Auto-Tuning with Early Stopping 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 introduce the AutoPas framework as a natural progression from existing MD engines, such as GROMACS, LAMMPS, and ls1 mardyn[1], 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 21.1% in specific 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 before the start of the simulation. As their implementation is determined 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 optimization 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 dynamic optimizations to adjust its implementation based on the simulation state and the actual hardware performance. This approach is favorable, 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 selected implementation.

This paper provides an overview of the AutoPas framework and the benefits and challenges of using dynamic auto-tuning in molecular dynamics simulations. Moreover, we investigate the potential of an early stopping mechanism and evaluate its performance in the context of the AutoPas framework.

[1]While it is possible to include AutoPas into ls1 mardyn and LAMMPS, we solely consider both standalone applications in this paper.

II. Related Work

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 an overview of these engines.

A. GROMACS

GROMACS implements a single, highly optimized variant of the Verlet Cluster List algorithm. The algorithm allows for flexible cluster sizes specifically designed for achieving good SIMD vectorization [2].

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 [2]. With suitable values for M and N, computations of $M \times N$ particle interactions can be performed with just two SIMD load instructions [3], 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 [3].

Tuning those parameters is time-consuming and relies on a detailed understanding of many low-level software optimization aspects of the different hardware platforms [2]. In GROMACS, the developers must create and maintain such performance models for each supported hardware platform, which is a time-consuming and labor-intensive process [2].

B. LAMMPS

LAMMPS utilizes a performance-optimized version of the Verlet List algorithm. To optimize cache performance, LAMMPS stores its neighbor lists in a global data structure split across multiple memory pages. Each page stores neighbor lists for multiple particles in a contiguous memory block, allowing for efficient allocation and deallocation of the neighbor lists [4]. All particle data is stored in a Structure of Arrays (SoA) layout [4], allowing for efficient vectorization of the force calculations.

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 [5]. Internally, ls1 mardyn uses the default data layout of *AoS* (Array of Structures) for storing particle data and converts it to an *SoA* (Structure of Arrays) layout when required. 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 [5].

While GROMACS, LAMMPS, and ls1 mardyn achieve remarkable performance through specialized and statically optimized implementations, their reliance on manual tuning and hardware-specific adjustments presents challenges. These limitations motivates the need for adaptive frameworks like AutoPas, which aim to dynamically optimize configurations during runtime, providing a more flexible and user-friendly approach to high-performance molecular dynamics simulations.

III. AutoPas

AutoPas[2] was developed on the basis of creating an efficient node-level particle simulation engine applicable to a wide range of scientific fields [6]. To achieve this goal, AutoPas uses a modular architecture that seamlessly integrates different algorithms and data structures into 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.

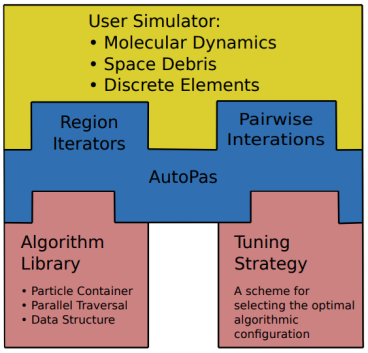


Fig. 1: AutoPas Library Structure as depicted by [7]

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 are combined 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 maintainable and can provide performance portability across a wide range of hardware platforms. The interchangeable nature of the implementations also ensures implicit backward compatibility, making it easy to study the effects of new hardware on existing implementations. 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 and can fully focus on the high-level aspects of their simulation [6] [8].

B. Tunable Parameters

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

Container

Containers are responsible for storing the simulation particles so 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[3] particles in neighboring cells (depicted in blue) must be considered, as all other particles are guaranteed to be outside the cutoff radius.
  LinkedCells introduces many spurious distance calculations (shown by arrows to gray particles), which can be reduced by using a smaller cell size factor [9]. LinkedCells are very cache-friendly as spatially close particles are stored together in memory [8].
- ***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

[2] https://github.com/AutoPas/AutoPas

[3] If $cellSizeFactor = cellSize/r_c = 1$.

to LinkedCells, each particle maintains its own list of potential neighbors, resulting in a 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 [10]. VerletLists have few spurious distance calculations but are less cache-friendly as neighboring particles are not stored together in memory [8]. This results in very inefficient vectorization [2].

- ***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 performing neighbor-list calculations on a cluster level [2]. This reduces the memory overhead significantly. However, it results in more spurious distance calculations as all particles in neighboring clusters must be considered when calculating forces. When M is chosen in accordance with the SIMD width of the system, efficient vectorization is possible [8] [2].

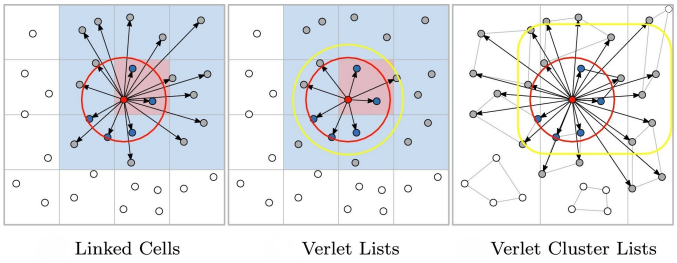


Fig. 2: Important container types as depited by [8]. The cutoff radius r_c is shown using a red circle. Arrows represent distance checks between particles. Only particles shown in blue contribute to the final force calculation.

Data Layout

The Data Layout describes how the particles are 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. However, accessing the properties of a single particle is more expensive, requiring multiple memory accesses. *AoS* is the opposite. It allows for efficient access to properties of a single particle to, e.g., send it to another MPI rank, but results in inefficient vectorization [8].

Newton 3

A common optimization in molecular dynamics involves applying Newton's third law to reuse force calculations between particle pairs, effectively reducing computational costs by half. However, not all parallel traversal patterns can safely implement this optimization without risking race conditions. For those that do, synchronization points must be added to ensure accurate force application, which in turn limits the potential for parallelism.

Traversal

Traversals are responsible for iterating over the particles in the simulation and calculating their interactions in a shared-memory environment [11]. 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. Newton's Third Law cannot be used with this traversal pattern since adjacent cells are processed simultaneously, which could lead to race conditions when applying forces. As no synchronization between cells is required, the C01 traversal pattern has the highest degree of parallelism [10].
- *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, which allows for a safe application of forces on neighboring cells and reduces scheduling overhead [10]. As the color groups must be processed sequentially, the C18 traversal introduces 18 synchronization points, which reduces the overall degree of parallelism [10].
- *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 [10].

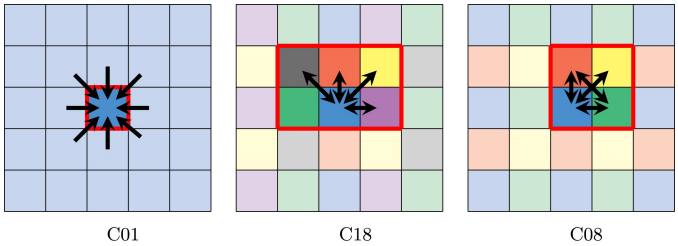


Fig. 3: Important Traversal Types as depicted by [10].

Load Estimator

Different load estimators can be used to estimate and fairly divide work across all available processing units. Maintaining a good workload distribution reduces the idle time of processing units and thus increases the overall simulation speed.

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 [9] [12], the cell size factor must be chosen carefully.

C. Auto-Tuning Framework

A 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 runtime or energy efficiency [8]. 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 currently available tuning strategies in AutoPas are:

FullSearch

The FullSearch strategy naively evaluates all possible configurations, thus always finding the best configuration.

RandomSearch

The RandomSearch strategy randomly selects configurations out of the entire 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 [13]. An improvement to better account for the discrete tuning space of AutoPas called *BayesianClusterSearch* is also available [13].

PredictiveTuning

The PredictiveTuning strategy extrapolates previously gathered timing measurements of a configuration to predict its performance in the current simulation state. This allows for an efficient rejection of inefficient configurations without evaluating them [14].

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 to make decisions [15].

FuzzyTuning

The FuzzyTuning strategy is similar to the RuleBasedTuning strategy but uses fuzzy logic systems to evaluate configurations. This allows for both an inter- and extrapolation of the rules to account for configurations not covered by the expert knowledge [16].

IV. Benefits of Auto-Tuning

Since no single configuration can deliver optimal performance across all simulation scenarios [6], dynamic performance tuning is essential for maintaining high efficiency across diverse simulation conditions. This section discusses some of the key benefits of using AutoPas's auto-tuning framework.

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 improve simulation times across diverse molecular dynamics simulation scenarios, both in the standalone application as well as in established MD engines such as ls1 mardyn and LAMMPS [11] [8]. 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 users to achieve optimal performance directly out of the box without requiring deep expertise in performance optimization. Different performance characteristics caused by varying hardware setups are automatically accounted for by the auto-tuning framework, allowing users to focus on the actual simulation [6].

The inherent user-friendliness is particularly valuable when integrating AutoPas into other simulation frameworks, as developers can treat the simulation engine as a black box and let the auto-tuning framework handle the optimization.

V. Drawbacks of Auto-Tuning

Suboptimal Configurations

A major drawback of auto-tuning in the way it is implemented in AutoPas is the inherent overhead caused by tuning phases. Even though it is an important characteristic of tuning strategies to infer and evaluate only those configurations that are likely to perform well, this inference is not always successful. Many configurations can turn out to be orders of magnitude slower than the optimal configuration [15] [16], making their evaluation extremely costly.

Advanced tuning strategies such as *RuleBasedTuning* or *FuzzyTuning* can mitigate this problem to some extent, however even they cannot guarantee always finding the best configuration, as the underlying expert knowledge for both strategies is expected to be highly incomplete.

This challenge highlights the need for additional optimization techniques, such as the early stopping mechanism introduced in section VI, to reduce the overhead caused by evaluating suboptimal configurations.

Figure 4 shows a typical timing profile of the ExplodingLiquid simulation for every available tuning strategy. The inherent overhead caused by tuning phases is very noticeable and is present in all tuning strategies. For some tuning strategies, the overhead results in a considerable increase in the total simulation time (see default cases in Figure 5a and Figure 5b).

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 aforementioned 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 an improved configuration. In such cases, the overhead of re-tuning is unnecessary and only increases the total simulation time. Figure 4 depicts this effect for the ExplodingLiquid simulation.

Further evaluation of the data obtained in [17] shows that only three out of 184 runs using slight variations of example scenarios provided by `md-flexible`[4] with both single and multi-node configurations 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.

More complex scenarios, most likely involving multiple MPI ranks and inhomogeneous particle distributions, are necessary to fully demonstrate the benefits of periodic re-tuning. Simulating such inhomogeneous scenarios in an MPI environment increases the chance of varying particle distribution across both MPI ranks and time steps, increasing the likelihood of benefitting from periodic re-tuning on each rank (See [7]).

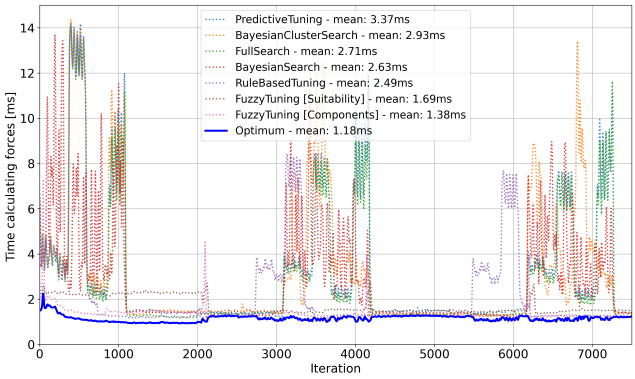


Fig. 4: Typical timing profile of the ExplodingLiquid simulation for every available tuning strategy. The plot clearly shows that all tuning strategies introduce overhead during the tuning phases. Data obtained from [17]

[4]`md-flexible` is a molecular dynamics simulation framework making use of the AutoPas library. It is included in the AutoPas repository and provides a wide range of example scenarios to demonstrate the capabilities of the AutoPas library.

VI. Early Stopping Optimization

To minimize some of the introduced drawbacks of the auto-tuning process, [15] [16] [18] 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 to 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 however requires fully evaluating some samples of a bad configuration.
- **Interrupting the Evaluation**
  A more fine-grained approach, proposed in [15], could interrupt the evaluation of a long-running configuration while it is still being evaluated. Such a change would require a big rewrite of AutoPas' internal structure and is 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[5]. 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, called $earlyStoppingFactor$, to determine how much slower a configuration can be compared to the best-known configuration before it is considered inefficient. The effect of this threshold will be evaluated in the following section.

Algorithm 1 Early Stopping Algorithm in AutoPas

```
1:  procedure CHECKEARLYSTOPPINGCONDITION( )
2:    if not ENOUGHCONFIDENCE( ) then
3:      return
4:    end if
5:    evidencePred ← ESTIMATERUNTIMEFROMSAMPLES( )
6:    bestEvidence ← GETLATESTOPTIMALCONFIGURATION( )
7:    slowdownFactor ← evidencePred / bestEvidence
8:    if slowdownFactor > earlyStoppingFactor then
9:      abort ← true
10:   end if
11: end procedure
12: procedure GETNEXTCONFIGURATION
13:   if not inTuningPhase then
14:     return (currentConfig, false)
15:   else if numSamples < maxSamples and not abort then
16:     return (currentConfig, true)
17:   else
18:     stillTuning ← TUNECONFIGURATION( )
19:     return (newConfig, stillTuning)
20:   end if
21: end procedure
```

[5]See https://github.com/AutoPas/AutoPas/pull/995 for the changes.

A. Evaluation: Exploding Liquid Simulation

To evaluate the performance of the early stopping mechanism, we perform benchmarks using the Exploding Liquid scenario[6] provided by `md-flexible`. The simulation consists of 1764 initially close-packed particles, rapidly expanding outwards and eventually hitting the simulation boundaries. All runs are performed using a single node of the CoolMUC4 supercomputer with 2 threads and are repeated five times.

FullSearch (Figure 5a)

When using the early stopping optimization together with the FullSearch strategy, the total simulation time can be reduced from 15.86 seconds to 12.58 seconds at $earlyStoppingFactor \approx 4$. This results in a reduction of the total simulation time by 20.7%.

PredictiveTuning (Figure 5b)

When using the early stopping optimization together with the PredictiveTuning strategy, the total simulation time can be reduced from 13.93 seconds to 10.99 seconds at $earlyStoppingFactor \approx 4$. This results in a reduction of the total simulation time by 21.1%.

FuzzyTuning (Figure 5c)

When using the early stopping optimization together with the FuzzyTuning strategy, no significant reduction in the total simulation time is observed.

B. Analysis and Discussion

Early Stopping Factor:

As expected, the benchmark results suggest that choosing a lower $earlyStoppingFactor$ generally results in shorter total simulation times, while higher thresholds lead to longer total simulation times, whenever *bad* configurations are evaluated. Thus the time savings correlate with the number of configurations that are stopped early.

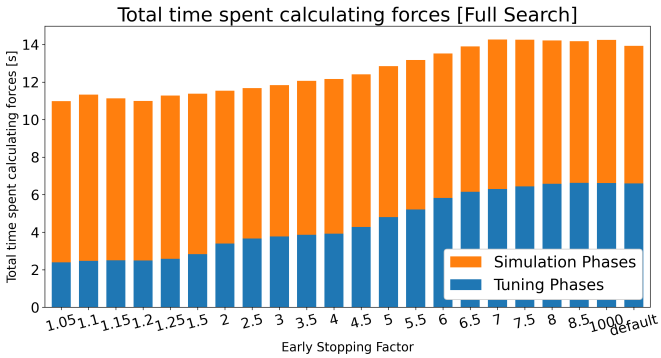
The limiting case of $earlyStoppingFactor \to \infty$ results in the same total simulation time as without the early stopping mechanism, as the early stopping mechanism is effectively disabled.

Interestingly, the limit $earlyStoppingFactor \to 1$ does not impact the total simulation time negatively. This suggests that the performance prediction is very accurate, even when very few samples are available, and allows for very aggressive early stopping.

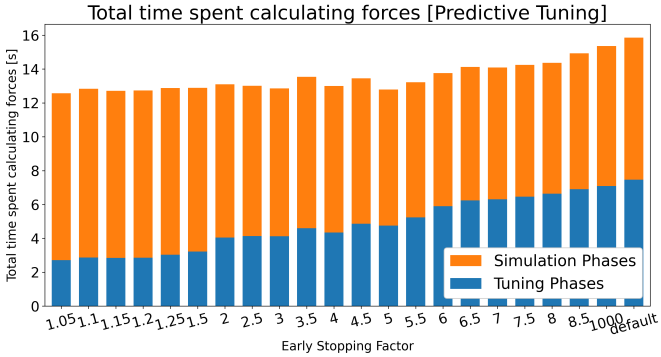
However, in general, an ideal $earlyStoppingFactor$ should be chosen high enough to provide a sufficient margin to account for measurement noise, while being low enough to stop the evaluation of slow configurations effectively. Based on the benchmark results, $earlyStoppingFactor \approx 1.5$ seems to be a good compromise, as lower thresholds would only reduce the total simulation time marginally.

It is however unclear whether such a threshold is universally applicable to all scenarios. Additional benchmarking across a broader range of use cases is needed to validate these preliminary findings.

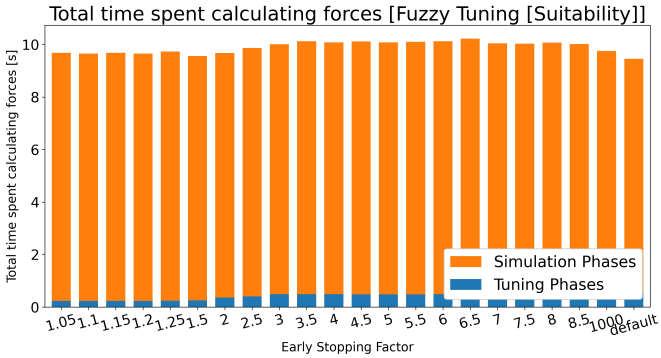
[6]The input file can be found at explodingLiquid.yaml. A video of the simulation is available at https://youtu.be/u7TE5KiSQ08.



(a) Total Simulation Time for Exploding Liquid Simulation using the FullSearch strategy with early stopping.



(b) Total Simulation Time for Exploding Liquid Simulation using the PredictiveTuning strategy with early stopping.



(c) Total Simulation Time for Exploding Liquid Simulation using the FuzzyTuning[Suitability] strategy with early stopping.

Effect on Tuning Strategies:

The evaluated benchmarks show that the FullSearch and PredictiveTuning strategies benefit drastically from early stopping. The FullSearch strategy naturally encounters many bad configurations and performs as expected. The PredictiveTuning strategy however can't show its full potential, since the benchmark is too short for the strategy to leave the initial data collection phase, during which it behaves similarly to the FullSearch strategy.

As the FuzzyTuning strategy performs nearly optimal even without early stopping, the use of the early stopping mechanism does not provide any further benefits.

Limitations and Future Work:

The current implementation of the early stopping mechanism starts with a fresh set of evidence samples in each tuning phase. This results in not using the information gathered in previous tuning phases, which could potentially help to stop evaluating configurations earlier.

This ensures that the $bestEvidence$ variable is always up-to-date with the current simulation state and prevents the early stopping mechanism from mistakenly stopping the evaluation of a configuration based on timing measurements that can no longer be achieved due to changes in the simulation state.

An improved version could keep track of a running average of timing measurements collected during the prior simulation phase, allowing the early stopping mechanism to start with a reasonable estimate of achievable performance, thus increasing the likelihood of stopping unsuitable configurations early.

VII. Conclusion

We presented an overview of the AutoPas framework and its auto-tuning capabilities and demonstrated 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 and demonstrated that such a mechanism can reduce the total simulation time by up to 21.1% in specific scenarios.

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 by accepting some performance overhead.
However, the successful integration of AutoPas 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] 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

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

[4] 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," 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0010465521002836

[5] 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

[6] 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

[7] 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.

[8] 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

[9] C. Menges, "Optimization and evaluation of the linked-cell algorithm," Bachelor's Thesis, Technical University of Munich, Aug 2019.

[10] 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

[11] 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

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

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

[14] J. M. Pelloth, "Implementing a predictive tuning strategy in autopas using extrapolation," Bachelor's Thesis, Technical University of Munich, Sep 2020.

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

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

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

[18] 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.