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 selected based on predefined performance models and are often 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: AutoPas heavily relies on dynamic optimizations and is able to adjust its implementation based on the current 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, its auto-tuning capabilities, 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 in its algorithm library, specificly designed to efficiently solve N-Body problems. The modular nature of AutoPas enables it to support more than a single *configuration*.

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 reduce the configuration search space according to a certain strategy. Figure 1 illustrates the high-level architecture of the AutoPas library.

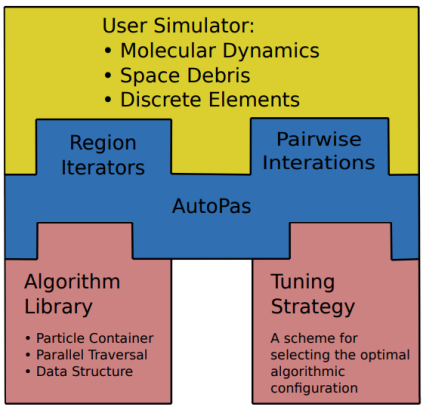

Fig. 1: AutoPas Library Structure as depicted by [3]

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.

A combination of different implementations for each key aspect of the simulation is called a *configuration*. Currently AutoPas supports the six tunable parameters: *Container*, *Data Layout*, *Newton 3*, *Traversal*, *Load Estimator* and *Cell Size Factor*.

Maintaining a library of different implementations is very beneficial for researchers and developers. The modular nature of AutoPas allows for an easy extension of the algorithm library with new, potentially hardware-specific, implementations for each key aspect of the simulation. The big search space of possible configurations associated with the algorithm library allows for a high degree of flexibility and provides performance portability across different hardware platforms [2]. Another benefit of this modular approach is presented with the implicit backward compatibility of the interchangable implementations. With the ever growing number of configurations, it is possible to test the feasibility of older implementations under new hardware [2].

B. Tunable Parameters

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 identification using just $O(N)$ distance calculations is possible [1]. The important container types depicted in Figure 2 will be shortly described below.

- ***Linked Cells***
  The Linked Cells algorithm maintains a grid of cells with a length of r_c (when $cellSizeFactor = 1$). 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.
  LinkedCell casues many spurious distance calculations (shown by many arrows to gray particles in the figure). As particles for a cell can be stored together in memory, LinkedCells are however very cache-friendly [4].
- ***Verlet Lists***
  The Verlet List 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. 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 in far higher memory consumptions and are less cache-friendly [4], resulting in inefficient vectorization [6].
- ***Verlet Cluster Lists***
  The Verlet Cluster Lists algorithm improves on the VerletList algorithm by grouping particles into clusters of size M ($M = 4$ in the figure). Maintaining the neighbor list and keeping track of buffer regions is done on a cluster level, reducing the memory overhead of the VerletList algorithm.
  As all particles in overlapping clusters need to be considered for the force calculation, the number of spurious distance calculations increases again. When M is chosen in accordance with the SIMD width of the system, efficient vectorization is possible [4].

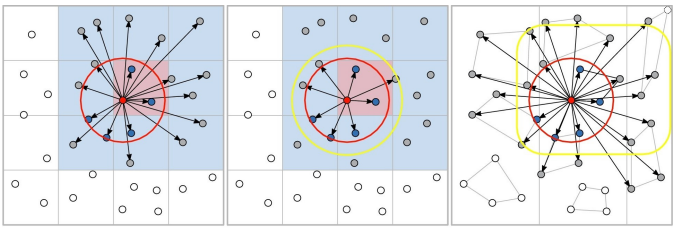

Fig. 2: Important Container Types as depited by [4]. The cutoff radius r_c is shown using a red circle. The arrows represent distance checks between particles. Only particles shown in blue contribute to the final force calculation.

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. However, it causes information about a single particle to be spread across multiple memory locations. *AoS* layout stores all information about a single particle together but prohibits efficient vectorization as filling vector registers requires gathering data from multiple memory locations.

Newton 3

Applying Newton's third law to the force calculations allows for a reduction of the number of force calculations by half, as the calculated force between two particles can be reused for the second particle. The optimization can be enabled or disabled in accordance with the interaction model and the traversal pattern.

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. No synchronization between cells is required, resulting in a high 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 that the cells are colored in a regular pattern, such 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* and *right* of it.
  During the force calculation, all available threads are working on a single color, and can therefore safely apply forces on neighboring cells.
  The color groups must be processed sequentially, resulting in 18 synchronization points. Each color can however be fully processed in parallel, still resulting in a high overall degree of parallelism [5].
- *C08*
  traversal pattern is similar to the C18 traversal pattern but uses a different coloring scheme with only eight colors. This reduces the number of synchronization points, resulting in a higher degree of parallelism at the cost of more scheduling overheads [5].

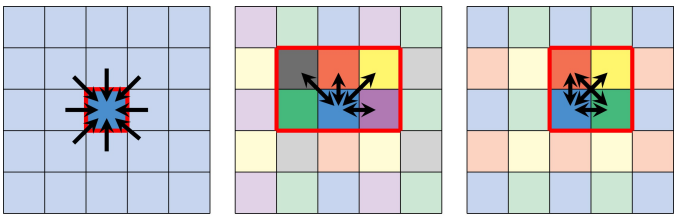

Fig. 3: Important Traversal Types as depicted by [5].

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 and thus increases the overall performance of the simulation.

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. By reducing the cell size factor, the number of spurious distance calculations can be reduced, at the cost of having to maintain more cells in memory [8]. A balance between the number of spurious distance calculations and the memory overhead must be found in order to achieve optimal performance.

C. Auto-Tuning Framework

As described previously, manual selection of suitable implementations for each tunable parameter is a daunting task and 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 maximize 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 of the naively evaluated configurations turn out to be orders of magnitude slower than the best-known Configuration, thus causing a drastic increase of the total simulation time [9] [10]. As AutoPas is developed further and new implementations are added to the algorithm library, the number of possible configurations will steadily increase, constantly exacerbating the problem of evaluating all configurations naively.

AutoPas attempts to mitigate this problem by using Tuning Strategies to select promising configurations. Tuning strategies are tasked with pruning the search space of possible configurations using certain rules or heuristics. Tuning strategies try to balance the trade of between encountering new, potentially better configurations with the cost of testing suboptimal configurations [3].

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. Therefore, the RandomSearch strategy causes less overhead than the FullSearch strategy, but is less likely to actually 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 the performance of previously evaluated configurations [11]. There also exists an improvement to account for the discrete tuning space of AutoPas called *BayesianClusterSearch* [11].

PredictiveTuning
The PredictiveTuning strategy extrapolates a previously gathered measurement of a container to predict the performance in the current simulation state.

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 simulation (called *LiveInformation*) to eliminate configurations that are unlikely to be the best Configuration [9].

FuzzyTuning

The FuzzyTuning strategy is similar to the RuleBasedTuning strategy, but uses a fuzzy logic system to evaluate the desirability of a configuration. This allows for both an interpolation 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], thereby providing compelling evidence for the effectiveness and importance of this 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 or parallel computing, which represents a significant advantage for the molecular dynamics community where researchers often need to focus on their scientific objectives rather than computational intricacies. This inherent user-friendliness is particularly valuable when integrating AutoPas into other simulation frameworks, as developers can leverage its sophisticated auto-tuning capabilities while maintaining a straightforward implementation path that minimizes the complexity traditionally associated with performance optimization in high-performance computing environments.

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 many suboptimal configurations quickly adds up. This is especially problematic as the performance of different configurations can span several orders of magnitude [9] [10], quickly leading to noticeable increases in the total simulation time.

Even though the tuning strategies employed by AutoPas are highly efficient, they still sometimes suggest suboptimal configurations. Rule-driven tuning strategies such as *RuleBasedTuning* and *FuzzyTuning* can mitigate this problem to some extent by making use of expert knowledge, however even they can not guarantee to find the best configuration in all cases, as the underlying expert knowledge is expected to be highly incomplete.

Consequently, there will always be some overhead caused by performing tuning phases.

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 than the current one. 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 refined configuration. Figure 4 shows this effect for the ExplodingLiquid scenario provided by the `md-flexible` framework.

Further evaluation of `md-flexible` data obtained in [12] using the provided example scenarios shows that only three out of 184 run show any changes in the best Configuration after the first 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 additional tuning phases are mostly causing unnecessary overhead.

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 can cause the load to be distributed unevenly across both MPI ranks and time steps, further increasing the potential benefits of performing periodical re-tuning on each rank (See [3]).

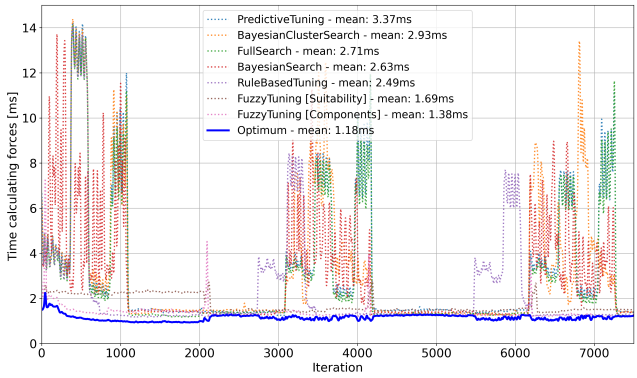

Fig. 4: Time spent calculating forces per iteration in the ExplodingLiquid Scenario. There exists a considerable overhead in the total simulation time caused by tuning phases. As the simulation is stable over time, the overhead of re-tuning is unnecessary. (Data obtained from [12])

V. Early Stopping Optimization

To minimize some of the introduced drawbacks of the auto-tuning process, [9] [10] [13] 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

procedure EVALUATECONFIGURATION(performance)
  $fastestTime \leftarrow \min(fastestTime, performance)$
  $slowdownFactor \leftarrow \frac{performance}{fastestTime}$
  if $slowdownFactor > allowedSlowdownFactor$ **then**
    $abort \leftarrow true$
  end if
end procedure
procedure GETNEXTCONFIGURATION
  if not $inTuningPhase$ **then**
    return $(currentConfig, false)$
  else if $numSamples < maxSamples$ **and** not $abort$ **then**
    return $(currentConfig, true)$
  else
    $stillTuning \leftarrow$ TUNECONFIGURATION( )
    return $(newConfig, stillTuning)$
  end if
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%.

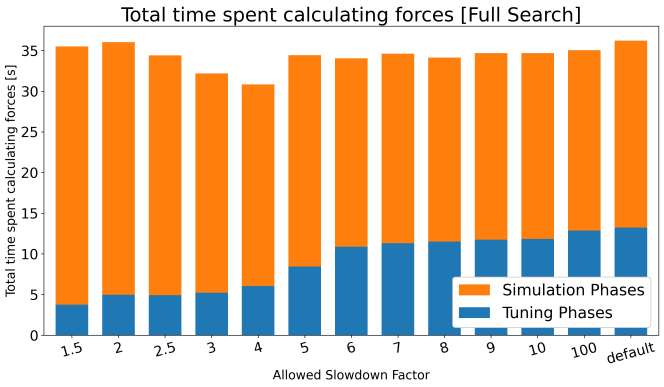

Fig. 5: Total Simulation Time for Exploding Liquid Simulation for different values of $allowedSlowdownFactor$ using the FullSearch strategy with Early Stopping.

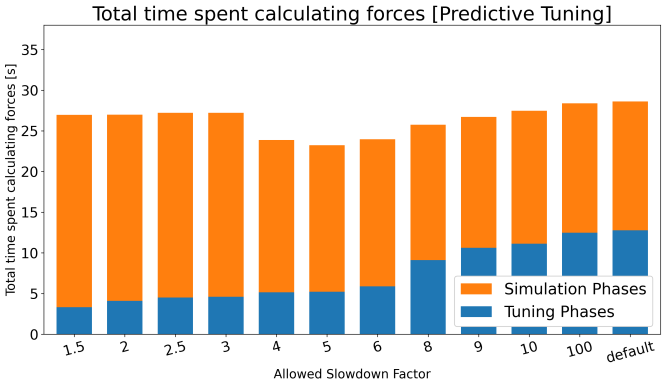

Fig. 6: Total Simulation Time for Exploding Liquid Simulation for different values of $allowedSlowdownFactor$ using the PredictiveTuning strategy with Early Stopping.

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

[13] 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.

[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