Computer Physics Communications 273 (2022) 108262

Contents lists available at ScienceDirect

Computer Physics Communications

www.elsevier.com/locate/cpc

N ways to simulate short-range particle systems: Automated algorithm selection with the node-level library AutoPas ☆,☆☆

Fabio Alexander Gratl [a,*], Steffen Seckler [a], Hans-Joachim Bungartz [a], Philipp Neumann [b]

[a] Chair for Scientific Computing in Computer Science, Technical University of Munich, Germany
[b] Chair for High Performance Computing, Helmut-Schmidt-Universität, Germany

ARTICLE INFO

Article history:
Received 18 November 2020
Received in revised form 20 November 2021
Accepted 12 December 2021
Available online 17 December 2021

Keywords:
HPC
N-body simulation
Molecular dynamics
Auto-tuning
Automatic algorithm selection
Dynamic tuning

ABSTRACT

AutoPas is an open-source C++ library delivering optimal node-level performance by providing the ideal algorithmic configuration for an arbitrary scenario in a given short-range particle simulation. It acts as a black-box container, internally implementing an extensive set of algorithms, parallelization strategies, and optimizations that are combined dynamically according to the state of the simulation via auto-tuning. This paper gives an overview of the high-level user perspective, as well as the internal view, covering the implemented techniques and features. The library is showcased by incorporating it into the codes LAMMPS and ls1 mardyn, and by investigating various applications. We further outline node-level shared-memory performance and scalability of our auto-tuning software which is comparable to LAMMPS.

Program summary

Program Title: AutoPas
CPC Library link to program files: https://doi.org/10.17632/9kdb2p76hv.1
Developer's repository link: https://github.com/AutoPas/AutoPas
Code Ocean capsule: https://codeocean.com/capsule/0391732
Licensing provisions: BSD 2-clause
Programming language: C++17, CMake 3.14
Nature of problem: The evaluation of the short-range pairwise interactions in an N-Body simulation can be achieved using many different algorithms and parallelization techniques. Depending on the nature of the scenario, its current state, and the forces of interest, the optimal algorithm configuration can differ greatly. Choosing this optimum is a non-trivial task even for experts. Furthermore, this optimum can change over the course of a simulation. Typically, a particle simulation software only implements one algorithm for force computation and is thus specialized for a certain type of simulation. It is up to the user to choose the program suitable for his needs.
Solution method: The AutoPas library implements a range of state of the art algorithms to find the relevant neighbors for the N-Body pairwise force calculation. It provides multiple shared-memory parallelization strategies using OpenMP and further algorithm optimization parameters that can all be set at run-time. A big burden for users persists in requiring the expert knowledge to pick the optimal solution procedure for a simulation. AutoPas removes this burden by tuning all aforementioned options automatically and dynamically. This way, simulation programs that make use of AutoPas give every domain scientist the possibility to make use of the most suitable algorithm configuration for arbitrary scenarios.

1. Introduction

N-body simulations are a set of important tools in various fields for many different applications. Examples are classical molecular dynamics used for the analysis of molecular behavior [1], smoothed particle hydrodynamics to approximate fluids for example in astrophysics [2], or biomedical applications [3] because they are easily usable even with complex geometries, or dissipative particle dynamics for complex fluids [4]. Although they are used for

☆ The review of this paper was arranged by Prof. Stephan Fritzsche.
☆☆ This paper and its associated computer program are available via the Computer Physics Communications homepage on ScienceDirect (http://www.sciencedirect.com/science/journal/00104655).
* Corresponding author.
E-mail address: f.gratl@tum.de (F.A. Gratl).

https://doi.org/10.1016/j.cpc.2021.108262
0010-4655/

completely different purposes, they all follow the same principle of simulating the behavior of particles defined by some form of interaction. These non-boned pairwise interactions are typically the computationally dominating part of particle simulations, and thus of particular interest for optimization. As will be discussed in Section 2.2, multiple different algorithms, e.g., Linked Cells or Verlet Lists, have been proposed to efficiently solve these interactions, however, no algorithm is optimal for every kind of scenario [5].

The open-source C++ node-level library AutoPas tackles this problem of generalization by providing a wide portfolio of algorithm combinations and optimizations. Domain scientist can not be expected to reliably know under which circumstances to select which algorithm. Therefore, AutoPas employs dynamic auto-tuning to determine the optimal algorithmic configuration at run-time. It can even adapt to evolving scenarios by autonomous on-the-fly adjustments of data structures and algorithms during simulation run-time.

This paper starts with a quick recap about short-range particle simulations, classical neighbor identification algorithms, and the algorithm problem in Section 2. Section 3 then discusses mature simulation packages that employ the previously discussed algorithms. Next, in Section 4, an extensive overview of the AutoPas library is presented from the out to the inside. First, from the perspective of a user, describing the high-level interface and how to use it in a simulation loop. After this, the implemented features like algorithms, data structures, and parallelization strategies are discussed. Subsequently, the perspective of a developer is presented in remarks on the internal software structure. In Section 5 the integration of AutoPas into different simulation codes is discussed and its behavior and performance in multiple simulations shown. Finally, conclusions are drawn and plans for future work laid out.

2. Theoretical background

2.1. Short-range particle simulations

In particle simulations, particles are advanced over time steps and interact with each other via force terms. Using Verlet integration [6] for Newton's Laws of Motion, these forces are translated to velocities and movements. The force exerted on a particle is typically the sum of pairwise forces. This makes a particle simulation an N-Body problem which has a computational complexity of $O(N^2)$, with N representing the number of particles. Many force potentials, however, quickly converge against zero with increasing distance between the particles. For these, usually, a cutoff radius is introduced. Interactions of particles whose distance exceeds this cutoff radius are not computed and assumed to be zero. This allows one to reduce the complexity to $O(N)$ without introducing a significant error. Such potentials are called short-range potentials [7]. There also exist techniques to compute long-range potentials efficiently [8]. However, this is not yet supported in AutoPas natively and is hence not subject to the following discussions.

In the case of molecular dynamics simulations, most commonly, the Lennard-Jones 12-6 potential is used:

$$U(r_{ij}) = 4\epsilon \left(\left(\frac{\sigma}{r_{ij}} \right)^{12} - \left(\frac{\sigma}{r_{ij}} \right)^{6} \right) \tag{1}$$

Here, r_{ij} represents the distance between two particles i and j. ϵ and σ are parameters that model particle properties and determine the strength and zero-crossing of the potential. The Lennard-Jones 12-6 potential has an attractive part, which models Van-der Waals forces, as well as a repulsive part that represents the Pauli repulsion [9].

To parallelize particle simulations on distributed systems like HPC clusters, the domain is split up and every node only stores a sub-domain. At the border between two such sub-domains, data needs to be duplicated to allow for the correct and complete calculation of forces. In practice, this means that all particles at a certain distance to the boundary of one sub-domain also need to be available in its neighboring domain. These cloned particles are called *halo particles*. During the force calculation, they are used as interaction partners like any other particle. However, in classical domain decomposition settings, no forces are computed for them, and they are updated from the sub-domain they are from.

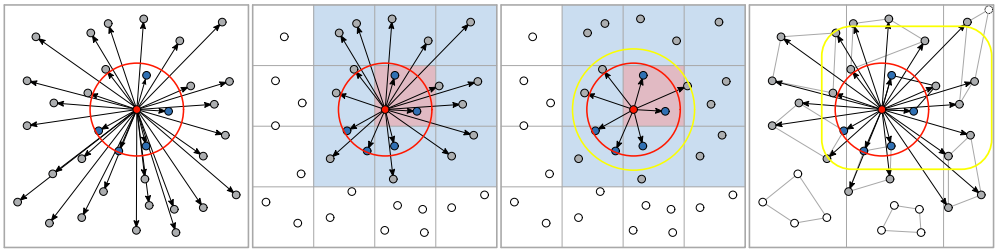
Although this paper is mainly concerned with molecular dynamics, these concepts apply for all kinds of particle simulations that contain short-range interactions like for example smooth particle hydrodynamics or dissipative particle dynamics.

2.2. Neighbor identification algorithms

At the heart of every particle dynamics simulation lies the algorithm for the identification of the neighboring particles to efficiently compute the short-range pairwise forces. The goal is for every particle to find all particles that are within the cutoff radius and therefore need to be considered for the force calculation of this particle. This choice of the algorithm has direct consequences for the data structures and layouts of how particles are stored. In the following, the four prototypes of these algorithms used in AutoPas are briefly presented.

Direct Sum The naive approach is to calculate the distances to all particles. This is depicted in Fig. 1a. Here, the force exerted on the red particle is calculated. For this all pairwise distances are calculated, visualized through the arrows. Particles that are outside the cutoff radius, depicted by the red circle are too far away and can be omitted from the force calculation. Since the distances need to be calculated for all particle pairs, this algorithm scales with $O(N^2)$ where N is the number of particles in the system. It is therefore only suitable for very small particle numbers. However, its main advantage lies in its simplicity, and the method does not invoke any additional memory or algorithmic/computational overheads.

Linked Cells To improve the scalability of pairwise computations the Linked Cells algorithm, also known as cell list [10], algorithm structures the spatial information of particles. The domain is subdivided into a Cartesian grid of cells with a mesh size bigger than or equal to the cutoff radius. Particles are sorted into these cells as can be seen in Fig. 1b. To identify the neighbors of the red particle, the Linked Cells algorithm looks at this particles' cell (red cell) and adjacent cells (blue cells) and calculates the pairwise distances to all contained particles, again depicted through arrows. This technique reduces the time complexity down to $O(N)$ [8] in case of homogeneous particle distributions. Another advantage is that the memory overhead for this algorithm only consists of maintaining a cell structure and is therefore independent of the actual number of particles. Since particles that are processed in sequence are also in sequence in memory, cache prefetching is possible and vectorization rather easy. The main disadvantages of Linked Cells are the overhead to keep moving particles sorted into their correct cells, as well as the number of superfluous distance calculations due to the rather poor approximation of the spherical cutoff region by the Cartesian cells as discussed and demonstrated, among others, in [11]. When using Linked Cells, AutoPas always uses a regular grid as described above. However, it shall be remarked that adaptive grid structures are conceivable [12], trading



(a) Direct Sum (b) Linked Cells (c) Verlet Lists (d) Verlet Cluster Lists

Fig. 1. Neighbor identification algorithms in short-range force calculation. The red circle symbolizes the cutoff radius. Particle colors indicate interaction intensity:
Red: Particle for which interactions are calculated.
Blue: Particles inside the red-colored particle's cutoff radius.
Gray: Potentially in range of the red particle, therefore the distance needs to be calculated. Arrows indicate when this is needed in every time step.
White: particles that are out of range and which are, thus, omitted.
(For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

more complex cell traversal schemes for a better approximation of the cutoff sphere, or accounting for inhomogeneous particle distributions in the domain.

Verlet Lists A completely different approach is not to rely on spatial information, but keeping track of a particle's interaction partners. For each particle, so-called Verlet or neighbor lists are generated that contain references to all particles within the cutoff range. [6] Since the generation of those lists is expensive it is desirable to reuse them in as many successive iterations as possible. Therefore, also particles that are slightly outside of the cutoff radius are included. This extension of the radius is called the Verlet skin and is visualized in Fig. 1c by the yellow circle. The combined length of the cutoff radius and Verlet skin will be referred to as interaction length. With this algorithm, in every iteration step, it is only necessary to compute the distances to the particles within the yellow circle. These are significantly fewer evaluations than evaluating all pairs in all neighboring cells in the case of the Linked Cells algorithm. This can directly be seen when comparing the number of arrows in Fig. 1b and Fig. 1c. A more formal comparison is drawn in [11] and [13]. The crucial point of this algorithm is the generation of the neighbor lists. Without further data structures one would have to fall back to the Direct Sum pattern of calculating all pairwise distances, which would make the Verlet Lists algorithm unusable for large numbers of particles. Therefore, one possibility is to build the Verlet Lists algorithm on top of Linked Cells. Linked Cells are used only for the periodic generation of the Verlet Lists, while for the actual force computation the lists are used. This means that for the generation of the neighbor list of the red particle in Fig. 1c only the distances to all particles in the blue and red cells need to be calculated. The first considerable disadvantage of this algorithm is its sizable memory footprint since for every particle a list of particle references needs to be saved. The second one is given by the lack of data locality. The spatial information in the lists is not sufficient to infer whether two particles are close in memory. Thus, distance and force calculations of successive particles can lead to loads from arbitrary memory locations resulting in bad cache behavior and difficulties for efficient vectorization.

Verlet Cluster Lists Building upon Verlet Lists, Páll and Hess developed an algorithm that mitigates some of the aforementioned drawbacks [14]. The algorithm is based on the observation, that neighboring particles will have very similar Verlet Lists. Therefore, a given number M of particles are combined into a cluster. Such clusters of, e.g., size four can be seen in Fig. 1d. This optimization does not only reduce the number of lists by a factor of $\frac{1}{M}$, but it also decreases their size, since entire clusters are tracked instead of individual particles. An additional and significant advantage is that when M is chosen as a multiple of a processor's vector length efficient vectorization becomes possible since the clusters inherently encode spatial information.

To construct these clusters the 3D domain is subdivided into a 2D grid along the x/y plane. The grid cell length is chosen as

$$\sqrt[3]{\frac{M}{N/V}} \tag{2}$$

where M is the aforementioned size of the clusters, N the total number of particles and V the volume of the domain. This effectively leads to a subdivision of the domain into towers as shown in Fig. 1d. Within each tower, clusters are formed according to the order of the particles along the z-axis.

Drawbacks of the Verlet Cluster Lists algorithm include the significantly more complex implementation and, compared to Verlet Lists, a higher number of distance calculations. The latter comes from the fact, that since only entire clusters interact, the search radius for potential interaction partners of one particle is not the cutoff radius with Verlet skin around the particle, but around the whole cluster. This difference can be seen in Fig. 1d where the yellow box depicts this radius. It is no longer a circle since it covers the area within the cutoff and Verlet skin radius around the bounding box of the cluster.

2.3. Algorithm selection problem

Although the algorithms presented in Section 2.2 appear incremental, each of them comes with a trade-off like higher memory consumption, ease of vectorization, or overhead through a complicated data structure. It is fair to say, that there is no silver bullet that will outperform all other approaches in every simulation scenario [15] [5] [16]. For this reason, AutoPas employs automated algorithm selection to independently select the optimal configuration for a given scenario.

The automated algorithm problem is the task to choose an algorithm for a given problem instance that maximizes an arbitrary performance metric [17]. Such metrics can consider precision, execution speed, memory consumption, etc. In AutoPas, currently, considerations are restricted to the time-to-solution.

Depending on whether the selection of the algorithm only happens once, or is reevaluated repeatedly during run-time, a distinction is made between static and dynamic algorithm selection. Since the scenarios in particle simulations can change drastically over the course of a simulation, AutoPas employs dynamic tuning in periodic intervals.

3. Related work

3.1. Particle simulations

Of course, there already exist particle simulators based on algorithms explained in Section 2.2, such as ls1 mardyn (Linked Cells-based) [18], LAMMPS (Verlet Lists-based) [19], and GROMACS (Verlet Cluster Lists-based) [20]. In the following, prominent examples shall be presented. Contrary to AutoPas, all of them use a fixed algorithm and are thus optimized for specific types of simulations.

3.1.1. ls1 mardyn

The open-source molecular dynamics simulation program ls1 mardyn[1] is actively used in academic research in the fields of thermodynamics and process engineering for many years [21] [18]. Written in C++ it features an MPI + OpenMP hybrid parallelization capable of scaling to thousands of compute nodes, and load balancing for highly heterogeneous particle distributions [22], or for use on heterogeneous hardware [23]. The code emphasizes memory efficiency and is thus based on the Linked Cells algorithm. This approach enabled the world's largest particle simulation [24] with more than $2 \cdot 10^{13}$ molecules. Its hand-crafted intrinsics implementation of the Lennard-Jones potential has proven very efficient especially when evaluating small molecules with multiple sites, like water in the TIP4P model [24].

3.1.2. LAMMPS

LAMMPS is a software package that focuses on materials modeling.[2] It uses the Verlet Lists approach and is built on the SoA data layout (see Section 4.2.1 for particles. A specific feature of LAMMPS is that almost all of its functionality is implemented in add-ons, called fixes, which makes the code very flexible to use. Since pointers to the aforementioned arrays are accessible from anywhere, it is easy to use them when developing new plugins. LAMMPS features an MPI parallelization based on domain decomposition, as well as different add-ons for shared-memory parallelization like OpenMP [19].

3.1.3. GROMACS

The Groningen Machine for Chemical Simulations (GROMACS) is another widely used software package with a strong focus on biological macromolecules like lipids and proteins[3] that are typically equilibrated in a box of water. While the simulation of macromolecules and thus long-range forces brings a whole new set of problems, the developers acknowledge that the short-range forces dominate their simulation time and are hence of most interest for optimization. To achieve high performance during the calculation of these non-bonded interactions, like the Lennard-Jones potential, it uses the Verlet Cluster Lists algorithm, with all data stored in SoA format, on top of highly optimized kernels written in intrinsic functions. GROMACS gained a significant speedup through the use of GPUs with CUDA. Their approach is to maximize concurrent usage of both CPU and GPU while giving each resource tasks for which they are optimized. As an example, while the GPU processes the neighbor lists for the short-range interactions of the local rank, the CPU already creates the neighbor lists for the non-local interactions [20].

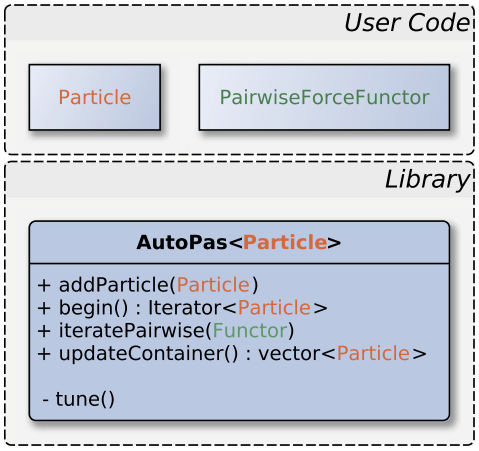


Fig. 2. Overview of the most important elements of the AutoPas interface and their relation to user code.

3.2. Algorithm selection in simulations: CoPA Cabana

As part of the Co-Design Center for Particle Applications (CoPA)[4] within the Exascale Computing Project, the Cabana toolkit was created.[5] Similar to AutoPas, it implements data structures like SoA or AoSoA and algorithms for particle simulations. The focus for pairwise iterations is set on the Verlet Lists approach, but also a Linked Cells interface is available. Cabana is intended to provide platform-independent high performance and is therefore built on top of the Kokkos C++ performance protability programming eco-system.[6] Since it is only a toolkit Cabana is not a simulator by itself just like AutoPas. There exist multiple proxy applications[7] to demonstrate how Cabana can be used to build simulations.

A decisive difference to AutoPas is that Cabana is a toolkit, meaning it provides a set of tools (algorithms and data structures) for a user to build a simulation from scratch and make all design decisions by himself. The combination of these tools then has to be decided statically at compile time via template arguments. AutoPas hides these actual tools behind its interface so that the user can focus directly on the simulator features while the library chooses suitable algorithms and data structures dynamically depending on the scenario and is capable of reevaluating these decisions during the simulation.

4. Implementation

4.1. User perspective

To use AutoPas, the user has to define two classes. First, a custom particle class is necessary that contains all properties and information that an individual particle shall possess. In order to facilitate compatibility, the new particle class should be derived from the base particle `autopas::ParticleBase` provided by AutoPas.

The second class a user has to implement contains the pairwise force calculation and is called *functor* throughout the following. Again, a base class `autopas::Functor` is available in AutoPas. This separation of user code and library code is sketched in Fig. 2.

[1] https://www.ls1-mardyn.de.
[2] https://lammps.sandia.gov.
[3] https://www.gromacs.org.
[4] https://exascale.llnl.gov/ecp-codesign.
[5] https://github.com/ECP-copa/Cabana.
[6] https://github.com/kokkos.
[7] https://github.com/ECP-copa/Cabana/wiki/Proxy-Apps.

```
1  AutoPas<ParticleType> autopas;
2  // set AutoPas options e.g. domain size:
3  autopas.setBoxMax({10,11,12});
4  autopas.init();
5
6  // fill scenario with particles
7  ParticleType p;
8  autopas.addParticle(p);
9
10 ForceFunctor functor;
11 // simulation loop
12 for(int i = 0; i < maxIterations; ++i) {
13   calculatePositions();
14   auto [leavingParticles, updated] = autopas.updateContainer();
15   if (updated) {
16       boundaryConditions(leavingParticles);
17   }
18   exchangeHalos();
19   autopas.iteratePairwise(&functor);
20   calculateVelocities();
21 }
```

Fig. 3. Simulation example, illustrating the use of AutoPas.

Some simulations might require multiple kinds of particles, e.g. two different types of atoms. One way to model this is by a type-ID flag in the particle class, so that all particles are of the same C++-type but can be distinguished for the force calculation in the functor. A possible implementation of this is provided through md-flexible, a MD simulation code that is shipped with the library.

An example of a simple simulation loop using AutoPas is shown in Fig. 3. Here, it can be seen that the AutoPas object is the single point of interaction with the library for users, and that it is intended to be used as a black-box particle container. At the start of the simulation program, an AutoPas object is created and configured. For almost all options, default values are provided and can be used. Users should only need to deviate from them if assumptions about the scenario can be made, which allow for reasonable restrictions on the algorithm configurations. If the program uses MPI and relies on some form of domain decomposition across the ranks, one AutoPas object per MPI rank has to be created. Details on the use of AutoPas in an MPI parallel program can be found in [22].

Directly after configuring the AutoPas object, the method `init()` (see line 4 in Fig. 3) is called, which among other things initializes the internal particle container. Only by then, particles can be added.

Looking at the main simulation loop, three interactions with the AutoPas object can be observed. To calculate the new positions or velocities, users need to iterate over all particles and update their values. For this, AutoPas provides iterators with `autopas.begin()`. Furthermore, iterators can be instructed to only iterate over halo or non-halo particles, or to only cover a certain spatial region of the domain.

Another iteration scheme is given by traversing particles pairwise, e.g., for the force calculation. AutoPas provides the method `iteratePairwise()` for this purpose, which takes the functor as argument. Note, that for more complex simulations it is supported to have multiple functors and thus multiple calls to `iteratePairwise()`.

Since AutoPas uses a Verlet-like interface, see [22], updating the container, as well as boundary and halo particle handling become more complicated. Depending on the internal particle container, which is currently in use, updating a container can mean rebuilding lists, resorting particles into cells, rebuilding clusters, etc., but this shall be of no concern for the user. A Verlet-like interface in this context means, that the particle container, similar to Verlet Lists as described in Section 2.2, remains valid for multiple iterations before being rebuilt. The consequences of this are, that all containers need to have a mechanism similar to the Verlet skin to account for particle drift over relevant boundaries and that particles can only be added or removed when the container is rebuilt. The second (boolean) return value of the function `updateContainer` signals, whether a rebuild actually took place. In contrast to reconfiguring the containers every few time steps only, halo particles have to be exchanged in every time step for parallel consistency.

4.2. List of features

4.2.1. Data layouts

The data layout defines how particle data are arranged in memory. There are two major ways to store structured data, which are both supported by AutoPas. See [11] for a more extensive overview.

Array of Structures (AoS) In this layout, particles are represented as a C++ object encapsulating properties like x/y/z positions, forces, etc. These objects are then stored subsequently in a `std::vector<Particle>`.

Structure of Arrays (SoA) Here, for each property like the x-position, a `std::vector<double>` is used with one entry for each particle.

It is also possible to create SoAViews. These are not copies of the actual SoA, but only a reference to a start and endpoint within an actual SoA. This is similar to the concept of `std::string_view`[8] from C++17.

In the AoS layout retrieving a single particle to, e.g., send it to a different MPI rank is simpler since it is only one random access on a vector. The advantage of the SoA layout is that successive particle information can be fetched from memory in one load operation. This is crucial for an efficient vectorization of the pairwise force calculation. Currently, AutoPas stores all particles in AoS format. When the SoA layout is chosen in every iteration the data that is required by the functor is copied to SoAs before the force calculation and back to the AoS afterward. This procedure is a result of AutoPas being the brainchild of ls1 mardyn. In [25] the usage of dynamically loaded SoAs in ls1 mardyn has been demonstrated successfully.

4.2.2. Containers

In AutoPas, a container serves two purposes: (1) managing in what data structures particles are stored and (2) how neighboring particles are identified. The containers implement the algorithms described in Section 2.2 and variations of them.

Direct Sum As explained in Section 2.2, the algorithm Direct Sum does not require any special data structure. Hence, all regular particles are stored in a single vector. Since halo particles need to be deleted regularly or sometimes traversed separately, a second array exists, which only stores the halo particles.

Linked Cells This container consists of a vector of uniform cells. Each cell holds a vector with its actual particles. Through this data structure, particles which are close in space and hence in the same cell, also end up close in memory. This preservation of spatial information in the data structure allows for efficient iteration of subregions of the domain, as well as for efficient vectorization of computations on particles. A disadvantage of this container is the fact that it actually models the space and not particle relations.

[8] https://en.cppreference.com/w/cpp/string/basic_string_view.

This means that if there is a region in the domain without particles, any traversal for this container still has to check the empty cells in that region.

Verlet Lists As laid out in Section 2.2, this container stores its particles in an instance of Linked Cells to efficiently build the neighbor lists. For this reason, the Verlet Lists container in AutoPas stores its particles in an instance of a Linked Cells container. The neighbor lists themselves are implemented as vectors of pointers to particles in the linked cell data structure. They are collected in a map of particle pointers, with each pointer mapping to a neighbor list.

Processing particles in SoA mode is a bit more complicated because entries of the neighbor lists are not sorted in memory. Since the Verlet Lists container is built on top of a Linked Cells container, particles are loaded in one SoA data structure cell by cell. When processing all particles in one neighbor list, these particles are not necessarily in consecutive memory locations. Therefore their data is first gathered in buffers and then the vectorized operations are applied.

Var Verlet Lists This container is a generalization of the previously described Verlet Lists container, and thus also builds on top of a Linked Cells container. As mentioned above the information about any ordering of the particles in the neighbor lists is a decisive point in the Verlet Lists approach. Therefore, this container provides an interface to easily swap out the implementation of the actual neighbor lists and their generation. When initialized with a given neighbor list, it will then manage any calls that involve interactions with it, like the pairwise iteration or rebuilding the lists. Currently, only the as_built style is implemented, which is described in Section 4.2.4. In the future, it is conceivable to implement the neighbor list style of the regular Verlet Lists container for this abstraction.

Verlet Lists Cells A problem with the Verlet Lists approach is that all neighbor lists are stored without any information about the spatial locality of the particles they belong to. To improve upon this, the implemented Verlet Lists Cells approach does not save all lists in one container wide vector but associates them to the cell where the corresponding particle is stored. Since the neighbor lists are organized very similar to Linked Cells, mostly the same kind of traversals can be applied. The exception being traversals that rely on the c08 base step (explained in Fig. 4c). This base step requires partial processing of the neighbors of cells other than the base cell. Restricting the processing of a cell in such a way would currently be highly inefficient, as the neighbor lists do not contain the information in which cell a neighbor particle resides.

Verlet Cluster Lists Implementing the idea of Section 2.2, this container consists of a 2D grid of towers in the xy-plane. Within these towers, particles are stored in one vector, sorted along the z-dimension together with a SoABuffer. Clusters consist only of a pointer to the first particle, a SoAView on the said buffer, and a vector of pointers to neighbor clusters.

If the number of particles in a tower happens to be not divisible by the size of clusters M, the size of clusters, dummy particles are inserted to complete the last cluster. These dummy particles do not participate in the force calculation. In Fig. 1d the particle marked in white on the top right is a dummy particle, which only exists to fill a cluster and does not exist in the other containers.

4.2.3. Optimization options

Cell Size Factor Already several years ago, it was shown that it can be beneficial to set up a Linked Cells container with cells that are shorter than the interaction cutoff [26]. This is because it can reduce the number of unnecessary distance calculations resulting from the poor approximation of the sphere describing the cutoff by the blocks of cells. For domains with few particles, it can be efficient to increase the size of cells. The advantage lies in less scheduling overhead for space with few particles and more data per cell for vectorization. These findings are in agreement with the idea of dynamically sized cells made in [12].

Since containers like Verlet Lists are built on top of Linked Cells they are also affected by this optimization parameter.

Newton3 Newton's third law of motion states, that for every force there exists a force of equal magnitude in the opposing direction [27]. For particle simulations, this means that for the forces F between the particles i and j, it must hold that:

$$F_{ij} = -F_{ji} \tag{3}$$

Thus, it is possible to replace half of the evaluations of the formula computing the pairwise force with this equation. However, this means that both variables which store the accumulated force for the respective particle have to be updated directly when computing F_{ij}, which has severe implications on parallelization patterns, cf. Section 4.2.4. It should be noted that, for example, in Smooth Particle Hydrodynamics, pairwise potentials are used which are not symmetric [28] and hence cannot be optimized with Newton's third law.

4.2.4. Sequential and parallel traversals

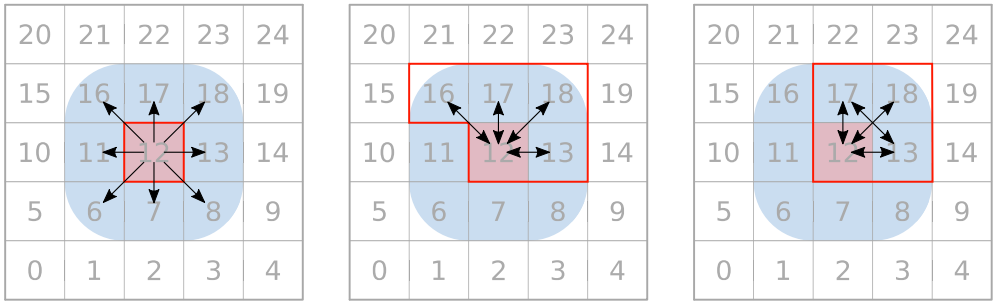
The traversal of the domain is inherently dependent on the container which is used to represent the domain. Therefore, in the following, traversals are grouped according to the containers that they have been designed for.

Traversal for Direct Sum

ds_sequential This traversal processes all node-local particle pairs sequentially in the order they were passed to the container.

Traversals for Linked Cells A shared memory parallelization of the traversal of the Linked Cells container typically works by distributing the cells over threads. As explained in Section 4.2.3, when Newton3 is disabled this becomes embarrassingly parallel. However, since using this optimization can save up to 50% of the force computations, it is desirable to make use of it. The resulting race conditions can be avoided by incorporating synchronization via domain coloring, or by locking mechanisms. All AutoPas shared memory parallelizations are implemented with OpenMP 4.5.

Technically every cell-based domain traversal can be divided into two parts: the traversal of the cells and the operations that are carried out on each cell. The latter will be called a base step in the following. During such a base step, all pairwise interactions of particles within the cell have to be performed, as well as those with the particles in neighboring cells. Currently, in AutoPas, three types of base steps are used which can be seen in Fig. 4. These steps were previously described in [29] and shall be explained here only briefly.



(a) c01 base step (b) c18 base step (c) c08 base step

Fig. 4. Base steps currently implemented in AutoPas. The red cell, also called the base cell, is the cell that is currently processed by the domain traversal. Its particles can interact with particles in the blue area, which is the bounding box of the cell extended by the cutoff radius of the force calculation. For the case which is shown here and in which the cell length is the same as the cutoff radius, all direct and diagonal neighbors potentially contain relevant particles.

c01 base step As seen in Fig. 4a, the c01 base step computes all interactions with all neighboring cells, abstaining from the use of the Newton3 optimization.

c18 base step Depicted in Fig. 4b, this variant makes use of Newton3 and only computes the interaction with the forward neighbors, meaning neighboring cells with a greater (lexicographically linearized) cell index. This increases the size of the box of cells that need to be guarded against race conditions, here shown in red, which reduces the degree of parallelism. On the other hand, the amount of cells that are needed for one base cell is decreased, which means it is more likely that all relevant particles can be kept in the lower level cache for the whole base step. In 3D space, the block is of size 3x2x2.

c08 base step Taking the idea of the c18 base step further the c08 base step replaces the computation of the forward diagonal interaction, here in Fig. 4c between cells 12 and 16, by the forward computation of the next cell, here 13 and 17. This decreases the local interaction area, which in return yields a higher degree of parallelism and cache reuse.

Now with these base steps, a number of different traversal patterns can be conceived:

lc_c01 This traversal applies the c01 base step on every cell within the domain. Since Newton3 cannot be applied it is embarrassingly parallel. The advantage is that this offers the maximal degree of parallelism since, given enough threads, all cells can be processed at the same time. This fine granular approach to parallelization without barriers also gives good load balancing opportunities, which is implemented with OpenMP's `dynamic` scheduling.

lc_c01_combined_SoA Here, the idea of the c01 traversal is picked up and extended with the special case in mind, that cells can be potentially smaller than the cutoff radius. Since it is inefficient to repeatedly traverse SoA buffers of many small cells, this traversal combines the SoAs of all cells around the base cell into one big SoA buffer similar to the sliding window mentioned in [12]. In Fig. 6a the blue cells within the red frame are in the red cell's sphere of influence so all the particles are added to the buffer. When the traversal progresses, in this case from cell 27 to 28 in Fig. 6b the particles from the yellow cells are out of range and can be discarded from the buffer while the green ones need to be added. For further details the interested reader is referred to the official documentation.[9]

lc_c18 When taking into account the Newton3 optimization the result of only having to compute interactions with half the number of particles is that every cell also only has to interact with half of its neighbors. This is implemented in the c18 base step described in Fig. 4b. Fig. 5a shows how this base step can be used in a coloring scheme in 2D. For all cells with the same color, the c18 base step can be computed in parallel. As an example, it can be seen that the green cell 38 only interacts with the cells 37, 43, 44 and 45, depicted by green arrows. This results in no overlap with any other green arrows from green cells. Using this pattern in 2D six colors are needed as seen in Fig. 5a, and 18 in 3D. For the 3D case, the same pattern needs to be stacked on the first twice, with new colors each, before the first color layer can be reused.

lc_c08 Similar to c18 the c08 pattern builds on an even smaller base step as described in Fig. 4c. The resulting domain coloring pattern is displayed in Fig. 5b. Thanks to the smaller size of the base step in 2D only four colors are needed resulting in a high degree of parallelism. Since the base step in 3D is a cube of 2x2x2 cells, the color pattern only has to be stacked once with new colors before the first set can be reused, resulting in eight colors. In general, the dependency of the number of colors C on the number of dimensions D can be described by

$$C = 2^D. \tag{4}$$

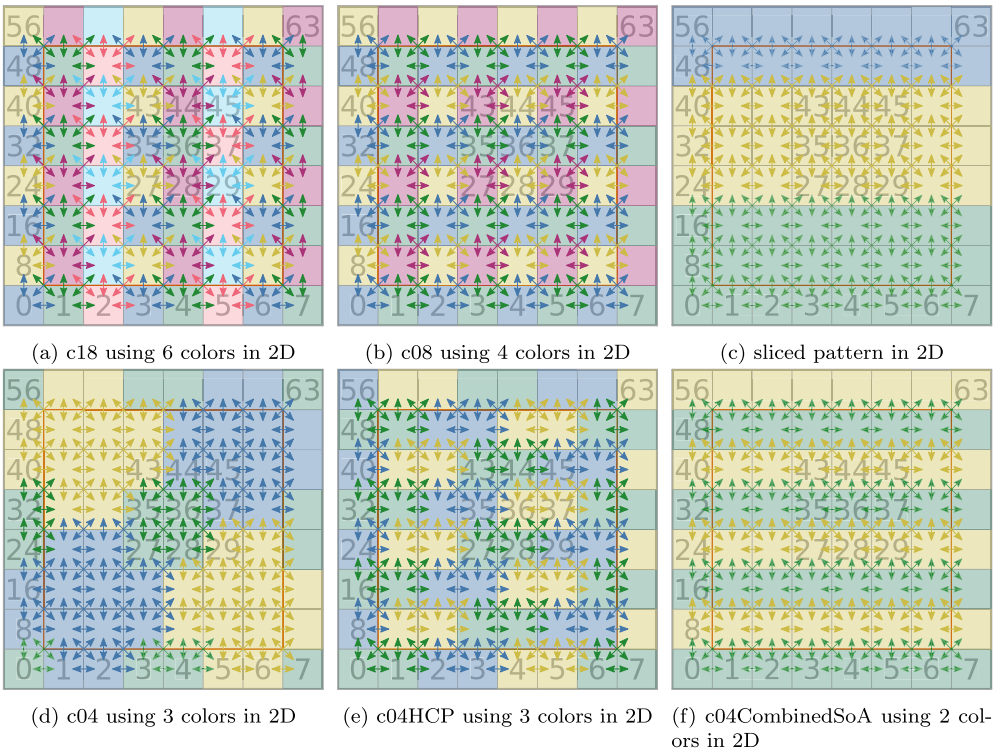
lc_c04_combined_SoA This traversal combines the ideas of the c01CombinedSoA traversal and the c08 base step. As seen in Fig. 5f the domain is colored in a stripe fashion, which reduces the degree of parallelism as each of these stripes is scheduled to one thread as a whole. This is used to combine the particles along the stripe in one sliding window like SoA buffer similar to the one described in lc_c01_combined_SoA.

lc_c04 In [24] a different approach was made, to reduce the number of color barriers to:

$$C = 1 + D \tag{5}$$

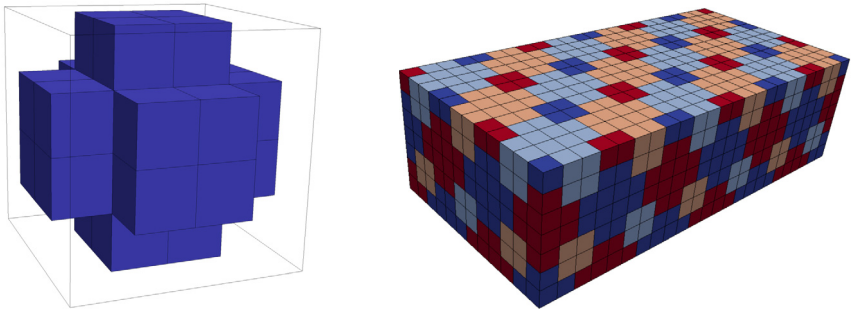
This was achieved by increasing the area a base step is applied to, which decreases the degree of parallelism. For the 2D case, this can be seen in Fig. 5d. All identically

[9] https://autopas.github.io/doxygen_documentation/git-master/classautopas_1_1LCC01Traversal.html.



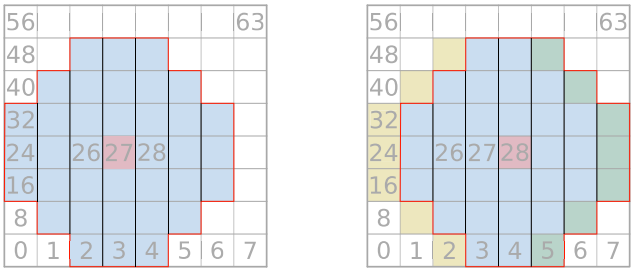
(a) c18 using 6 colors in 2D (b) c08 using 4 colors in 2D (c) sliced pattern in 2D

(d) c04 using 3 colors in 2D (e) c04HCP using 3 colors in 2D (f) c04CombinedSoA using 2 colors in 2D



(g) c04 patch in 3D Source [15] (h) c04 pattern in 3D Source [15]

Fig. 5. Coloring patterns.



(a) Base cell 27 (b) Base cell 28

Fig. 6. Progression of the ring buffer in the traversal lc_c01_combined_SoA.

colored patches can be processed in parallel, however, only one thread can work on a single patch to eliminate race conditions. The c08 base step is applied to every cell, as indicated by the arrows in Fig. 5d. In the 2D case, there are two types of color patches, here, green and yellow/blue, however, in the 3D case all patches are of equal size. The shape can be observed in Fig. 5g. These patches are then arranged in a staggered grid fashion and clipped at the end of the domain resulting in the 3D pattern seen in Fig. 5h.

lc_c04HCP Inspired by the hexagonal closest packing pattern this four-way coloring approach was first suggested in [15]. In contrast to c04, this pattern only uses one type of 1x2x3 block of base cells that is repeated in a staggered fashion as is displayed in Fig. 5e. Again, the c08 base step is used and the blocks within a color are scheduled dynamically. The smaller blocks lead to a higher degree of parallelism and more fine-grain scheduling.

lc_sliced In contrast to the dynamically scheduled coloring approaches described above, the sliced traversal reduces the scheduling work to a minimum by assigning all cells statically. As depicted in Fig. 5c, the domain is subdivided into slices of equal size as far as possible. The number of slices equals the number of available threads, but every slice has at least two layers of cells in the dimension in which the domain was sliced. Every thread starts with the first layer of cells in its slice and locks this section. The lock is released as soon as all cells in this plane are processed. Processing here means again that the c08 base step is applied. Cells are processed first in the directions perpendicular to the direction of the dimension that was sliced. This way the locks are released as soon as possible. When a thread reaches its last layer the base step

would access cells from the next slice. Therefore, the lock of this next slice needs to be acquired. However, since all threads release their locks as quickly as possible actual waiting is highly unlikely and should only occur in cases of high load imbalance between the slices. Despite having no load balancing mechanisms this traversal is highly interesting since it represents the absolute minimum of scheduling overhead and thus provides very good performance in homogeneous scenarios with sufficiently many cell layers [24].

For Verlet Lists

vl_list_iteration The pairwise traversal of the regular Verlet Lists container follows a similar fashion as the Direct Sum container. This means that without Newton3 the problem is embarrassingly parallel and the neighbor lists are dynamically distributed over the available threads for processing. Parallel processing with the Newton3 optimization is not supported for this container.

For Var Verlet Lists

vvl_as_built All traversals for this container depend on the structure of the neighbor lists and how they are built. The neighbor lists for this traversal are built in parallel using a c08 traversal on the underlying Linked Cells container. It is recorded, which thread creates which neighbor list during what color phase of the traversal. This information is then used to schedule the neighbor lists in the exact same way as they were built for the next pairwise iterations. This allows for optional optimization with Newton3, as well as a static load balancing.

For Verlet Lists Cells

vlc_c01 This traversal processes all neighbor lists stored in a cell. Since it does not support Newton3 all cells can be processed in parallel. As mentioned in the description of the container, the neighbor lists are created by applying a special functor with the chosen traversal to the cells. During the creation of the lists, for every particle pair P_i, P_j the distance $\overline{P_i P_j}$ has to be evaluated. If the distance is within the interaction length, the particles are considered neighbors. Both particles are appended to the neighbor list of the respective other particle and then, all neighbor lists of a cell are processed. This results in a cell-wise base step like c01 as described in Fig. 4a.

vlc_c18 Similar to the vlc_c01 traversal, all neighbor lists of every cell are processed. The important difference lies in the creation of the lists. For the particle pair P_i, P_j, where P_i is from cell i and P_j from cell j with $i < j$, only P_j is placed in the list of P_i but not the other way around, thus creating only a list of forward-facing neighbors. Processing all lists of a cell is then equivalent to a c18 base step. The same 18-way domain coloring is applied as for lc_c18 seen in Fig. 5a.

vlc_sliced This traversal works the same way as its counterpart for Linked Cells. Depending on whether the Newton3 optimization is used, lists are created to mimic the c18 or c01 base step.

For Verlet Cluster Lists

vcl_clusterIteration Disregarding the Newton3 optimization, this traversal schedules towers to threads. Each thread then processes all clusters in the given towers. Since the scheduling is done dynamically, some load balancing in the xy-plane is achieved.

vcl_c01Balanced Very similar to vcl_clusterIteration, Newton3 is disregarded resulting in an embarrassingly parallel traversal of clusters. However, in contrast to vcl_clusterIteration, here a static but load-balanced scheduling is chosen. A cluster's computational cost is estimated using the number of entries in its neighbor list. With this, threads are assigned a range of clusters for processing.

vcl_c06 In order to make use of Newton3 optimization, a domain coloring approach is used by this traversal. Depending on whether the Newton3 optimization is used, the neighbor lists are built either containing all or only forward neighbor clusters. The coloring is applied on the 2D grid of the towers and follows, in the case with Newton3 optimization, the c18 base step pattern, as shown in Fig. 4b. Since the coloring extends in two dimensions, only six colors are needed. If the Newton3 optimization is disabled, the base step corresponds to c01.

4.2.5. Auto-tuning strategy

Full Search Currently, AutoPas supports an exhaustive search approach. It is assumed, that consecutive iterations of the simulation are sufficiently similar to yield comparable times for the pairwise force calculation. Therefore, in successive iterations, different configurations are used and their execution times are measured. This way, no iteration is wasted or computed multiple times, and the tuning overhead is minimized.

During the measurements, significant fluctuations might occur, which might be induced by hardware, the operating system, or the number of particles. To prevent them from distorting the measurements, the same configuration can be measured over a given number of iterations. These obtained times then have to be reduced to a single value, called evidence, by means of e.g. average, mean, or absolute minimum.

The configuration with the lowest evidence is subsequently used for a given number of iterations. Afterward, the process of finding the optimum starts over again, allowing AutoPas to dynamically react to changing scenarios. This technique of tuning is described in [11].

4.2.6. Further options

Log Level AutoPas like many other software projects features a logging mechanism that is initialized when the first AutoPas object is instantiated. The interesting part is that when setting the log level to debug, information about the current, internally used algorithms and the autotuning is shown.

4.3. Remarks on software engineering

One of the first and most fundamental design decisions that have to be made when designing a molecular dynamics simulation is in what kind of data structure the particles are stored. This is important because it will, for example, affect how the neighbor identification algorithms can access other particles, or how a traversal can iterate over them. The decision thus affects the interactions of the most performance-critical parts of the code. In AutoPas the decision was made to build all components around the neighbor identification algorithm. Therefore, those are referred to as containers within this project, because for every algorithm, specific data layouts, like cell-wise spatially sorted data organization, one huge list, or sorted towers are required.

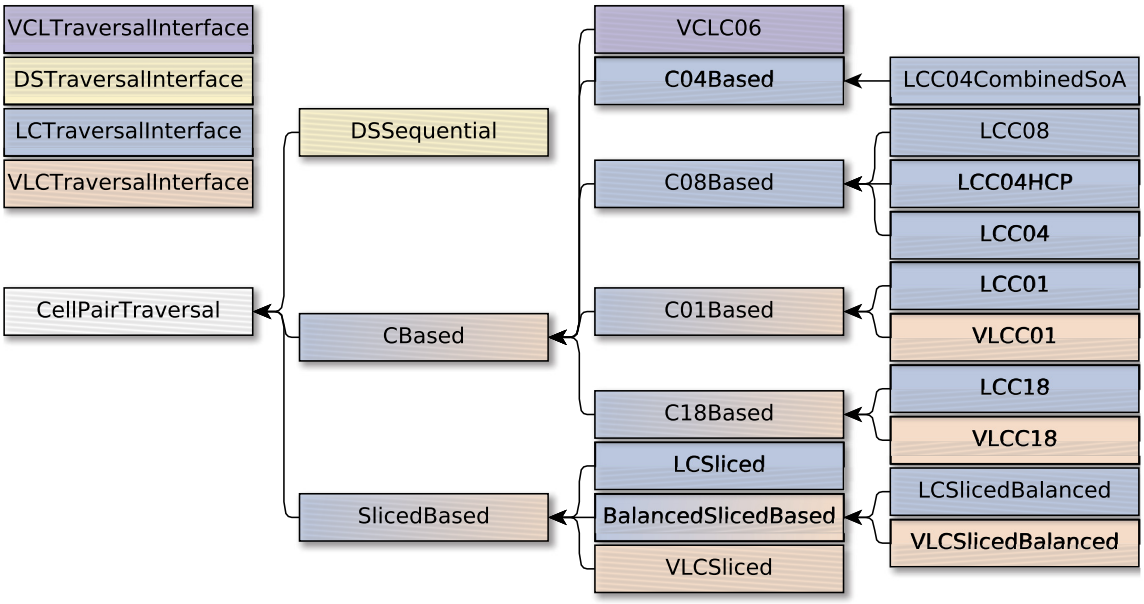


Fig. 7. Overview of hierarchical implementation of cell pair traversals. Colors indicate relations to particle containers. Boxes with a color transition hint that this abstract class is used in traversals for multiple containers. Only classes without any incoming relations, i.e., the ones on the right, are actual traversals. All actual traversals also inherit from the traversal interface with the matching color.

Especially traversals share a great part of functionality since all of them technically realize the same thing, but in different ways. AutoPas heavily leverages object-oriented coding, making use of (multiple-) inheritance, polymorphism, and abstractions combined with template meta-programming to avoid code duplication and encapsulating functionalities while still maintaining performance. In Fig. 7 inheritance relationship of all traversals for cell-based containers is shown. It can be seen, that all traversals are derived from two base classes: one that describes common functionality related to the container, and one that encapsulates traversal techniques. For example, going from left to right in Fig. 7, on a first level the distinction of the type of traversal is made, either color- or slice-based. In the color-based branch, the next step is to distinguish the base step types. The last level brings the distinction for the container by using multiple inheritances.

Another fundamental idea of AutoPas is to be usable as a black-box container. However, since the internal particle container, its traversals, data layouts etc. can potentially change in any iteration, the need arises for one common interface. The `AutoPas` Class provides this in a facade-pattern-like [30] way. Especially when starting a pairwise iteration several additional mechanisms like container validation and the auto-tuning need to be triggered. Since this shall not be left to the user this is also hidden behind the facade of the `AutoPas` class as sketched in Fig. 8. Here it is shown how the call to `iteratePairwise()` is propagated through the internal structure to the actual particle container. Along this way, mechanisms for container validity and the auto-tuning, marked in yellow and red respectively, are triggered.

To the outside, this facade pattern acts in a Verlet-Like fashion. This means, that the particle container is only updated at regular intervals. An update might mean for example rebuilding of neighbor lists, resorting of particles into cells, or rebuilding the cluster structure. For further information on this interface behavior see [22].

Internally, AutoPas also needs homogeneous ways to create algorithm elements like containers and traversals. For this, the factory pattern [30] is employed. An example of this can be seen in Fig. 8. Here the class `TraversalSelector` generates arbitrary traversal objects by being passed a `TraversalOption` enumerator from the current configuration.

A more complicated aspect is the creation and handling of the SoAs for two reasons. First, there should be as little need as possible for the user to manually interact with them since AutoPas is intended to be used by application scientists. Second, depending on the user-defined particle and functor, the SoA may have to map an arbitrary amount of data. This is solved by code generation through template meta-programming and sketched in Fig. 9. The user has to define the particle class and functor class(es) that shall be used. Within the particle, a few symbols need to be defined. `AttributesNames` is an enumeration type that should have an enumerator for each particle attribute that is used in the functor. `SoAArraysType` defines for each enumerator in `AttributesNames` what data type the corresponding field has. The utility functions `get()` and `set()` then strike the real bridge from the enumerators to the actual member fields of the particle. An exemplary implementation can be seen in Fig. 10, that uses templates. Next, the user has to specify which of those attributes have to be accessed by the functor. A distinction is made between input and output values. This is done by instructing the methods `getNeededAttr()` and `getComputedAttr()` to return an array of the aforementioned attribute enumerators. Now, AutoPas has all the information to generate code that moves the particle data to and from the SoA buffers for the force calculation, by using pack expansions of template parameters as illustrated in Fig. 11.

5. Example applications and results

5.1. Integration into simulation codes

5.1.1. md-flexible

To get a quick impression of AutoPas, the library is shipped together with a few example codes. md-flexible is a simple molecular dynamics simulation built around AutoPas. It simulates the Lennard-Jones 12-6 potential on single-site particles with Störmer-Verlet time integration. Features include a thermostat, periodic

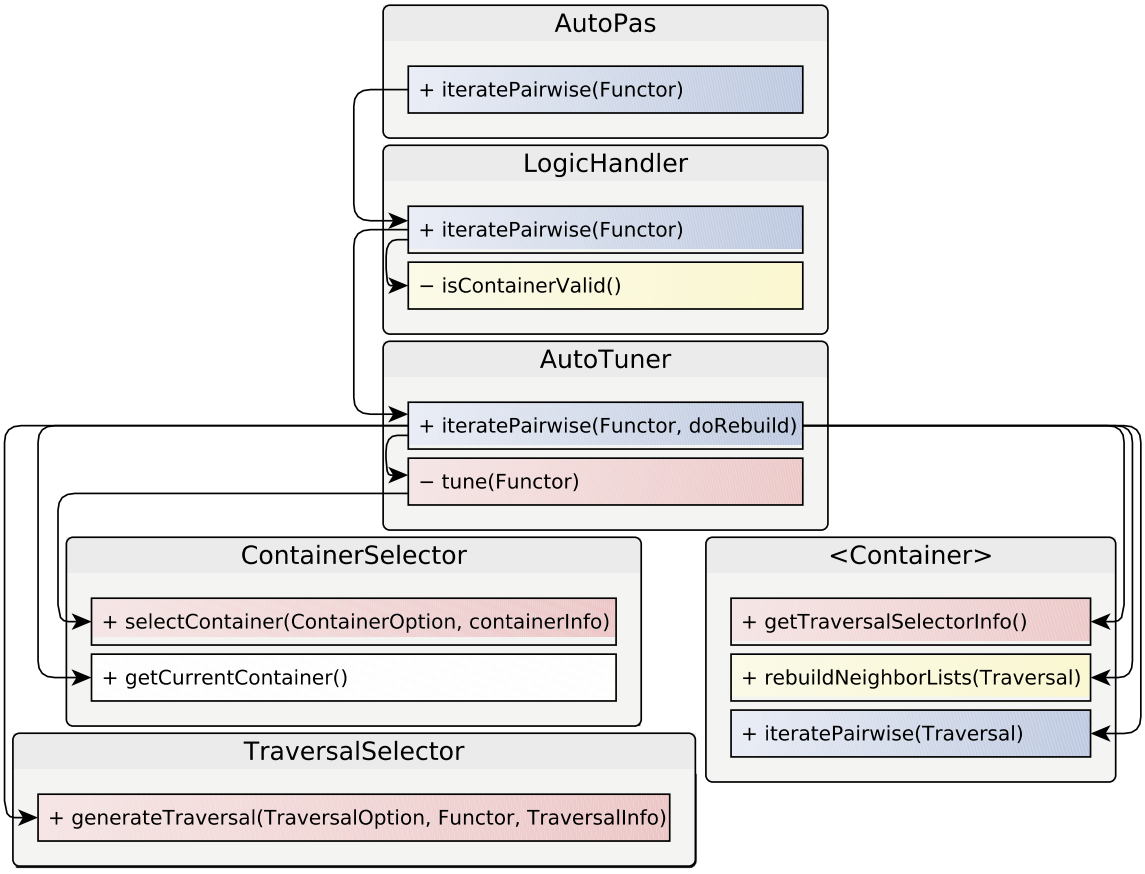


Fig. 8. Call trace of `iteratePairwise()` from the main interface to the actual particle container. Colors indicate to what functional group functions belong. Blue: pairwise iteration with the functor. Yellow: Container validity. Red: Auto-Tuning. White: Utility.

boundary conditions, multiple particle types, checkpointing, and VTK output for visualization. Its main purpose is the showcasing of AutoPas to first time users, but also for developers to provide an easy way to test AutoPas in an actual simulation code. The ease of exploring is achieved by exposing every `set()` function of AutoPas to the input of md-flexible and providing sane defaults for every option so that the user only has to configure what he is interested in. Together with generators for simulation scenarios, this paves the way for rapid prototyping, benchmark executions, and, generally, exploring AutoPas. More information on md-flexible can be found in the online documentation.[10]

5.1.2. ls1 mardyn

Integrating AutoPas into ls1 mardyn[11] was relatively straight forward since both are based on the concept of a central particle container. This container of ls1 mardyn was replaced with an AutoPas object. All calls interacting with the container had to be mapped to the interface of AutoPas. The parallel force calculation was mapped to `iteratePairwise()` to make use of the auto-tuning capabilities. The remaining accesses can be resolved through the AutoPas (region-)iterators. A particle class was implemented, which maps from the existing particle model of ls1 mardyn to the one required by AutoPas. The Lennard-Jones interactions are calculated with the `LJFunctor` that is shipped with the library. This, currently, limits functionality, since this example functor is not designed for multi-centered particles. It needs to be stressed, that this is not a limitation of AutoPas itself but only of the example functor which is considered user-code. Providing an alternative version that supports multi-centered particles is subject of future-work.

The biggest obstacle during the integration was the implementation of the MPI domain decomposition. At that time ls1 mardyn featured a simple domain decomposition (SDD) and a k-d tree-based decomposition (KDD). The first splits the domain into equally sized blocks, with one block for each MPI process. The second recursively divides the domain into blocks of equal load. This load is estimated by creating a model for the time needed per cell, thus restricting the granularity of the decomposition to the size of a linked cell. Since AutoPas is a black-box particle container, it cannot be assumed to contain cells, rendering the way the local load is estimated inapplicable. Also, the fact that a changing scenario might trigger the use of different algorithms, renders process-local performance predictions infeasible. This means that the implementation of the KDD in ls1 mardyn is not compatible with AutoPas. To still achieve load balancing for MPI, the approach of diffusive load balancing was implemented, using A Load balancing Library (ALL)[12] developed at the Jülich Supercomputing Center. Here, the domain is first divided similar to the SDD. Over time, the computational load of those blocks is aligned by shifting their borders, effectively extending or shrinking the part of the domain that a single MPI rank covers. This approach lends itself very good to AutoPas since the decomposition makes no assumptions on any cell or grid structure and the load balancing is purely based on mea-

[10] https://autopas.github.io/doxygen_documentation_md-flexible/git-master/.
[11] Git Version: f3b89b56ce70c899b4e07649f3b651643b963d4f.
[12] https://e-cam.readthedocs.io/en/latest/Meso-Multi-Scale-Modelling-Modules/index.html#all-a-load-balancing-library.

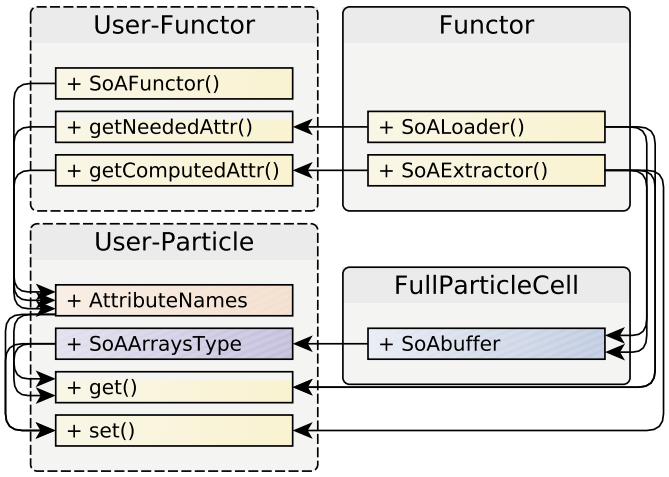


Fig. 9. Code generation of SoA utility functions with C++17 Templates.
Colors indicate the type of the symbol:

Orange	Enumeration Type
Purple	Type
Yellow	Function
Blue	Field

Arrows indicate that a symbol is used.
Gray boxes with a dashed border indicate user code while a solid border indicates library code. `User-Functor` is supposed to be a derived class of `Functor`. The user has to implement the shown methods, enumeration, and type for the particle class. As required by the interface, `get` and `set` map the enumerators of `AttributeNames` together with the corresponding types of `SoAArraysType` to the member fields of the particle class. Additionally, the user has to specify the `AttributeNames` of the Particle fields that are needed or computed by the functor. From all this information, AutoPas then generates the layout of the actual SoABuffers in the cells, as well as the `SoALoader()` and `SoAExtractor()` methods.

surements of execution times. For more information on AutoPas in ls1 mardyn see [22].

5.1.3. LAMMPS

AutoPas was successfully integrated into LAMMPS as an accelerator for node-level simulations. The source code of the integration is publicly available on GitHub.[13]

Similar to the integration into ls1 mardyn a particle class and a class acting as an interface for the AutoPas object had to be implemented. In contrast to ls1 mardyn the provided `LJFunctor`, could not be used directly, but needed to be extended to support functionalities like particle types that do not interact or calculating the non-diagonal entries of the virial tensor. As a proof of concept, some styles and fixes have been adapted to be compatible with AutoPas, like *nve*, *temp/rescale*, *aveforce*, and *indent*, but the remaining ones can be converted similarly. Integrating AutoPas into LAMMPS was slightly more challenging because LAMMPS does not follow the concept of a central particle container but directly works on the arrays from the SoA representation of the particles. The key rationale is to replace any access to LAMMPS' particle SoAs with an access to AutoPas. Due to this SoA representation, some interfaces, in particular, those for the (periodic-)boundary conditions, work with particle indices. AutoPas, on the other hand, does not provide random access to particles but follows an iterator pattern approach. To comply with the internal interfaces of LAMMPS, an index structure was set up, mapping an index to a particle pointer. These index structures need to be rebuilt whenever the AutoPas internal particle container is updated, changed, or particles are added.

[13] https://github.com/AutoPas/lammps-autopas.

5.2. Experiments

All experiments were executed on the CoolMUC2[14] system at Leibnitz Rechenzentrum (LRZ).[15] It consists of 812 dual CPU Haswell nodes with 14 cores per CPU running at 2.6 GHz and hyperthreading. Each node holds 64 GB of DDR4 RAM.

5.2.1. md-flexible: falling drop

The falling drop scenario is a canonical teaching example [8]. It simulates a sphere of particles, that is accelerated by a gravity potential, and drops into a bed of particles, as seen in Fig. 12. The top and bottom sides of the domain are restricted by reflecting boundary conditions, simulated by a wall of particles with infinite mass. All other sides are subject to periodic boundary conditions. The shock of the impact (12d) is reflected at the bottom of the domain and scatters back particles from the original drop, as well as from the bed (12e). After some time, the particles begin to equilibrate at the bottom of the domain as they are subject to gravity (12f).

The scenario shown here consists of about 18.000 particles and was performed on one node of CoolMUC2 only making use of AutoPas' internal OpenMP parallelization.

Fig. 13a shows the data gathered by the auto-tuner over the course of the whole simulation for each configuration. From the small confidence intervals, it can be seen that for this scenario the performance of each configuration remained rather constant. However, large time differences can be seen with the slowest configuration (lc_sliced) taking, on average, 13.7 times longer per iteration than the fastest (vvl_as_built). The bad performance of the configurations using the traversals lc_sliced, lc_c04, or vcl_c06 can be explained by their low grade of parallelism. When dividing the domain, the Linked Cells algorithm creates a grid of 17x11x14 cells, leaving space for only eight slices, since they need a minimal thickness of two cells for their locking logic. This means that only eight of the 28 threads of the node are used which explains why this configuration is more than seven times slower than most others. A similar argument can be made for lc_c04 and vcl_c06 since the blocks they schedule are rather large which limits their ability for load balancing.

In Fig. 13b the performance of the two best configurations for this scenario is shown over the course of the simulation. These are the only configurations that are picked by the auto-tuner consistently over multiple execution runs. Although the shown values are the minimum of three iterations, the measurement of the time for a single iteration is rather noisy, as can be seen by the very inconsistent course of the graphs. When creating smoothed curves from the data, the trend can be observed that at the start of the simulation, the configurations are more similar in performance, with vvl_as_built slightly improving over time. In the beginning, although the smoothed line for vvl_as_built remains below that for lc_c08, it cannot be stated that it is faster due to the significant overlap in the confidence intervals. At this time the auto-tuner sometimes picks one or the other, while later on it tends to pick vvl_as_built more, which is in agreement with the observations from the figure.

5.2.2. ls1 mardyn: Vapor Liquid mixture

A reoccurring case of an application for ls1 mardyn is the simulation of interfaces of molecule phases of different density [21,31]. The Vapor-Liquid mixture scenario shown in Fig. 14a is an example of this type of experiment. It is composed of about 55.000 particles distributed over two denser liquid phases of different sizes and an

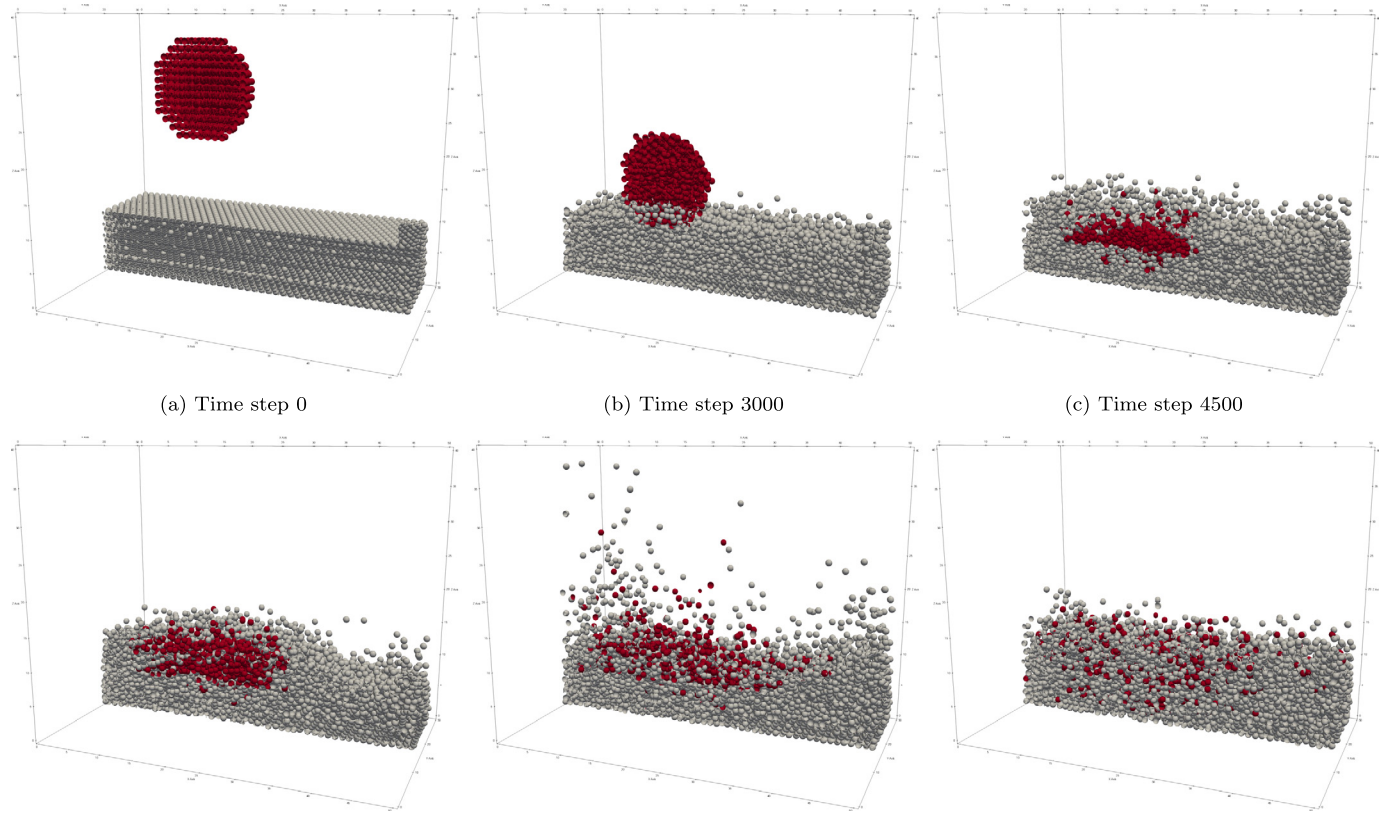
[14] https://doku.lrz.de/display/PUBLIC/CoolMUC-2.
[15] https://www.lrz.de/.

```
1  template <AttributeNames attribute>
2  constexpr typename std::tuple_element<attribute, SoAArraysType>::type::value_type get() const {
3      if constexpr (attribute == AttributeNames::id) {
4          return id;
5      } else if constexpr (attribute == AttributeNames::posX) {
6          return r[0];
7      } else // all other cases ...
8  }
```

Fig. 10. Potential implementation of the `get()` function that maps from an enumerator to member fields. Instead of a regular argument, the function receives a template (line 1). This is then used in a `constexpr if` construct (line 3 ff.) to determine which member field to return. Since the member fields may not have the same types, the return type is retrieved from the `SoAArraysType` tuple by accessing the N-th element, where N is the number specified through the `attribute` enumerator (line 2). Note that through the usage of templates, the compiler creates one instantiation for each attribute without any `if`-branching.

```
1  void SoALoader(ParticleCell<Particle> &cell, SoA<SoAArraysType> &soa) {
2      SoALoaderImpl(cell, soa, std::make_index_sequence<Impl_t::getNeededAttr().size()>{});
3  }
4
5  template <typename cell_t, std::size_t... I>
6  void SoALoaderImpl(cell_t &cell, SoA<SoAArraysType> &soa, std::index_sequence<I...>) {
7
8      auto const pointer = std::make_tuple(soa.template begin<Impl_t::getNeededAttr()[I]>()...);
9
10     auto cellIter = cell.begin();
11     for (size_t i = 0; cellIter.isValid(); ++cellIter, ++i) {
12         ((std::get<I>(pointer)[i] = cellIter->template get<Impl_t::getNeededAttr()[I]>()), ...);
13     }
14 }
```

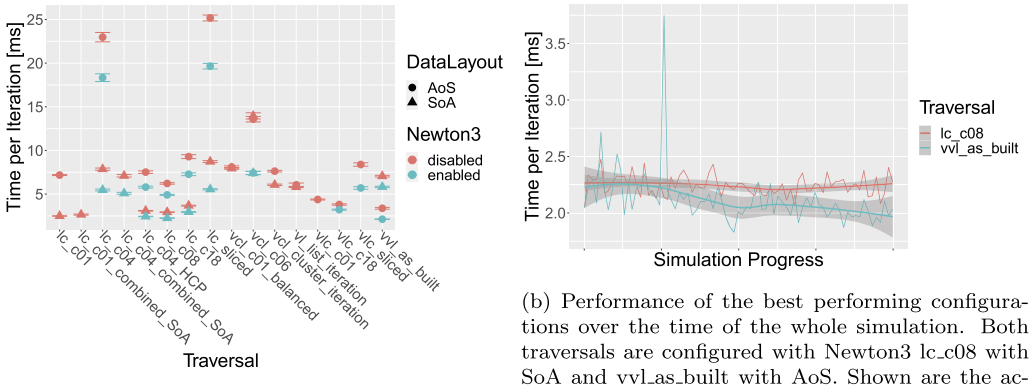
Fig. 11. Potential implementation for `SoALoader`. The implementation is split into an interface method (lines 1-3) and the actual implementation (line 5-14). This is necessary to create a packed index sequence that covers all attributes that need to be loaded (line 2). Each index is used to create a pointer to an array within the SoA (line 8). These pointers are created in a tuple by expanding the packed sequence of indices and using them as indices for the needed attributes (line 8). The for loop (line 11) iterates over all particles in a given cell and stores their data in consecutive positions in the SoA. To execute this for all relevant fields of the particle, a fold expression over the index sequence is applied with the comma operator (line 12). I is an index sequence that acts as an array index for the needed attributes. This is effectively an iteration over all needed attributes.



(a) Time step 0 (b) Time step 3000 (c) Time step 4500

(d) Time step 5500 (e) Time step 9500 (f) Time step 35000

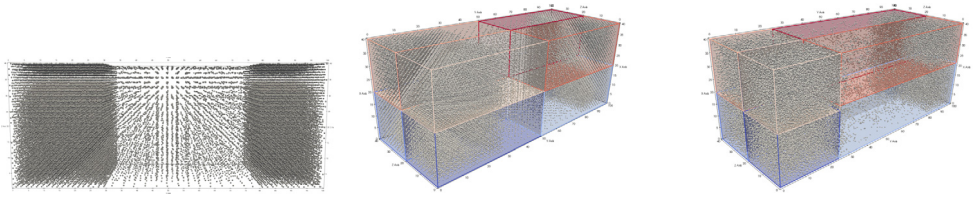
Fig. 12. Cross-section of the falling drop scenario colored by particle type. This highlights the distribution of the drop material (red) after the impact. Not shown are the particle walls for the reflecting boundaries on the top and bottom side of the domain.



(a) Overall mean times and 95% confidence intervals of all configurations.

(b) Performance of the best performing configurations over the time of the whole simulation. Both traversals are configured with Newton3 lc_c08 with SoA and vvl_as_built with AoS. Shown are the actual measurements, as well as a smoothed trend line with the 95% confidence interval.

Fig. 13. Analysis of the performance of algorithm configurations in the falling drop simulation on CoolMUC2.



(a) t = 0 (b) t = 500 (c) t = 66000

Fig. 14. Vapor-Liquid Mixture: MPI parallel with diffusive load balancing. The color represents the number of the rank with dark blue (front bottom) as zero until dark red (back up) as seven.

enclosed sparse gaseous phase. The phases are at their respective boiling and melting densities and are thus in an equilibrium [32]. This scenario was computed on four nodes of CoolMUC2 with two MPI ranks per node and 14 OpenMP threads per rank. The resulting MPI decomposition of the domain can be observed in Fig. 14b and Fig. 14c. In Fig. 14b the initial regular decomposition can be seen which is not in balance, because of the asymmetry of the scenario. Over the course of the simulation, the diffusive load balancer shrinks the sub-domains of the ranks containing the denser liquid phase, which accounts for the large part of the total computational cost of the simulation. The decomposition converges to the state seen in Fig. 14c, with four small sub-domains containing the denser liquid phase and four containing the gas and the sparse liquid phase.

Since every MPI rank runs an independent instance of AutoPas, it is interesting to look at the individual tuning behaviors. AutoPas is allowed to try all algorithms available by default. Fig. 15a shows the subset of configurations that were actually picked on each rank and how often relatively. On all ranks, most of the time, Verlet Lists Cells with the c01 traversal is the preferred choice. Looking at the actually measured run-times per iteration in Fig. 15b, this seems to be a reasonable choice. The wide gray area around the smoothed curves, representing the 95% confidence interval, indicates the high amount of noise in these measurements. These come from the very short iteration times of less than 10 ms.

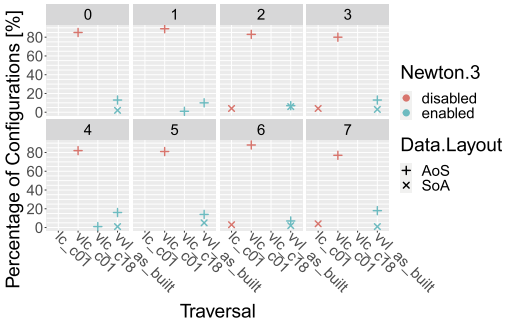
5.2.3. LAMMPS: Lennard-Jones melt

The *melt* or *Lennard-Jones liquid benchmark* was chosen since it is a benchmark specified by LAMMPS itself.[16] Here, it features a very densely packed block of about 8 million single-site particles with periodic boundary conditions.

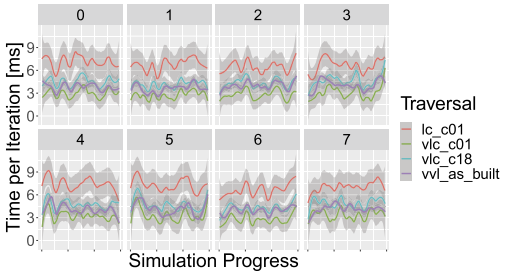
Fig. 16a displays the comparison of total wall time of the strong scaling on CoolMUC2. Shown is the performance of LAMMPS using different accelerator packages for the node-level parallelization: *KOKKOS*, *USER-OMP*, and *USER-AUTOPAS*. For AutoPas, two graphs are shown, one with auto-tuning over the default options enabled and another one fixed to a single configuration. This configuration was found to be optimal or near-optimal on all thread counts during the run with tuning enabled and represents the maximal possible performance using AutoPas in this scenario. When simulating more iterations without re-tuning the relative difference between the AutoPas lines will diminish, because the run-time is dominated by the iterations run with the same, optimal, configuration. Abstaining from re-tuning after an initial tuning phase is sensible in this scenario, since this scenario is not expected to change its physical structure, thus there is no reason for a change in the optimal configuration.

Although AutoPas starts with a single-thread performance, which is slower by a factor two without- respectively almost three with auto-tuning, it surpasses the other packages when using all resources of the full node due to better overall scaling. Without tuning, AutoPas reaches its peak performance with 56 threads while the peak of the OpenMP package is at 28 threads, which only corresponds to the full node without hyperthreading. Compared to the OpenMP package the best total execution time of AutoPas is 16% faster and 58% faster compared to the Kokkos package. The shortcomings of all packages can be narrowed down when looking at the breakdown timers in Fig. 16b and Fig. 16c. Here it is apparent that the low single-thread performance of AutoPas lies in the pairwise iteration. However, when utilizing the full node, AutoPas even has a marginally better performance than the other packages. Without auto-tuning, AutoPas, using 56 threads, is 15% faster than OpenMP and 35% faster than Kokkos, both reaching their peak for the pairwise force calculation at 28 threads. The timer *pair* covers the pairwise force calculation, as well as the rebuilding of the neighbor lists. Looking at Fig. 16c, the performance

[16] https://lammps.sandia.gov/bench.html#lj.

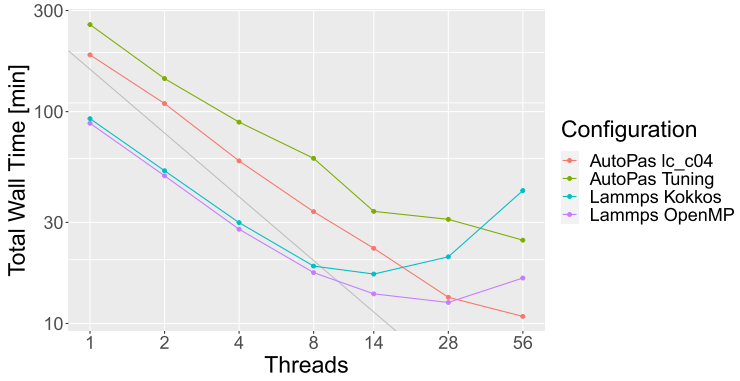


(a) Distribution of configurations picked per MPI Rank throughout the simulation.

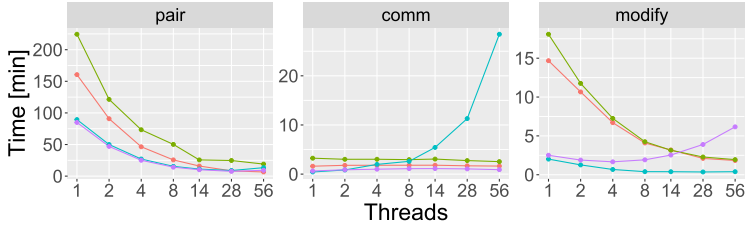


(b) Smoothed (LOESS) trend of time per iteration for best configurations measured per rank. The gray area shows the 95% confidence interval highlighting the high instability of measurements.

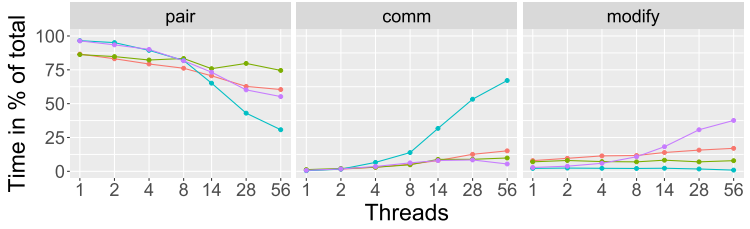
Fig. 15. Analysis of the configurations picked on each rank and their performance.



(a) Total wall time for 1000 time steps (log-log scale). The gray line represents linear scaling.



(b) Breakdown of absolute time the simulation spent in different parts.



(c) Relative portion of run-time in different parts of the simulation.

Fig. 16. Strong scaling of LAMMPS' Lennard-Jones Melt benchmark on a single node on CoolMUC2. Shown is a detailed comparison of four configurations:
1. (green) LAMMPS with integrated AutoPas using auto-tuning over default options.
2. (orange) LAMMPS with integrated AutoPas only using the optimal configuration.
3. (blue) Vanilla LAMMPS using the Kokkos accelerator.
4. (purple) Vanilla LAMMPS using the omp accelerator.
Using vectorized Linked Cells, AutoPas is able to outperform LAMMPS in this benchmark due to overall better scaling behavior.

decline of the *KOKKOS* package seems to be in the part of the simulation covered by the timer *comm*, as it is responsible for almost 70% of the total run-time. This timer observes the communication of forces and particles at the boundaries of ranks. Since this simulation was executed on one rank only, this degenerates to the application of the periodic boundary condition which is mainly copy operations. Looking at the timer *modify*, again it can be seen, that AutoPas has sub-optimal single-thread performance, but also the *USER-OMP* package seems to lose performance at high thread counts. This timer measures the time needed to update the particles, so the simple traversal of particles. A potential reason for this performance might be that the parallelization makes use of the

Newton3 optimization (see Section 4.2.3), by duplicating the data for each thread and then reducing it within critical sections. While this provides very good throughput for a low number of threads, scalability is limited.

6. Conclusion

The open-source node-level auto-tuning library AutoPas was presented in this paper. An overview was given, covering its algorithm portfolio including Linked Cells, multiple forms of Verlet Lists, and GROMACS-inspired Verlet Cluster Lists, as well as multiple parallelization strategies for each. The software design of the library was detailed, focusing on internal structurings, like data structures, containers, traversals and code generation aspects via *C++* templates.

AutoPas was integrated into mature simulation codes achieving reasonable performance and demonstrating algorithmic flexibility. With the integration into ls1 mardyn, an MPI parallel simulation was shown, featuring a mix of Linked Cells and Verlet Lists in combination with diffusive load balancing. Using the integration into LAMMPS, the close to linear scaling behavior of AutoPas was demonstrated, slightly outperforming other accelerator packages that are shipped with LAMMPS.

Thanks to automatic algorithm selection, AutoPas provides the optimal algorithmic configuration even for dynamically changing scenarios for arbitrary short-range N-Body simulations independent of the users' experience.

7. Future work

As a young project, AutoPas shows potential with regard to various research directions. Features under current development contain general performance improvements, as well as more efficient algorithms for auto-tuning. Here, the goal is to decrease the number of iterations spent testing inefficient configurations while still finding an "at-least near-optimal" one. This is especially important since regular contributions are extending the algorithm portfolio, leading to a larger search space. For the future, a deep integration of Kokkos will be investigated to provide solid GPU support and provide more platform-independent performance. In this work, the focus was put on the software design and exemplary simulation results, demonstrating AutoPas' capabilities after integration into mature simulation codes. A more comprehensive study of more benchmark scenarios would be highly desirable in the future.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

This project is a group effort that goes beyond the authors of the paper. Special thanks go to Nikola Tchipev for initial ideas, as well as all other contributors.[17] Financial support for this project was provided by the Federal Ministry of Education and Research, Germany, project "Task-based load balancing and auto-tuning in particle simulations" (TaLPas), grant numbers 01IH16008A and 01IH16008B. The authors acknowledge the provision of compute resources of CoolMUC2 provided by LRZ.

References

[1] J.D. Durrant, J.A. McCammon, BMC Biol. 9 (1) (2011) 1–9.
[2] V. Springel, Annu. Rev. Astron. Astrophys. 48 (2010) 391–430.
[3] M. Toma, Significances Bioeng. Biosci. 1 (1) (2017) 1–4.
[4] P. Espanol, P.B. Warren, J. Chem. Phys. 146 (15) (2017) 150901.
[5] C. Fraga Filho, L. Schuina, B. Porto, Arch. Comput. Methods Eng. (2019) 1–15.
[6] L. Verlet, Phys. Rev. 159 (1) (1967) 98.
[7] R.E. Mickens, Found. Phys. 9 (3–4) (1979) 261–269.
[8] M. Griebel, S. Knapek, G. Zumbusch, Numerical Simulation in Molecular Dynamics: Numerics, Algorithms, Parallelization, Applications, vol. 5, Springer Science & Business Media, 2007.
[9] J.E. Lennard-Jones, in: Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences, vol. 106, The Royal Society, 1924, pp. 463–477.
[10] D.C. Rapaport, R.L. Blumberg, S.R. McKay, W. Christian, Comput. Phys. 10 (5) (1996) 456.
[11] F.A. Gratl, S. Seckler, N. Tchipev, H.-J. Bungartz, P. Neumann, in: 2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), IEEE, 2019, pp. 748–757.
[12] W. Eckhardt, Efficient hpc implementations for large-scale molecular simulation in process engineering, Dissertation, Institut für Informatik, Technische Universität München, München, Verlag Dr. Hut, ISBN 978-3-8439-1746-9, Jun. 2014.
[13] P. Gonnet, J. Comput. Chem. 28 (2) (2007) 570–573.
[14] S. Páll, B. Hess, Comput. Phys. Commun. 184 (12) (2013) 2641–2650.
[15] N.P. Tchipev, Algorithmic and implementational optimizations of molecular dynamics simulations for process engineering, 2020.
[16] F.P. Brooks Jr, Computer 20 (4) (1987) 10–19.
[17] J.R. Rice, in: Advances in Computers, vol. 15, Elsevier, 1976, pp. 65–118.
[18] C. Niethammer, S. Becker, M. Bernreuther, M. Buchholz, W. Eckhardt, A. Heinecke, S. Werth, H.-J. Bungartz, C.W. Glass, H. Hasse, J. Vrabec, M. Horsch, J. Chem. Theory Comput. 10 (10) (2014) 4455–4464.
[19] S. Plimpton, J. Comput. Phys. 117 (1) (1995) 1–19.
[20] M.J. Abraham, T. Murtola, R. Schulz, S. Páll, J.C. Smith, B. Hess, E. Lindahl, SoftwareX 1 (2015) 19–25.
[21] T. Hitz, M. Heinen, J. Vrabec, C.-D. Munz, J. Comput. Phys. 402 (2020) 109077.
[22] S. Seckler, F.A. Gratl, M. Heinen, J. Vrabec, H.-J. Bungartz, P. Neumann, J. Comput. Sci. 50 (2021) 101296.
[23] S. Seckler, N. Tchipev, H.-J. Bungartz, P. Neumann, in: 2016 IEEE 23rd International Conference on High Performance Computing, 2016, pp. 101–110.
[24] 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, P. Neumann, Int. J. High Perform. Comput. Appl. (2018) 1094342018819741.
[25] W. Eckhardt, A. Heinecke, R. Bader, M. Brehm, N. Hammer, H. Huber, H.-G. Kleinhenz, J. Vrabec, H. Hasse, M. Horsch, M. Bernreuther, C.W. Glass, C. Niethammer, A. Bode, H.-J. Bungartz, in: International Supercomputing Conference, Springer, 2013, pp. 1–12.
[26] W. Mattson, B.M. Rice, Comput. Phys. Commun. 119 (2–3) (1999) 135–148.
[27] I. Newton, Philosophiae Naturalis Principia Mathematica, vol. 1, Edmond Halley, 1687.
[28] J. Bonet, M.X. Rodríguez-Paz, J. Comput. Phys. 209 (2) (2005) 541–558.
[29] N. Tchipev, A. Wafai, C.W. Glass, W. Eckhardt, A. Heinecke, H.-J. Bungartz, P. Neumann, in: European Conference on Parallel Processing, Springer, 2015, pp. 774–785.
[30] E. Gamma, Design Patterns: Elements of Reusable Object-Oriented Software, Pearson Education India, 1995.
[31] S. Werth, M. Horsch, H. Hasse, J. Mol. Liq. 235 (2017) 126–134.
[32] J. Vrabec, G.K. Kedia, G. Fuchs, H. Hasse, Mol. Phys. 104 (09) (2006) 1509–1527.

[17] https://github.com/AutoPas/AutoPas/graphs/contributors.