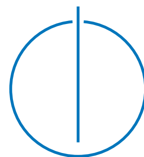
DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics

Integration of the C++ Node-Level AutoTuning Library AutoPas in the Large-scale Atomic/Molecular Massively Parallel Simulator (LAMMPS)

Sascha Sauermann



DEPARTMENT OF INFORMATICS

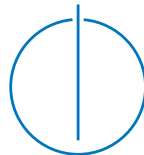
TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics

Integration of the C++ Node-Level AutoTuning Library AutoPas in the Large-scale Atomic/Molecular Massively Parallel Simulator (LAMMPS)

Integration der C++ Node-Level AutoTuning Bibliothek AutoPas in den Large-scale Atomic/Molecular Massively Parallel Simulator (LAMMPS)

Author:	Sascha Sauermann
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisors:	Fabio Gratl, M.Sc.
	Steffen Seckler, M.Sc. (hons)
Submission Date:	August 15, 2020



I confirm that this master’s thesis is my own work and I have documented all sources and material used.

Munich, August 15, 2020 Sascha Sauermann

Acknowledgments

I want to thank all people that supported me throughout my thesis and my degree in general. My sincere gratitude goes to my advisor Fabio Gratl for his excellent guidance, continuous support, and valuable feedback as well as keeping all discussions productive but also light-hearted and amusing. A huge thanks also to Steffen Seckler for giving me further insights into the AutoPas library.

And special thanks to all developers and contributors of LAMMPS and AutoPas.

Abstract

Molecular dynamics simulations are a typical example of an N-body problem. They have immense computational costs and require a high degree of parallelism to be solvable in reasonable time on current hardware.

The performance of different data structures and algorithms highly depends on the scenario and available hardware. It might even change over the time of a simulation. Thus it is not always obvious what to select before starting the simulation. The AutoPas library solves this by performing auto-tuning and (re-)selecting the best combination at runtime.

The goal of this thesis was to integrate AutoPas into the *Large-scale Atomic/Molecular Massively Parallel Simulator* (LAMMPS). We then show how to make the existing functionality compatible with AutoPas by focusing on some of the provided example scenarios. Additionally, we compare the node-level performance of the AutoPas integration with the OMP parallelization of the *USER-OMP* and *KOKKOS* packages provided with LAMMPS.

We achieve a successful integration with reasonably good performance that is easily extendable to support more of LAMMPS's features. While we do not outperform the provided packages with the current, still unoptimized version of AutoPas, our integration shows a significantly better speedup for high core counts.

Contents

Acknowledgments **iii**

Abstract **iv**

1. Introduction **1**

2. Theoretical Background **2**
2.1. Time Integration 2
2.2. Modeled Forces 2
2.3. Efficient Algorithms for the Short-Range N-body Problem 3

3. Technical Background **5**
3.1. LAMMPS 5
3.2. AutoPas 7
3.3. Kokkos 9

4. Implementation **10**
4.1. Simple Integration 10
4.2. Full Integration 10
4.2.1. LAMMPS Package Creation 11
4.2.2. AutoPas Accelerator 12
4.2.3. AutoPas-ifying of LAMMPS Classes 12
4.2.4. Implementation Problems & Pitfalls 15
4.2.5. Usage of the AutoPas Accelerator 17

5. Scenarios **20**
5.1. Lennard-Jones Melt 20
5.1.1. Scenario Description 20
5.1.2. Necessary Styles & Features 21
5.1.3. Performance Comparison 22
5.1.4. Causes for Loss of Performance 31
5.2. Lennard-Jones Crack 35
5.2.1. Scenario Description 35

5.2.2. Necessary Styles & Features 35
5.3. Lennard-Jones Obstacle Flow 37
5.3.1. Scenario Description 37
5.3.2. Necessary Styles & Features 37

6. Evaluation of the Integration Effort **40**

7. Conclusion & Future Work **43**

A. Appendix **44**

List of Figures **51**

List of Listings **52**

List of Tables **53**

Notes **53**

Bibliography **55**

1. Introduction

Molecular dynamics (MD) simulations require an enormous computational effort for any life-sized scenario and thus are quite an interesting topic for computer-science. Running these simulations includes parallelizing them over many nodes of a supercomputer and across multiple cores on the CPUs (or other hardware) of each node. This node-level performance is of utmost importance for a good scaling simulation.

Large-scale Atomic/Molecular Massively Parallel Simulator (LAMMPS) is a huge MD code that can be used to perform simulations of many different scenarios. It also comes with support for running highly parallel simulations using MPI and node-level parallelism like OpenMP, CUDA, etc. LAMMPS uses Verlet lists and an SoA-like architecture to efficiently compute forces of short-range potentials.

There exist many different algorithms and data structures for these computations whose performance can vary greatly depending on the available hardware or the simulated scenario. The best approach can even change over time when, e.g., the particle distribution of the simulation changes significantly. Therefore, selecting the fastest settings as a human often requires extensive domain knowledge, a good understanding of the different algorithms available, and some experimentation. [4]

Leaving this difficult decision to the computer, called automated algorithm selection, is a sub-problem of the process called auto-tuning. The latter is the overarching goal of the AutoPas library, which provides a black-box approach for performing the required computations. The user only has to define what should be calculated without the need to care about the specific algorithm used or the internal data structure. AutoPas will then select the most efficient configuration at runtime and can even change it at regular intervals if needed.

For this thesis, we integrated AutoPas as another node-level parallelization into LAMMPS. In general, it can be selected by the user to run the same simulations that are possible with LAMMPS. We show this by looking at three of the example scenarios that are shipped with LAMMPS. We also compare the performance with the OMP based node-level parallelizations provided in the *USER-OMP* and *KOKKOS* packages that come bundled with LAMMPS.

2. Theoretical Background

Molecular Dynamics (MD) Simulations compute the physical movement and interactions between small particles or molecules. This allows domain scientists insights into structural behavior or thermodynamic properties like energy, temperature, and pressure. They are commonly used in areas like chemistry, biology, astrophysics, and material science. [1, 6, 7, 11]

As in general interactions between all particles have to be considered, MD simulations are a typical example of an N-body problem. Thus, they have quadratic computational complexity. As life-size simulations can easily reach an unimaginable amount of particles (e.g., around 10^{21} (sextillion) molecules in a single drop of water), they require way more efficient methods and still a huge computational effort. [5]

2.1. Time Integration

The movement of particles is governed by *Newton's Equations of Motion*, a system of second-order *ordinary differential equations* (ODEs). A commonly used numerical solver for them is the *Störmer-Verlet Method*. There exist two variants of it that are less susceptible to rounding errors:

Leapfrog Scheme The new velocities are calculated with an offset of half a timestep at time $t + \frac{\delta t}{2}$ from the old velocities and forces. Then the positions are calculated at $t + \delta t$ from these velocities. Finally, the new forces can be computed. If the velocities at the full timesteps are needed, they have to be calculated separately.

Velocity-Störmer-Verlet Method The new positions are calculated from the old velocities and forces at $t + \delta t$. Then the new forces are computed. Finally, the velocities are updated from the old velocities and new and old forces. Thus this method requires an additional force vector in memory.

See reference [5] for a more in-depth explanation and formulas.

2.2. Modeled Forces

The forces that act between particles can be split into two groups:

- Short-range interactions like the *Van der Waals Forces*
- Long-range interactions like electrostatic forces

Each requires a different approach to be solved efficiently. We'll only focus on the short-range interactions in this thesis.

A common short-range potential is the *12-6 Lennard-Jones Potential*, which models an attraction between particles based on the *Van der Waals Forces* and a repulsion based on the *Pauli principle*. It is parameterized by a *potential well* ε and a *zero crossing* σ. It results in strong repulsive forces up to the energy minimum of $-\varepsilon$ and attractive forces past that minimum. For larger distances, it falls off quickly and stays close to zero, which results in very weak attractive forces. See Figure 2.1 for an example and reference [5] for formulas.

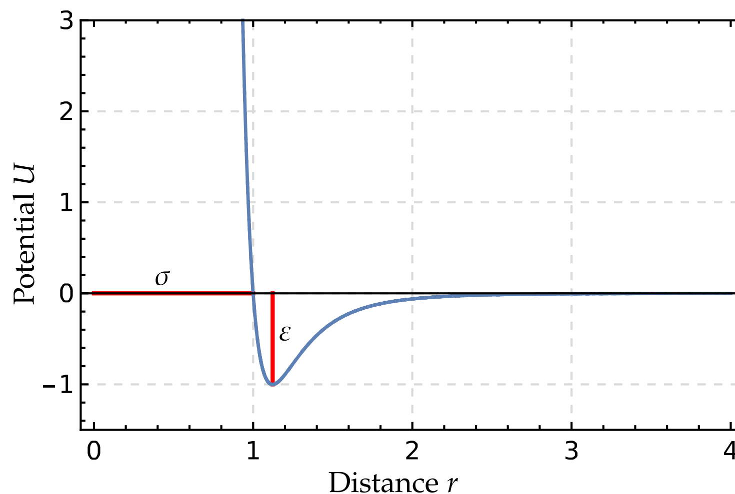


Figure 2.1.: The *12-6 Lennard-Jones Potential* for a *potential well* $\varepsilon = 1$ and *zero crossing* $\sigma = 1$. Particles are repelled for distances smaller than ~ 1.15 units and attracted for larger ones. Particles further apart than ~ 2.5 units apply a negligible force on each other.

2.3. Efficient Algorithms for the Short-Range N-body Problem

All short-range potentials like the *Lennard-Jones Potential* discussed in the previous section converge to zero for larger distances. We can thus define some distance r_c where the potential is sufficiently close to zero. This distance is called the *cutoff radius*. All interactions between particles that are further apart than this, can be ignored without

accumulating a significant error in the calculation of the individual particle forces. It can be necessary though to apply a correction term for the energy of the whole system.

There exist different efficient methods for determining particle pairs that could have a distance from each other smaller than the cutoff radius, so that the overall complexity is reduced to be linear in the number of particles. [5]

Two common methods are:

Linked Cells The domain is divided into cuboidal cells and particles are assigned to the cell that they reside in. Only the distances of particles in cells that are in close proximity are computed for deciding if the forces between them need to be calculated. It is common to choose the cell width bigger than or equal to the cutoff radius, such that only neighboring cells have to be considered.

Therefore only a small constant number of possible interaction partners have to be considered for each particle, reducing the complexity of this algorithm to $O(n)$. [5, 9]

Verlet Lists For each particle, neighbor lists are constructed that contain all sufficiently close particles. Because particle are moving in a simulation, they have to be rebuilt when new particles come in range. By increasing the distance that is used for constructing these lists by a so-called *Verlet skin*, the number of simulation steps between rebuilds can be increased. The force computation then only considers particles from those neighbor lists when calculating distances and consequently the pairwise interactions.

Constructing the neighbor lists requires $O(n^2)$ operations as distances between all particles have to be calculated. This can be improved by combining this approach with the Linked Cells algorithm to build these lists. [2, 12, 13]

3. Technical Background

This chapter describes the structure and data-flow of the MD code LAMMPS and the capabilities of the node-level auto-tuning library AutoPas that was integrated into LAMMPS for this thesis.

We are using the LAMMPS version `stable_7Aug2019` and the AutoPas version of the commit `918ce7c356250df60b9a1e217482a0c8d4f7bf0d`.

3.1. LAMMPS

LAMMPS[1], which is an acronym for *Large-scale Atomic/Molecular Massively Parallel Simulator*, is a molecular dynamics C++ code developed by the *Sandia National Laboratories* financed by the *US Department of Energy*. It can be used to model atoms or general particles for solid-state materials, soft matter, coarse-grained materials, or mesoscopic systems. [10]

The core simulation loop consists of the steps expected for an MD simulation, see Figure 3.1 for a class overview:

1. Time integration
2. Periodic boundary conditions
3. Leaving particle exchange
4. Sending particle ghost copies
5. Neighbor list rebuild
6. Calculation of pairwise forces
7. Force reverse communication
8. Output

All particle data is stored in C-style arrays that are accessible via a global singleton. This results in code that accesses everything directly in memory via the particle index without

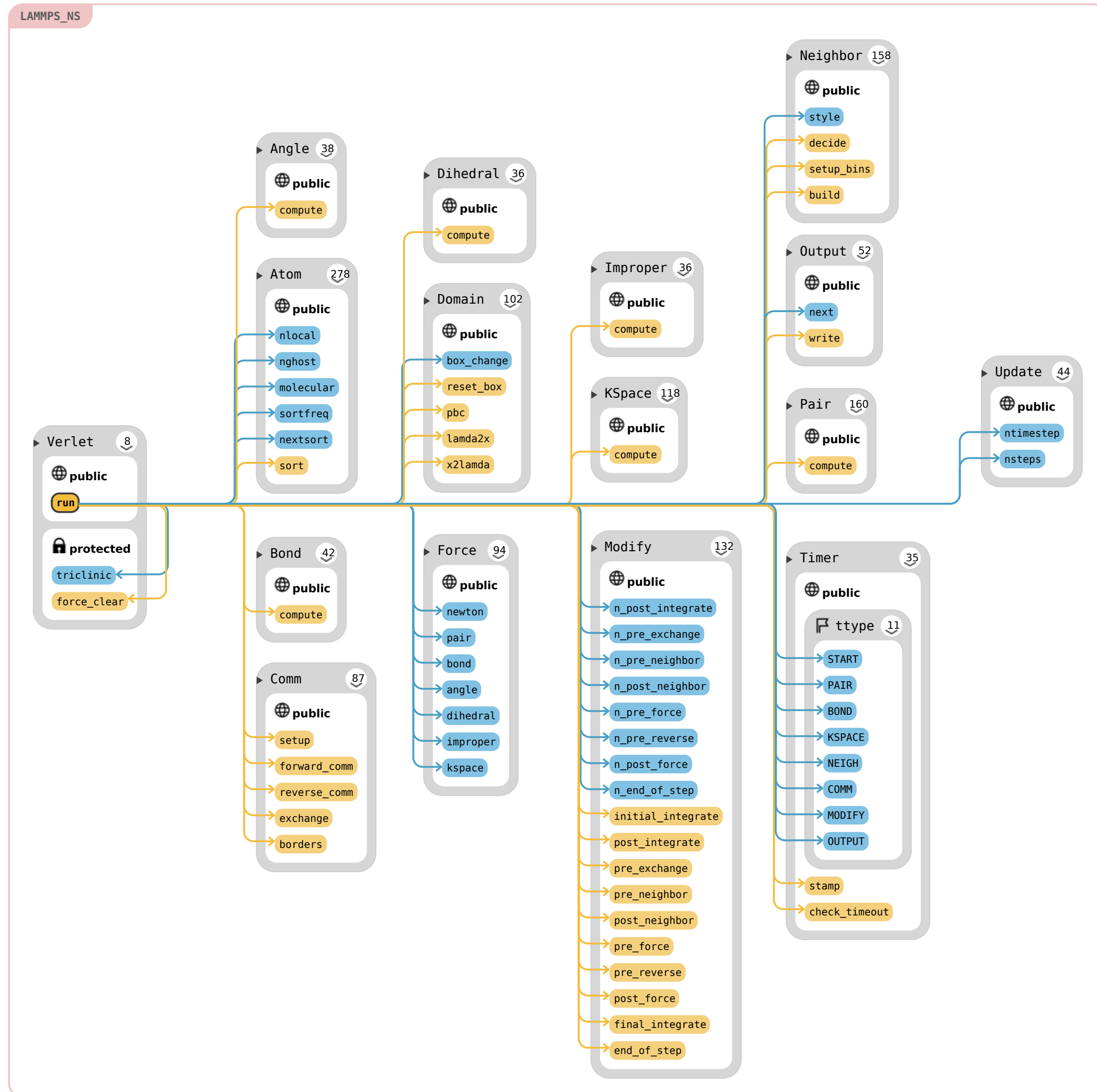


Figure 3.1.: **LAMMPS - Verlet::run method:** Overview of all classes used and functions called from the run method of the Verlet integrator class. This method contains the main simulation loop and performs all relevant steps as listed in section Section 3.1.

any indirections. This is quite fast but results in code that has no encapsulation at all and thus is difficult to maintain or change. See Figure A.7 for an example showing this.

LAMMPS supports grid-based domain decomposition using MPI with two different partition styles. A grid-based `brick` style and a `tiled` style based on cuboids of different sizes.

LAMMPS is highly extensible by user-created code and comes packaged with a lot of optional functionality. It uses the following definitions for the description of its modular structure:

Style Styles are the features of LAMMPS, like new particle types, force computations, or operations being performed every time step (so-called fixes). Every style is defined by its type (e.g., pair, fix, angle) and a name used for selecting it in the input file (e.g., pair `lj/cut` or fix `nve`). The slash is the naming scheme used by LAMMPS to define a variant of a similar style. It is implemented by inheriting from the style type base class and integrated into the CMake build system by supplying the name in a preprocessor macro. Styles can then be enabled in the input file using their name.

Package A package is a collection of new styles or variants of existing styles, e.g., for an accelerator. It is defined by a separate CMake file listing the source files, external dependencies, etc. Packages can be enabled in any combination at configure time by defining the `PKG_<NAME>` flag when running CMake.

Accelerator An accelerator is a package providing a parallelized implementation for all (or possibly only some) available styles utilizing a certain architecture or hardware type to the fullest. These for example include executing in parallel on CPUs with OMP, GPUs with CUDA, or special optimizations for Intel Xeon Phis. To define an accelerated variant of a style, the accelerator suffix is appended to the style name (e.g., `lj/cut/omp`). Accelerated styles can be selected manually in the input file by using their name or can be enabled globally by using the `-sf` command line flag.

3.2. AutoPas

AutoPas is a C++ node-level auto-tuning library for particle simulations developed in the context of the TaLPas[2] project. It is still in active development and not yet available as a stable release.

AutoPas provides a black-box approach for performing MD simulations. It handles the storage of particles for a given domain and provides iterators for accessing them. Pairwise particle interactions can be implemented efficiently by writing so-called functors

that are then applied by AutoPas to particle pairs considering the cutoff radius. The public interface of the library can be seen in Figure 3.2.

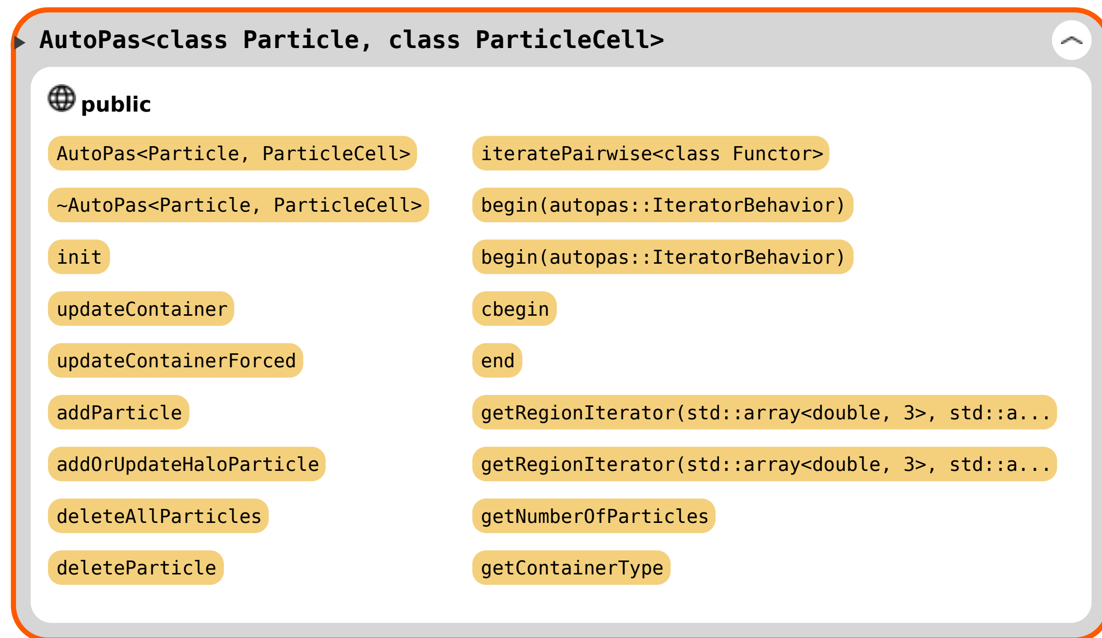


Figure 3.2.: **AutoPas - Public interface:** This figure shows the interface of the AutoPas framework without the getters and setters for the configuration options. The remaining functions include methods for adding and removing particles, iterating all particles, iterating parts of the domain, and iterating particle pairs with functors. Further, the container can be updated when particles have been moved, which might rebuild the internal neighbor structure and return particles that left the domain.

The internal algorithms and data structure can be switched while keeping the interface for the user consistent. AutoPas can change the following things:

Container Object storing and managing all particles defining the algorithm how particle neighbors are found like *Linked Cells* or *Verlet Lists* (see Section 2.3)

Traversal Algorithm how the pairwise force calculations are parallelized on the hardware

Data Layout Array-of-Structures (AoS) or Structure-of-Arrays (SoA)

Optimization Switches Other optimizations like using Newton's third law to remove up to half of the force calculations but introducing parallelization limitations

The decision which algorithms to use for the aspects mentioned above can be left to the auto-tuner. It will re-evaluate the simulation performance at set intervals and tries to select the fastest combination available for the next simulation steps. Functor implementations can also define if they support Newton3 optimizations, which then can be considered in the auto-tuning process. [4]

Currently, AutoPas is limited to short-range potentials that can be modeled using a cutoff radius. It also always uses an AoS data layout internally that is converted to SoA every time step when needed. As LAMMPS always uses *Verlet Lists* with an SoA layout, it might have an inherent advantage in all scenarios where SoA is the preferred layout.

3.3. Kokkos

Kokkos is a C++ programming model for writing performance portable code. It provides a hardware-independent interface, but the same code gets compiled and optimized for different architectures and node configurations. Currently, it supports CUDA, OpenMP, Pthreads, and HPX as backends.

The two core concepts Kokkos is based on are: [3]

1. The dispatching of architecture optimized parallel kernels for performing computations.

2. And the management of the data structure such that these kernels have an optimal layout for the target device.

In general, this means Kokkos is similar to AutoPas, as it also allows the same code to run efficiently on different systems. While AutoPas does auto-tuning at runtime to select the best configuration, Kokkos performs the optimizations at compile time. This is more or less like compile-time static auto-tuning.

4. Implementation

The following chapter explains how AutoPas was integrated into LAMMPS and how to extend the current implementation.

4.1. Simple Integration

The first approach for the integration of AutoPas is to just replace the computation of the pairwise forces. For this, we added a pair style for the 12-6 Lennard-Jones potential with an AutoPas suffix: `lj/cut/autopas`

The AutoPas container is kept alive for the whole simulation to not lose the auto-tuning information and the force computation was then simply performed as follows:

1. Copy particles from LAMMPS's C-style arrays into AutoPas container
2. Compute pairwise forces using a functor
3. Copy particles back to the C-style arrays

While this allowed us to keep all LAMMPS code except for the classes calculating the pairwise forces and run any scenario already supported by LAMMPS, this approach had major drawbacks and did not scale well for a higher number of particles.

Not only is the copy operation costly but the internal data structure of AutoPas (e.g., *Verlet Lists* or *Linked Cells*) has to be rebuild after inserting particles. This for example removed the possibility to use a *Verlet Skin* and keep the neighbor lists for multiple steps. It would also double the memory requirements and still would not parallelize any other parts of the core LAMMPS code.

4.2. Full Integration

The code of the AutoPas integration can be found in the `ssauermann/lammps-autopas` GitHub repository[3].

4.2.1. LAMMPS Package Creation

Our LAMMPS package is defined by a CMake file in the `cmake/Modules/Packages` directory. The package is enabled with the `PKG_USER-AUTOPAS` flag when configuring LAMMPS. The AutoPas library is included via the CMake *FetchContent* module directly from its GitHub repository[4].

LAMMPS provides a system for automatically discovering sources files that overwrite existing styles in form of the `RegisterStylesExt` function. Any additional file for the accelerator structure must be added manually to the list of source files. See Listing 4.1 for an excerpt from the package CMake file.

```
set(USER-AUTOPAS_SOURCES_DIR ${LAMMPS_SOURCE_DIR}/USER-AUTOPAS)

add_definitions(-DLMP_AUTOPAS)

# sources that are not a style
set(USER-AUTOPAS_SOURCES
${USER-AUTOPAS_SOURCES_DIR}/autopas.cpp
${USER-AUTOPAS_SOURCES_DIR}/atom_autopas.cpp
${USER-AUTOPAS_SOURCES_DIR}/atom_vec_autopas.cpp
${USER-AUTOPAS_SOURCES_DIR}/comm_autopas.cpp
${USER-AUTOPAS_SOURCES_DIR}/domain_autopas.cpp
${USER-AUTOPAS_SOURCES_DIR}/neighbor_autopas.cpp
${USER-AUTOPAS_SOURCES_DIR}/output_autopas.cpp
)

set_property(GLOBAL PROPERTY "AUTOPAS_SOURCES" "${USER-AUTOPAS_SOURCES}")

# detects styles which have USER-AUTOPAS version
RegisterStylesExt(${USER-AUTOPAS_SOURCES_DIR} autopas AUTOPAS_SOURCES)

get_property(USER-AUTOPAS_SOURCES GLOBAL PROPERTY AUTOPAS_SOURCES)

list(APPEND LIB_SOURCES ${USER-AUTOPAS_SOURCES})
include_directories(${USER-AUTOPAS_SOURCES_DIR})
```

Listing 4.1: Excerpt of the CMake file defining the AutoPas package for LAMMPS. Overwritten styles are discovered by the `RegisterStylesExt` function (l.19).

4.2.2. AutoPas Accelerator

The core of our AutoPas integration is the *AutoPas accelerator* class `AutoPasLMP` in `src/USER-AUTOPAS/autopas.cpp` that manages the AutoPas container from the external library and provides the interface to the LAMMPS code. The accelerator is initialized in `src/lammps.cpp` when the `-autopas on <options>` argument is set when running LAMMPS. All parameters following this flag are forwarded to the accelerator class until the next argument starting with a dash. See Subsection 4.2.5 for further instructions on how to use the accelerator.

When running an input file using the AutoPas accelerator, the particles are only copied into the AutoPas container for the execution of the `run` command and copied back afterward. Thus, any style implementation has to check the initialization status of the accelerator before performing operations and provide a fallback in case the code is executed from a different command (e.g., `compute`). See Listing 4.3 for implementation advice.

Additionally, we have to overwrite some of the LAMMPS core classes that are not part of the style system:

- `Atom` - Atom sorting not possible/necessary with AutoPas
- `CommBrick` - Modified communication code
- `Domain` - AutoPas-ifying and custom handling of domain resizing
- `Neighbor` - Disabled creation of neighbor lists as AutoPas manages those
- `Output` - Copy back of particles from AutoPas container required

They are initialized instead of the base class in `src/lammps.cpp` when the AutoPas accelerator is defined.

To circumvent compile errors when the `USER-AUTOPAS` package is not loaded, we provide a dummy interface in `src/accelerator_autopas.h`. It uses preprocessor macros to include the real classes when `LMP_AUTOPAS` is defined and otherwise just provides class stubs.

4.2.3. AutoPas-ifying of LAMMPS Classes

As described earlier, the particles are now stored inside the AutoPas container instead of in C-style arrays. Because the internal memory layout of the particles can change, a memory mapping is not possible. Thus most styles need slight adaptions to work with the AutoPas accelerator.

The basic steps that had to be performed to create an AutoPas compatible style are as follows:

1. Find all usages of the particle arrays in the class. These are the `atom->x`, `atom->v` and `atom->f` arrays. If none of those arrays are used, the style will very probable work out of the box with AutoPas.

2. Create a style variant of this class for the AutoPas accelerator by inheriting from the style and adding `/autopas` to the end of the style name in the style definition macro.

3. Overwrite all affected methods

See Listing 4.2 for an example header file.

```
#ifdef FIX_CLASS
FixStyle(temp/rescale/autopas,FixTempRescaleAutoPas)
#else
#ifndef LMP_FIX_TEMP_RESCALE_AUTOPAS_H
#define LMP_FIX_TEMP_RESCALE_AUTOPAS_H
#include "fix_temp_rescale.h"

namespace LAMMPS_NS {

class FixTempRescaleAutoPas : public FixTempRescale {
public:
        using FixTempRescale::FixTempRescale;
        void end_of_step() override;
};

};
#endif
#endif
```

Listing 4.2: **FixTempRescaleAutoPas header file:** `FixStyle` (l.2) registers this class as the AutoPas version of the `temp/rescale` fixture in the CMake build system. The class itself can then just inherit from the original fixture and overwrite all methods using the C-style arrays. The constructors can simply be forwarded (l.11) as they need no changes.

Make sure to provide a fallback implementation for the case when AutoPas is not initialized as discussed in the previous section. In most cases a call to the method of the base class is sufficient. See Listing 4.3 for an example.

```
auto Derived::foo() overwrite {
        if (!autopas->is_initialized()) {
                return Base::foo();
        }
        // Code with AutoPas here
}
```

Listing 4.3: **Fallback implementation when AutoPas is uninitialized:** AutoPas only gets initialized when LAMMPS executes the `run` command. As methods can also be executed from other commands (e.g., `compute`), it is necessary to provide a fallback implementation when AutoPas is not available. For this, a call to the base class is mostly sufficient.

Depending on the specific way the particles are accessed, different features of the accelerator can be used. To gain the maximal performance they should be considered in the following order when AutoPas-ifying LAMMPS code:

- Pair functor
- Region iterator
- Particle iterator
- Random access by index

See Listing 4.4 for an example of an AutoPas-ified particle loop.

Pair Functor

When replacing a force calculation that iterates over all particle pairs within a cutoff radius using LAMMPS's neighbor lists an AutoPas functor should be used. The iteration is then performed by calling `iterate_pairwise(functor)`. For further information on how to implement such a functor refer to the official AutoPas documentation[5].

Region Iterator

When replacing a loop over particles that also include a check if the particle position is inside a given slab or region of the domain, region iterators can be used. These iterators are accessed via `iterate_region(...)` and for convenience with different parameters also as `particles_by_slab(...)`. These iterators are used (and can be limited to local or ghost atoms) similarly to the particle iterator (See Section 5).

Particle Iterator

When all or a range of particles are accessed with a ranged based for loop, a particle iterator can be used to replace it:

- Replace the index-based loop through the particle arrays by an AutoPas iterator
- Add the `#pragma omp parallel` pragma to the loop with the necessary OMP data-sharing attributes when free of race conditions
- Get and set the position, velocity and force vectors via the iterator
- Access any other particle data in the arrays by index via the `localID` provided from the iterator

Depending on the range that should be iterated, different iterator behaviors have to be selected:

- 0 → nlocal — use `iterate<ownedOnly>()`
- nlocal → nlocal + nghost — use `iterate<haloOnly>()`
- 0 → nlocal + nghost — use `iterate<haloAndOwned>()`
- Any subset of the above — additionally check the index of particle returned by the iterator

Random Access by Index

If none of the above implementations is possible, index-based random access is available by calling `particle_by_index(index)` but can be quite slow. More in-depth refactoring of the code segment should be considered instead if it is a performance-critical path.

When accessing the particles in this way, an index to particle mapping is created on the first call. As the internal data structure of AutoPas can change, this cache is invalidated when performing certain operations like adding particles or rebuilding the neighbor lists.

4.2.4. Implementation Problems & Pitfalls

In the previous section, we presented a general approach for converting styles to an AutoPas compatible version. This section discusses some classes that needed special attention to work and other pitfalls that can occur.

```
#pragma omp parallel default(none) shared(factor)
for (auto iter = lmp->autopas->iterate<autopas::ownedOnly>();
iter.isValid(); ++iter) {
        auto v = iter->getV();
        int idx{iter->getLocalID()};
        if (atom->mask[idx] & groupbit) {
                if (which == BIAS) temperature->remove_bias(idx, v.data());
                v[0] *= factor;
                v[1] *= factor;
                v[2] *= factor;
                if (which == BIAS) temperature->restore_bias(idx, v.data());
                iter->setV(v);
        }
}
```

Listing 4.4: **Excerpt of the FixTempRescaleAutoPas source file:** The particle loop through all local particles is implemented with an AutoPas particle iterator. The mask of the particle is accessed via its ID and one of the C-style arrays.

Differing virial definitions AutoPas already provides a functor that calculates the Lennard-Jones potential. LAMMPS expects different values for the calculation of virials and global energy than that functor calculates though. Thus it could not be used as-is and had to be adapted.

Excluding particle interactions LAMMPS allows the user to define which particles should not interact with each other. This can be done for example by specifying type pairs to exclude with the `neigh_modify` command. LAMMPS then considers these exclusions when building the neighbor lists. As AutoPas does not use the same lists and does not support excluding types from the pair iterations directly, the force functor implementations have to perform additional checks. This was implemented by pre-generating an interaction map for all combinations of particle types that return a factor (either zero or one) which is multiplied with the result of the force computation. This approach was chosen because it still allows for the auto-vectorization of loops in the functor with recent compilers.

Shrink-wrapped boundary LAMMPS provides the possibility to define a domain boundary as shrink-wrapped such that the domain extends and retracts to accommodate the farthest particle. AutoPas can not simply resize its domain but has to recreate

the whole container. This loses all insights already gained for auto-tuning.

Therefore we limited the shrink-wrapping to only grow the domain and never shrink it. The possible performance degradation of parts of the domain being empty can be negated by the auto-tuning choosing a matching algorithm and data structure. Additionally, we noticeably increased the size the domain grows each time it gets to small, compared to the LAMMPS default, to minimize the frequency of this occurring.

Particle random access AutoPas provides no random access to particles via some kind of index. The domain decomposition and communication code of LAMMPS heavily rely on index lists though. Thus we would have to either completely rewrite those classes from scratch or provide index-based access. Although a rewrite would probably give better performance, the second, simpler approach was implemented by creating a temporary mapping from index to particle. This mapping can be created in a single iteration through all particles and allows retrieving a particle in constant time afterward. Similarly to LAMMPS's global to local ID mappings, this is either implemented using an array or a hashmap, depending on the number of global particles. See Section 5 for usage information.

4.2.5. Usage of the AutoPas Accelerator

When LAMMPS was built with the PKG_USER-AUTOPAS flag, the AutoPas accelerator can be used when running the program. To enable the package the -autopas on command-line switch has to be added. This also invokes a package autopas command with default settings.

When the user wants to customize the behavior of AutoPas they can set further options either by adding a package autopas <options> command to the input file or appending the options to the command-line switch like this: -autopas on <options>.

To enable the AutoPas enabled styles, they can either be selected by hand at every style selection by appending the /autopas suffix in the input file (e.g., fix nve/autopas) or enabled globally with the -suffix (-sf) command-line argument like this: -suffix autopas. Keep in mind that most non-AutoPas styles do not work when AutoPas is enabled with autopas on because particles are stored differently. Thus it is highly recommended to use the suffix flag to use the AutoPas version of styles wherever available.

A complete call of LAMMPS using the AutoPas accelerator might therefore look like this:

```
lmp -i <input_file> -autopas on t c04,c08 i 1000 n enabled -sf autopas
```

See Table 4.1 for a full list of options. All options that are considered for auto-tuning, allow setting multiple values by giving a list separated by any of the , ; | / characters.

Option	Short	Tuning	Possible Values	Description
log			`trace, debug, info, warn, err, critical, off`	Logging level
newton	**n**	✓	`enabled, disabled`	Newton3 options
traversals	**t**	✓	`c08, sliced, c18, c01, directSum, verlet-sliced, verlet-c18, verlet-c01, cuda-c01, verlet-lists, c01-combined-SoA, verlet-clusters, c04, var-verlet-lists-as-build, verlet-clusters-coloring, c04SoA, verlet-cluster-cells, verlet-clusters-static, balanced-sliced, balanced-sliced-verlet, c04HCP`	Traversal options
containers	**c**	✓	`DirectSum, LinkedCells, VerletLists, VerletListsCells, VerletClusterLists, VarVerletListsAsBuild, VerletClusterCells`	Container options
data	**d**	✓	`AoS, SoA`	Data layout options
interval	**i**		`<integer>`	Tuning interval
strategy	**s**		`bayesian-Search, bayesian-cluster-Search, full-Search, random-Search, active-harmony, predictive-tuning`	Tuning strategy
selector			`Fastest-Absolute-Value, Fastest-Mean-Value, Fastest-Median-Value`	Evidence selector
vc_size			`<integer>`	Verlet-cluster size

Option	Short	Tuning	Possible Values	Description
samples			<integer>	Number of tuning samples
evidence			<integer>	Max evidence for tuning
pred_ror			<double>	Relative optimum range for predictive tuning
pred_mtpwt			<integer>	Max tuning phases without test for predictive tuning
bayesian_af			upper-confidence-bound, mean, variance, probability-of-improvement, expected-improvement	Acquisition function for bayesian based searches
csf		✓	<double>	Cell size factors
estimator		✓	none, squared-particles-per-cell, neighbor-list-length	Load estimator for balanced traversals
notune				Strictly checks if only a single configuration is selected

Table 4.1.: **Options for the AutoPas accelerator:** Either the long or short form can be used when giving an option followed by the value. When the setting is auto-tunable, multiple options can be given by a list separated by any , ; | / character. The values do not have to match exactly but are matched with the *Needleman-Wunsch algorithm* to find the closest one.

5. Scenarios

In the previous chapter, we have seen how AutoPas was integrated as an accelerator and how to make styles AutoPas compatible. For this thesis, we looked at three different example input files shipped with LAMMPS, converted all styles used in them, and made some additional adaptions where necessary. This chapter explains for each of the three scenarios which styles were necessary to AutoPas-ify and which additional features had to be implemented on the accelerator. The converted scenarios perform an identical simulation to the original ones with the same results.

We also compare the node-level performance of runs using AutoPas with two accelerator packages that come with LAMMPS: *USER-OMP*[6] and *KOKKOS*[7] (with OMP backend).

The benchmarks were executed on the *CoolMUC2* and *CoolMUC3* clusters of the *Leibniz Supercomputing Centre (LRZ)*[8]. The former uses the same *Haswell* CPU based hardware as the recently decommissioned *SuperMUC Phase 2* and the latter has a *Knights-Landing* manycore architecture. See Table 5.1 for a detailed hardware overview.

	CoolMUC2	CoolMUC3
Architecture	Haswell	Knights Landing
Processor Type	Intel Xeon E5-2697 v3	Intel Xeon Phi 7210F
Nominal Frequency (turbo)	2.6 (3.6) GHz	1.3 (1.4) GHz
Cores per Node	2×14	64
Threads per Core	2	4
Vector Instruction Set (width)	AVX2 (256 Bit)	AVX-512 (512 Bit)
Memory per Node	64 GB DDR4	96 GB DDR4 + 16 GB HBM

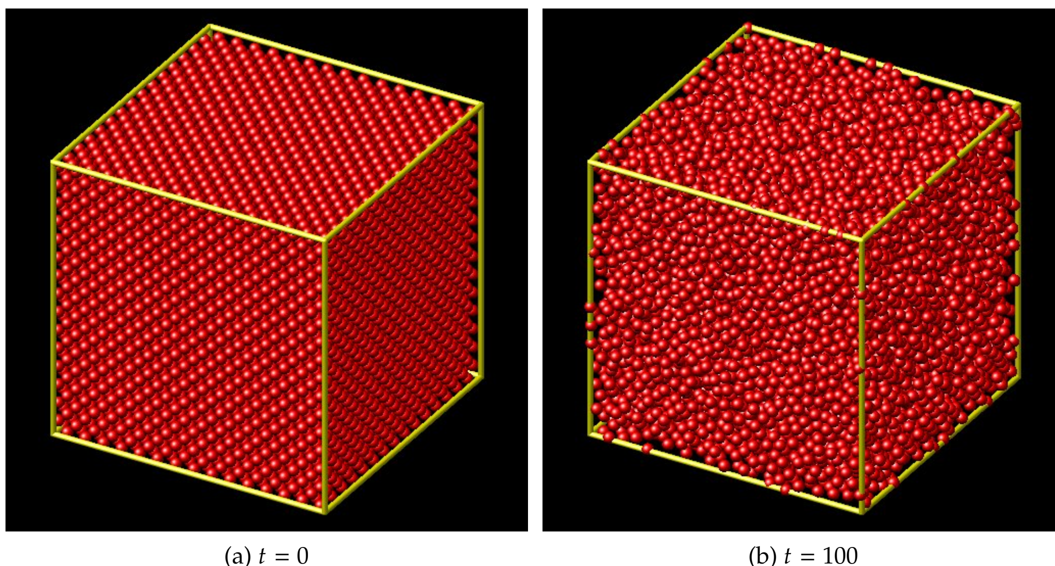
Table 5.1.: **Specifications of the CoolMUC2[9] and CoolMUC3[10] clusters.**

5.1. Lennard-Jones Melt

5.1.1. Scenario Description

The first scenario we converted is used typically for benchmarks as it does not change much over time. We use the 12-6 Lennard-Jones potential (see Section 2.2) because it is

quite common for MD simulations. We simply use $\sigma = \epsilon = 1$ and a cutoff radius of 2.5. The domain is completely filled with atomic particles on a face-centered cubic lattice. They start with random velocities such that a temperature of 1.44 is reached for the system. The boundaries are periodic and a thermostat is applied every 100 steps. The rebuild frequency of the neighbor lists is set to 20 steps and they use a skin of 0.3. See Figure 5.1 for an example of this setup.



(a) $t = 0$ (b) $t = 100$

Figure 5.1.: **Lennard-Jones Melt Scenario:** The fully periodic domain is filled with a regular arrangement of particles with random velocities. We use atomic particles and the 12-6 Lennard-Jones potential with $\sigma = \epsilon = 1$ and a cutoff radius of 2.5. Because of the high density, this is a good scenario for benchmarks and performance comparisons between the different implementations.

5.1.2. Necessary Styles & Features

In addition to the core accelerator integration as described in Section 4.2, the following styles were AutoPas-ified to run this scenario.

[Atom Style] atomic An atom style defines the attributes that make up every particle and handles their memory layout. It is responsible for packing/unpacking the data into buffers for in- and inter-process communication. Atomic is the simplest style, just consisting of position, velocity, force, tag, type, mask, and image attributes.

The most significant change for the AutoPas-ified implementation is that the position, velocity, force, tag, and type are only stored inside of AutoPas and no longer in the C-style arrays. Because the remaining attributes are not needed in any AutoPas functor, they can be kept as is.

Packing particles into buffers is done using particle iterators where possible but sometimes requires accessing particles by index. When unpacking, the particles are directly inserted into the AutoPas container.

[Pair Style] lj/cut This class implements the 12-6 Lennard-Jones potential with a cutoff radius as described in Section 2.2. It was implemented by writing an AutoPas functor for calculating the pairwise interactions between particles.

[Fix] nve This fix performs the leapfrog integration to update the velocities and positions of the system. Because the number of particles (N), volume (V), and energy (E) are constant the trajectories are consistent with the microcanonical ensemble. Particle iterators were used for performing this calculation in parallel for all particles.

Other changes to the input file that were necessary to be compatible with AutoPas are:

Atom Sorting LAMMPS can resort the C-style arrays to improve the locality of succeeding particles. We do not support sorting because the iteration order is given by AutoPas anyway.

Index Map We need to enable the global particle IDs to perform mappings between particles in the AutoPas container and their corresponding index in the C-style arrays.

5.1.3. Performance Comparison

To compare the performance of the AutoPas integration we use the LJ melt scenario similar to the benchmarks provided in the `bench/KEPLER` folder with LAMMPS which results can also be found on the LAMMPS website[11].

We use a domain size of 214.988^3 filled with 8388608 atoms on a single node. LAMMPS was compiled with the *Intel C++ Compiler 19.0.5* with the matching vectorization optimizations enabled for both clusters. The affinity for the OMP threads is set to thread scatter for CoolMUC2 and to thread balanced for CoolMUC3 using the `KMP_AFFINITY` environment variable of the *Intel OpenMP Runtime Extensions*. We simulate 1000 steps to amortize the auto-tuning costs of AutoPas better than just running 100 steps.

We perform strong scaling for 1 to 56 threads for CoolMUC2 and 1 to 256 threads for CoolMUC3. All measurements were only taken three times because they had a small variance and subsequently averaged. The compared configurations are shortened as follows in the plot legends:

AutoPas-FullSearch AutoPas with the default settings, trying most possible combination of algorithms and data-structures when auto-tuning. Only configurations that most likely are not a good choice (e.g., *Direct Sum*) are skipped.

AutoPas-Bayesian AutoPas with the default settings but using a bayesian search for auto-tuning. This approach finds good settings faster than the full search but is not guaranteed to find the optimum.

AutoPas-C04 AutoPas using the *C04* traversal with the *Linked Cells* container, SoA data layout, and Newton3 optimizations enabled. No tuning is performed. This is the fastest combination of settings in this scenario for higher thread counts.

AutoPas-NoN3 AutoPas using the balanced *sliced verlet traversal* with the *Verlet Lists Cells* container, AoS data layout, and Newton3 optimizations disabled. This is a configuration chosen by the auto-tuner for higher core counts in this scenario when disabling Newton3 optimizations explicitly.

OMP USER-OMP package parallelization. Implemented with an MPI-like approach using separate copies of the data per thread. Therefore, the pairwise forces can be computed using Newton3 optimizations without synchronization. Afterward, global properties are reduced in a `#pragma omp critical` section. The reduction of per-atom properties is performed in parallel. [8]

Kokkos KOKKOS package parallelization with the OMP backend. Kokkos is set to use half neighbor lists with Newton3 optimizations enabled, which is the default. Atomic operations are used to prevent race conditions.

Kokkos-NoN3 KOKKOS package parallelization with the OMP backend. Kokkos is set to use full neighbor lists with Newton3 optimizations disabled. This is recommended in the documentation for pair-wise potentials as it needs no synchronization.

Chosen configurations by auto-tuning

The auto-tuner consistently chooses a *Verlet Lists* based configuration for a low number of threads and a *Linked Cells* based configuration for more threads.

Figure 5.2 and Figure 5.3 give an overview of what configuration was chosen for this scenario by the tested auto-tuning approaches for the different thread counts as well as the time they took for one timestep. If not all three runs chose the same configuration, the most common one is shown.

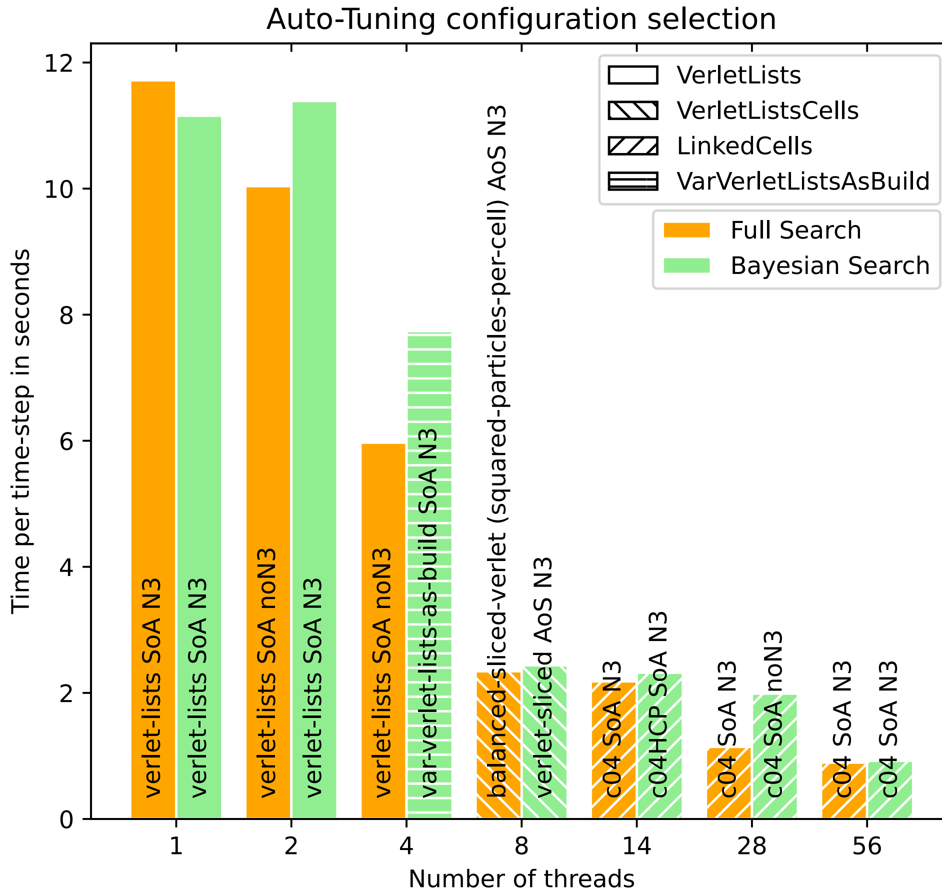


Figure 5.2.: **Auto-tuner configuration selection CoolMUC2:** The auto-tuner selected these AutoPas configurations for the Lennard-Jones melt scenario with about 8 million particles when simulated on the CoolMUC2. The patterns show the selected container, while the traversal, data layout, and if the Newton3 optimizations are enabled, is written on each bar.

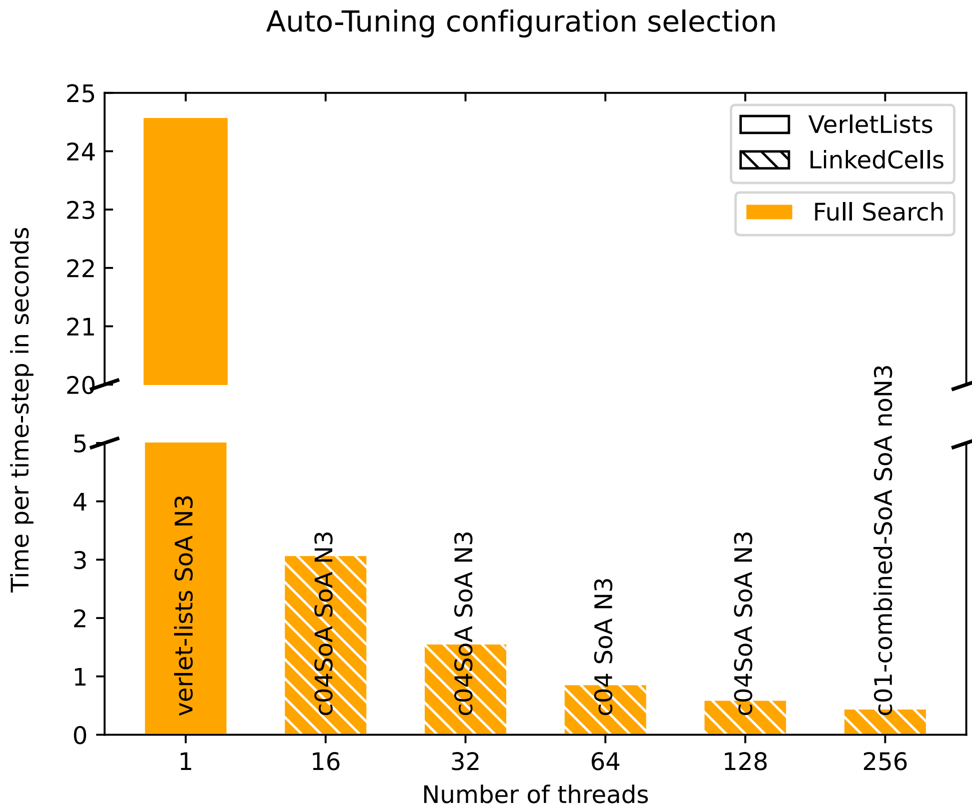


Figure 5.3.: **Auto-tuner configuration selection CoolMUC3:** The auto-tuner selected these AutoPas configurations for the Lennard-Jones melt scenario with about 8 million particles when simulated on the CoolMUC3. The patterns show the selected container, while the traversal, data layout, and if the Newton3 optimizations are enabled, is written on each bar.

Atom-Timesteps per Second

The main metric for the performance comparison is the number of timesteps per second that can be performed. To stay consistent with the LAMMPS benchmark website, we plot the atom-timesteps per second on the y-axis. This means a value of 4 million for our about 8 million particle system equals the simulation running at 0.5 steps per second.

Figure 5.4 and Figure 5.5 show the plots for running on CoolMUC2 and CoolMUC3 respectively. The OMP package outperforms both AutoPas and Kokkos which is not unexpected as it is optimized low-level code specialized on a small core count. AutoPas is only slower by a factor of 2-4. We can also see that Kokkos stops improving somewhere around 14 (32) threads and only AutoPas even scales past the physical core count utilizing hyper-threads.

Similar relations can be shown when plotting the total simulation time as seen in Figure A.3 and Figure A.4.

Speedup

Next, we can look at the parallelization efficiency and speedup of the different accelerators to compare how parallelizable the code is. For MD codes this is often influenced by the amount of time spent communicating instead of computing.

The speedup can be defined as the time the simulation takes sequentially divided by the time using multiple threads n: t_1/t_n. Ideally, the performance would scale linearly but this is not the case for most algorithms. Another way to show the same concept is by plotting the parallel efficiency, by dividing the speedup by the number of threads. This should stay as close to the value one as possible.

Figure 5.6 and Figure A.5 show the speedup while Figure 5.7 and Figure A.6 show the parallel efficiency of different configurations to their sequential performance on CoolMUC2 and CoolMUC3. We plot both for the total simulation time as well as for just the time spent on computing the pairwise forces.

AutoPas leads in this metric compared to the OMP and Kokkos package, when running with one thread per physical core. It reaching about 50-60% efficiency looking at total simulation time, and even more than 90% for the force computation. Enabling tuning reduces the efficiency quite dramatically especially for less than 14 threads. When using the bayesian search the efficiency still stays around or above OMP and Kokkos for more than 14 threads though. On the CoolMUC3 even the full search is more efficient than the references for 64 or more threads.

These performance drop-offs introduced by auto-tuning might disappear when running longer simulations so that the amount of time spent on auto-tuning and therefore on computing with slower configurations is less relevant.

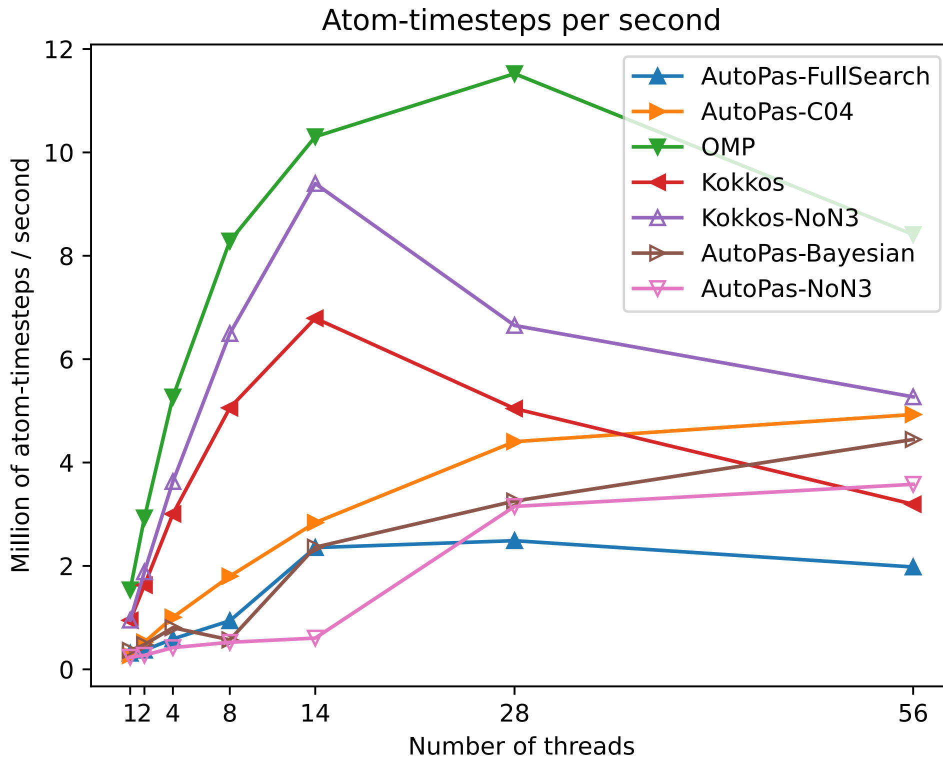


Figure 5.4.: **Atom-timesteps per second on CoolMUC2 (higher is better):** The OMP package outperforms both Kokkos and AutoPas in any configuration. Kokkos scales quite good up to 14 threads, which equals 7 threads per CPU. Using the full neighbor lists with Newton3 disabled shows better performance than using half neighbor lists with Newton3. AutoPas is the slowest of the compared methods but not by a large factor. It is also the only accelerator that still gains performance from utilizing the hyper-threads (56 threads). Disabling Newton3 optimizations only has disadvantages for AutoPas contrary to Kokkos.

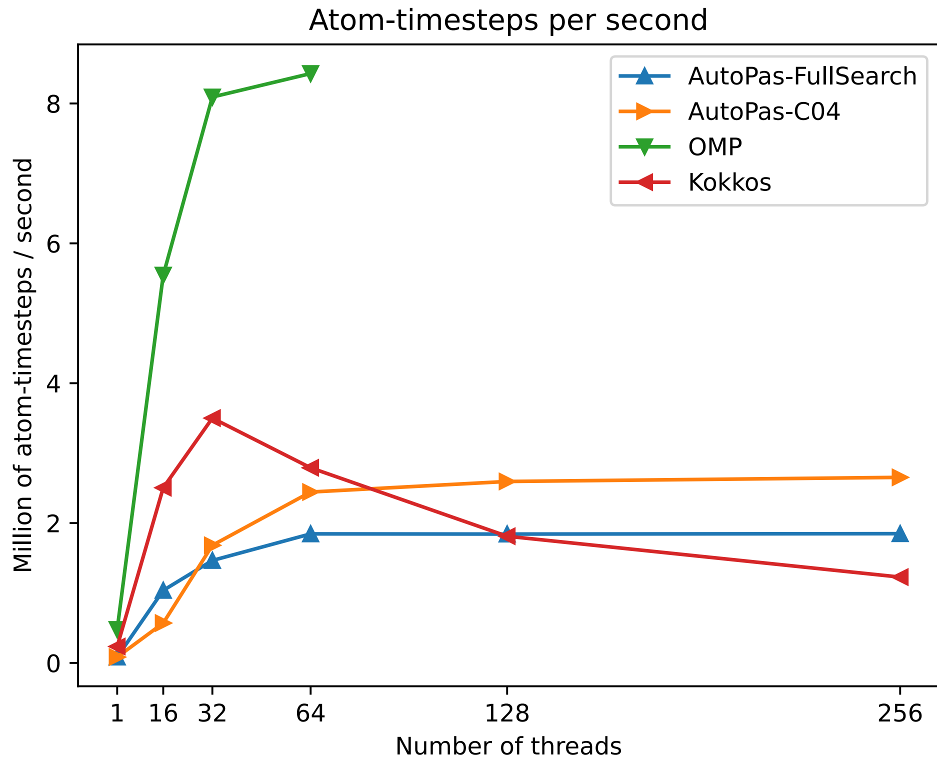


Figure 5.5.: **Atom-timesteps per second on CoolMUC3 (higher is better):** The OMP package is clearly the fastest accelerator here, but it produces a segmentation failure with 128 or 256 threads for this large scenario. Kokkos is a little bit faster than AutoPas but does not scale past 32 cores. AutoPas can utilize the hyper-threads (128, 256) slightly but does not gain a lot from them.

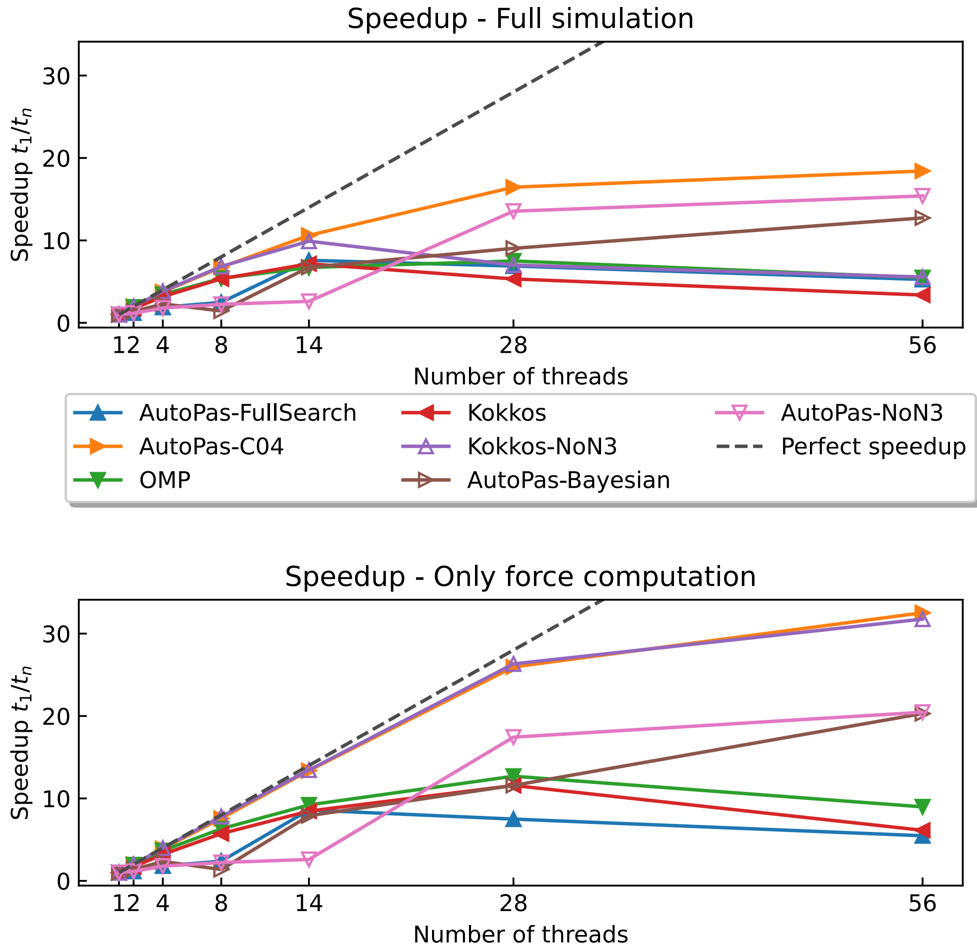


Figure 5.6.: **Speedup on CoolMUC2 (higher is better):** The upper plot shows the speedup for the total simulation, the lower one when calculated just using the force computation times.
AutoPas can reach quite a good speedup when limited to a single configuration. AutoPas-C04 is always effective but the AutoPas-NoN3 configuration only improves for more than 14 cores. With auto-tuning, the speedup is comparable to OMP and Kokkos for high thread counts but quite bad for a small number of threads. The speedup drops off in general for more than 14 cores.
Looking just at the forces shows that the AutoPas-C04 as well as Kokkos-NoN3 configuration scale nearly perfectly up to 28 cores and only drop off when using hyper-threads. This shows that the theoretical peak performance is not limited by the force computation for those two configurations but more likely by communication. The force computation of the OMP package scales only slightly better than the whole simulation.

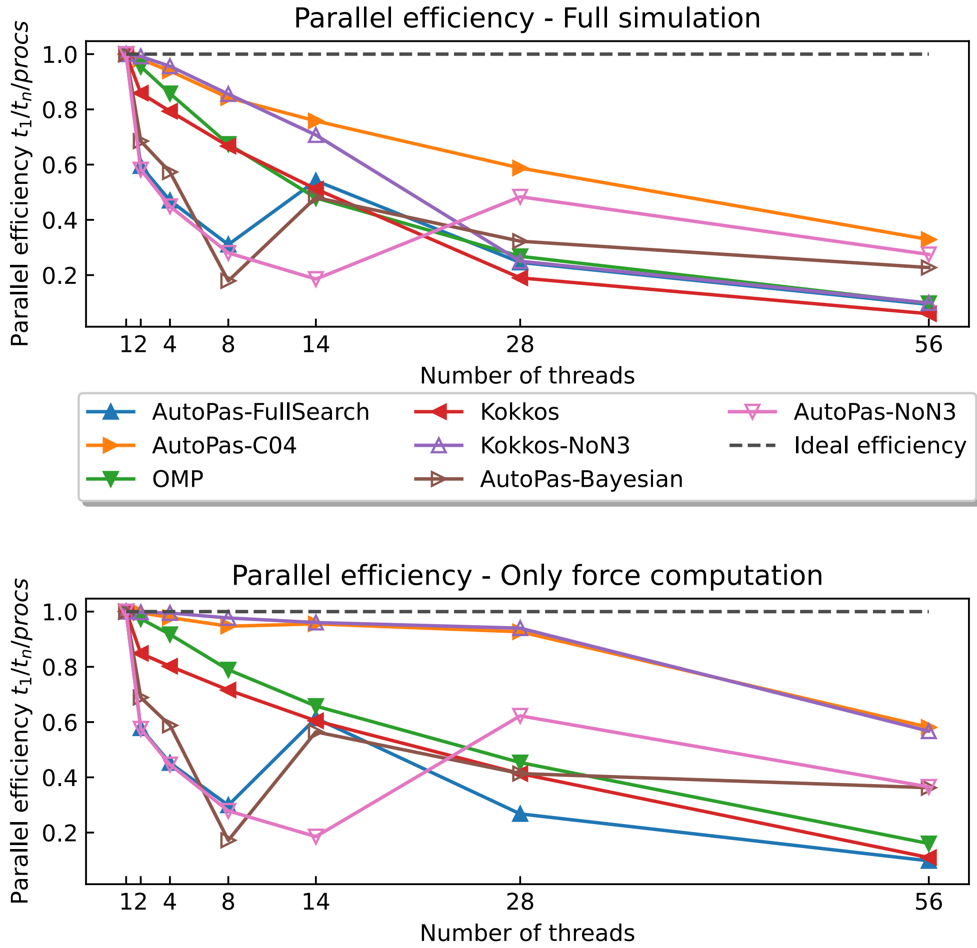


Figure 5.7.: **Parallel efficiency on CoolMUC2 (higher is better):** The upper plot shows the parallel efficiency when calculated for the total simulation time, the lower one when just looking at the time spent in computing the pairwise forces.
For the full simulation, only the parallel efficiency of AutoPas-C04 and Kokkos-NoN3 is very good for up to 14 cores, while the OMP and Kokkos configurations are acceptable. Afterward, only AutoPas with a fixed setting reaches sufficient parallelization efficiency. When auto-tuning is enabled AutoPas shows a surprisingly bad efficiency for low thread counts.
The efficiency of the pairwise forces shows that AutoPas-C04 and Kokkos-NoN3 can scale excellently but all other configurations are just slightly more efficient than the full simulation. Thus, communication overhead affects those two the most.

When comparing just the force speedups, we can see that AutoPas-C04 as well as Kokkos-NoN3 scale nearly perfectly up to the physical core count and just drop off when utilizing hyper-threads. This is a clear sign that the scaling issues we see with both are not introduced by the force calculation.

LAMMPS in general does not seem to be parallelizable super efficiently at node level just using OMP, especially for high thread counts. This matches findings of the USER-OMP package implementation and the current documentation: MPI parallelization is preferred and OMP is used to keep the number of MPI ranks manageable while still utilizing all cores. OMP also gets very useful for high node counts or when communication bandwidth is the limiting factor. [8]

Memory

Another metric for comparing the accelerator packages is the amount of main memory they use. While current supercomputers have a reasonable amount of RAM per node, there exist cache and bandwidth limitations that might influence performance. High memory usage might also limit the size of possible simulations.

While LAMMPS always uses verlet-lists, AutoPas offers multiple methods that can vary a lot in their memory consumption. Figure 5.8 and Figure A.1 show the memory used by the different configurations over varying thread counts.

We can see that the OMP packages and Kokkos scale linearly with the number of threads. AutoPas stays constant for the C04 configuration utilizing a *Linked Cells Container*, and the NoN3 configuration using a *Verlet List Cells* container.

Determining the memory usage for the auto-tuning configurations is more difficult as they can change over time. The values plotted are always for the configuration the tuner finally selected and thus was used for the majority of the simulation. Any *Linked Cells* based configuration resulted in very little memory usage and *Verlet Lists* based configurations used the most memory. See Figure 5.2 and Figure 5.3 for the exact configurations that were selected.

5.1.4. Causes for Loss of Performance

As we have seen in the previous section, AutoPas is a bit slower than Kokkos and especially the OMP package but shows a better speedup. We now want to take a deeper look into possible reasons that limit the performance of the AutoPas integration.

When we observe how the simulation time is spent we can see that the relative amount of communication increases significantly for more than 8 cores. This affects not only AutoPas but also Kokkos. See Figure 5.9 and Figure A.2 for plots of this from the runs on CoolMUC2 and CoolMUC3.

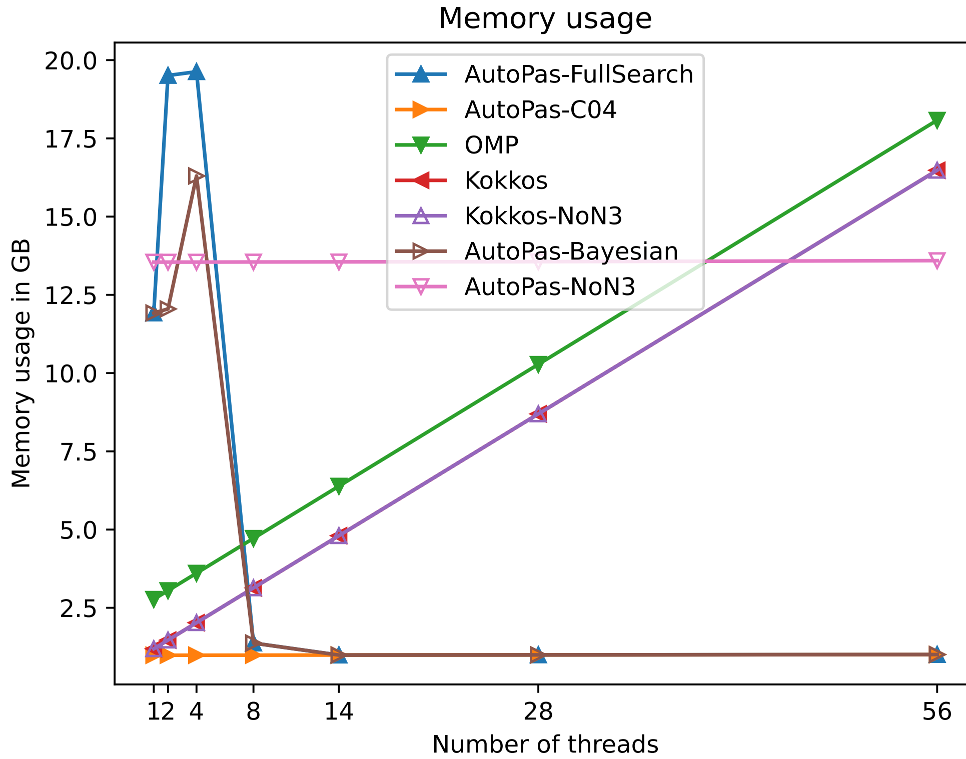


Figure 5.8.: **Memory usage on CoolMUC2 (lower is better):** The OMP and Kokkos packages that use *Verlet Lists* scale linearly with the number of threads. The memory usage of AutoPas highly depends on the internal data structure and can change over the simulation when auto-tuning is enabled. For the two configurations that perform auto-tuning only the memory usage of the configuration that was finally selected is plotted.
Therefore the memory usage can be anything from around 1GB for *Linked Cells* based configurations (here: AutoPas-C04) up to 20 GB for *Verlet Lists* (here: AutoPas with *Full Search* for 2 and 4 threads). The AutoPas-NoN3 run uses a *Verlet Lists Cells* configuration that results in constant, but relatively high memory usage.

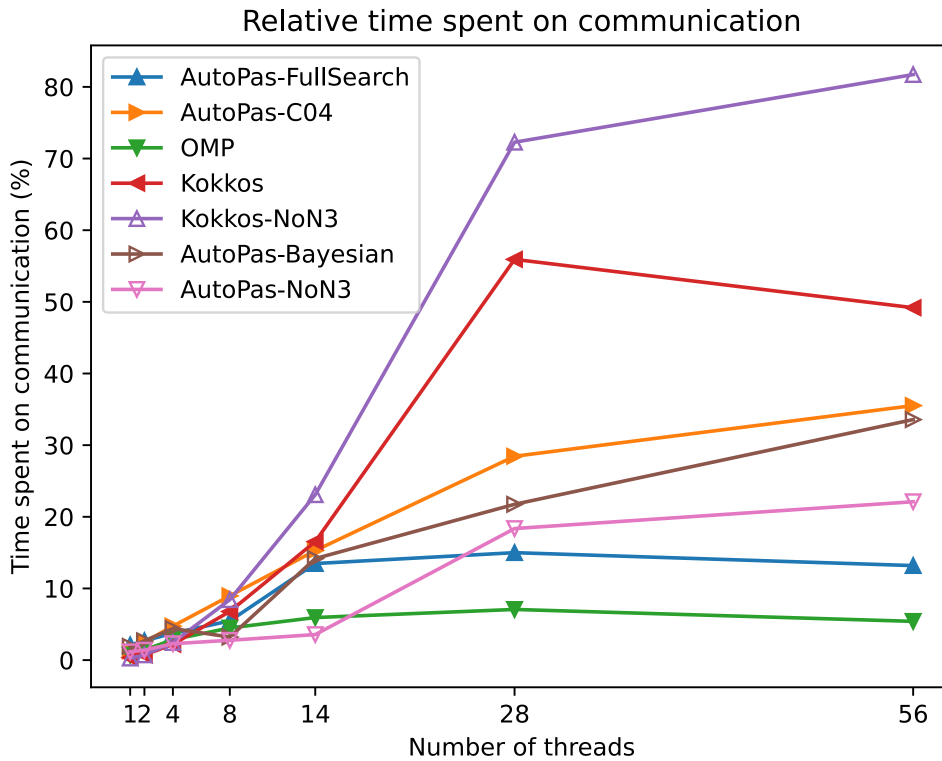


Figure 5.9.: **Relative time spent on communication on CoolMUC2 (lower is better):** This plot shows how much of the simulation time is spent on communication. Up to 8 threads, the amount of communication required is still below 10% for all configurations. Starting with 14+ threads, it quickly explodes. This is especially the case for Kokkos but also for AutoPas. Only the OMP package can steadily remain below 10%.

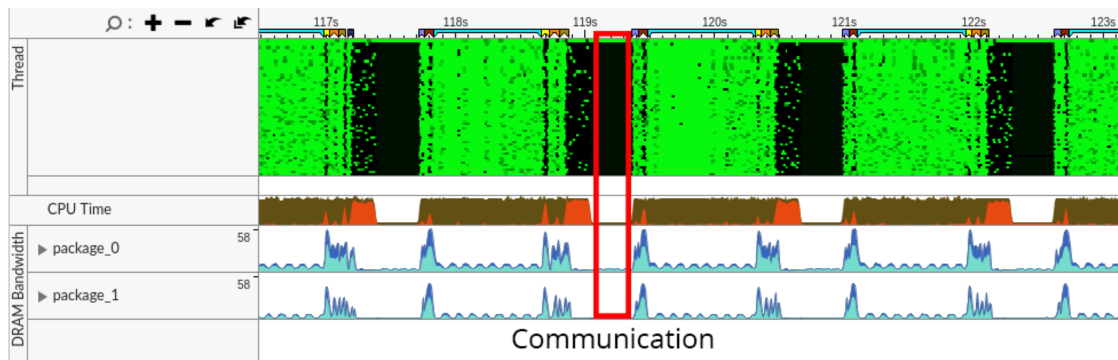


Figure 5.10.: **Intel VTune overview for the Lennard-Jones Melt scenario, the `C04` configuration, and 54 threads:** This excerpt from the profiler shows the parallel efficiency of the OMP threads as well as the memory bandwidth used. The red rectangle shows the time frame the communication code is executed for one iteration. This section is clearly neither compute nor memory- bound at the moment. Thus replacing this sequential section of the code with an optimized and parallelized version should bring a huge performance boost.

The communication of LAMMPS is split into four categories:

1. Sending of atom positions in steps where the neighbor lists are not rebuild
2. Sending ghost particles for periodic boundaries and neighboring processes
3. The force communication back to the particles these ghosts originate from
4. Exchanging particles with neighboring processes that leave the local domain

As we are not using multiple processes with MPI we have no particle exchange (4). And because AutoPas currently always calculates a *Full Shell*, i.e. particle interactions with the ghost layer are fully calculated in each process, no force exchange (3) is performed in the AutoPas configurations. Therefore, in each time step either (1) or (2) is performed.

When integrating AutoPas we had to modify the communication code a little bit but we kept it mostly as-is for simplicity. It is therefore not parallelized at all and uses a lot of slow index-based particle accesses (see Subsection 4.2.4).

Using the Intel VTune profiler for this scenario and 54 threads, we can see that this is indeed one problem resulting in reduced parallel performance (see Figure 5.10). Therefore the communication code should be replaced with a parallelized version which then can also remove the index-based accesses. For this mainly the `CommAutoPas` and `AtomVecAutoPas` classes have to be changed.

With the profiler, we can also see that about 33.5% of the time is spent in serial sections. This is for a large part connected to the AutoPas particle iterators. The performance of these can vary greatly depending on the internal container and might improve in the future.

5.2. Lennard-Jones Crack

5.2.1. Scenario Description

In this scenario a plane of atomic particles is modeled that is torn apart from one edge. For this, two groups of particles that do not interact with each other are attached to one edge of the plane such that a gap is created in the middle (see Figure 5.11). The edges orthogonal to the one that is torn are reinforced with a boundary also made up of particles but not affected by forces. One boundary is moving away from the other, tearing on the plane until it cracks.

At first, there appears a visible gap between the non-interacting particles (green and blue). Then a small crack will appear on one edge of the plane (red). Finally, the internal structure of the plane will shatter in multiple locations and its regular grid-like structure will deform.

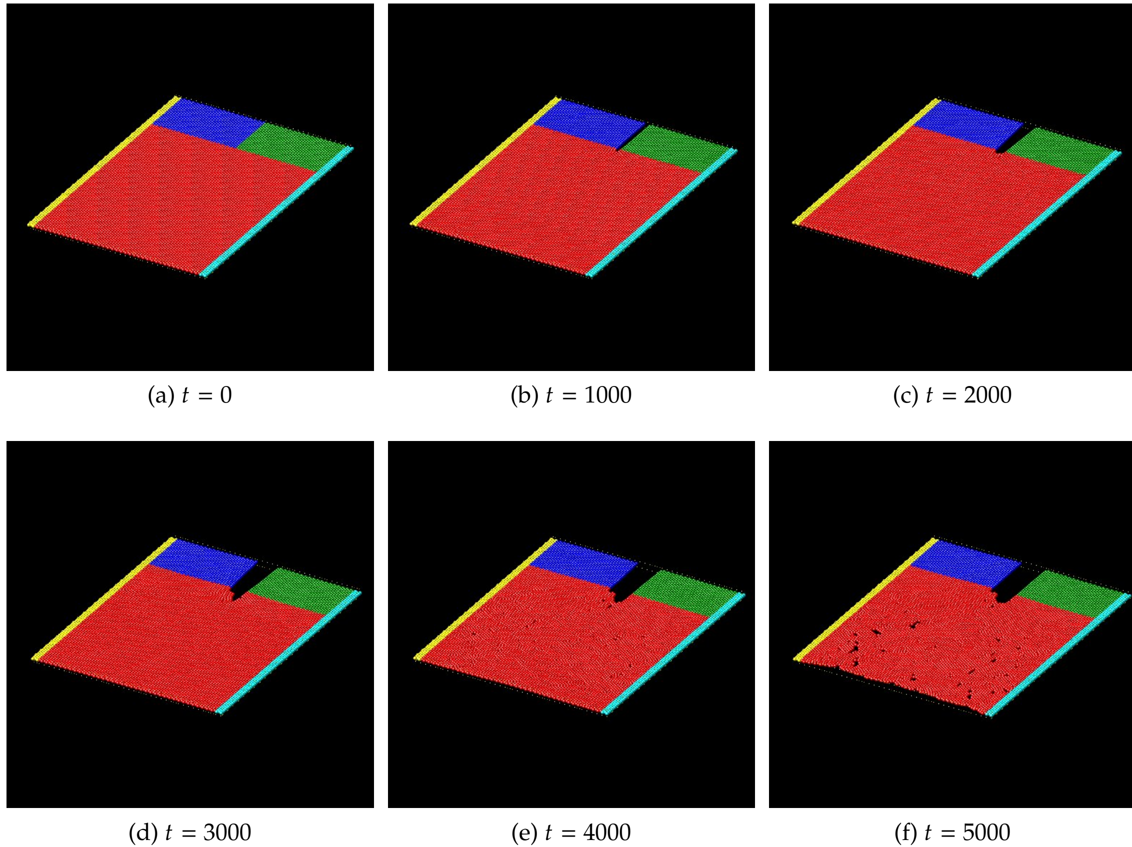
Forces are again modeled using the 12-6 Lennard-Jones potential $\sigma = \epsilon = 1$ and a cutoff radius of 2.5. The domain is shrink-wrapped (see Subsection 4.2.4) in both dimensions and scales with the farthest particles.

5.2.2. Necessary Styles & Features

The following styles and features had to be converted/implemented for this scenario in addition to the ones described in Subsection 5.1.2:

[Fix] enforce2d This fix explicitly sets the velocity and force of particles in the z-dimension to zero. This ensures the particles stay in a plane when running 2D simulations. As AutoPas only supports 3D simulations, we use this fix to run this 2D scenario inside a 3D domain.

[Fix] setforce With this fix the force of particles can be set to an arbitrary value. This is used to set the forces on the boundary particles (yellow and cyan in Figure 5.11) to zero so they can not be pushed by the particles of the plane. Thus the yellow boundary stays static and the cyan boundary is only moved by its initial velocity while pulling apart the plane.



(a) $t = 0$ (b) $t = 1000$ (c) $t = 2000$

(d) $t = 3000$ (e) $t = 4000$ (f) $t = 5000$

Figure 5.11.: **Lennard-Jones Crack Scenario:** The red plane of atomic particles is torn apart at the point of contact of the blue and green plane extension. The blue and green particles do not interact with each other which allows them to slide apart when the cyan boundary is moved away from the yellow one and hereinafter tearing on the red plane until it cracks. The boundaries are not affected by forces. For particle interactions, the 12-6 Lennard-Jones potential with $\sigma = \epsilon = 1$ and a cutoff radius of 2.5 is used.

[Compute] temp This computation calculates the temperature of a group of particles. It is used to compute the initial temperature of the mobile (red, blue, and green) particles. This temperature is then set as the target temperature of the thermostat.

Shrink-Wrapping Boundary This scenario uses a shrink-wrapped boundary which requires resizing the domain dynamically. The drawbacks of doing that with AutoPas were discussed in Subsection 4.2.4. As the edges are moving apart slowly in this scenario, the amount of required resizes is quite small.

Excluding Particle Interactions The blue and green particles (see Figure 5.11) should not interact with each other. LAMMPS solves this by not including them when building the neighbor lists. As AutoPas does not use LAMMPS's neighbor lists, the AutoPas functor now has to consider an interaction matrix. The drawbacks were discussed in Subsection 4.2.4.

5.3. Lennard-Jones Obstacle Flow

5.3.1. Scenario Description

This scenario models a 2D *Poiseuille flow* through a channel that contains two circular obstacles and widens over time. It uses atomic particles in a periodic and shrink-wrapped domain and a 12-6 Lennard-Jones potential with $\sigma = \epsilon = 1$ and a cutoff radius of 1.12246. The channel is bound in the shrink-wrapped dimension using particles as a wall. The forces applied to these particles are set to zero every step so they can not be moved by the particles in the channel. The channel is then filled with particles, except for the spaces taken up by the obstacles. See Figure 5.12 for a visualization.

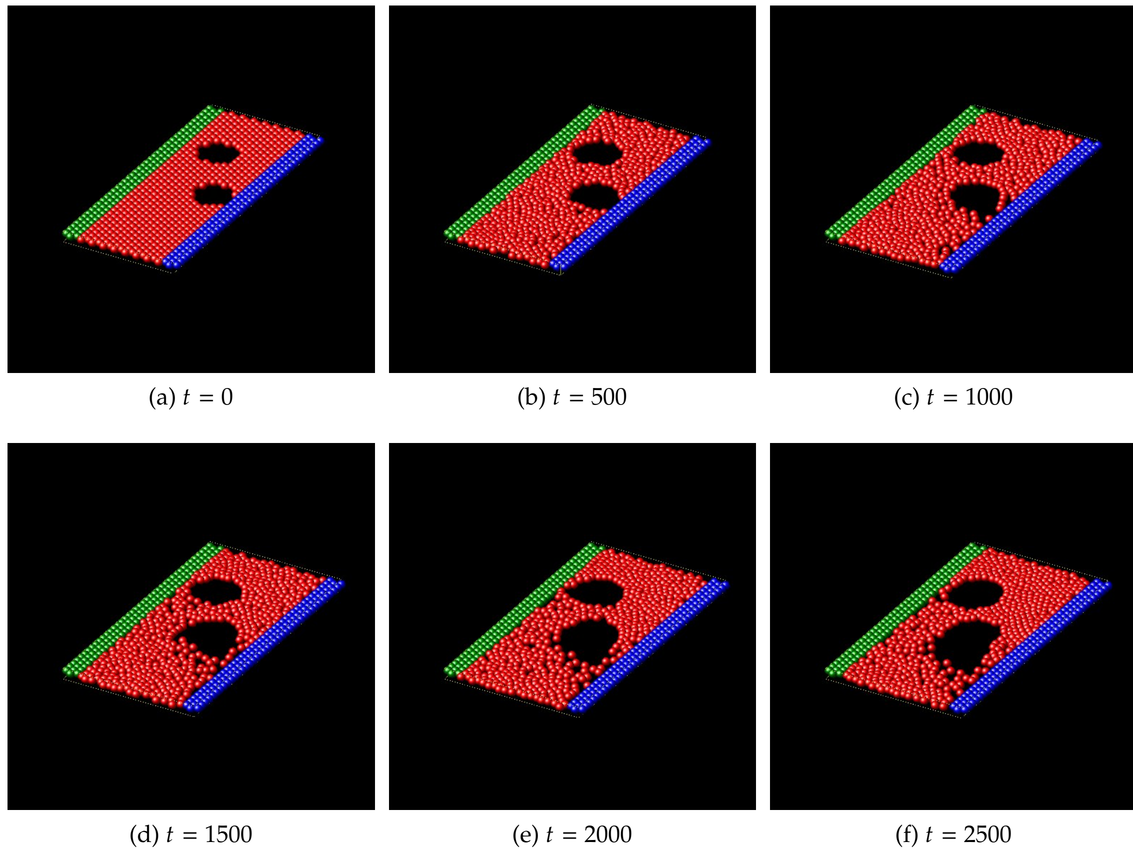
The particles of the flow start by moving around the obstacles filling in all available space. The wider the channel gets, the more they leave a shadow-like area behind the obstacles with a lower particle density.

5.3.2. Necessary Styles & Features

In addition to the styles mentioned in the previous sections ([Atom Style] atomic, [Pair Style] lj/cut, [Fix] nve, [Fix] enforce2d, [Fix] setforce, [Compute] temp), the following ones were converted:

[Fix] temp/rescale This fix explicitly rescales the velocities of particles to reset their temperature. Here it keeps the temperature of the particles in the flow constant.

[Fix] aveforce With this an external force can be applied over a group of particles. The existing forces of the particles are averaged component by component and the new



(a) $t = 0$ (b) $t = 500$ (c) $t = 1000$

(d) $t = 1500$ (e) $t = 2000$ (f) $t = 2500$

Figure 5.12.: **Lennard-Jones Obstacle Flow Scenario:** A 2D channel, which is bound by the green and blue particles, encloses the red particles that move along it, as well as two circular obstacles. The blue boundary is pushed over time to widen the channel. The domain is periodic in the direction of the channel and shrink-wrapped in the other. All particles are atomic and use a 12-6 Lennard-Jones potential with $\sigma = \epsilon = 1$ and a cutoff radius of 1.12246. Forces applied to the boundaries are negated.

external force is added. It is used to widen the channel over time by pushing the blue boundary particles away from the green ones.

[Fix] addforce The fix simply adds a constant force to particles every step. The flow is created with this by pushing the red particles along the channel while the thermostat keeps them at a constant velocity.

[Fix] indent This fix creates a cylindrical, spherical, or planar obstacle that repels particles. In this scenario, it is used to create the circular obstacles in the flow.

6. Evaluation of the Integration Effort

In this chapter, we want to give an overview of how much effort the AutoPas integration took and how much of LAMMPS had to be changed. For this, we analyze how many lines of code were added and removed using `git diff`. Listing 6.1 shows the diff summary comparing the current master with the original LAMMPS version. The following changes were excluded from the overview as they are not really a part of the core integration:

- New example inputs
- OMP data-sharing attributes added to the *USER-OMP* package that are necessary to compile when enabling both the *USER-OMP* and *USER-AUTOPAS* package.
- Other things like `.gitignore` and `.gitattributes`

The core files of LAMMPS consist of roughly around 220,000 lines of code, excluding any packages. The *KOKKOS* package itself is around 100,000 and the *USER-OMP* package around 80,000 additional lines.

Our integration required about 6000 changes in 56 files, of which only around 100 lines in 16 files are modifications of the original LAMMPS code. These are mostly needed to replace classes that are not part of the style system with AutoPas specific versions (see Subsection 4.2.2 for a list), similar to the Kokkos integration. Or they were minor changes required for overriding methods of styles in the AutoPas version (e.g., changing access specifiers from `private` to `protected`).

Everything else is new files, mostly inside the `src/USER-AUTOPAS` directory, which will not affect the compilation when the package is disabled. Currently, only a small selection of styles is supported though, compared to the other two packages.

Many of the new files, especially the fix and compute styles, are just a slightly modified copy of the base style with replaced particle accesses as described in Subsection 4.2.3. Additionally the `autopas_lj_functor.h` file implementing the Lennard-Jones functor used for the `lj/cut` pair style is mostly provided from AutoPas with some changes in the virial calculations as seen in Subsection 4.2.4.

Thus, removing all of these from the summary leaves us with about 3550 changed lines of code in 35 files (23 new) for the most basic form of the integration. In summary, it was not trivial to integrate AutoPas into LAMMPS, but taking a similar approach to the Kokkos accelerator package it is doable without major changes to the original code.

Duplicating code for the re-implementation of every style is not desirable but necessary and can be done following the simple step-by-step guide of Subsection 4.2.3. If required, this also allows for further fine-tuned optimizations of the code to utilize AutoPas to its fullest.

```
cmake/CMakeLists.txt                                |    6 +
cmake/Modules/Packages/USER-AUTOPAS.cmake (new)     |   80 ++
src/USER-AUTOPAS/atom_autopas.cpp (new)             |   11 +
src/USER-AUTOPAS/atom_autopas.h (new)               |   16 +
src/USER-AUTOPAS/atom_vec_atomic_autopas.cpp (new)  |  518 ++++++++
src/USER-AUTOPAS/atom_vec_atomic_autopas.h (new)    |   77 ++
src/USER-AUTOPAS/atom_vec_autopas.cpp (new)         |  204 +++
src/USER-AUTOPAS/atom_vec_autopas.h (new)           |   54 +
src/USER-AUTOPAS/autopas.cpp (new)                  |  608 +++++++++
src/USER-AUTOPAS/autopas.h (new)                    |  272 ++++
src/USER-AUTOPAS/autopas_lj_functor.h (new)         | 1326 ++++++++++++++++++++
src/USER-AUTOPAS/autopas_particle.h (new)           |   37 +
src/USER-AUTOPAS/comm_autopas.cpp (new)             |  471 +++++++
src/USER-AUTOPAS/comm_autopas.h (new)               |   50 +
src/USER-AUTOPAS/compute_temp_autopas.cpp (new)     |   86 ++
src/USER-AUTOPAS/compute_temp_autopas.h (new)       |   26 +
src/USER-AUTOPAS/domain_autopas.cpp (new)           |  507 ++++++++
src/USER-AUTOPAS/domain_autopas.h (new)             |   38 +
src/USER-AUTOPAS/fix_addforce_autopas.cpp (new)     |  155 +++
src/USER-AUTOPAS/fix_addforce_autopas.h (new)       |   23 +
src/USER-AUTOPAS/fix_aveforce_autopas.cpp (new)     |  149 +++
src/USER-AUTOPAS/fix_aveforce_autopas.h (new)       |   27 +
src/USER-AUTOPAS/fix_enforce2d_autopas.cpp (new)    |   65 +
src/USER-AUTOPAS/fix_enforce2d_autopas.h (new)      |   24 +
src/USER-AUTOPAS/fix_indent_autopas.cpp (new)       |  195 +++
src/USER-AUTOPAS/fix_indent_autopas.h (new)         |   24 +
src/USER-AUTOPAS/fix_nve_autopas.cpp (new)          |   40 +
src/USER-AUTOPAS/fix_nve_autopas.h (new)            |   29 +
src/USER-AUTOPAS/fix_setforce_autopas.cpp (new)     |  131 ++
src/USER-AUTOPAS/fix_setforce_autopas.h (new)       |   27 +
src/USER-AUTOPAS/fix_temp_rescale_autopas.cpp (new) |   83 ++
src/USER-AUTOPAS/fix_temp_rescale_autopas.h (new)   |   24 +
src/USER-AUTOPAS/neighbor_autopas.cpp (new)         |  174 +++
src/USER-AUTOPAS/neighbor_autopas.h (new)           |   19 +
```

```
src/USER-AUTOPAS/output_autopas.cpp (new)           |   25 +
src/USER-AUTOPAS/output_autopas.h (new)             |   24 +
src/USER-AUTOPAS/pair_lj_cut_autopas.cpp (new)      |   59 +
src/USER-AUTOPAS/pair_lj_cut_autopas.h (new)        |   41 +
src/USER-AUTOPAS/verlet_autopas.cpp (new)           |   41 +
src/USER-AUTOPAS/verlet_autopas.h (new)             |   31 +
src/accelerator_autopas.h (new)                     |   86 ++
src/domain.h                                        |    8 +-
src/dump_custom.cpp                                 |    6 +-
src/finish.cpp                                      |    2 +
src/fix_addforce.h                                  |    2 +-
src/fix_aveforce.h                                  |    2 +-
src/fix_enforce2d.cpp                               |    4 +-
src/fix_indent.h                                    |    2 +-
src/kspace.cpp                                      |    2 +-
src/lammps.cpp                                      |   44 +-
src/lammps.h                                        |    4 +
src/neighbor.h                                      |    3 +-
src/output.h                                        |    9 +-
src/suffix.h                                        |    1 +
src/timer.h                                         |    2 +-
src/variable.cpp                                    |   12 +-
56 files changed, 5959 insertions(+), 27 deletions(-)
```

Listing 6.1: **Git diff summary showing all changed files:** For existing files, the combined number of additions and deletions, and for new files, the total number of lines in them are shown on the right. Plus means additions and minus signals deletions.
In total, the core part of the integration took about 6000 lines of codes. Roughly half of it is modified copies of LAMMPS styles and the AutoPas Lennard-Jones functor. Only about 100 lines were changed in the original LAMMPS code.

7. Conclusion & Future Work

AutoPas was successfully integrated into LAMMPS as an accelerator and can be used like any other parallelization method LAMMPS offers. While making all styles and functionality LAMMPS provides compatible with AutoPas is out of scope for this thesis, they can be simply converted following the steps described in Subsection 4.2.3. This has been tested by converting some example scenarios bundled with LAMMPS as shown in chapter 5.
Nevertheless, there are some functionalities like the shrink-wrapping boundaries that can only be adapted to work with AutoPas by making more significant changes and might come with performance drawbacks.
When comparing the node-level performance of the AutoPas version with the OMP parallelizations provided in the *USER-OMP* and *KOKKOS* (OMP backend) packages, we find that AutoPas is consistently slower than the references but only by a small factor. It shows excellent scaling for high core counts though and provides some configurations with a significantly smaller memory footprint.
While the MPI code was mostly adapted for AutoPas, it requires more work and testing to perform comparisons of runs spanning multiple nodes. As the communication code in general often accesses particles by random access, it is a performance bottleneck and requires rewriting it from scratch for an optimized AutoPas version.
We only compared the performance of AutoPas on CPUs but GPUs are also supported. One might consider comparing it against the GPU accelerator or KOKKOS accelerator (GPU backend) of LAMMPS.
There are a lot of classes that are untested for their AutoPas compatibility and will often require an adaption. These additional challenges provide a basis for possible future research.

A. Appendix

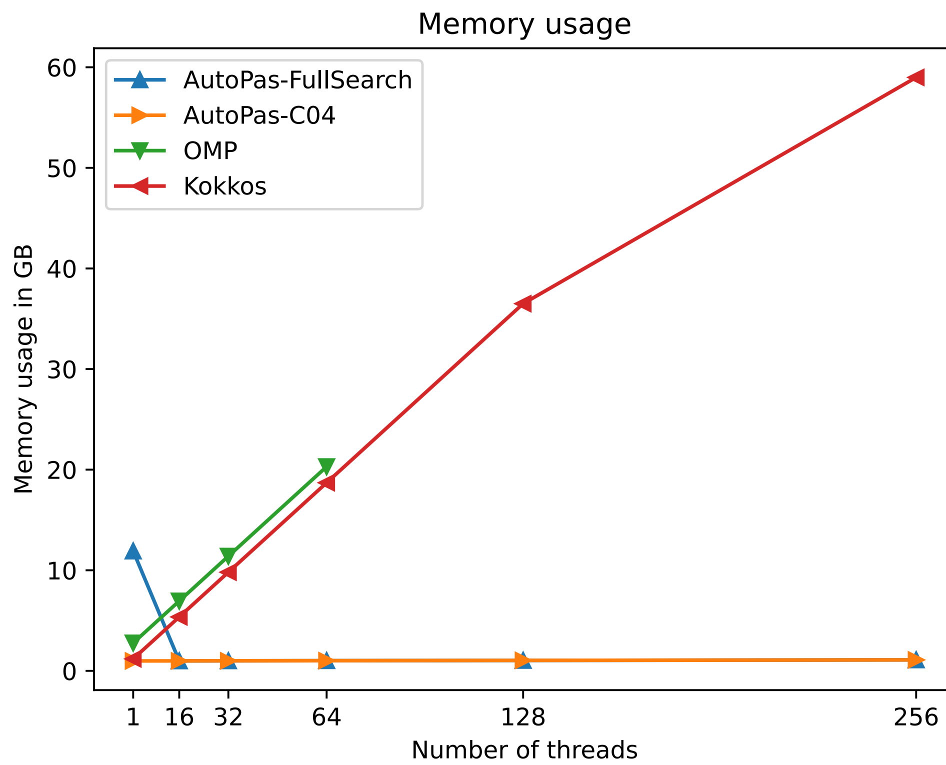


Figure A.1.: **Memory usage on CoolMUC3 (lower is better):** See Subsection 5.1.3

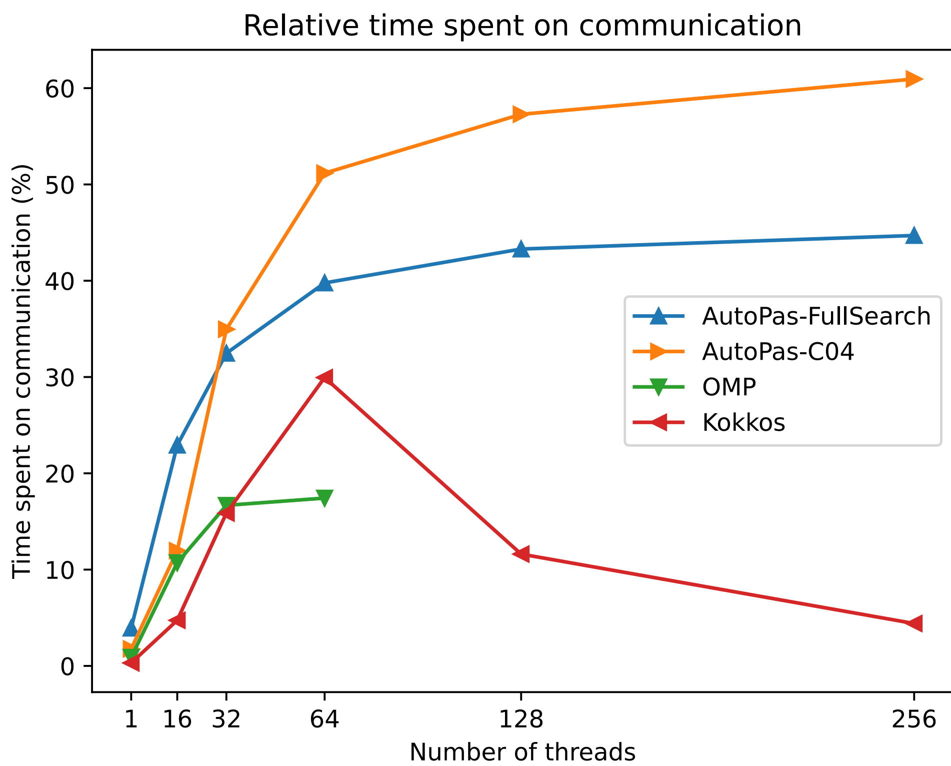


Figure A.2.: **Relative time spent on communication on CoolMUC3 (lower is better):** See Subsection 5.1.4

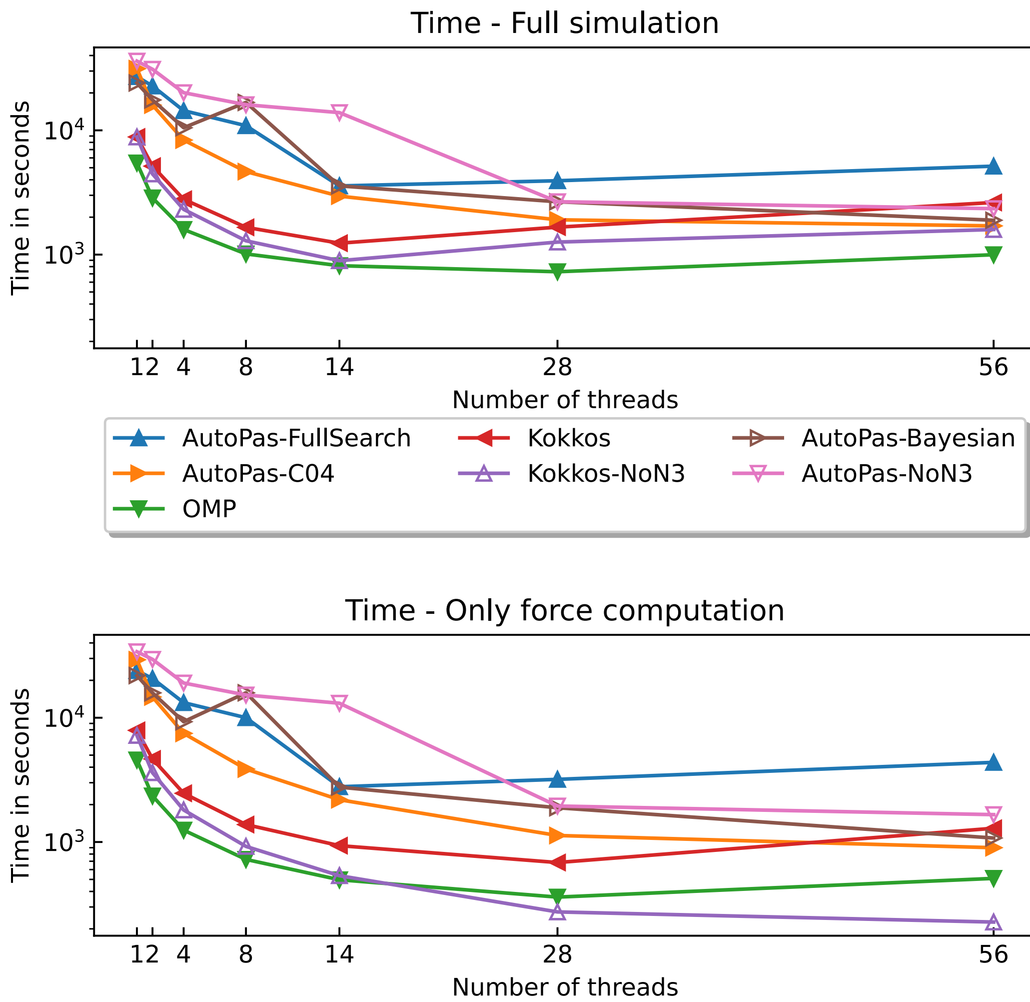


Figure A.3.: **Simulation times on CoolMUC2 (lower is better):** See Subsection 5.1.3

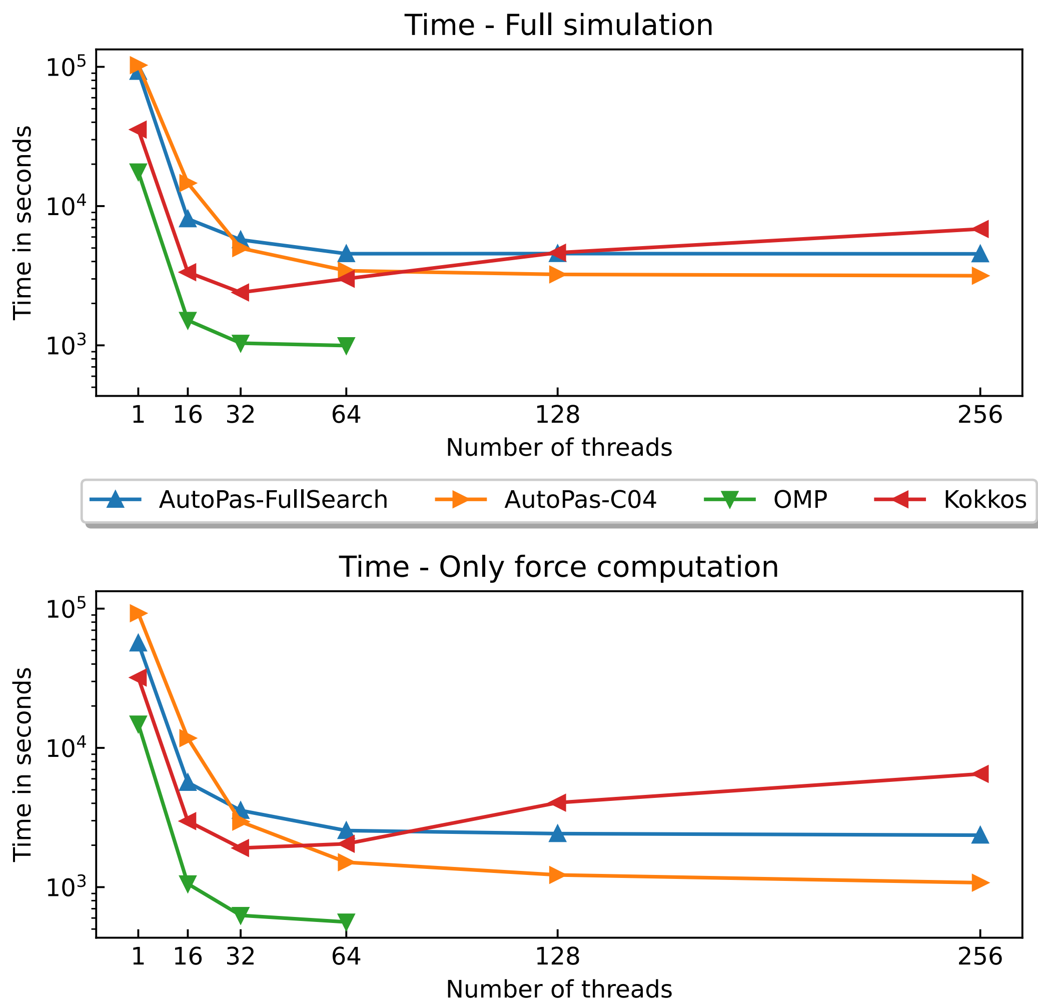


Figure A.4.: **Simulation times on CoolMUC3 (lower is better):** See Subsection 5.1.3

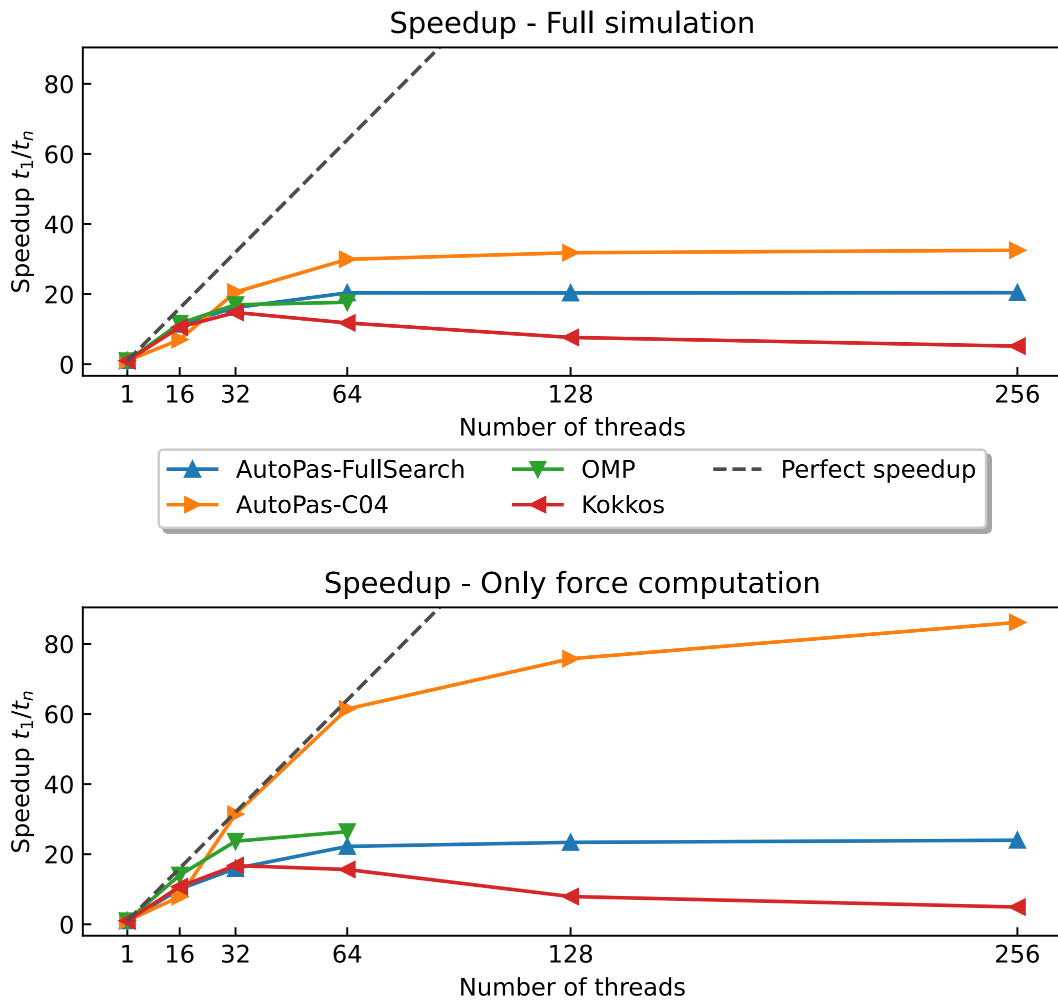


Figure A.5.: **Speedup on CoolMUC3 (higher is better):** See Subsection 5.1.3

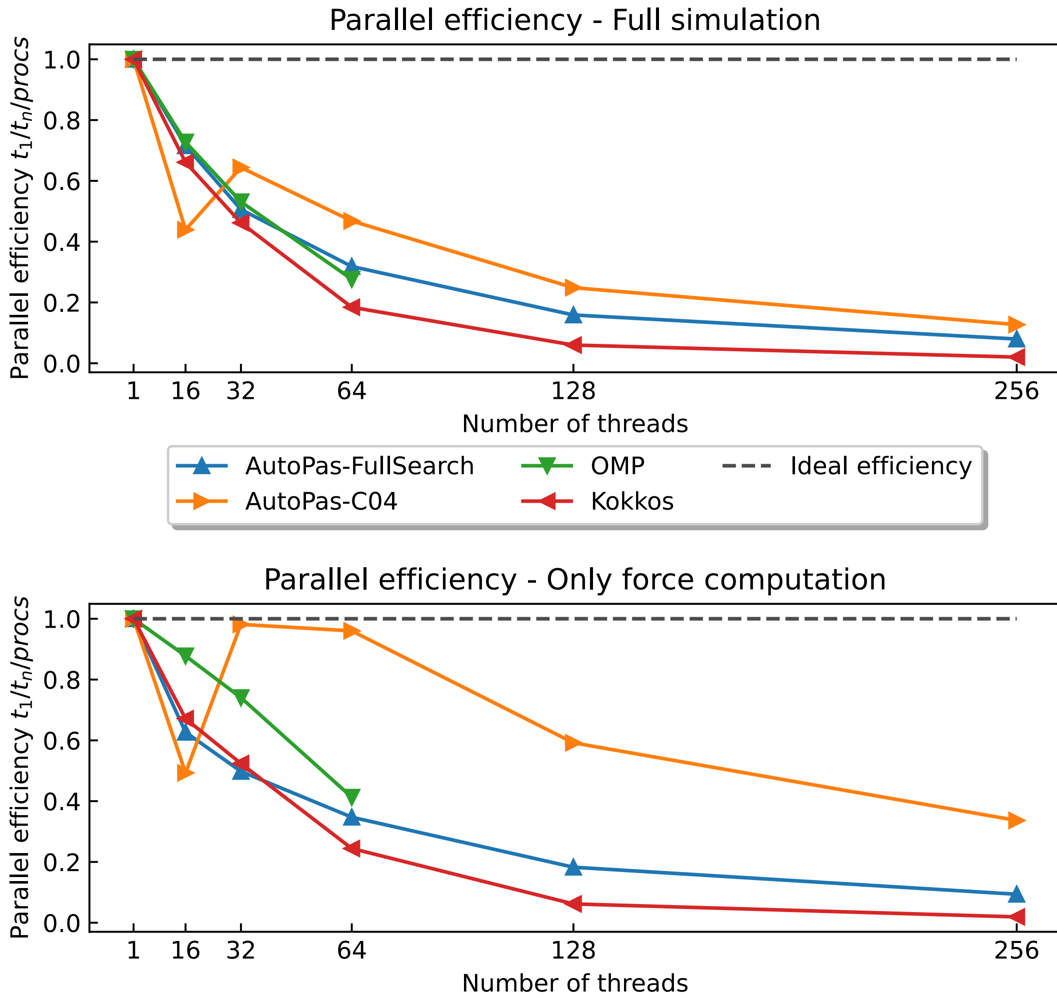


Figure A.6.: **Parallel efficiency on CoolMUC3 (higher is better):** See Subsection 5.1.3

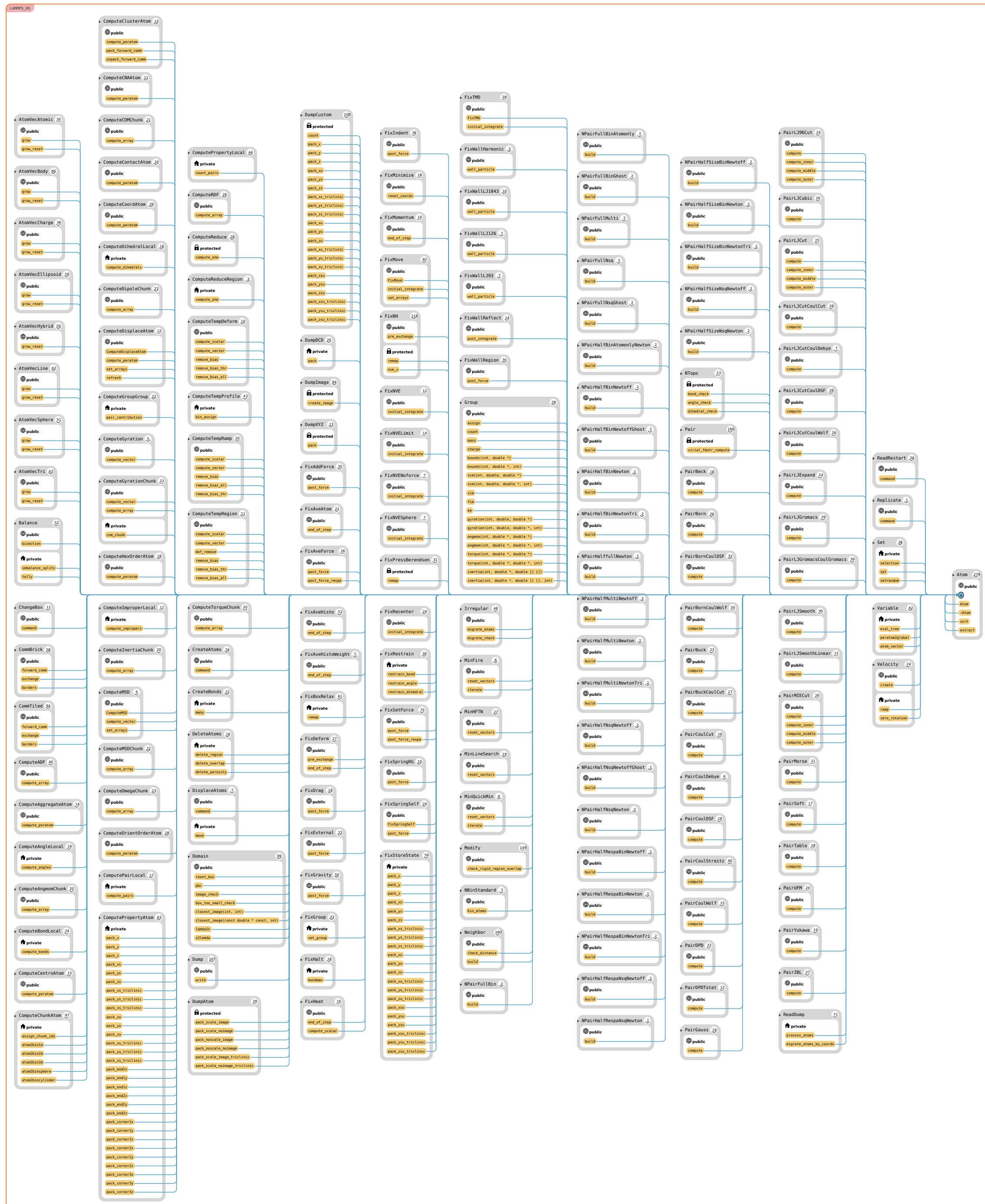


Figure A.7.: **LAMMPS - Direct accesses to the particle positions:** This figure shows all classes and methods that directly access the array storing the particle positions. In total there are 413 references in 175 different core files of LAMMPS, not including any optional packages.

List of Figures

2.1. Lennard-Jones Potential 3

3.1. LAMMPS - Verlet::run method 6
3.2. AutoPas - Public interface 8

5.1. Lennard-Jones Melt Scenario 21
5.2. Auto-tuner configuration selection CoolMUC2 24
5.3. Auto-tuner configuration selection CoolMUC3 25
5.4. Atom-timesteps per second on CoolMUC2 27
5.5. Atom-timesteps per second on CoolMUC3 28
5.6. Speedup on CoolMUC2 29
5.7. Parallel efficiency on CoolMUC2 30
5.8. Memory usage on CoolMUC2 32
5.9. Relative time spent on communication on CoolMUC2 33
5.10. Intel VTune overview for the Lennard-Jones Melt, C04, 54 threads 34
5.11. Lennard-Jones Crack Scenario 36
5.12. Lennard-Jones Obstacle Flow Scenario 38

A.1. Memory usage on CoolMUC3 44
A.2. Relative time spent on communication on CoolMUC3 45
A.3. Simulation times on CoolMUC2 46
A.4. Simulation times on CoolMUC3 47
A.5. Speedup on CoolMUC3 48
A.6. Parallel efficiency on CoolMUC3 49
A.7. LAMMPS - Direct accesses to the particle positions 50

List of Listings

4.1. CMake file defining the AutoPas package 11
4.2. FixTempRescaleAutoPas header file 13
4.3. Fallback implementation when AutoPas is uninitialized 14
4.4. FixTempRescaleAutoPas source file 16

6.1. Git diff summary showing all changed files 42

List of Tables

4.1. AutoPas accelerator options 19

5.1. Specifications of the CoolMUC2 and CoolMUC3 clusters 20

Notes

1. https://lammps.sandia.gov
2. https://wr.informatik.uni-hamburg.de/research/projects/talpas/start
3. https://github.com/ssauermann/lammps-autopas/
4. https://github.com/AutoPas/AutoPas
5. https://www5.in.tum.de/AutoPas/doxygen_doc/master/
6. https://lammps.sandia.gov/doc/Speed_omp.html
7. https://lammps.sandia.gov/doc/Speed_kokkos.html
8. https://www.lrz.de/english/
9. https://doku.lrz.de/display/PUBLIC/CoolMUC-2
10. https://doku.lrz.de/display/PUBLIC/CoolMUC-3
11. https://lammps.sandia.gov/bench.html

Bibliography

[1] K. Binder. *Monte Carlo and molecular dynamics simulations in polymer science*. Oxford University Press, 1995.

[2] A. A. Chialvo and P. G. Debenedetti. "On the use of the Verlet neighbor list in molecular dynamics." In: *Computer Physics Communications* 60.2 (1990), pp. 215–224. ISSN: 0010-4655. DOI: https://doi.org/10.1016/0010-4655(90)90007-N.

[3] H. C. Edwards, C. R. Trott, and D. Sunderland. "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns." In: *Journal of Parallel and Distributed Computing* 74.12 (2014). Domain-Specific Languages and High-Level Frameworks for High-Performance Computing, pp. 3202–3216. ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2014.07.003.

[4] F. A. Gratl, S. Seckler, N. Tchipev, H.-J. Bungartz, and P. Neumann. "AutoPas: Auto-Tuning for Particle Simulations." en. In: *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. Rio de Janeiro: IEEE, May 2019. ISBN: 9781728135106. DOI: 10.1109/ipdpsw.2019.00125.

[5] M. Griebel, S. Knapek, and G. Zumbusch. *Numerical simulation in molecular dynamics*. Springer Berlin, 2007.

[6] W. F. van Gunsteren and H. J. C. Berendsen. "Computer Simulation of Molecular Dynamics: Methodology, Applications, and Perspectives in Chemistry." In: *Angewandte Chemie International Edition in English* 29.9 (1990), pp. 992–1023. DOI: 10.1002/anie.199009921. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/anie.199009921.

[7] M. Karplus and G. A. Petsko. "Molecular dynamics simulations in biology." In: *Nature* 347.6294 (1990), pp. 631–639.

[8] A. Kohlmayer. "Implementation of Multi-level parallelism in LAMMPS for improved scaling on petaflops supercomputers." In: *TMS Annual Meeting, San Diego, CA, USA*. Vol. 27. 2011.

[9] W. Mattson and B. M. Rice. "Near-neighbor calculations using a modified cell-linked list method." In: *Computer Physics Communications* 119.2 (1999), pp. 135–148. ISSN: 0010-4655. DOI: https://doi.org/10.1016/S0010-4655(98)00203-3.

[10] S. Plimpton. “Fast Parallel Algorithms for Short-Range Molecular Dynamics.” In: *Journal of Computational Physics* 117.1 (1995), pp. 1–19. ISSN: 0021-9991. DOI: `https://doi.org/10.1006/jcph.1995.1039`.

[11] A. Simon, M. Rapacioli, M. Lanza, B. Joalland, and F. Spiegelman. “Molecular dynamics simulations on [FePAH]+ -complexes of astrophysical interest: anharmonic infrared spectroscopy.” In: *Phys. Chem. Chem. Phys.* 13 (8 2011), pp. 3359–3374. DOI: `10.1039/C0CP00990C`.

[12] L. Verlet. “Computer "Experiments" on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules.” In: *Phys. Rev.* 159 (1 July 1967), pp. 98–103. DOI: `10.1103/PhysRev.159.98`.

[13] Z. Yao, J.-S. Wang, G.-R. Liu, and M. Cheng. “Improved neighbor list algorithm in molecular simulations using cell decomposition and data sorting method.” In: *Computer Physics Communications* 161.1 (2004), pp. 27–35. ISSN: 0010-4655. DOI: `https://doi.org/10.1016/j.cpc.2004.04.004`.