms and data structures can be interchanged seamlessly within the simulation engine. As it is not capable of running simulations on its own, AutoPas serves as an intermediary layer between user-provided simulation code and implementations specifically designed to efficiently solve N-Body problems. The modular nature of AutoPas enables it to combine different implementations and create a wide range of so-called *configurations*.

To eliminate the need for manual configuration selection and enable dynamic optimization, AutoPas provides an auto-tuning framework. This framework periodically assesses different configurations and selects optimal ones based on performance metrics. This selection process is managed by *TuningStrategies* that systematically suggest promising configurations to evaluate. Figure 1 illustrates the high-level architecture of the AutoPas library.

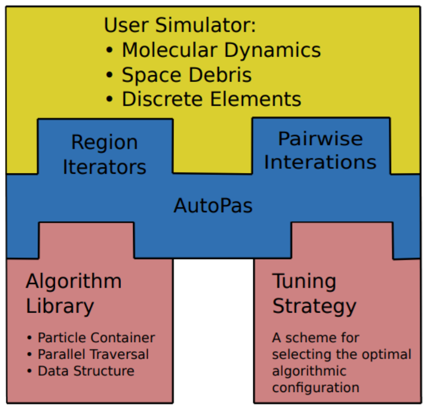

Fig. 1: AutoPas Library Structure as depicted by [3]

A. Algorithm Library

All different algorithmic implementations for solving N-Body problems are part of the so-called *Algorithm Library* of AutoPas. The Algorithm Library contains different implementations for certain key aspects of the simulation, such as neighbor identification, traversal patterns, memory layouts and optimization techniques. Currently AutoPas supports the six tunable parameters: *Container*, *Data Layout*, *Newton 3*, *Traversal*, *Load Estimator* and *Cell Size Factor* which combine to form a configuration.

Maintaining a library of different implementations has several benefits: As the library is modular, it is straightforward to add new, potentially hardware-specific, implementations for each key aspect of the simulation. This ensures that the simulation engine remains maintanable and can provide performance portability across a wide range of hardware platforms [2]. The interchangable nature of the implementations also ensures an implicit backward compatibility, making it easy to study effects of new hardware on existing implementations [2]. Users of the library also benefit from this approach. As AutoPas is able to automatically select the best configuration for their specific use case and hardware setup, users do not need to have a deep understanding of the underlying hardware or software optimizations. By treating the simulation engine as a black box, users can focus on the high-level aspects of their simulation.

B. Tunable Parameters

This section provides a brief overview of the six tunable parameters available in AutoPas.

Container

Containers are responsible for storing the particles of the simulation such that relevant neighbor particles can be determined efficiently. As AutoPas focuses on short-range interactions with a force cutoff radius r_c, efficient neighbor identifications using just $O(N)$ distance calculations are possible [1]. The container types depicted in Figure 2 will be shortly described below.

- ***Linked Cells***
  The Linked Cells algorithm maintains a grid of cells, each storing a list of particles located within the cell. When calculating forces for a particle, only particles in neighboring cells (depicted in blue) need to be considered, as all other particles are guaranteed to be outside the cutoff radius.
  LinkedCells introduces many spurious distance calculations (shown by many arrows to gray particles in the figure) which can be reduced by using a smaller cell size factor. LinkedCells are very cache-friendly as spatially close particles are stored together in memory [4].
- ***Verlet Lists***
  The Verlet Lists algorithm uses a second radius $r_v = r_c + \Delta_s$ (yellow circle) and considers all particles within this radius as potential neighbors. Contrary to LinkedCells, each particle maintains its own list of potential neighbors resulting in a way higher memory overhead. As the bigger radius r_v provides a buffer region, it is possible to only rebuild the neighbor-list every n simulation steps, as long as no particle can move from outside $r_c + \Delta_s$ to inside r_c unnoticed [5]. VerletLists have very few spurious distance calculations but result are less cache-friendly as neighboring particles are not stored together in memory [4]. This results in very inefficient vectorization [6].
- ***Verlet Cluster Lists***
  The Verlet Cluster Lists algorithm improves on the VerletLists algorithm by grouping particles into clusters of size M ($M = 4$ in the figure) and perforing neighbor-list calculations on a cluster level. This reduces the memory overhead significantly, however results in more spurious distance calculations as all particles in neighboring clusters need to be considered when calculating forces. When M is chosen in accordance with the SIMD width of the system, efficient vectorization is possible [4].

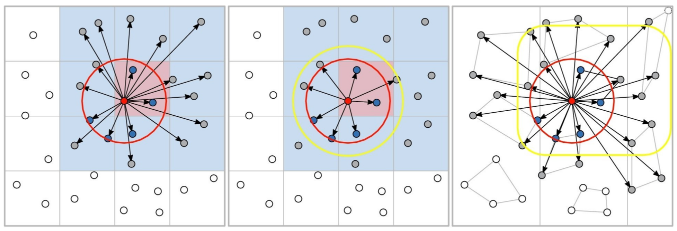

Fig. 2: Important container types as depited by [4]. The cutoff radius r_c is shown using a red circle. The arrows represent distance checks between particles. Only particles shown in blue contribute to the final force calculation.

Data Layout

The Data Layout describes how the particle data is stored in memory. Possible choices are *SoA* (Structure of Arrays) and *AoS* (Array of Structures). *SoA* allows for better vectorization as properties of multiple particles can be loaded efficiently into SIMD registers. Accesing properties of a single particle is however more expensive as it requires multiple memory accesses. *AoS* is the opposite, allowing for efficient access to properties of a single particle to e.g. send it to another MPI rank, but results in inefficient vectorization [4].

Newton 3

A common optimization in molecular dynamics simulations is the application of Newton's third law, which allows reusing force calculations between two particles, reducing the number of force calculations by a factor of two. As Newton 3 can cause race conditions in parallel environments, not all traversal patterns support this optimization. The optimization can be enabled or disabled in accordance with the traversal pattern and the interaction model.

Traversal

Traversals are responsible for iterating over the particles in the simulation and calculating their interactions in a shared-memory environment [7]. The traversal pattern determines to which extent force calculations can be parallelized and whether optimizations, such as Newton 3, can be applied. The traversal patterns depicted in Figure 3 will again be shortly introduced below.

- *C01*
  The C01 traversal pattern processes each cell independently, resulting in an embarrassingly parallel traversal pattern. Newton 3 can not be used in this traversal pattern, as neighboring cells can be processed in parallel, which could result in race conditions when applying forces. As no synchronization between cells is required, the C01 traversal pattern has the highest degree of parallelism.
- *C18*
  The C18 traversal pattern uses color assignments to ensure that no race conditions occur when using Newton 3. Figure 3 shows the regular color assignments, ensuring that no two cells of the same color share common neighbors. To ensure that forces are only applied once when using Newton 3, each cell only applies forces to cells *above* or *right* of it.
  During the force calculation, all available threads work on a single color, and can therefore safely apply forces on neighboring cells.
  As the color groups must be processed sequentially, the C18 traversal introduces 18 synchronization points, which reduces the overall degree of parallelism [5].
- *C08*
  The C08 traversal pattern is similar to the C18 traversal pattern but uses a different coloring scheme with only eight colors, reducing the number of synchronization points to eight. The C08 traversal pattern has a degree of parallelism between the C01 and C18 traversal patterns [5].

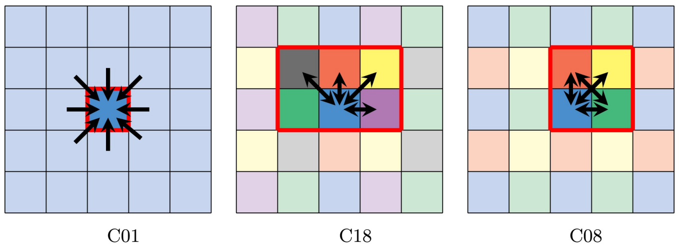

Fig. 3: Important Traversal Types as depicted by [5].

Load Estimator

To ensure an even distribution of work across all available processing units in a distributed-memory environment, different load estimators can be used. A good workload distribution reduces the idle time of processing units which reduces the total simulation time.

Cell Size Factor

The default cell size factor of 1 results in a cell size of r_c (see Figure 2). As discussed previously, the large area of neighboring cells causes many spurious distance calculations, which can be reduced by using a smaller cell size factor. As decreasing the cell size factor also increases the memory overhead due to the higher number of cells to be maintained in memory [8], the cell size factor should be chosen carefully.

C. Auto-Tuning Framework

Manual selection of suitable implementations for each tunable parameter would require extensive domain knowledge that is challenging to acquire and maintain under the constantly changing software and hardware landscape. To address this issue, AutoPas performs automated algorithm selection to optimize specific performance metrics, such as simulation speed or energy efficiency [4]. Internally AutoPas periodically initiates so-called *tuning-phases* in which promising configurations are evaluated, in order to determine the best Configuration for the current simulation state. The winning Configuration is then used until the next tuning phase is initiated.

The key to efficient tuning phases is the ability to efficiently determine promising configurations. The naive approach of evaluating all possible configurations is infeasible in practice, as many configurations turn out to be orders of magnitude slower than the optimal configuration [9] [10].

AutoPas attempts to mitigate this problem by using Tuning Strategies to only evaluate promising configurations. Tuning strategies are tasked with reducing the search space of possible configurations using certain rules or heuristics.

The currently available tuning strategies in AutoPas are:

FullSearch

The FullSearch strategy naively evaluates all possible configurations, thus always finding the best Configuration. As many of the possible configurations tend to be suboptimal [10], the FullSearch strategy often causes a considerable overhead.

RandomSearch

The RandomSearch strategy randomly selects configurations out of the full search space, causing less overhead than the FullSearch strategy at the cost of potentially missing the best Configuration.

BayesianSearch

The Bayesian Search strategy is similar to the RandomSearch strategy, however, it uses a Bayesian optimization algorithm to select the next configuration to evaluate based on previous measurements [11]. An improvement to better account for the discrete tuning space of AutoPas called *BayesianClusterSearch* is also available [12].

PredictiveTuning

The PredictiveTuning strategy extrapolates previously gathered measurements of a configuration to predict the performance in the current simulation state. This allows for an efficient selection of promising configurations without the need to evaluate them.

RuleBasedTuning
The RuleBasedTuning strategy uses a set of rules to discard undesirable configurations immediately. The rules are based on expert knowledge in a *if-then* fashion and use aggregate statistics of the simulation state(called *LiveInformation*) to make decisions [9].

FuzzyTuning
The FuzzyTuning strategy is similar to the RuleBased-Tuning strategy, but uses fuzzy logic systems to evaluate configurations. This allows for both an implicit inter- and extrapolation of the rules to account for configurations that are not covered by the expert knowledge [10].

III. Benefits of Auto-Tuning

Since no single configuration can deliver optimal performance across all simulation scenarios [2], performance tuning is essential for maintaining high efficiency across diverse simulation conditions. The auto-tuning approach implemented in AutoPas offers some key advantages:

Performance Improvements

The most compelling advantage of auto-tuning is the significant performance improvements it can achieve. It has been shown many times that AutoPas can provide significant performance improvements across diverse molecular dynamics simulation scenarios, both in the standalone application aswell as in established MD engines such as ls1 mardyn and LAMMPS [7] [4]. Those improvements provide compelling evidence for the effectiveness and importance of the auto-tuning approach.

Accessibility and Ease of Use

AutoPas's tuning framework enables scientists to achieve optimal performance directly out of the box, without requiring deep expertise in performance optimization. The inherent user-friendliness is particularly valuable when integrating AutoPas into other simulation frameworks, as developers can simply treat the simulation engine as a black box and let the auto-tuning framework handle the optimization.

IV. Drawbacks of AutoTuning

Suboptimal Configurations

A major drawback of auto-tuning in the way it is implemented in AutoPas is the inherent overhead caused by tuning phases. As the tuning process requires evaluating many configurations, the overhead of evaluating suboptimal configurations quickly adds up. This is especially problematic bad configurations, performing orders of magnitude worse than the optimal configuration, quickly lead to noticeable increases in the total simulation time.

Even though the tuning strategies employed by AutoPas are highly efficient, they still tend to suggest suboptimal configurations from time to time. More advanced tuning strategies such such as *RuleBasedTuning* or *FuzzyTuning* can mitigate this problem to some extent, however even they can not guarantee to always finding the best configuration, as the underlying expert knowledge for both strategies is expected to be highly incomplete. Figure 4 shows typical timing profile of a simulation for every available tuning strategy. The overhead caused by tuning phases is very noticeable, and is present in all tuning strategies.

As no tuning strategy is capable of just selecting optimal configurations, the tuning process will always introduce some overhead.

Periodic Re-Tuning

Even though AutoPas is capable of performing periodic auto-tuning, it is often beneficial to just execute a single tuning phase right at the beginning of the simulation.

Ideally the beforementioned overhead of evaluating suboptimal configurations leads to discovering a better configuration at the end of the tuning phase. However, many scenarios, especially homogeneous ones with simple interaction models, tend to behave fairly stable over time making it very likely that re-tuning does not lead to a improved configuration. In such cases, the overhead of re-tuning is unnecessary and increases the total simulation time unnecessarily. Figure 4 shows this effect for the ExplodingLiquid scenario provided by the `md-flexible` framework.

Further evaluation of the data obtained in [13] shows that only three out of 184 run using slight variations of example scenarios of `md-flexible` show any changes in the best Configuration after the initial tuning phase. This indicates that all currently provided example scenarios of `md-flexible` are incapable of demonstrating the benefits of periodic re-tuning and that performing additional tuning phases on these scenarios only increases the total simulation time.

To fully demonstrate the benefits of periodic re-tuning, more complex scenarios, most likely involving multiple MPI ranks and inhomogeneous particle distributions, are necessary. Simulating inhomogeneous scenarios in an MPI environment increases the chance of variying particle distribution across both MPI ranks and time steps, increasing the benefits of performing periodical re-tuning on each rank (See [3]).

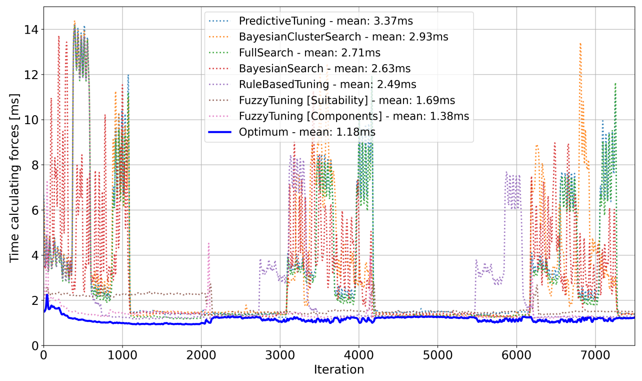

Fig. 4: Typical timing profile of the ExplodingLiquid simulation for every available tuning strategy. The plot clearly shows that all tuning strategies introduc overhead introduced during tuning phases. (Data obtained from [13])

V. Early Stopping Optimization

To minimize some of the introduced drawbacks of the auto-tuning process, [9] [10] [12] suggest that an *early stopping* mechanism could be beneficial for the AutoPas framework. The primary goal of such a mechanism would be to detect tuning iterations that take much longer than the currently best-known configuration and stop the evaluation of those configurations early. There are two approaches to this problem:

- **Stopping Further Samples**
  As AutoPas evaluates a configuration multiple times to reduce measurement noise, a simple way to implement early stopping would be to stop the evaluation of further samples as soon as it is clear that the performance is significantly worse than the best-known configuration.
  This approach may not be as effective as it still requires fully evaluating the first sample of a bad configuration.
- **Interrupting the Evaluation**
  A more fine-grained approach, proposed in [9] could interrupt the evaluation of a long running configuration while it is still being evaluated.
  The Implementation of this approach would require a big rewrite of AutoPas' internal structure, and is therefore not feasible in the short term.

To get a first impression of the potential benefits of an early stopping mechanism, we implemented the first approach into the AutoPas framework. The changes to the existing codebase are minimal, as the early-stopping mechanism can be implemented using existing functionality. Algorithm 1 shows the main changes to the `AutoTuner.cpp` file.

Both described approaches require a user-defined threshold for the $allowedSlowdownFactor$ that determines when the evaluation of a configuration should be stopped. As finding optimal thresholds is non-trivial and may depend on the simulation scenario and the tuning strategy, suitable thresholds will be determined empirically in subsection V-A.

Algorithm 1 Early Stopping Algorithm in AutoPas

```
1: procedure EVALUATECONFIGURATION(performance)
2:   $fastestTime \leftarrow \min(fastestTime, performance)$
3:   $slowdownFactor \leftarrow \frac{performance}{fastestTime}$
4:   if $slowdownFactor > allowedSlowdownFactor$ then
5:     $abort \leftarrow true$
6:   end if
7: end procedure
8: procedure GETNEXTCONFIGURATION
9:   if not $inTuningPhase$ then
10:    return $(currentConfig, false)$
11:  else if $numSamples < maxSamples$ and not $abort$ then
12:    return $(currentConfig, true)$
13:  else
14:    $stillTuning \leftarrow$ TUNECONFIGURATION( )
15:    return $(newConfig, stillTuning)$
16:  end if
17: end procedure
```

A. Evaluation: Exploding Liquid Simulation

To evaluate the performance of the early stopping mechanism we perform a benchmark using the *Exploding Liquid* scenario provided by the `md-flexible` framework. The simulation consists of 1764 initially close-packed particles which rapidly expand outwards and eventually hit the simulation boundaries. All benchmarks are performed on a single node of the CoolMUC2 supercomputer using 14 threads. To ensure reproducibility, all runs are repeated three times.

FullSearch (Figure 5)
When using Early Stopping optimization together with the FullSearch strategy the total simulation time can be reduced from 36.22 seconds to 30.84 seconds, at $allowedSlowdownFactor \approx 4$. This results in a reduction of the total simulation time by 14.8%.

PredictiveTuning (Figure 6)
When using Early Stopping optimization together with the PredictiveTuning strategy the total simulation time can be reduced from 28.62 seconds to 23.23 seconds, at a $allowedSlowdownFactor \approx 5$. This results in a reduction of the total simulation time by 18.9%.

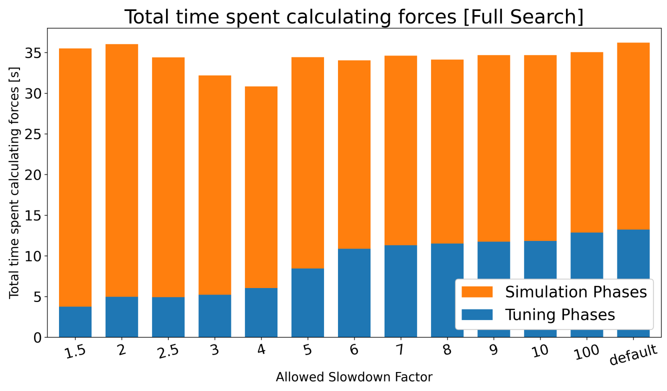

Fig. 5: Total Simulation Time for Exploding Liquid Simulation for different values of $allowedSlowdownFactor$ using the FullSearch strategy with Early Stopping.

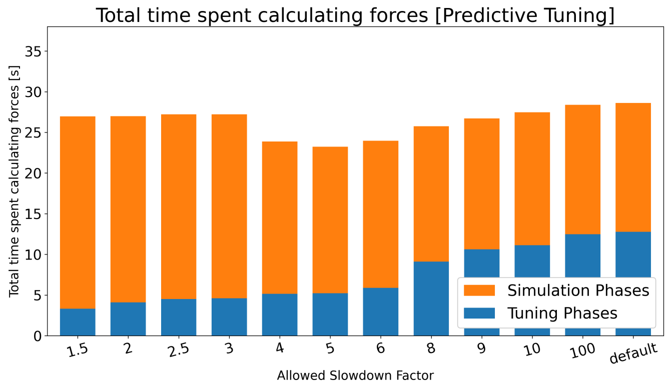

Fig. 6: Total Simulation Time for Exploding Liquid Simulation for different values of $allowedSlowdownFactor$ using the PredictiveTuning strategy with Early Stopping.

B. Analysis and Discussion

Optimal Thresholds: The evaluated benchmarks show that using the early stopping mechanism can reduce the total simulation time for both the FullSearch and PredictiveTuning strategies. However, there only exists a narrow range of $allowedSlowdownFactor$ values where the early stopping mechanism is beneficial. Outside of this range, the total simulation time is comparable to performance of the simulation without the early stopping mechanism.

This is expected, as the two limiting cases both result in undesirable behavior: $allowedSlowdownFactor \to 1$ results in tuning phases with very few samples per configuration, as even small noise in the performance measurements causes the early stopping mechanism to abort the evaluation of a configuration, prohibiting reasonable estimates of the actual performance of a configuration. On the other hand, $allowedSlowdownFactor \to \infty$ results in the early stopping mechanism never aborting a configuration, which is equivalent to not using the early stopping mechanism at all. Picking a suitable threshold corresponds to finding a balance between the two extremes.

From the executed benchmarks, we deduce that the optimal threshold for the early stopping mechanism is around 4-5 for the *Exploding Liquid* scenario. However, the optimal threshold probably varies between different simulation scenarios and tuning strategies, and further benchmarks are required to determine the optimal threshold for other scenarios. It is however noteworthy early stopping mechanism never caused a significant increase in the total simulation time, even when using suboptimal thresholds. This indicates that the performance measurements were precise enough to correctly identify the suitability of a configuration.

Combination with Tuning Strategies: The evaluated benchmarks showed that a combination of the early stopping mechanism with good tuning strategies is beneficial and additionaly reduces the total simulation time. It is expected that good tuning strategies benefit more from the early stopping mechanism, as good configurations are evaluated earlier resulting in a faster convergence of the $fastestTime$ variable. This allows for the early stopping mechanism to abort more unsuitable configurations, further reducing the total simulation time.

Limitations and Future Work: The current implementation of the early stopping mechanism resets the $fastestTime$ variable prior to each tuning phase resulting in a complete loss of the information gathered in previous iterations. This ensures that the $fastestTime$ variable is always up-to-date with the current simulation state.

A simple improvement to the early stopping mechanism would be to not fully reset the $fastestTime$ variable, but instead only reset it to a running average of timing measurements throughout the simulation-phase. This would allow for the early stopping mechanism to start of with a reasonable estimate of achievable performance, increasing the likelihood of aborting unsuitable configurations early.

VI. Comparison with Other MD Engines

Established MD engines such as GROMACS, LAMMPS, and ls1 mardyn have been developed over many years and have been optimized to achieve high performance in their respective use cases. This section provides a short overview of those engines and highlights the differences in their implementations.

A. GROMACS

Contrary to AutoPas, GROMACS only implements a single, highly optimized Verlet Cluster List scheme variant with flexible cluster sizes specifically designed for good SIMD vectorization.

Gromacs allows setting the vectorization parameters for the cluster size M and the number of particles in neighbor groups N statically to tune the force calculations to the SIMD width of the system [6]. With suitable values for M and N, computations of $M \times N$ particle interactions can be performed with just two SIMD load instructions [14], drastically reducing the number of memory operations required for the force calculations and reaching up to 50% of the peak flop rate on all supported hardware platforms [14].

Gromacs defaults to $M = 4$ and selects $N \in \{2, 4, 8\}$ depending on the SIMD width of the system. However, Finding good values is very time-consuming and depends on a detailed understanding of many low-level software optimization aspects of the different hardware platforms [6]. In GROMACS, the developers have to manually tune this tuning.

B. LAMMPS

LAMMPS implements a single, highly optimized variant of the Verlet List scheme. The neighbor list is stored globally inside a multiple-page data structure. Inside each page, vectors of neighboring particles J for multiple particles I are stored inside a contiguous memory block [15], efficiently loading the neighbor list into the cache. All particles are stored in a *SoA* (Structure of Arrays) data layout [15].

C. ls1 mardyn

ls1 mardyn differs from the previously mentioned MD engines as it uses the Linked Cells algorithm for particle interactions. Using LinkedCells provides a better memory efficiency than GROMACS and LAMMPS, allowing for simulations of massive particle systems [16]. Internally, ls1 mardyn uses the default data layout of *AoS* (Array of Structures) for the particle data. However, particular branches aimed at simulating massive particle systems can use a RMM (Reduced Memory Mode) layout in combination with a *SoA* (Structure of Arrays) data layout, allowing for simulations of up to twenty trillion atoms [16].

To overcome the limitations of a single implementation, AutoPas was successfully integrated into ls1 mardyn, providing significant speedups in specific scenarios [7].

VII. Conclusion

We have presented an overview of the AutoPas framework and its auto-tuning capabilities, demonstrating the benefits and challenges of dynamic auto-tuning in molecular dynamics simulations. Moreover, we investigated the potential of a naive early stopping mechanism to reduce some of the inherent overhead caused by tuning phases. Early measurements show that the early stopping mechanism can provide a reduction of the total simulation time of up to 18.9% when using the PredictiveTuning strategy without causing additional overhead.

The comparison with established MD engines such as GROMACS, LAMMPS, and ls1 mardyn reveals a fundamental trade-off in software design: while these engines achieve excellent performance through highly specialized implementations, AutoPas offers greater flexibility and adaptability through its modular architecture and dynamic optimization capabilities. The success of AutoPas's integration into ls1 mardyn and LAMMPS demonstrates that these approaches can be complementary rather than mutually exclusive.

References

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

[2] N. P. Tchipev, "Algorithmic and implementational optimizations of molecular dynamics simulations for process engineering," Ph.D. dissertation, Technische Universität München, 2020. [Online]. Available: https://mediatum.ub.tum.de/1524715

[3] S. J. Newcome, F. A. Gratl, P. Neumann, and H.-J. Bungartz, "Towards the smarter tuning of molecular dynamics simulations," in *SIAM Conference on Computational Science and Engineering (CSE23)*. SIAM, Feb 2023.

[4] F. A. Gratl, S. Seckler, H.-J. Bungartz, and P. Neumann, "N ways to simulate short-range particle systems: Automated algorithm selection with the node-level library autopas," *Computer Physics Communications*, vol. 273, p. 108262, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S001046552100374X

[5] S. J. Newcome, F. A. Gratl, P. Neumann, and H.-J. Bungartz, "Towards auto-tuning multi-site molecular dynamics simulations with autopas," *Journal of Computational and Applied Mathematics*, vol. 433, p. 115278, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0377042723002224

[6] S. Páll and B. Hess, "A flexible algorithm for calculating pair interactions on simd architectures," *Computer Physics Communications*, vol. 184, no. 12, pp. 2641–2650, 2013. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0010465513001975

[7] S. Seckler, F. Gratl, M. Heinen, J. Vrabec, H.-J. Bungartz, and P. Neumann, "Autopas in ls1 mardyn: Massively parallel particle simulations with node-level auto-tuning," *Journal of Computational Science*, vol. 50, p. 101296, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877750320305901

[8] M. Papula, "Implementing the linked cell algorithm in autopas using references," Master's thesis, Technical University of Munich, Sep 2020.

[9] T. Humig, "Project report: Exploring performance modeling in autopas," Project Report, Technical University of Munich, Oct 2023.

[10] M. Lerchner, "Exploring fuzzy tuning technique for molecular dynamics simulations in autopas," Bachelor's Thesis, Technical University of Munich, Aug 2024.

[11] J. Nguyen, "Mixed discrete-continuous bayesian optimization for auto-tuning," Master's thesis, Technical University of Munich, Oct 2020.

[12] HobbyProgrammer, "Skip (or even timeout) extremely long running iterations of configurations during tuning," https://github.com/AutoPas/AutoPas/issues/673, 2022, accessed: 2024-11-06.

[13] M. Lerchner, "Autopas fuzzy tuning - bachelor thesis," 2024, accessed: 2024-11-26. [Online]. Available: https://github.com/ManuelLerchner/AutoPas-FuzzyTuning-Bachelor-Thesis

[14] *Solving Software Challenges for Exascale*. Stockholm, Sweden: Springer, 2015, revised Selected Papers.

[15] A. P. Thompson, H. M. Aktulga, R. Berger, D. S. Bolintineanu, W. M. Brown, P. S. Crozier, P. J. in 't Veld, A. Kohlmeyer, S. G. Moore, T. D. Nguyen, R. Shan, M. J. Stevens, J. Tranchida, C. Trott, and S. J. Plimpton, "Lammps - a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales," *Computer Physics Communications*, vol. 271, p. 108171, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0010465521002836

[16] N. Tchipev, S. Seckler, M. Heinen, J. Vrabec, F. Gratl, M. Horsch, M. Bernreuther, C. W. Glass, C. Niethammer, N. Hammer, B. Krischok, M. Resch, D. Kranzlmüller, H. Hasse, H.-J. Bungartz, and P. Neumann, "Twetris: Twenty trillion-atom simulation," *The International Journal of High Performance Computing Applications*, vol. 33, no. 5, pp. 838–854, 2019. [Online]. Available: https://doi.org/10.1177/1094342018819741