

Algorithm Selection and Auto-Tuning in AutoPas

Manuel Lerchner
Technical University of Munich
Munich, Germany

Abstract—Molecular dynamics (MD) simulations face significant computational challenges that require highly optimized simulation engines to deal with the enormous number of particles present in modern simulations. Naturally, researchers have put much effort into developing algorithms and frameworks that can efficiently simulate these systems. This paper examines the auto-tuning capabilities of AutoPas, a modern MD framework, and provides a comparative analysis with other prominent MD engines such as GROMACS and LAMMPS. We analyze the static and dynamic optimization approaches and evaluate their effectiveness in various simulation scenarios. Furthermore, we investigate a possible improvement to AutoPas' auto-tuning capabilities by introducing an early stopping mechanism to reduce the overhead of parameter space exploration.

Index Terms—molecular dynamics, auto-tuning, algorithm selection, performance optimization, GROMACS, LAMMPS

I. INTRODUCTION

Molecular dynamics simulations represent a computational cornerstone in various scientific fields, from materials science to biochemistry. These simulations typically use complex and computationally intensive interaction models acting on enormous numbers of particles to ensure accurate results. For simulation engines to be practical, they must be highly optimized to handle the computational load efficiently and utilize available resources effectively.

Prominent optimization techniques used in modern molecular dynamics (MD) engines fall into two main categories: static and dynamic optimization. Static optimizations rely on predefined configurations and performance models, often fine-tuned for specific hardware architectures. These optimizations include strategies like memory layout optimization, vectorization, and architecture-specific instruction use (e.g., SIMD).

Modern compiler frameworks like Kokkos and SYCL further abstract hardware-specific optimizations, enabling more portable code across different hardware platforms (e.g., GPUs, CPUs). Kokkos provides a performance-portable parallel programming model that supports diverse high-performance computing environments, while SYCL, a standard by the Khronos Group, facilitates single-source C++ for heterogeneous platform.

In contrast, dynamic optimizations adjust parameters based on the current simulation state and the actual hardware performance. Unlike static optimizations, set before the simulation begins, dynamic optimizations allow for adjustments throughout the simulation. This approach enables MD engines to periodically measure and respond to actual performance, optimizing parameters like load balancing, cache locality, and

communication patterns to improve efficiency under complex and possibly changing conditions.

In particular, we will focus on the auto-tuning capabilities of AutoPas, a modern MD framework that focuses on dynamic optimization techniques to achieve high performance in complex and possibly changing simulation scenarios. To evaluate their effectiveness in various simulation scenarios, we will compare AutoPas's auto-tuning capabilities with other prominent MD engines, such as GROMACS and LAMMPS. We will also investigate a possible improvement to AutoPas' auto-tuning capabilities by introducing an early stopping mechanism 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 applications [1]. To support these various simulations, AutoPas is built on a modular software architecture that allows different algorithms and data structures to be used interchangeably in the underlying simulation engine. AutoPas acts as a middleware between the simulation code provided by the user and various implementations of algorithms and data structures, which are chosen dynamically based on performance criteria by so-called *tuning strategies*. Figure 1 shows a high-level overview of the AutoPas library structure.

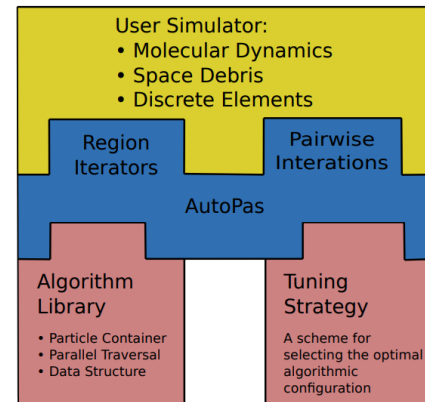


Fig. 1: AutoPas Library Structure as depicted by [2]

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

CITE:
Studies of
dynamic
tuning
in MD
simulation
or load
balancing
techniques

clusion

cient
n-
ation
Opti-
zation
techniques

lecular
dynamics
simulations,
or
similar

Kokkos
System:
impre-
cise
for-
mance
stability

as neighbor identification, traversal patterns, and memory layouts.

Together they form a so-called *Configuration* fully describing the internal implementation of the engine. Such a configuration is a 6-tuple consisting of implementations for: *Container*, *Traversal*, *Load Estimator*, *Data Layout*, *Newton 3*, and *Cell Size Factor*.

An obvious benefit of this modular approach is the ability to easily swap out implementations for certain simulation aspects without changing the entire codebase, allowing for straightforward investigation of different implementations and quickly adapting to new hardware or simulation scenarios.

Another benefit of this modular approach is presented with the ever-growing configuration library of AutoPas. As the library is constantly updated and new implementations targeting specific hardware or simulation scenarios are added, it is possible to easily test the feasibility of older implementations under new hardware [1].

1) *Container*: The *Container* is responsible for storing the particles of the simulation such that relevant neighbor particles can be determined efficiently. As AutoPas focuses on short-range interactions and uses a cutoff radius r_c , neighbor identification using just $O(N)$ distance calculations is possible [3]. Such efficient neighbor identification is required to make the simulation of large particle systems feasible. Figure 2 shows important container types used in AutoPas.

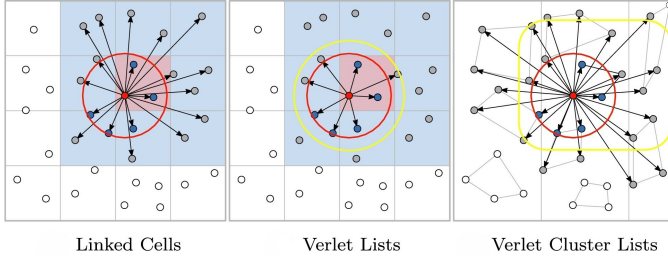


Fig. 2: Important Container Types as depicted by [4]. The cutoff radius r_c is shown using a red circle.

- The *LinkedCells* container maintains a grid of cells with a length of $r_c * cellSizeFactor$, where each cell contains a list of particles. Only neighboring cells (depicted as blue cells) need to be considered when calculating the forces between particles. All other particles are guaranteed to be outside the cutoff radius and can be ignored. *LinkedCells* typically have a significant overhead of useless distance calculations (shown by the many arrows in the figure) but are very cache-friendly [4].
- The *VerletList* container uses a second radius $r_v = r_c + \Delta_s$ (yellow circle) and considers all particles within this radius as potential neighbors. By introducing this buffer region, it is also possible to only rebuilt the neighbor-list every N simulation step, as long as no particle can move from outside $r_c + \Delta_s$ to inside r_c unnoticed [5]. *VerletLists* can drastically reduce spurious distance calculations but cause far higher memory consumptions and

are less cache-friendly [4]. This results in inefficient vectorization as memory operations and data shuffling can take longer than the actual force calculations [6].

- The *Verlet Cluster Lists* container improves on the *VerletList* container by grouping particles into clusters of size M ($M = 4$ in the figures). When calculating the forces between particles, all particles from overlapping clusters within the r_v radius need to be considered.

Naturally, this increases the number of spurious distance calculations again. However, the number and size of the maintained lists can be reduced drastically. Furthermore, M can be chosen to make efficient vectorization possible [4].

2) *Traversal*: *Traversals* is responsible for iterating over the particles in the simulation and calculating their interactions in a shared-memory environment [7]. The traversal pattern is a key aspect of the simulation engine, as it determines to which extent force calculations can be parallelized and whether optimizations, such as Newton 3, can be applied.

Newton 3 is a common optimization technique in MD simulations, which states that the force between two particles is equal and opposite. This allows for a reduction of the number of force calculations by half, as the force acting between two particles only needs to be calculated once. However, this optimization is only sometimes possible, as it could introduce race conditions in neighboring cells that are processed in parallel.

Figure 3 shows important traversal patterns used in AutoPas.

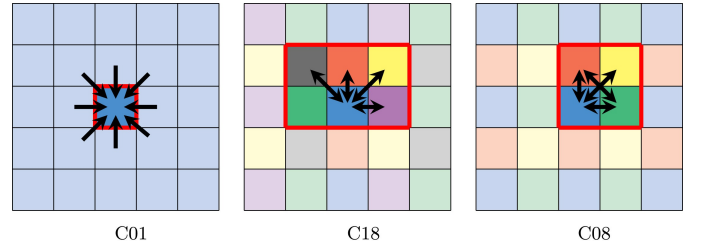


Fig. 3: Important Traversal Types as depicted by [5]. Arrows represent force calculations between cells. A double-sided arrow indicates that the Newton 3 optimization is applied.

- The *C01* traversal pattern is a simple traversal pattern that processes each cell independently. Using Newton 3 to calculate forces between cells is impossible, as neighboring cells can be processed in parallel, possibly resulting in race conditions. This traversal pattern is embarrassingly parallel, with no synchronization between cells.
- The *C18* 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 neighbors. To ensure that forces are only applied once, each cell only applies Newton 3 forces to cells *above* and *right* of it. During the force calculation, all available threads are assigned to a single color, and can therefore safely apply

the force obtained by Newton 3 on neighboring cells.

The color groups must be processed sequentially, but the cells within a color group can be processed embarrassingly parallel again.

Due to the 18 synchronization points, it is less parallelizable than the C01 traversal pattern but causes fewer scheduling overheads as threads can work on bigger chunks of data [5].

- The 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].

3) *Data Layout*: The *Data Layout* is responsible for how the particle data is stored in memory. The choice of data layout can significantly impact the simulation engine's performance, as it determines the memory access patterns and the potential for vectorization. Possible choices are *SoA* (Structure of Arrays) and *AoS* (Array of Structures). *SoA* is typically more cache-friendly and allows for better vectorization. However, it causes information about a single particle to be spread across multiple memory locations. *AoS* on the other hand stores all information about a single particle together but prohibits efficient vectorization as filling vector registers requires gathering data from multiple memory locations.

B. Auto-Tuning Framework

Manually selecting the best 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]. The auto-tuning framework of AutoPas periodically initiates so-called *tuning-phases* in which it measures. It evaluates promising¹ configurations in order to determine the best Configuration for the current simulation state [3]. 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. Just using the naive approach of evaluating all possible configurations turns out to be 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 [8] [9]. As AutoPas is developed further and new implementations are added to the algorithm portfolio, the number of possible configurations will steadily increase, exacerbating the problem of evaluating all configurations naively.

AutoPas attempts to mitigate this problem by using so-called *TuningStrategies*, which prune the search space of possible configurations based on specific criteria [3]. Tuning strategies are tasked with pruning enough configurations to

make the tuning phase feasible, but not too many, to avoid missing the optimal Configuration. Tuning strategies need to balance the trade-off between potentially finding a better configuration and the cost of potentially encountering worse configurations during exploration [2].

C. Static Optimization Techniques

Compilers apply static optimization techniques to generate efficient machine code for hardware architecture. As modern compilers are aimed at generating good average performance across a wide range of hardware platforms, default optimization settings are often not optimal for a specific hardware platform and application [10], and some extra effort is required to tune the compiler settings to the specific hardware and application.

Another way to improve the performance of the generated code is profile-guided optimizations. Profile-guided optimizations use prior performance data collected during a simulation run to optimize the generated code for the specific workload during the following compilation.

Apart from the static optimization already performed by modern compilers, AutoPas can perform basic tuning at compile time. As AutoPas however primarily aims at dynamic optimization, the static optimization capabilities are limited to a set of compile-time flags that can be used to statically turn on or off certain features of the simulation engine, such as SIMD instructions, MPI support, or Auto-Vectorization [3]. Those flags primarily aim at ensuring that the simulation engine can utilize the full potential of the hardware it is run on, but they are not used directly to tune the performance of the simulation engine.

1) *Kokkos Integration*: There are, however, attempts to expand the static optimization capabilities of AutoPas beyond just compile-time flags. One such attempt is the integration of the Kokkos library into AutoPas, aiming to provide performance portability across different hardware platforms and be able to run existing algorithms on GPGPUs [11].

Kokkos is a performance-portable parallel programming model supporting diverse high-performance computing environments [12]. Kokkos provides high-level C++ abstractions for parallel execution and memory management, enabling developers to write performance-portable code that can be compiled for different hardware architectures, including GPUs, Intel Xeon Phi, or many-core CPUs [13].

The integration is still experimental and incapable of targeting systems other than conventional multi-core CPUs [11].

Full integration of Kokkos into AutoPas could be beneficial as classical MD Simulation engines such as LAMMPS have demonstrated performance benefits from using Kokkos, particularly on extensive simulations on GPUs [13]. Supporting various hardware platforms could further demonstrate the usefulness of the AutoPas framework, as the performance of the algorithm library is highly dependent on the underlying hardware [7] and changes in the optimal Configuration are expected.

¹Promising configurations are suggested by so-called *TuningStrategies*

III. DEMONSTRATION OF BENEFITS OF AUTOTUNING

Even though AutoPas is designed to perform periodic auto-tuning, it is often sufficient to perform a single tuning phase at the beginning of the simulation, as many scenarios tend to behave fairly stable over time. Simulating an inhomogeneous scenario can sometimes benefit from re-tuning the Configuration after several simulation steps, especially when using an MPI environment with multiple nodes. Inhomogeneous scenarios can cause the load to be distributed unevenly across the nodes and time steps, further increasing the potential benefits of periodically re-tuning the Configuration on each node.

The currently provided example runs from `md-flexible` are insufficient to demonstrate the benefits of periodic re-tuning. More complex scenarios, most likely involving multiple nodes and a high number of particles, are required to demonstrate the benefits of periodic re-tuning.

IV. EARLY STOPPING OPTIMIZATION

As identified by [14] [8] [9], overhead caused by evaluating suboptimal configurations during the tuning phase can be a significant bottleneck in the performance of the AutoPas framework. Even though the tuning strategies employed by AutoPas are highly efficient, they still suggest many configurations that could be more optimal. Rule-driven tuning strategies such as *RuleBasedTuning* and *FuzzyTuning* can mitigate this problem to some extent, but due to the complexity of particle simulations, those rule bases are expected to be highly incomplete.

All mentioned sources suggest that some form of *early stopping* mechanism could benefit 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:** Currently, AutoPas supports testing a certain parameter configuration multiple times to get a more accurate and stable performance measurement. A simple way to implement early stopping will be to stop evaluating further samples of a configuration if the performance of a sample is significantly worse than the best-known Configuration. The implementation of this approach is relatively simple, but it is coarse-grained as all started samples would still be evaluated fully.
- **Interrupting the Evaluation:** A more fine-grained approach, as proposed in [8] would be to interrupt the evaluation of a configuration as soon as it is clear that the performance is significantly worse than the best-known Configuration. This is way more difficult to implement, as it would require the ability to interrupt the evaluation of a configuration at any time. Especially in an MPI environment with multiple nodes, aborting and resetting the simulation to a consistent state would require a lot of synchronization and communication work.

Both mentioned approaches require a user-defined threshold for the maximum allowed slowdown of a configuration before it should be stopped. This threshold will be determined empirically in subsection IV-B.

To get a first impression of the potential benefits of an early stopping mechanism, we implemented the first approach in the AutoPas framework. The changes to the existing code-base are minimal, and the early-stopping mechanism can be implemented using existing functionality. Algorithm 1 shows the implementation of the early stopping mechanism in the AutoPas framework.

A. Implementation

The early stopping mechanism is triggered by the new `CheckEarlyStopping` function, which is called after the performance of a configuration has been measured. The function compares the performance of the current Configuration to the best-known performance encountered in the current tuning phase. If the performance of the current Configuration is significantly worse than the best-known performance, the `abort`

flag is set to `true`. The existing `GetNextConfiguration` function is modified slightly to trigger a re-tuning of the Configuration if the abort flag is set. The abort flag is reset during re-tuning.

Algorithm 1 Early Stopping Algorithm in AutoPas

```

1: procedure CHECKEARLYSTOPPING(performance)
2:    $fastestTime \leftarrow \min(fastestTime, performance)$ 
3:    $slowdownFactor \leftarrow \frac{performance}{fastestTime}$ 
4:   if  $slowdownFactor > maxAllowedSlowdown$  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

```

B. Evaluation: Exploding Liquid Simulation

This section evaluates the performance of the early stopping mechanism described in Algorithm 1. The performance of the early stopping mechanism is evaluated for different values of the maximum allowed slowdown factor to determine the optimal threshold for the early stopping mechanism.

The benchmark is performed with the *Exploding Liquid* scenario in the *md-flexible* framework. The simulation consists of 1764 initially close-packed particles that are simulated with a Lennard-Jones potential. During the simulation, the particles rapidly expand outwards and eventually hit the simulation boundaries².

All benchmarks are performed using 14 threads on a single node of the CoolMUC2³ supercomputer and are repeated 3 times to ensure reproducibility.

1) *Full Search*: This section evaluates the effect of the early stopping mechanism when using the FullSearch strategy. The FullSearch strategy evaluates all possible configurations in each tuning phase without pruning the search space.

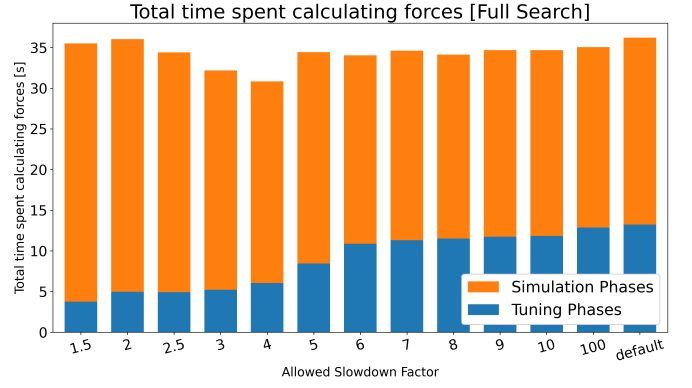


Fig. 4: Total Simulation Time for Exploding Liquid Simulation with Early Stopping divided into tuning and simulation phases. The total simulation time is minimal at a maximum allowed slowdown factor of ≈ 4 .

When comparing the total simulation time of the FullSearch strategy with and without early stopping, it can be seen that the early stopping mechanism can reduce the total simulation time from 36.22 seconds to 30.84 seconds when using a maximum allowed slowdown factor of 4. This is a reduction of 14.8% in the total simulation time.

2) *Predictive Tuning*: Since the effect of the early stopping mechanism is expected to change under different tuning strategies, the effect of the early stopping mechanism is evaluated symbolically for the predictive tuning strategy. The predictive tuning strategy extrapolates previous performance measurements to predict the performance of configurations in the current tuning phase.

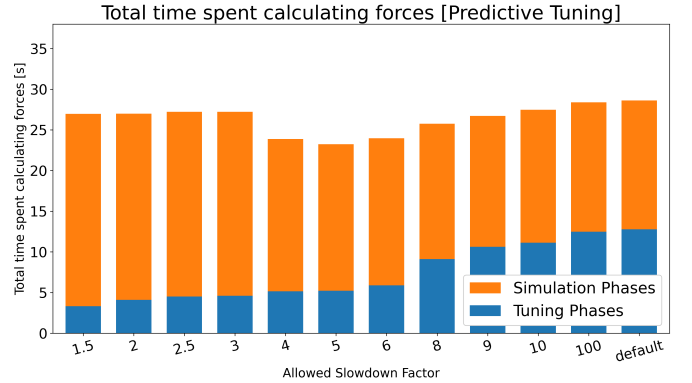


Fig. 5: Total Simulation Time for Exploding Liquid Simulation with Early Stopping divided into tuning and simulation phases. The total simulation time is minimal at a maximum allowed slowdown factor of ≈ 5 .

Compared to Predictive Tuning without early stopping, the early stopping mechanism can reduce the total simulation time from 28.62 seconds to 23.23 seconds when using a maximum allowed slowdown factor of 5. This is a reduction of 18.9% in the total simulation time. The optimal threshold for the

²A simulation video can be found at <https://youtu.be/u7TE5KiSQ08>

³CoolMUC-2 is a supercomputer located at the Leibniz Supercomputing Centre in Garching, Germany. See: <https://doku.lrz.de/coolmuc-2-11484376.html>

early stopping mechanism is around 5 for the *Exploding Liquid* scenario.

C. Analysis and Discussion

All evaluated benchmarks show that a slim range of optimal thresholds exists for the early stopping mechanism, which actually reduces the total simulation time. This is expected, as the two corner cases $\text{maxAllowedSlowdown} \rightarrow 1$ and $\text{maxAllowedSlowdown} \rightarrow \infty$ are expected to perform poorly. $\text{maxAllowedSlowdown} \rightarrow 1$ essentially results in tuning phases with just one sample per Configuration, which is not enough to get a reasonable estimate of the performance of a configuration and causes simulation phases with suboptimal configurations. On the other hand, $\text{maxAllowedSlowdown} \rightarrow \infty$ results in the early stopping mechanism never aborting a configuration, which is equivalent to constantly evaluating all configuration samples, even if the Configuration is known to be suboptimal.

Consequently, the optimal threshold for the early stopping mechanism is a trade-off between the overhead of evaluating suboptimal configurations and the risk of missing optimal configurations due to noise in the performance measurements.

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.

The benchmark for the predictive tuning strategy shows that the early stopping mechanism results in a higher reduction of the total simulation time compared to the FullSearch strategy. This

It is, however, noteworthy that the early stopping mechanism never caused a significant slowdown of the simulation, even if poor thresholds were chosen.

We conclude that the (naive) early stopping mechanism is a valuable addition to the AutoPas framework, as it can significantly reduce the total simulation time without causing any significant slowdowns.

D. Future Work

As the current implementation is only capable of stopping further samples of a configuration, the next step would be to extend the early stopping mechanism to dynamically blacklist certain implementations.

A crude way to implement this would be to create a set of parameters most influential to the performance of the simulation (for example $\{\text{Particle Container}, \text{Data Layout}\}$) and blacklist configurations that use specific values if there is enough evidence that a configuration with those values is not a good choice for the current simulation scenario (potentially if more than $X\%$ of samples using specific values are stopped early).

The parameter *Particle Container* should be included in the set of parameters to blacklist, as it is known to be highly influential to the performance of the simulation [3].

V. COMPARISON WITH OTHER MD ENGINES

Well-established MD engines such as GROMACS, LAMMPS, and ls1 mardyn have been developed over many years and 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 [15], 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 [15].

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 [16], efficiently loading the neighbor list into the cache. All particles are stored in a *SoA* (Structure of Arrays) data layout [16].

LAMMPS supports accelerator packages using either CUDA, OpenCL, ROCm/HIP or OpenMP threads, hand-optimized code, generically optimized code, and code using the KOKKOS library [17].

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 [18]. 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 [18].

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

VI. CONCLUSION

REFERENCES

- [1] 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>
- [2] 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.
- [3] 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.
- [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] T. Humig, "Project report: Exploring performance modeling in autopas," Project Report, Technical University of Munich, Oct 2023.
- [9] M. Lerchner, "Exploring fuzzy tuning technique for molecular dynamics simulations in autopas," Bachelor's Thesis, Technical University of Munich, Aug 2024.
- [10] G. Fursin, "Collective tuning initiative," 2014. [Online]. Available: <https://arxiv.org/abs/1407.3487>
- [11] L. Gärtner, "Integrating kokkos into autopas for hardware agnostic particle simulations," Master's thesis, Technical University of Munich, Feb 2022.
- [12] H. Carter Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202–3216, 2014, domain-Specific Languages and High-Level Frameworks for High-Performance Computing. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0743731514001257>
- [13] LAMMPS, "Speeding up lammmps with kokkos," *LAMMPS Documentation*, 2024. [Online]. Available: https://docs.lammps.org/Speed_kokkos.html
- [14] 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.
- [15] *Solving Software Challenges for Exascale*. Stockholm, Sweden: Springer, 2015, revised Selected Papers.
- [16] 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>
- [17] S. Seckler, "Algorithm and performance engineering for hpc particle simulations," Ph.D. dissertation, Technische Universität München, 2021. [Online]. Available: <https://mediatum.ub.tum.de/1616999>
- [18] 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>

CONTENTS

I	Introduction	1
II	AutoPas	1
II-A	Algorithm Library	1
II-A1	Container	2
II-A2	Traversal	2
II-A3	Data Layout	3
II-B	Auto-Tuning Framework	3
II-C	Static Optimization Techniques	3
II-C1	Kokkos Integration	3
III	Demonstration of Benefits of AutoTuning	4
IV	Early Stopping Optimization	4
IV-A	Implementation	4
IV-B	Evaluation: Exploding Liquid Simulation	5
IV-B1	Full Search	5
IV-B2	Predictive Tuning	5
IV-C	Analysis and Discussion	6
IV-D	Future Work	6
V	Comparison with Other MD Engines	6
V-A	GROMACS	6
V-B	LAMMPS	6
V-C	ls1 mardyn	6
VI	Conclusion	7
	References	7