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 focuses on the AutoPas framework, a modern MD framework 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 to reduce the overhead of parameter space exploration. Our evaluation shows that an early stopping mechanism can reduce the total simulation time by up to 18.9% when using the predictive tuning strategy.**

***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, 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. Therefore, the computational complexity of these simulations grows rapidly with the number of particles, requiring highly optimized simulation engines to guarantee feasible simulation times. 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. They are selected before the simulation begins and remain constant throughout the simulation. Static optimizations include automatic optimizations performed by modern compilers (e.g., loop unrolling, inlining, auto-vectorization), conditional compilation based on the target hardware (e.g. SIMD) [1], or manual selection of simulation parameters based on expert knowledge.

Dynamic optimizations adjust parameters based on the current simulation state and the actual hardware performance. Unlike static optimizations, dynamic optimizations allow for adjustments throughout the simulation. This approach is favourable, as the engine can adapt and optimize itself without external intervention and can adapt itself to the progressing simulation. However, dynamic optimizations come at the cost of increased complexity and potential overhead, as the engine must periodically re-evaluate its configuration based on the current simulation state.

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. 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 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 [2]. Therefore, 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 capable of solving N-Body problems efficiently. As all implementations are (mostly) interchangeable, AutoPas can provide a wide range of so called *Configurations* which fully describe the internal implementation of the engine. To lift the burden of selecting suitable configurations from the user, and to allow for dynamic optimization, AutoPas provides an auto-tuning framework that periodically evaluates different configurations and selects the best one based on certain performance metrics. This selection is performed by so-called *TuningStrategies* that try to prune the search space of possible configurations according to a specific strategy. Figure 1 shows a high-level overview of the AutoPas library structure.

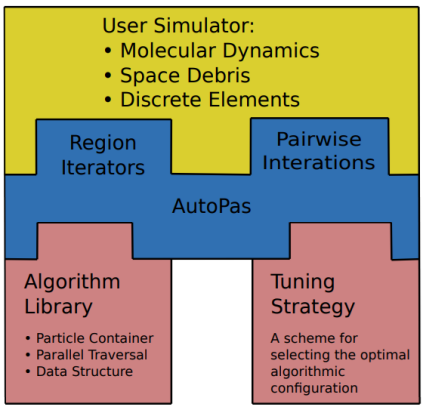

Fig. 1: AutoPas Library Structure as depicted by [3]

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*, *Traversal*, *Load Estimator*, *Data Layout*, *Newton 3*, and *Cell Size Factor*.

As implementations for a tunable parameter are (mostly) interchangeable, it is straightforward to create new, potentially hardware-specific, implementations for each key aspect of the simulation, allowing for both a bigger search space of possible configurations and 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].

The next sections will provide a detailed overview of prominent tunable parameters and the different implementations available in the AutoPas framework.

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, neighbor identification using just $O(N)$ distance calculations is possible [1], drastically reducing the computational complexity of the simulation. Figure 2 shows important container types used in AutoPas.

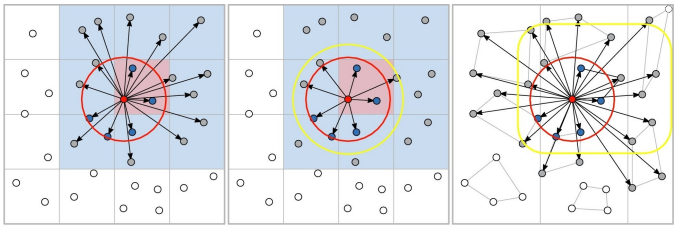

Fig. 2: Important Container Types as depited by [4]. The cutoff radius r_c is shown using a red circle. The arrows represent distance checks between particles. Only particles shown in blue contribute to the final force calculation.

LinkedCells

The LinkedCells 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].

VerletLists

The VerletList 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].

VerletClusterList

The Verlet Cluster Lists algorithm improves on the VerletList algorithm by grouping particles into clusters of size M ($M = 4$ in the figures). 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].

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. Figure 3 shows important traversal patterns used in AutoPas.

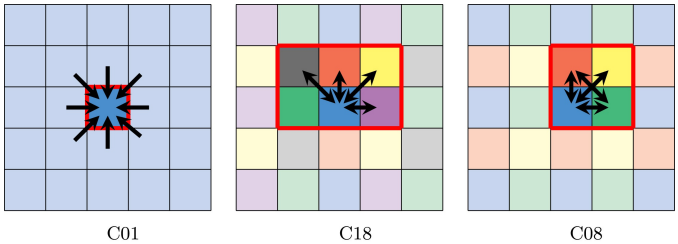

Fig. 3: Important Traversal Types as depicted by [5].

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 the force obtained by Newton 3 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].

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* 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

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 [1]. 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 [8] [9]. 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 enough configurations to make the tuning phase feasible, but not too many, to avoid missing the optimal Configuration. Tuning strategies balance the trade-off between potentially finding a better configuration and the cost of potentially encountering worse configurations during further exploration [3].

Add som tuning strategie

C. Static Optimization Techniques

AutoPas can perform basic tuning at compile time by using compile-time flags to conditionally compile certain features of the simulation engine, such as SIMD instructions, MPI support, or to enable auto-vectorization. Those flags primarily aim at ensuring that the simulation engine can utilize the full potential of the hardware it is run on.

1) Kokkos Integration: There are, 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 [10].

Kokkos is a performance-portable parallel programming model supporting diverse high-performance computing environments [11]. 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 Phis, or many-core CPUs [12].

The integration is still experimental and incapable of targeting systems other than conventional multi-core CPUs [10].

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 [12]. 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.

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 [13] [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 codebase 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

procedure CHECKEARLYSTOPPING(performance)
  $fastestTime \leftarrow \min(fastestTime, performance)$
  $slowdownFactor \leftarrow \frac{performance}{fastestTime}$
  if $slowdownFactor > maxAllowedSlowdown$ **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

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 boundaries[1].

All benchmarks are performed using 14 threads on a single node of the CoolMUC2[2] 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.

[1]A simulation video can be found at https://youtu.be/u7TE5KiSQ08

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

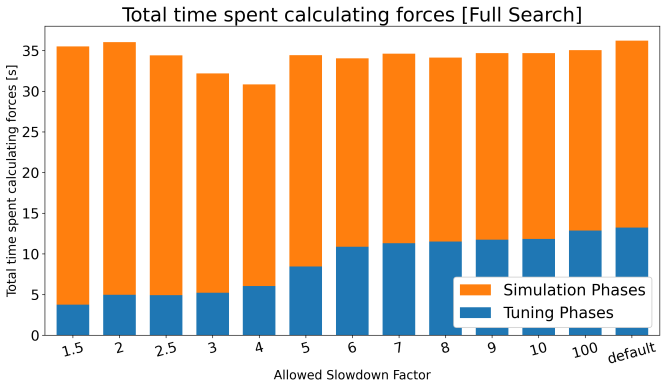

Fig. 4: Total Simulation Time for Exploding Liquid Simulation with Early Stopping divided into tuning and simulation phases. The total simulation time is minimal at a maximum allowed slowdown factor of ≈ 4.

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.

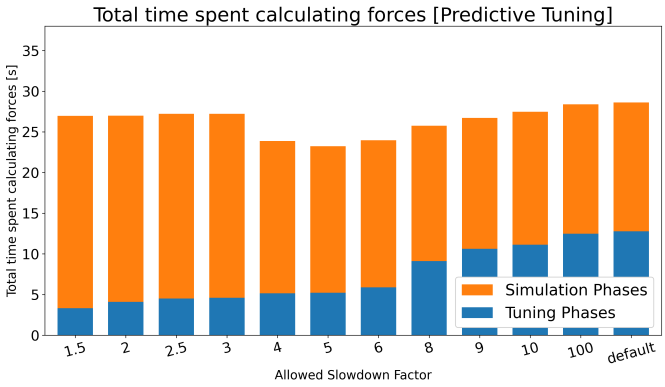

Fig. 5: Total Simulation Time for Exploding Liquid Simulation with Early Stopping divided into tuning and simulation phases. The total simulation time is minimal at a maximum allowed slowdown factor of ≈ 5.

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

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 $maxAllowedSlowdown \to 1$ and $maxAllowedSlowdown \to \infty$ are expected to perform poorly. $maxAllowedSlowdown \to 1$ 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, $maxAllowedSlowdown \to \infty$ 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 {*Particle Container*, *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 [1].

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 [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].

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 [16].

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 [17]. 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 [17].

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

clusion

VI. Conclusion

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] 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] L. Gärtner, "Integrating kokkos into autopas for hardware agnostic particle simulations," Master's thesis, Technical University of Munich, Feb 2022.

[11] 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

[12] LAMMPS, "Speeding up lammps with kokkos," *LAMMPS Documentation*, 2024. [Online]. Available: https://docs.lammps.org/Speed_kokkos.html

[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] 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

[17] 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