AutoPas[⋆] in ls1 mardyn[,⋆⋆]: Massively Parallel Particle Simulations with Node-Level Auto-Tuning

Steffen Seckler[a,*], Fabio Gratl[a], Matthias Heinen[b], Jadran Vrabec[b], Hans-Joachim Bungartz[a], Philipp Neumann[c]

[a] *Technical University of Munich, Department of Informatics, Boltzmannstr. 3, 85748 Garching b. München, Germany*
[b] *Technical University of Berlin, Thermodynamics and Process Engineering, Ernst-Reuter-Platz 1, 10587 Berlin, Germany*
[c] *Helmut Schmidt University, Department of Mechanical Engineering, High Performance Computing, Holstenhofweg 85, 22043 Hamburg, Germany*

Abstract

Due to computational cost, simulation software is confronted with the need to always use optimal building blocks — data structures, solver algorithms, parallelization schemes, and so forth — in terms of efficiency, while it typically needs to support a variety of hardware architectures. AutoPas implements the computationally most expensive molecular dynamics (MD) steps (e.g., force calculation) and chooses on-the-fly, i.e., at run time, the optimal combination of the previously mentioned building blocks. We detail decisions made in AutoPas to enable the interplay with MPI-parallel simulations and, to our knowledge, showcase the first MPI-parallel MD simulations that use dynamic tuning. We discuss the benefits of this approach for three simulation scenarios from process engineering, in which we obtain performance improvements of up to 50%, compared to the baseline performance of the highly optimized ls1 mardyn software.

Keywords: `AutoPas`, `ls1 mardyn`, molecular dynamics, particle simulations, MPI, auto-tuning

1. Introduction

Molecular Dynamics (MD) simulations regularly go hand in hand with very cost-intensive computations to achieve accurate and reliable results. Different approaches have been followed to reduce the cost of these simulations. Efforts comprise, for example, tuning specific algorithms and programs to particular types of hardware [1, 2, 3, 4, 5, 6]. In some cases, computationally more favorable

⋆The source code can be found at `https://github.com/AutoPas/AutoPas`.
⋆⋆The source code can be found at `https://github.com/ls1mardyn/ls1-mardyn`.
*Corresponding author
Email address: `steffen.seckler@tum.de` (Steffen Seckler)
URL: `www5.in.tum.de` (Steffen Seckler)

coarse-grained models, such as continuum models or hybrid particle-continuum / multiscale models, can be applied [7, 8, 9].

In case of purely particle-based simulations, it has, however, already been shown that depending on hardware and simulation scenario different algorithms provide optimal performance [1]. Therefore, specific tuning efforts invested into one algorithm might prove inefficient as other algorithms might perform significantly better for a different MD scenario, even when they are not optimized to the same level.

In preceding work of our group [10], the idea of providing multiple algorithms and then tuning over these algorithms at runtime has been picked up for n-body simulations. We introduced the node-level library AutoPas, integrated it into the highly-parallel MD program ls1 mardyn [11] and demonstrated significant speed-ups for node-level simulations simply by switching between different traversal strategies.

In the present paper, we are going beyond node-level scope [10] of auto-tuned MD simulations, considering MD systems distributed on multiple compute nodes. Implications of this extension in contrast to a node-level system are outlined and discussed. We lay out modifications to AutoPas and ls1 mardyn that enable load-balanced, auto-tuned MD simulations on massively parallel compute clusters. We show that allowing each MPI rank to independently choose the optimal algorithm configuration can yield significant performance benefits especially for heterogeneous particle distributions.

Furthermore, we showcase performance results, load balancing and auto-tuning on the basis of three very different scenarios for which we achieved performance benefits of up to 50%.

1.1. Outline

In the next section, we give a short overview of related work, describing two libraries similar to AutoPas, give a short introduction to static and dynamic auto-tuning, and provide examples that use these techniques. We continue with a brief description of AutoPas and the modifications of AutoPas itself to allow for its usage in multi-node systems (Section 3). Section 4 introduces ls1 mardyn and provides information on adaptations of ls1 mardyn that were made in order to use AutoPas with its full functionality. After a short description of the three considered scenarios in Section 5, results of this integration are presented in Section 6. Finally, we draw conclusions and provide an outlook on the further development of AutoPas in Section 7.

2. Related Work

There is a great variety of particle simulation software available, including Gromacs [12], LAMMPS [13], NAMD [14], ESPResSo [15] for MD, and Gadget [16], GIZMO [17] and SPHysics [18] for astrophysics. With regard to providing particle simulation technology via a library approach, two MPI-parallel libraries

OpenFPM[1] and FDPS[2] have been published recently [19, 20]. In this section, we shortly introduce these two libraries, followed by a brief overview of auto-tuning techniques.

2.1. Particle Frameworks

Like AutoPas, both OpenFPM and FDPS try to help researchers in the development of particle-based simulations. In contrast to AutoPas, OpenFPM also supports mesh-based approaches and combinations of mesh and particle applications, while FDPS provides means for the calculation of long-range particle interactions. AutoPas features auto-tuning on building blocks for short-range MD, which the other two do not provide. All three of them have in common that the actual particle interaction is interchangeable and can be provided by the user. For this purpose, OpenFPM provides iterators over particles and their neighbors, while FDPS and AutoPas use a functor-like interface to inject the kernel into the library. The internally employed node-level data structures of the libraries also vary. FDPS uses an adaptive octree-based structure, OpenFPM relies on a (statically selected) Verlet or Cell-list, while AutoPas dynamically tunes over multiple data structures. In contrast to OpenFPM and FDPS, AutoPas focuses purely on the node-level and does not provide any MPI functionality.

2.2. Auto-Tuning

Static auto-tuning is performed once, at the latest, at the startup of program execution. This can include optimizations by the compiler, e.g., through profile-guided optimization[3], optimization of compiler flags [21], or static algorithm selection, e.g., through micro-benchmarking [22, 23]. In contrast to static auto-tuning, dynamic auto-tuning can be performed multiple times during a program execution to adapt to changes in the simulation domain. One framework that can be used for dynamic auto-tuning is Active Harmony [24, 25]. It provides a server-client infrastructure to explore a search space, where the server decides the next point in the search space that should be sampled by the client(s). Active Harmony uses the Nelder-Mead algorithm [26] to explore almost arbitrary search spaces and provides a relatively simple interface. We are currently in the process of integrating Active Harmony into AutoPas as one possible tuning algorithm, but this is not in the scope of this paper.

3. AutoPas: Enabling Auto-Tuning on Multi-Node Systems

In this section, we first describe AutoPas itself, followed by the modifications of AutoPas to allow for its usage in an MPI-parallel setup.

Container	Traversal	Advantages	Disadvantages	DLB	CS
Direct Sum	direct sum	low overhead for managing particles	$O(n^2)$, bad parallelizability		
Linked Cells	sliced	very low scheduling overhead	each chunk of work consists of at least two slices of cells	no	XL
	c01 1-way-coloring	only one barrier, best parallelizability	no support for Newton3, bad caching	yes	XS
	c04 4-way-coloring	4 barriers, caching, low scheduling overhead	lower parallelizability than c08	yes	M
	c08 8-way-coloring	best parallelizability with Newton3 support	more barriers than c04	yes	S
	c18		many barriers	yes	S-M
VL Global	verlet (v.)		poor Newton3 support	yes	XS
VL Cells	v.-sliced	very low scheduling overhead	bad for small domains	no	XL
	v.-c18		many barriers	yes	XS
	v.-c01	only one barrier	no Newton3 support	yes	S-M
VL Build	v.-build	good parallelizability	static LB not necessarily accurate	partial	

Table 1: Traversals implemented in AutoPas, specifying whether dynamic load balancing (DLB) is supported on the node level or not. A large work chunk size (CS) entails lower scheduling overhead, a small CS leads to better parallelizability. VerletLists (VL) use neighbor lists to mark potential interaction partners for the force calculation. These lists are then stored differently depending on the container: VL Global stores them globally, while VLCells stores them for each cell individually. VL Build uses a linked cells traversal for the list generation and stores the interacting pairs for each color and thread separately to prevent race conditions. Its load balancing is static and given by the load balancing of the list generation. As the VL containers use indirect and thus unordered memory accesses, they provide worse vectorizability compared to the linked cells container, but need to carry out fewer distance calculations.

3.1. AutoPas

AutoPas is a node-level library that enables dynamic auto-tuning on the node-level for short-ranged n-body simulations, by employing algorithm and parameter selection through the tuning of the following items:

Particle Container The container stores and administrates the actual particle data. In AutoPas, a DirectSum container (storing the particles in a global vector), a linked cells container and different variants of so-called Verlet or neighbor list containers [27, 28, 29] are implemented. Except for the first one, all reduce the neighbor interaction from an $O(n^2)$ to an $O(n)$ complexity by employing a cutoff radius for the short-ranged interactions.

Traversal The traversal defines the order in which pairs of particles inside of the container are processed for short-range force evaluations, including shared-memory parallelism. The purposes of the traversals are (a) to prevent race conditions without the need of atomics, force buffers or similar options, (b) to provide static or dynamic load balancing on the node-level and (c) to provide different levels of granularity for parallelization. An overview of the traversals in AutoPas can be found in Table 1. The traversals often use x-way-coloring on the cell structures for safe multi-threading (refer to [1] for examples).

Data Layout Two different data layouts are implemented in AutoPas. The Array of Structure (AoS) data layout always stores an array (or vector), wherein multiple particles are stored as a structure (or class). The Strucure of Arrays (SoA) data layout will convert the particle data into multiple arrays, one for each property, to allow for better vectorization (refer to [10] for details).

Newton3 This option allows to enable or disable the application of Newton's third law of motion; for particles i and j, the equality $F_{ij} = -F_{ji}$ is applied for the corresponding force F_{ij} acting between them. Enabling it reduces the number of necessary computations by roughly a factor of two. Disabling it increases the amount of possible parallelization, as a force calculation only results in a force update of one particle, because all particles can be updated in parallel.

Currently, AutoPas tunes the algorithm parameters at runtime at fixed time step intervals and tests the performance of all valid combinations of parameters, which we refer to as configuration. To minimize tuning overhead, AutoPas acquires performance samples over multiple time steps of the actual simulation. During each time step, the force calculation, which is defined through a so-called functor, is performed using one configuration and the time for its execution is

[1] `http://openfpm.mpi-cbg.de/`
[2] `https://github.com/FDPS/FDPS`
[3] `https://docs.microsoft.com/en-us/cpp/build/profile-guided-optimizations`

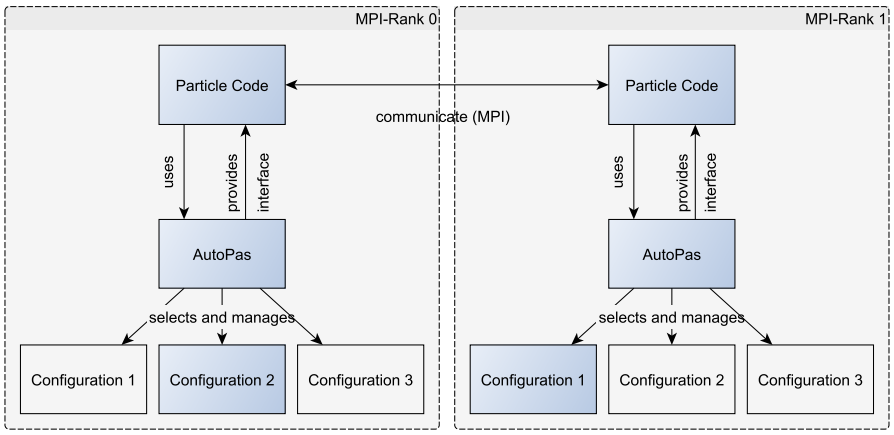


Figure 1: Overview of the usage of AutoPas in MPI-parallel particle simulations. For each MPI rank, a dedicated AutoPas object is used. Each AutoPas instance will then determine the optimal configuration – including the choice of container – on its own and select it.

used as one sample for the chosen configuration. When a specified amount of samples has been collected for the current configuration, the next configuration is tested. Once all configurations are tested, the configuration with the best performance is selected and used until the next tuning is triggered. Currently, AutoPas uses a fixed number of samples that can be input by the user. The sampling and auto-tuning process is designed to be oblivious to a user of AutoPas and thus hidden within the library.

3.2. One Common Interface

AutoPas is intended as a node-level library that does not handle any communication across multiple nodes. The communication is left to the user code (see Figure 1).

For AutoPas to be usable across multiple ranks of a particle simulation package, it is necessary to provide a common interface which AutoPas and all its internal containers abide to, allowing for a well-defined particle exchange. The need for this interface becomes apparent when the different containers used within AutoPas are inspected in more detail: On the one hand, linked cells containers typically require an update[4] of the container structure every time step. On the other hand, Verlet list containers, which store a list of potentially interacting particles for each particle, can only be efficient if the neighbor list structure is updated sufficiently rarely. This implies that particles contained in Verlet list containers have to remain in the same container over a couple of

[4]An update of the container is the resorting of every particle into the correct cell, that actually covers the current particle position. For Verlet lists, this entails a rebuild of the neighbor lists.

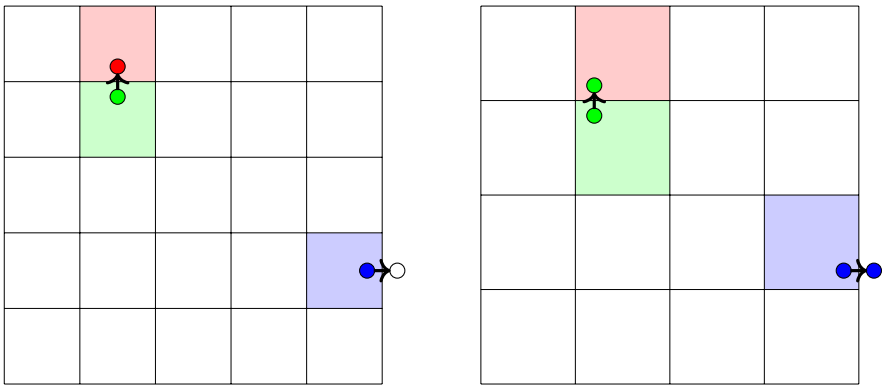


Figure 2: Illustration of the particle ownership and corresponding behavior in linked cells and Verlet list containers. The cells of the linked cells container are updated every time step, ensuring that the particles are always sorted into the correct cells. For the Verlet list container, the structure of the stored particles is only updated every few time steps. A particle with a specific color is owned by the cell with the same color. (a) Linked cells container. The particle is sorted into another cell or is removed from the container, if its position has changed. (b) Verlet lists container with underlying linked cells structure. The illustration shows an iteration without sorting: while the particle positions change, the particles are not sorted into other cells and are also not removed from the container. The cell size of Verlet lists (b) is larger than that of linked cells (a), as the cell width is at least $r_c + r_{skin}$, compared to r_c for linked cells.

time steps, even if their position is already slightly outside of the domain of the container, while particles that move outside of a linked cells container should be removed from it immediately. These differences are illustrated in Figure 2. Having differently acting containers in one MPI simulation complicates the particle exchange or disturb its performance. As we want AutoPas to be able to freely choose the container type, it is necessary to provide one common interface for all of them. For this purpose, two different approaches were considered: either using a linked cells-like approach, where every particle will belong to the process in which domain it resides, or using a Verlet list-like approach, where particles will only be moved between processes every few time steps. In this paper, we will refer to the former as AlwaysUpdateInterface and to the latter as RegularUpdateInterface.

First, we discuss the AlwaysUpdateInterface and its advantages, disadvantages and possible implementation variants. In this approach, a particle is owned[5] by a specific process if its position lies within the domain that is governed by that process. Therefore, leaving particles[6] have to be communicated among

[5] A particle is owned by exactly one process, which is the process that manages its particle position updates, etc. The particle may still appear as a ghost/halo particle in other processes.

[6] Particles that leave the domain governed by one process and enter the domain of another process are called leaving particles.

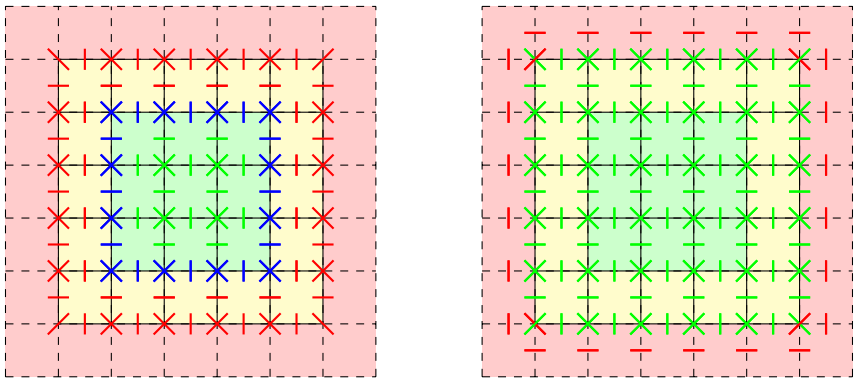
processes every time step. This is the default for linked cells-like containers and has no implications for the linked cells approach or for "direct summation", i.e., for brute-force n^2 particle-particle interaction evaluations. For Verlet-list containers, there are three variants to implement the AlwaysUpdateInterface interface: either (a) the neighbor lists have to be updated every time step, or (b) it needs to be explicitly tracked which process owns a particle, or (c) there has to be a mix between linked cells and Verlet list containers.

At first glance, variant (a) seems to completely destroy the performance of the Verlet lists, since rebuilding them is typically a rather expensive operation. However, this is not necessarily true, as only the container and the neighbor lists for particles close to the process boundaries need to be updated. Unfortunately, in the strong scaling limit for small domains, this will always be suboptimal. In addition, the implementation of a partial update has to be established for every position modifying particle operation and potential failures arising from an erroneous update will be comparably hard to debug, thus raising the software maintenance overhead.

Option (b) allows the Verlet list container to actually not update the inner state, except for marking specific particles as owned or not-owned. Therefore, no resorting of particles into cells or an update of the neighbor lists is required. This approach is viable, since relevant particles are already present as halo particles. However, neighbor lists also have to be built for all halo particles and the interactions of halo particles with other halo particles have to be considered, as their owned state might change without a change of the interaction lists. This, again, is suboptimal in the strong scaling limit because almost only halo-halo interactions are calculated in this case, which normally can be ignored. For Verlet list-like containers, storing the owned state explicitly is mandatory, as halo particles and particles that have left a container can otherwise not be distinguished. Consequently, particles may get lost or duplicated. For a linked cells container, the explicit tracking of owned particles is not necessary, as the owned state is given by the cell the particle resides in.

Option (c) is similar to option (a), but instead of rebuilding the interaction lists on the boundaries, those interactions can be calculated in a linked cells fashion. In the strong scaling limit, this is the same as using the linked cells algorithm and will thus provide the same performance. Option (c), however, provides a relatively complex data structure that needs to be managed, as some interactions have to be calculated using a cell-based approach, while other interactions need to be calculated using a neighbor list approach. The different options are illustrated in Figure 3. Options (a) and (c) are particularly difficult to implement for containers that use interaction lists of clusters (e.g. the novel VerletClusterLists in AutoPas that resemble the algorithms used in Gromacs [30]).

We now consider the usage of the RegularUpdateInterface. Using this interface, the container structure and (for Verlet lists) the according neighbor lists are only updated every few time steps. For linked cells this can be easily implemented by simply not updating the container structure and thus not resorting the particles. To maintain physical correctness, the cell size of the



(a), (c) (b)

Figure 3: Comparison of the three options for the AlwaysUpdateInterface in terms of the corresponding Verlet list implementation. Different cell types are marked with different colors: halo cells (red), boundary cells (yellow), inner cells (green). Option (a): Neighbor lists of particles that have potential interactions with particles in boundary or halo cells are rebuilt every time step (blue and red lines). The neighbor lists for the other cells remain untouched. Assuming that the memory location of particles inside the boundary cells is left untouched when other particles are removed or added to the container, one can additionally reuse the neighbor lists for interactions across cells that are indicated along the blue lines. Option (c) is very similar to option (a), but instead of rebuilding the neighbor lists at the boundary, the interactions on the boundary are directly calculated using the linked cells algorithm. Option (b): When a particle moves outside of a Verlet list container, the particle is marked as "non-owned" and remains in the container. A copy of this particle is sent to the other process. Using this mechanism all neighbor lists remain valid. This container's main downside is the additional need for interactions (red lines) among multiple halo cells, as particles inside of them can also be owned.

linked cells needs to be increased (if particles interact with those in neighboring cells only), such that all particle pair interactions are calculated, even if the position of the particle changed slightly over a few time steps. This increase in cell size will reduce the effective hit rate for force computations among particle pairs in (neighboring) linked cells due to the decreasing volumetric ratio $volume(\text{cutoff sphere}) : volume(\text{linked cell})$. For small particle densities, it will, however, reduce the overhead of iterating through the cells, as there are fewer. A RegularUpdateInterface is relatively easy to maintain, but has some drawbacks for the user, because the particles no longer necessarily lie in the container one would expect. Due to the imprecise location of the particles, the user will have to take extra care of the correct exchange of particles. Moreover, using a RegularUpdateInterface will specify the rebuild frequency over the entire domain and for all AutoPas objects and will not allow for a node-wise selection (or tuning) of the rebuild frequency.

Following this argumentation, we decided to implement the RegularUpdateInterface option, mainly due to better code maintainability (compared to options (a) and (c)) and since better performance is expected (compared to option (b)), even though this approach is a bit more advanced. Support for the

AlwaysUpdateInterface option (b) within AutoPas might still be considered in future works.

3.3. Interface Details

In this section, we describe the simulation loop of an MPI-parallel program using AutoPas. An outline of this loop is depicted in Listing 1. The interface

```
1  while(/*not done*/){
2    // 1. Update the container.
3    auto [invalidParticles, updated] = autoPas.updateContainer();
4    // 'invalidParticles' is now a std::vector of particles that should be removed
         ↪ from the container.
5    // 'updated' indicates whether the container was updated or not.
6
7    // 2. Exchange the leaving particles (if necessary)
8    // If 'updated' is false then 'invalidParticles' is empty and the exchange of
         ↪ leaving particles can be skipped.
9    if (updated) {
10         exchangeLeavingParticles(std::move(invalidParticles));
11   }
12
13   // 3. Exchange the halo particles.
14   exchangeHaloParticles(autoPas);
15
16   // 4. Calculate the pairwise interactions.
17   autoPas.iteratePairwise(functor);
18
19   // 5. Update position and velocities of the particles.
20   doPositionAndVelocityUpdate(autoPas);
21
22   // 6. Do other things (output, ...).
23 }
```

Listing 1: Outline of a simulation loop using AutoPas.

RegularUpdateInterface was implemented for AutoPas. Therefore, leaving particles will only be communicated among different MPI processes every few time steps. To make it easy for the user to decide whether leaving particles have to be communicated or not, the method *updateContainer()* of AutoPas will return both a *std::vector* of particles that have been removed from the container and an additional flag that indicates whether the container has been updated. This flag is provided to be able to skip the exchange of leaving particles. Without the flag, the user does not have a clear indicator whether or not a container update was performed because even in time steps in which the container is updated, no particle might actually have left the container. The flag guarantees that all

processes will execute the required communication steps. Internally, on a call to *updateContainer()*, AutoPas will update the container only if the neighbor lists have to be rebuilt in this step (indicated by the rebuild frequency) or if the configuration will be changed with the next call to *iteratePairwise()*.

After the potential exchange of leaving particles (l. 10), the halo particles need to be updated (l. 14). Therefore, the relevant particles have to be identified first, before they are sent to other processes. For the correct identification of the halo particles, AutoPas provides *RegionIterators* to efficiently and in an OpenMP-parallel way iterate over particles within a specific region. In addition, particles from other processes have to be received, before they are added to the container. For the RegularUpdateInterface, halo particles may already exist. In this case, the existing particle should be updated with the properties of the received particle. To combine these two actions into one, AutoPas was extended to provide the method *addOrUpdateHaloParticle(haloParticle)*, which will either add the halo particle (if the container has been updated in this step) or will find the appropriate, already existing, particle and update its position and other properties.

Once the halo has been successfully updated, the pairwise interaction can be executed using *iteratePairwise(functor)* (l. 17).

AutoPas further provides the function *updateinterfaceForced()* which enforces an update of the container, ignoring the rebuild frequency. This method is needed before rebalancing steps.

4. ls1 mardyn: Extensions and Load Balancing

In this section, we describe the modifications made in ls1 mardyn to support AutoPas and its RegularUpdateInterface in both single-node and multi-node runs. We therefore first shortly discuss ls1 mardyn itself and its previous changes [10]. Then we describe the changes that we implemented in ls1 mardyn to support the new interface of AutoPas in single and multi-node applications. Additionally, we describe new load balancing methods that were implemented in ls1 mardyn to enable the support of auto-tuned containers.

4.1. ls1 mardyn

ls1 mardyn is a highly-parallel MD library for chemical engineering and energy technology applications [11]. It is capable of simulating small rigid molecules up to extreme scales and was used for the largest MD simulation to date, containing more than $2 \cdot 10^{13}$ particles [1]. ls1 mardyn uses a linked cell data structure to handle efficient vectorization of the force calculations and utilizes OpenMP for an efficient node-level performance. It natively supports three OpenMP traversals, which are similar to the c08, sliced and c04 traversal of AutoPas. The traversal type can be selected upon program startup. Dynamic load balancing through MPI is supported by employing a k-d tree-based domain decomposition [31].

4.2. Previous AutoPas integration

As described in Ref. [10], ls1 mardyn was extended with an additional particle class and a wrapper around AutoPas, which replaces the previously used particle container. The particle class hereby implements both the interface needed by ls1 mardyn and the interface needed by AutoPas.

Additional changes include a new particle iterator, which was created to wrap the behavior of AutoPas and make it compatible with ls1 mardyn. For the force calculation, we use the functors provided by AutoPas instead of creating our own ones, as they provide all features needed for the scenarios considered in this paper.

4.3. Single-Node Modifications

With the new interface of AutoPas, a few additional changes to ls1 mardyn were necessary. For single-node usage, these included mainly the handling of the particle exchange on the periodic boundaries.

Leaving Particle Exchange

As the RegularUpdateInterface requires particles to remain inside their specific cells, the periodic boundary conditions are only allowed to be applied in those time steps in which the container is updated. This indicates that particles do no longer necessarily remain inside the domain, but can be slightly outside of it (by at most a distance of $r_{\text{skin}}/2$). Methods acting on specific sub-regions of the domain had to be updated to incorporate this change.

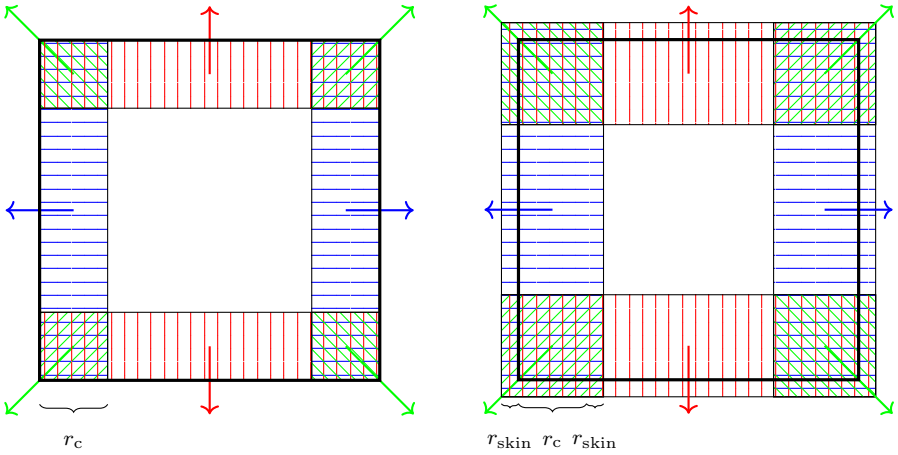
Halo Particle Exchange

In ls1 mardyn, halo particles need to be communicated in 26 different spatial directions (cf. Figure 4a). For the integration of AutoPas, these regions had to be enlarged by the skin radius r_{skin} in all directions (cf. Figure 4b). This change was necessary because (a) all interaction partners within $r_{\text{c}} + r_{\text{skin}}$ have to be known and (b) because particles can be slightly outside (by at most $r_{\text{skin}}/2$) of the domain bounding box. In principle, the extension of the regions in the direction outside of the domain only needs to be $r_{\text{skin}}/2$, but for simplicity reasons and because there are no severe negative performance impacts for the considered scenarios so far, we always enlarged the regions by r_{skin} in all spatial directions.

4.4. Multi-Node Modifications

To make ls1 mardyn compatible with the interface of AutoPas in MPI-parallel simulations, the neighbor exchange had to be modified similarly to the single-node case.

First, instead of communicating particles that move between processes in every time step, the communication is triggered only every N_{rebuild} steps. Second, the halo regions had to be adapted in a very similar fashion to the single-node case by increasing the halo regions by r_{skin} in all directions.



(a) Old behavior of the halo exchange (also new behavior if the skin is zero).

(b) Behavior of the halo exchange if the skin is non-zero.

Figure 4: Illustration showing the regions that include the boundary particles which need to be sent for a correct update of the halo particles and the spatial directions in which they need to be sent.

4.5. Load Balancing

For heterogeneous particle distributions, load balancing is necessary to ensure good performance. ls1 mardyn features k-d tree based load balancing that leverages and builds upon ls1 mardyn's linked cell structure [31]. This method has the advantage that in a single rebalancing step, a very good load balancing is achievable. The method, however, requires knowledge of the time needed for the calculations of every cell and is limited to the "cell view" – which might be too coarse in the strong scaling limit.

This knowledge is hard to obtain when using AutoPas, as, depending on the number of cells and properties of the other cells, different algorithms are used and thus the time spent for the calculation of a cell is dependent on other cells in the same partition. AutoPas does not provide information about the underlying data structure, which varies depending on the selected configuration. Moreover, k-d tree-based balancing may severely alter the topology within an MPI-parallel simulation. Hence, drastic fluctuations in the partitioning (ls1 mardyn) may change the locally optimal algorithm configuration (AutoPas). This makes the load of a subdomain unpredictable, rendering global load balancing infeasible.

However, if the partitioning varies only slightly from one time step to the next, the performance of the algorithms will only deviate marginally. This is typically the case in MD scenarios, where immediate drastic changes occur over hundreds of time steps in the worst case. Hence, other load balancing strategies may be preferable; an evaluation of various strategies can be found in Ref. [32]. We therefore extended ls1 mardyn to support diffusive load balancing, where changes to the subdomains are typically rather small, and thus, drastic changes

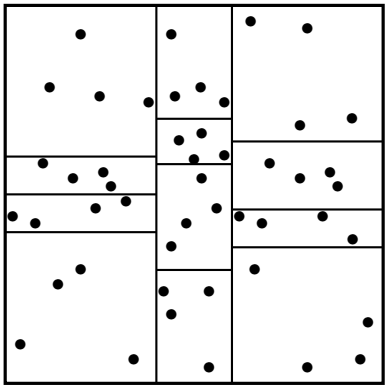


Figure 5: Multisection method for 12 ranks. First, the domain is divided into three parts along the horizontal direction, then each of these partitions is again divided into four subpartitions. In this schematic, the number of particles estimates the load of a subdomain.

of the performance for a specific partition are unlikely.

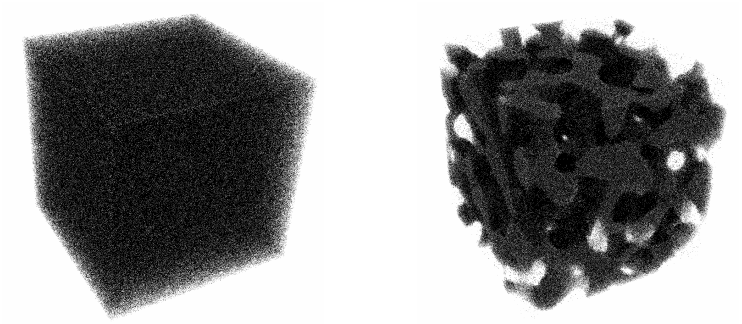
For load balancing, we are using the *A Load-balancing Library* (ALL)[7] that is developed at the Jülich Supercomputing Center and which provides a diffusive multisection method (called staggered grid in the library). This method is similar to an orthogonal recursive bisection method [33], but instead of always splitting a region into two parts, it allows splitting the domain in multiple subsections at once. It also limits the depth of the tree to the dimension of the domain, such that a splitting is only done once for each dimension. A schematic of that approach is depicted in Figure 5.

With the integration of diffusive load balancing, ls1 mardyn is no longer bound to cell-wise domain decomposition, allowing for more precise partitioning. In addition, the load no longer needs to be estimated for each cell, but is measured for an entire process instead. This makes the load calculations significantly easier, but cannot resolve the load distribution within an MPI process. The technical integration of ALL into ls1 mardyn was straightforward, as it is provided as an header-only library and the interface matches well with our previous load balancers. As the library does not share detailed information about the neighboring processes, information that describes the necessary communication partners is provided through global communication.

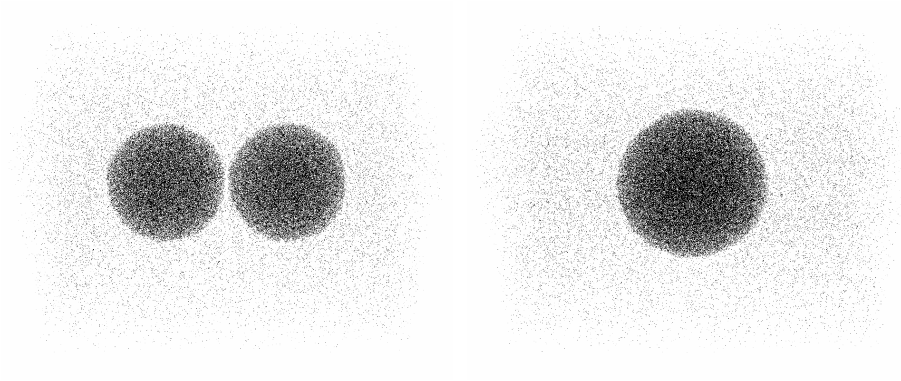
5. Scenario Descriptions

To test the performance of the load-balanced ls1 mardyn software with AutoPas integration, we considered a variety of scenarios. All of these scenarios are using single-site molecules that interact with the Lennard-Jones 12-6

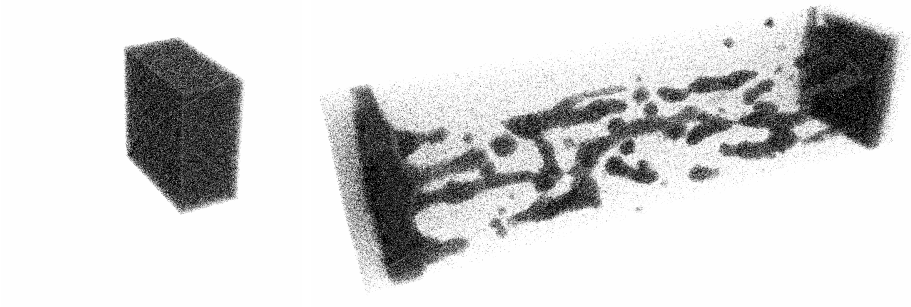
[7] `https://e-cam.readthedocs.io/en/latest/Meso-Multi-Scale-Modelling-Modules/modules/ALL_library/tensor_method/readme.html`



(a) Spinodal decomposition scenario at the start (l.) and the end (r.) of the simulation.
`https://youtu.be/yarl9028dEc`



(b) Droplet coalescence scenario at the start (l.) and the end (r.) of the simulation.
`https://youtu.be/1tlxDapmgbI`



(c) Exploding liquid scenario at the start (l.) and the end (r.) of the simulation.
`https://youtu.be/u7TE5KiSQ08`

Figure 6: Three scenarios that were studied in this work.

potential.

The first scenario is a **spinodal decomposition**, i.e., a system which is first equilibrated at a supercritical temperature and a density close to the critical one. Then, a velocity scaling thermostat is applied to set the temperature far below the critical one. The fluid immediately becomes unstable and decomposes into vapor and liquid phases. Throughout the simulation, the system thus changes from a homogeneous to a heterogeneous state (cf. Figure 6a). We have chosen this scenario to compare with results from Ref. [10].

The second scenario is a **droplet coalescence**. The initial configuration was set up with two neighboring liquid droplets in equilibrium with their surrounding vapor. During simulation, the two droplets merge and form a larger droplet (cf. Figure 6b).

The third scenario is an **exploding liquid** that comprises a compressed and hot liquid film exposed to vacuum. The film rapidly expands and then disintegrates into filaments and droplets (cf. Figure 6c).

We have chosen the last two scenarios because they possess large inhomogeneities and each subdomain might require a different algorithm so that they are particularly interesting for auto-tuning. In comparison to the second scenario, the third scenario develops much faster and the dynamic load balancing takes an important role which we wanted to evaluate.

All experiments were conducted on SuperMUC-NG, a cluster at the Leibniz Supercomputing Centre[8] in Garching, Germany and, as of November 2019, the 9th fastest supercomputer worldwide[9]. The cluster contains 6336 nodes, each containing two sockets with a 24-core Skylake Xeon Platinum 8174 processor. For the present simulations, we have executed ls1 mardyn with one process per socket, i.e., two processes per node. We pinned the threads of each process to one socket, corresponding to the NUMA domains.

6. Results

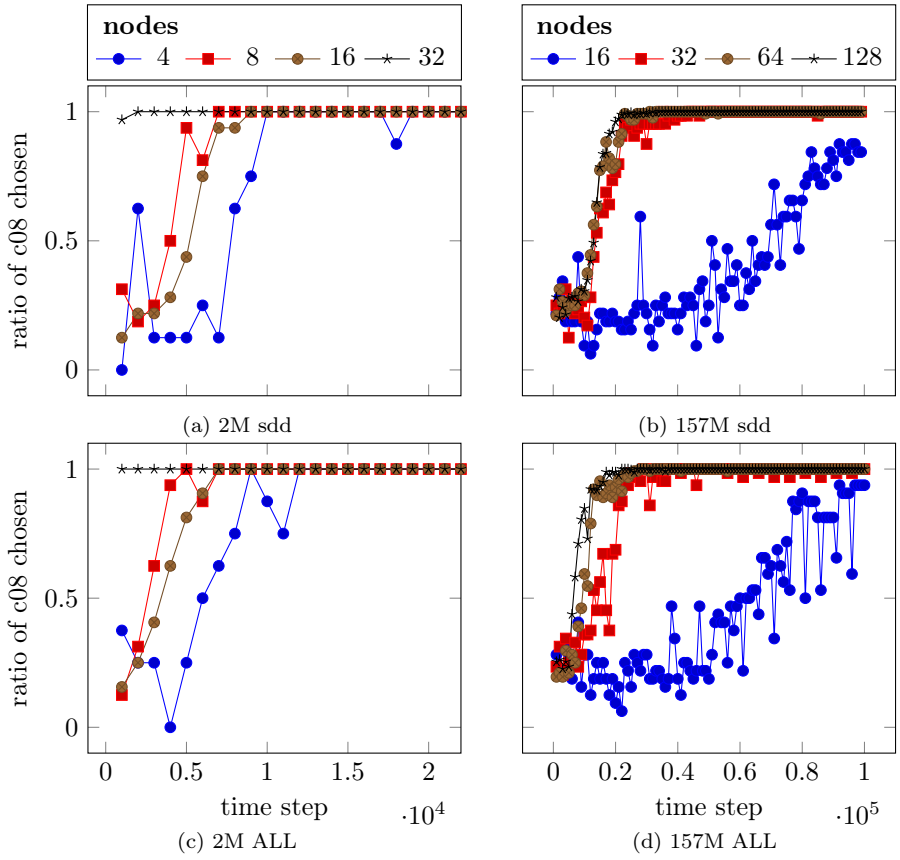
6.1. Spinodal Decomposition

First, we have investigated the performance of ls1 mardyn for the spinodal decomposition scenario. We compared auto-tuning and the performance of ls1 mardyn with and without AutoPas for different load balancing schemes.

To compare to the results from Ref. [10], we have performed a very similar experiment and examined the switching behavior of AutoPas for the spinodal decomposition scenario when choosing between the c08 and sliced traversal. The results are depicted in Figure 7. For almost all scenarios and node counts the simulation starts with the sliced traversal being more beneficial than the c08 traversal, with the c08 traversal becoming faster in the later stages of the simulation. This is in accordance with the results from Ref. [10] and can be attributed to a very homogeneous particle distribution at the beginning of the

[8] LRZ: `https://www.lrz.de/`

[9] `https://www.top500.org/lists/2019/11/`



(a) 2M sdd (b) 157M sdd

(c) 2M ALL (d) 157M ALL

Figure 7: Relative amount of processes for which AutoPas selects the c08 traversal in the spinodal decomposition scenario. AutoPas was only allowed to select either the c08 or the sliced traversal. A value of 1 indicates that all processes were using the c08 traversal, a value of 0 that all processes were using the sliced traversal. Simulations have been carried out for a small scenario with $2 \cdot 10^6$ particles (left) and a larger scenario with $157 \cdot 10^6$ particles for both a Cartesian domain decomposition (top) and a load-balanced decomposition (bottom), for which rebalancing was triggered every 5000 time steps. For this experiment, tuning was triggered every 1000 time steps. The plot for the 2M scenario only shows the initial $2 \cdot 10^4$ time steps, to better illustrate the switching of the auto-tuning behavior, even though the entire simulation took $1 \cdot 10^5$ time steps.

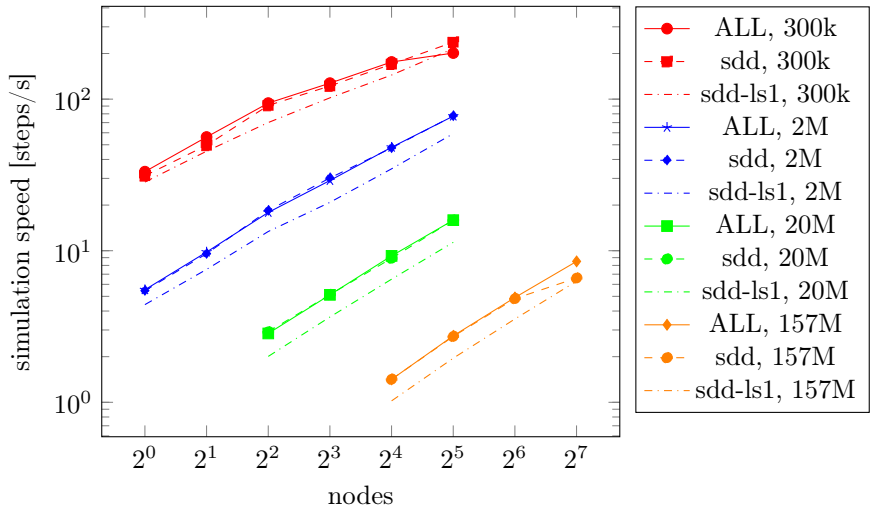


Figure 8: Strong scaling for the spinodal decomposition scenario ($1 \cdot 10^5$ time steps) comparing ls1 mardyn with AutoPas to ls1 mardyn without AutoPas. ALL: ls1 mardyn with ALL-based load balancing and using AutoPas. sdd: ls1 mardyn with AutoPas and with standard domain decomposition. sdd-ls1: ls1 mardyn without AutoPas and with standard domain decomposition.

simulation and a heterogeneous distribution towards the end of the simulation (cf. Figure 6a), for which the c08 traversal is more appropriate due to dynamic work scheduling on the node-level.

However, it can be seen that the behavior varies significantly depending on the size of the simulation and the number of processes. Especially for small scenarios and many processes (nodes), the c08 traversal is chosen by AutoPas even at the start of the simulation. This can be explained by a higher degree of potential parallelization for the c08 traversal compared to the sliced traversal: The work chunks of the c08 traversal are single cells, whereas the sliced traversal is one-dimensional and distributes one-dimensional slices of the domain as work units. The only exception is the 157M scenario with few processes, where the sliced traversal remains superior for more than half of the simulation.

Figure 7 also shows the switching behavior with load balancing. It can be seen that the c08 traversal is selected earlier and more often compared to the non-load-balanced approach. To some extent, this can be explained by the fact that some subdomains are getting smaller and thus the traversal with a better parallelizability (c08) in terms of dynamic work load distribution on the node-level is chosen.

A comparison of the simulation performance between the Cartesian domain decomposition and the diffusive load balancing (using ALL) is displayed in Figure 8. Except for the smallest scenario ($3 \cdot 10^5$ particles, 32 nodes), diffusive load balancing can yield some performance benefits compared to the non-load-balanced simulations. The relatively small performance benefits are within our

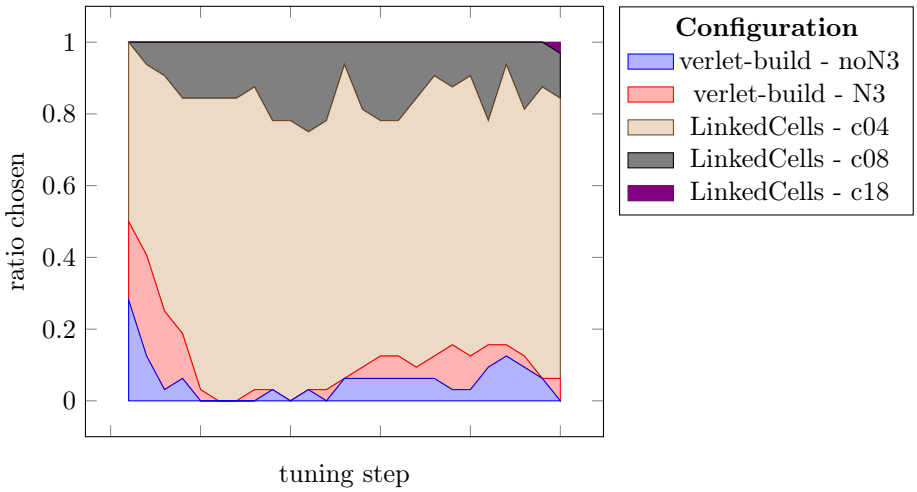


Figure 9: Relative amount of 32 processes (16 nodes), for which AutoPas selected a specific configuration for the droplet coalescence scenario and load balancing with the ALL library.

expectations as the scenario does not provide large inhomogeneities throughout the entire domain, but contains smaller inhomogeneities that are distributed throughout the entire simulation domain.

The performance of the original ls1 mardyn code, i.e., without AutoPas, is compared to the one of this code using AutoPas in Figure 8. It can be seen that the simulation is sped-up through AutoPas by about 50% in comparison to the original ls1 mardyn code which does not contain auto-tuning.

6.2. Droplet Coalescence

For the droplet coalescence scenario, we have allowed AutoPas to tune over all available tuning options, with the exception of the DirectSum approach, as it is not beneficial and its costs are too high. A statistics of the chosen scenario can be seen in Figure 9. At the beginning, roughly 50% of all processes employ the VL Build container. This behavior changes relatively quickly until this container is used by around 10-20% of the processes.

This change in behavior can be explained by looking at single ranks (cf. Figure 10). Most of the outer processes, i.e., processes calculating only the gaseous phase that consists of few particles per volume, are using the Verlet list traversal because it allows for a quick traversal of the according particle pairs. In contrast to the linked cell traversals, this does not incorporate the overhead of iterating through the cells, but uses the list of particle pairs directly.

Due to load balancing, many of these outer processes will cover larger domains of the simulation volume, as their computational load is small at the beginning and some of these processes will also have to cover parts of the droplet. Due to these two changes, these processes have to take more care of load balancing at the node-level and the c04 traversal of the linked cells container is

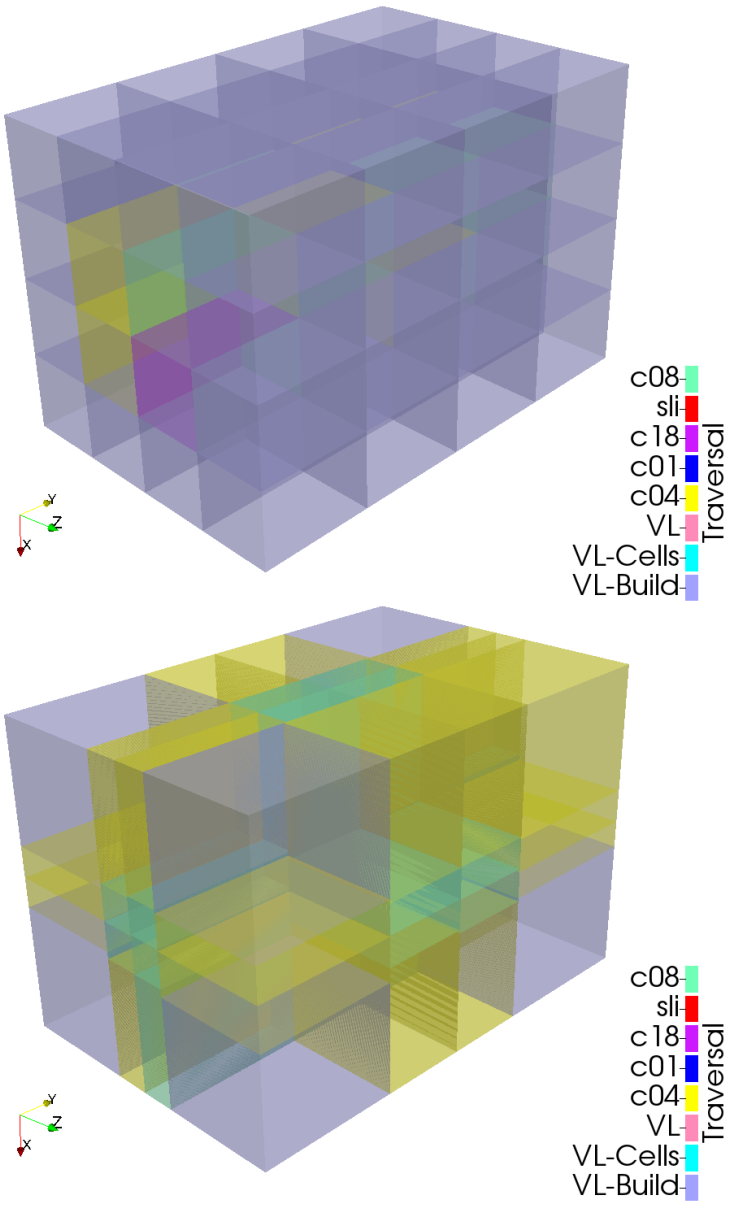


Figure 10: Configurations chosen for the droplet coalescence scenario and decomposition of a simulation on 32 nodes (64 processes) when using ls1 with enabled AutoPas and ALL load balancing at the beginning of the simulation (top) and after some rebalancing steps (bottom).

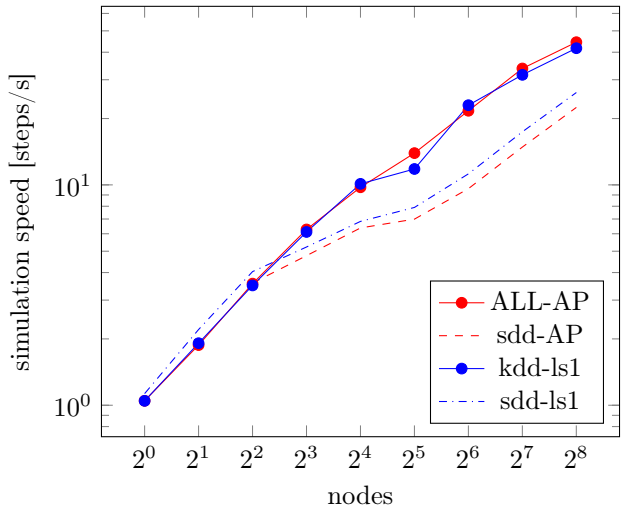


Figure 11: Strong scaling for the droplet coalescence scenario using ls1 mardyn with and without AutoPas. The performance of AutoPas with the ALL load balancing is compared with the original code with k-d tree-based load balancing.

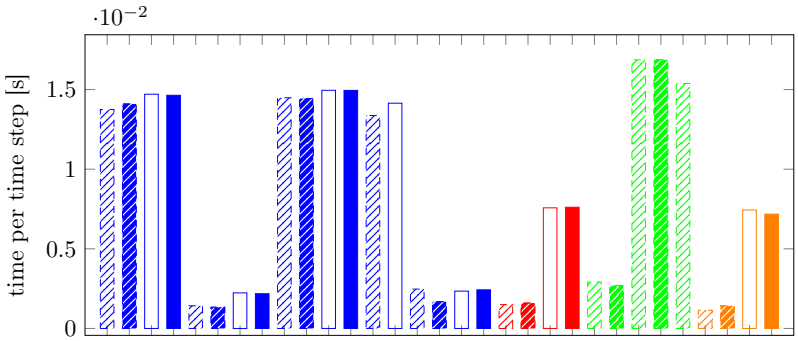
chosen instead, since it has a lower overhead when traversing through many cells, compared to the c08 traversal.

The volume of the inner ranks will shrink over time as they are calculating regions with a high computational load. In comparison to the outer ones, some ranks will cover relatively small subdomains. As the c08 traversal provides finer-grained load balancing, this traversal was chosen in this case.

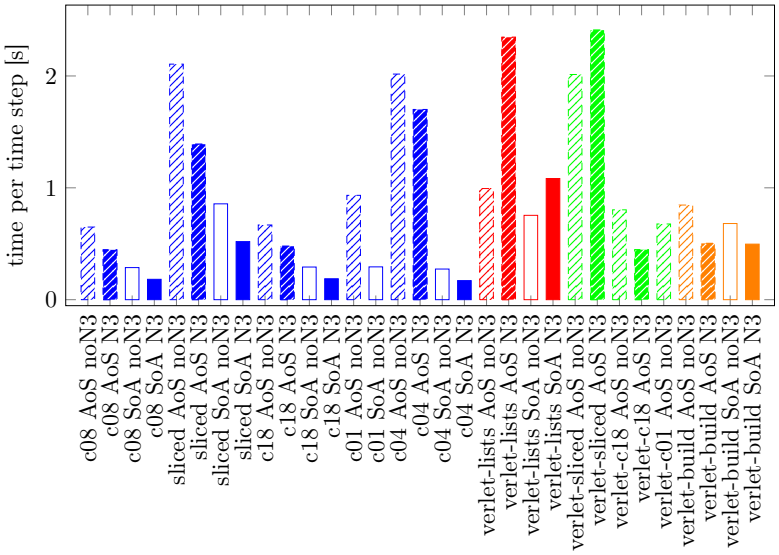
In Figure 11, the scaling behavior of ls1 mardyn with and without AutoPas is shown both with enabled and disabled load balancing. It can be seen that the ALL load balancing is mostly on par with the kdd load balancing and that AutoPas has no large impact on the overall performance. Comparing the load-balanced with the non-load-balanced calculations, it can be seen that they provide the same performance up to four compute nodes. With eight or more nodes, the load-balanced simulations significantly outperform the non-load-balanced ones. Up to four nodes, the load distribution using Cartesian decomposition (sdd) is optimal because the two droplets are centered exactly in the middle of the domain and thus the load distribution up to two subdomains in each direction, i.e., eight subdomains in total, is optimal. This correlates to four nodes, as they each have two NUMA domains, and one process was used per NUMA domain, i.e., two processes per node.

6.3. Exploding Liquid

First, we illustrate the heavily varying performance of the different subdomains in the exploding liquid scenario. Figure 12 shows the sampling results for the different algorithm configurations for two subdomains when simulating



(a) Rank 0, simulating entirely vacuum.



(b) Rank 1, simulating the liquid film and parts of the vacuum.

Figure 12: Simulation time for one time step for the different configurations at the beginning of the simulation when simulating the exploding liquid scenario with three MPI processes. Due to the Cartesian domain decomposition, the three ranks are arranged along the y axis in an equidistant manner. This corresponds to the first time step, when using the ALL load balancer (cf. Figure 13). The colors mark different container types (blue=Linked Cells, red=VL Global, green=VL Cells, orangs=VL Build, see also Table 1). The results for rank 2 are almost identical to rank 0 and are thus not shown.

with three processes. Ranks 0 and 2 represent subdomains which initially are entirely made up of vacuum, while rank 1 contains the liquid film.

For the processes simulating the vacuum regions (Figure 12a) the c08, c18 and c01 traversals for the linked cells and the v18 and v01 traversals for the Verlet list container performed worse than the other traversals due to their relatively large overhead for dynamic scheduling. In contrast, the sliced traversal performed well, as it does not provide dynamic scheduling, and the c04 traversal

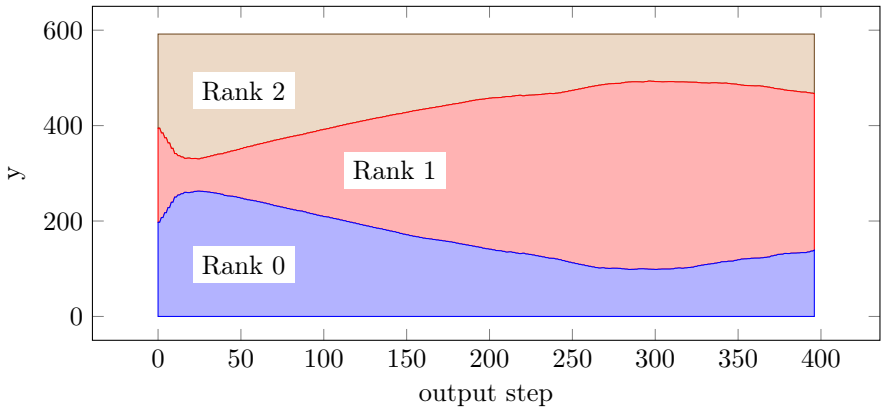


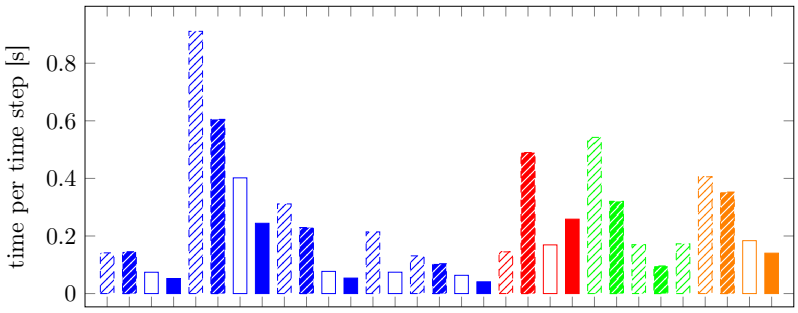
Figure 13: Time evolution of the decomposition of the exploding liquid scenario when using the ALL load balancer on three MPI ranks. The liquid expands over time in y direction, with the domain decomposition following.

performed reasonably well, as it always schedules larger chunks of data. For the vacuum region, the Verlet build traversal performed best, as it directly iterates over pairs of particles inside of a neighbor list, whereas all linked cell traversals include the overhead of iterating over the empty cells.

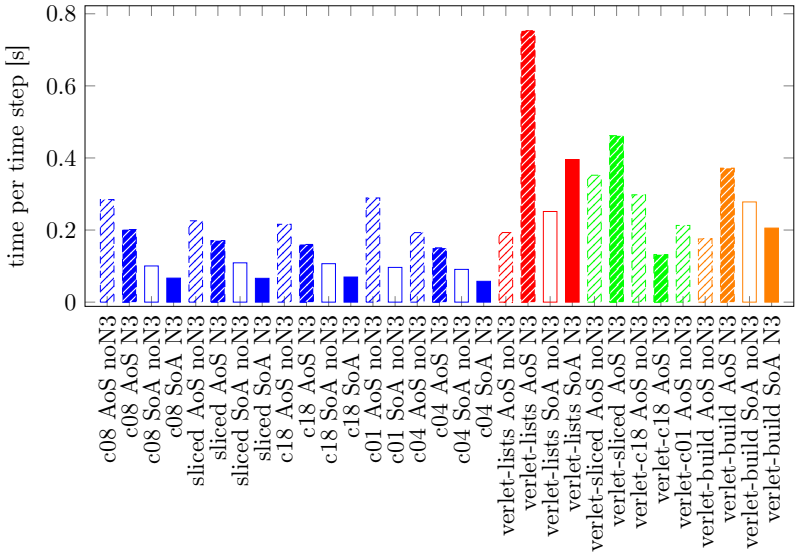
For the process that calculates the subdomain containing the liquid film (Figure 12b), the requirements for efficient traversals are completely different. First, vectorization and thus an SoA data structure become favorable, as the film consists of a dense set of particles. Second, load balancing at node-level through dynamic scheduling is now profitable. In addition, the use of Newton's third law of motion is beneficial because it halves the amount of required calculations. For these reasons, all linked cell traversals, with the exception of the sliced traversal, perform well in their SoA form with enabled Newton3. Due to a worse vectorizability, the Verlet-like traversals provide a poorer performance than the linked cells traversals.

We have repeated the above measurements with ALL load balancing and checked the measurements for different traversals once a good load balancing was reached and the liquid film expanded only slightly, i.e., when the inner rank covered the smallest volume (cf. Figures 13 and 14). In contrast to the previous measurements, a good load balancing between the two optimal choices was observed. In the load-balanced case, the subdomain of the inner rank only included a part of the liquid phase volume, while the outer ranks contained both parts of the very dilute gas, as well as parts of the liquid. Because of this behavior, the inner process (rank 1) did not have to care about load balancing so that all linked cells traversals perform equally well. However, both the Newton3 optimization and vectorization were important for an efficient processing. For the outer process, load balancing was essential, making the sliced traversal perform worst.

The performance of ls1 mardyn with and without AutoPas is shown in



(a) Rank 0, simulating mostly vacuum with some parts of the liquid film.



(b) Rank 1, simulating only the liquid film.

Figure 14: Simulation time for one time step for the different configurations at the point of the simulation, where the inner subdomain is the smallest (cf. Figure 13), when simulating the exploding liquid scenario with three MPI processes. The colors mark different container types (blue=Linked Cells, red=VL Global, green=VL Cells, orangs=VL Build, see also Table 1). The results for rank 2 are almost identical to rank 0 and are thus not shown.

Figure 15 for the exploding liquid scenario. Overall, using AutoPas and dynamic load balancing with ALL provides the best performance for all node counts, as ALL provides a diffusive load balancing that is able to better follow the expanding liquid compared to the k-d tree based decomposition.

The kdd load balancing of ls1 mardyn cannot properly follow the explosion and thus generally results in the worst performance compared to the other three options, which can be explained by a poor load prediction, which is especially complicated for this very inhomogeneous scenario.

Except for very large node counts (starting at 128 nodes, i.e., $3 \cdot 10^4$ particles

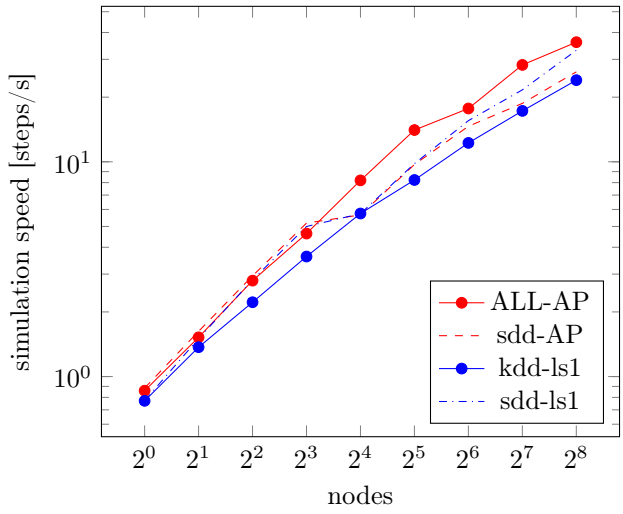


Figure 15: Strong scaling for the exploding liquid scenario with $8 \cdot 10^6$ particles, both for ls1 mardyn with and without AutoPas and for simulations using load balancing (ALL, resp. kdd) and not using load balancing (sdd).

per process), the performance of ls1 mardyn with and without AutoPas provides a similar performance when the Cartesian domain distribution is used. However, the version without AutoPas provided the better performance for large node counts. This can be explained by a variety of factors. The following most negatively influence the performance of communication:

1. The iterators inside of AutoPas have an additional layer of indirection that arises through the hiding of the container structure.
2. The RegionIterators that are used to iterate over particles in a specific region, e.g., to select particles for communication, provide less performance. These internally work by iterating over all particles in cells that could potentially accommodate such particles. As the cells using AutoPas are always larger (at least $r_c + r_{skin}$) compared to the original ls1 mardyn (at least r_c) the iterators have to consider (and skip) more particles and thus provide less performance.
3. Inside of the halo exchange, particles are only updated and not deleted and reinserted when using AutoPas. This requires a search of the particles which should be updated.

Additionally, the larger minimal cell size introduces more calculations when using AutoPas, especially in the strong scaling limit.

For this scenario, AutoPas and ALL load balancing can provide an improvement in speed of up to 43% (32 nodes) compared to the original ls1 mardyn code.

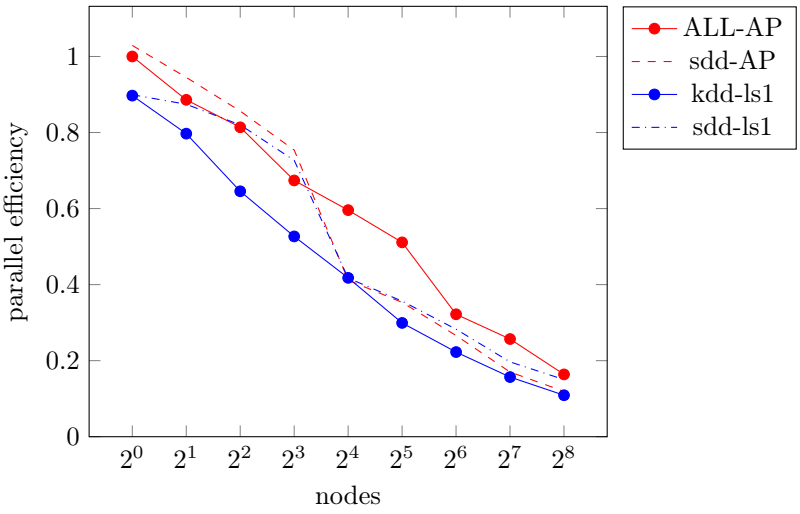


Figure 16: Parallel efficiency for the exploding liquid scenario with $8 \cdot 10^6$ particles, both for ls1 mardyn with and without AutoPas and for simulations with load balancing (ALL or kdd) and without load balancing (sdd). The efficiency is normalized by the performance of the ALL-AutoPas(AP) approach on a single node.

7. Summary and Outlook

7.1. Summary

We have discussed details of the integration of the node-level library AutoPas into the code ls1 mardyn which enable the first parallel MD simulation that employs dynamic auto-tuning. We have provided an overview of the auto-tuning capabilities of AutoPas inside of MPI-parallel simulations and have successfully demonstrated its potential for selecting different algorithm configurations for different subdomains of highly heterogeneous simulations. Additionally, we have shown that AutoPas can provide significant speedups over the original ls1 mardyn both for the exploding liquid (up to 43% speedup) and for the spinodal decomposition scenario (up to 50% speedup), while similar performance was achieved for the droplet coalescence scenario. This is remarkable, with ls1 mardyn being a highly-optimized code already, and thus underpins both the potential of and the need for dynamically adapting software in simulation technology.

7.2. Outlook

While a speedup compared to the original ls1 mardyn was almost always observable, it was often not optimal. Especially for the exploding liquid scenario, only very low parallel efficiencies are achieved (cf. Figure 16). The main reason for this is a rapidly developing change of the particle distribution, which the load balancing could not accurately follow due to a relatively low rebalancing

frequency and because diffusive load balancing can only react to load imbalances after they have been observed.

The rebalancing frequency is currently limited by the time AutoPas needs to tune and then select a configuration. As AutoPas is currently able to use around 35 different configurations, it needs 350 time steps to select the best one when ten samples are collected for each configuration. To provide performance benefits that outweigh the cost of tuning, the simulation has to continue for a significant amount of time, e.g., 4000 time steps, without changing the domain distribution. To circumvent this long tuning phase, we are currently developing a variety of methods to speed it up, e.g., by using machine learning or Bayesian statistics.

Another approach to provide a better load balancing is currently being developed by the team behind ALL and is described in Ref. [34]. This approach aims to provide faster convergence compared to the current diffusive approach of the staggered grid within the ALL library.

8. Acknowledgements

We thank the Federal Ministry of Education and Research, Germany, for providing financial support of this work through the project "Task-based load balancing and auto-tuning in particle simulations" (TaLPas)[10], grant numbers 01IH16008A and 01IH16008B. We give thanks to the LRZ (projects pn56mo and pr94ta), the HLRS and the Gauss Centre for Supercomputing (GCS-MDDC) for providing compute resources for this work.

References

[1] 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, TweTriS: Twenty trillion-atom simulation, The International Journal of High Performance Computing Applications 33 (5) (2019) 838–854. arXiv:https://doi.org/10.1177/1094342018819741, doi:10.1177/1094342018819741.
URL https://doi.org/10.1177/1094342018819741

[2] S. Pronk, S. Páll, R. Schulz, P. Larsson, P. Bjelkmar, R. Apostolov, M. R. Shirts, J. C. Smith, P. M. Kasson, D. van der Spoel, B. Hess, E. Lindahl, GROMACS 4.5: a high-throughput and highly parallel open source molecular simulation toolkit, Bioinformatics 29 (7) (2013) 845–854. arXiv:http://oup.prod.sis.lan/bioinformatics/article-pdf/29/7/845/17343875/btt055.pdf, doi:10.1093/bioinformatics/btt055.
URL https://doi.org/10.1093/bioinformatics/btt055

[10] http://www.talpas.de

[3] S. Páll, M. J. Abraham, C. Kutzner, B. Hess, E. Lindahl, Tackling Exascale Software Challenges in Molecular Dynamics Simulations with GROMACS, in: S. Markidis, E. Laure (Eds.), Solving Software Challenges for Exascale, Springer International Publishing, Cham, 2015, pp. 3–27.

[4] W. M. Brown, P. Wang, S. J. Plimpton, A. N. Tharrington, Implementing molecular dynamics on hybrid high performance computers – short range forces, Computer Physics Communications 182 (4) (2011) 898–911. doi:https://doi.org/10.1016/j.cpc.2010.12.021.
URL http://www.sciencedirect.com/science/article/pii/S0010465510005102

[5] W. M. Brown, A. Semin, M. Hebenstreit, S. Khvostov, K. Raman, S. J. Plimpton, Increasing Molecular Dynamics Simulation Rates with an 8-Fold Increase in Electrical Power Efficiency, in: SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2016, pp. 82–95. doi:10.1109/SC.2016.7.

[6] W. M. Brown, J.-M. Y. Carrillo, N. Gavhane, F. M. Thakkar, S. J. Plimpton, Optimizing legacy molecular dynamics software with directive-based offload, Computer Physics Communications 195 (2015) 95–101. doi:https://doi.org/10.1016/j.cpc.2015.05.004.
URL http://www.sciencedirect.com/science/article/pii/S001046551500171X

[7] R. E. Rudd, J. Q. Broughton, Coarse-grained molecular dynamics and the atomic limit of finite elements, Physical Review B 58 (1998) R5893–R5896. doi:10.1103/PhysRevB.58.R5893.
URL https://link.aps.org/doi/10.1103/PhysRevB.58.R5893

[8] S. T. O'Connell, P. A. Thompson, Molecular dynamics–continuum hybrid computations: A tool for studying complex fluid flows, Physical Review E 52 (1995) R5792–R5795. doi:10.1103/PhysRevE.52.R5792.
URL https://link.aps.org/doi/10.1103/PhysRevE.52.R5792

[9] P. Neumann, H. Flohr, R. Arora, P. Jarmatz, N. Tchipev, H.-J. Bungartz, MaMiCo: Software design for parallel molecular-continuum flow simulations, Computer Physics Communications 200 (2016) 324–335. doi:https://doi.org/10.1016/j.cpc.2015.10.029.
URL http://www.sciencedirect.com/science/article/pii/S0010465515004129

[10] F. A. Gratl, S. Seckler, N. Tchipev, H. Bungartz, P. Neumann, AutoPas: Auto-Tuning for Particle Simulations, in: 2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2019, pp. 748–757. doi:10.1109/IPDPSW.2019.00125.

[11] 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, ls1 mardyn: The Massively Parallel Molecular Dynamics Code for Large Systems, Journal of Chemical Theory and Computation 10 (10) (2014) 4455–4464, pMID: 26588142. arXiv:https://doi.org/10.1021/ct500169q, doi:10.1021/ct500169q.
URL https://doi.org/10.1021/ct500169q

[12] H. Berendsen, D. van der Spoel, R. van Drunen, GROMACS: A message-passing parallel molecular dynamics implementation, Computer Physics Communications 91 (1) (1995) 43–56. doi:https://doi.org/10.1016/0010-4655(95)00042-E.
URL http://www.sciencedirect.com/science/article/pii/001046559500042E

[13] S. Plimpton, Fast Parallel Algorithms for Short-Range Molecular Dynamics, Journal of Computational Physics 117 (1) (1995) 1–19. doi:https://doi.org/10.1006/jcph.1995.1039.
URL http://www.sciencedirect.com/science/article/pii/S002199918571039X

[14] J. C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kalé, K. Schulten, Scalable molecular dynamics with NAMD, Journal of Computational Chemistry 26 (16) (2005) 1781–1802. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/jcc.20289, doi:10.1002/jcc.20289.
URL https://onlinelibrary.wiley.com/doi/abs/10.1002/jcc.20289

[15] F. Weik, R. Weeber, K. Szuttor, K. Breitsprecher, J. de Graaf, M. Kuron, J. Landsgesell, H. Menke, D. Sean, C. Holm, ESPResSo 4.0 – an extensible software package for simulating soft matter systems, The European Physical Journal Special Topics 227 (14) (2019) 1789–1816. doi:10.1140/epjst/e2019-800186-9.
URL https://doi.org/10.1140/epjst/e2019-800186-9

[16] V. Springel, The cosmological simulation code gadget-2, Monthly Notices of the Royal Astronomical Society 364 (4) (2005) 1105–1134. arXiv:https://academic.oup.com/mnras/article-pdf/364/4/1105/18657201/364-4-1105.pdf, doi:10.1111/j.1365-2966.2005.09655.x.
URL https://doi.org/10.1111/j.1365-2966.2005.09655.x

[17] P. F. Hopkins, A new class of accurate, mesh-free hydrodynamic simulation methods, Monthly Notices of the Royal Astronomical Society 450 (1) (2015) 53–110. arXiv:https://academic.oup.com/mnras/article-pdf/450/1/53/18507840/stv195.pdf, doi:10.1093/mnras/stv195.
URL https://doi.org/10.1093/mnras/stv195

[18] A. Crespo, J. Domínguez, B. Rogers, M. Gómez-Gesteira, S. Longshaw, R. Canelas, R. Vacondio, A. Barreiro, O. García-Feal, DualSPHysics:

Open-source parallel CFD solver based on Smoothed Particle Hydrodynamics (SPH), Computer Physics Communications 187 (2015) 204–216. doi:https://doi.org/10.1016/j.cpc.2014.10.004.
URL http://www.sciencedirect.com/science/article/pii/S0010465514003397

[19] P. Incardona, A. Leo, Y. Zaluzhnyi, R. Ramaswamy, I. F. Sbalzarini, Openfpm: A scalable open framework for particle and particle-mesh codes on parallel computers, Computer Physics Communications 241 (2019) 155–177. doi:https://doi.org/10.1016/j.cpc.2019.03.007.
URL http://www.sciencedirect.com/science/article/pii/S0010465519300852

[20] M. Iwasawa, A. Tanikawa, N. Hosono, K. Nitadori, T. Muranushi, J. Makino, Implementation and performance of FDPS: a framework for developing parallel particle simulation codes, Publications of the Astronomical Society of Japan 68 (4) (2016) 54 (1–22). doi:10.1093/pasj/psw053.
URL http://dx.doi.org/10.1093/pasj/psw053

[21] G. Fursin, Collective Tuning Initiative: automating and accelerating development and optimization of computing systems, in: GCC Developers' Summit, Montreal, Canada, 2009, pp. 1–28.
URL https://hal.inria.fr/inria-00436029

[22] R. Ewald, Automatic Algorithm Selection for Complex Simulation Problems, Vieweg, 2011.

[23] R. Ewald, J. Himmelspach, M. Jeschke, S. Leye, A. M. Uhrmacher, Flexible experimentation in the modeling and simulation framework JAMES II—implications for computational systems biology, Briefings in Bioinformatics 11 (3) (2010) 290–300. arXiv:https://academic.oup.com/bib/article-pdf/11/3/290/606206/bbp067.pdf, doi:10.1093/bib/bbp067.
URL https://doi.org/10.1093/bib/bbp067

[24] C. Tapus, I-Hsin Chung, J. K. Hollingsworth, Active Harmony: Towards Automated Performance Tuning, in: SC '02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing, 2002, pp. 44–44. doi:10.1109/SC.2002.10062.

[25] A. Tiwari, J. K. Hollingsworth, Online Adaptive Code Generation and Tuning, in: 2011 IEEE International Parallel Distributed Processing Symposium, 2011, pp. 879–892. doi:10.1109/IPDPS.2011.86.

[26] J. A. Nelder, R. Mead, A Simplex Method for Function Minimization, The Computer Journal 7 (4) (1965) 308–313. arXiv:https://academic.oup.com/comjnl/article-pdf/7/4/308/1013182/7-4-308.pdf, doi:10.1093/comjnl/7.4.308.
URL https://doi.org/10.1093/comjnl/7.4.308

[27] L. Verlet, Computer "Experiments" on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules, Phys. Rev. 159 (1967) 98–103. doi:10.1103/PhysRev.159.98.
URL https://link.aps.org/doi/10.1103/PhysRev.159.98

[28] A. A. Chialvo, P. G. Debenedetti, On the use of the Verlet neighbor list in molecular dynamics, Computer Physics Communications 60 (2) (1990) 215–224. doi:https://doi.org/10.1016/0010-4655(90)90007-N.
URL http://www.sciencedirect.com/science/article/pii/001046559090007N

[29] P. Gonnet, Pairwise verlet lists: Combining cell lists and verlet lists to improve memory locality and parallelism, Journal of Computational Chemistry 33 (1) (2012) 76–81. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/jcc.21945, doi:10.1002/jcc.21945.
URL https://onlinelibrary.wiley.com/doi/abs/10.1002/jcc.21945

[30] M. J. Abraham, T. Murtola, R. Schulz, S. Páll, J. C. Smith, B. Hess, E. Lindahl, GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers, SoftwareX 1-2 (2015) 19–25. doi:https://doi.org/10.1016/j.softx.2015.06.001.
URL http://www.sciencedirect.com/science/article/pii/S2352711015000059

[31] S. Seckler, N. Tchipev, H. Bungartz, P. Neumann, Load Balancing for Molecular Dynamics Simulations on Heterogeneous Architectures, in: 2016 IEEE 23rd International Conference on High Performance Computing (HiPC), 2016, pp. 101–110. doi:10.1109/HiPC.2016.021.

[32] B. Hendrickson, K. Devine, Dynamic load balancing in computational mechanics, Computer Methods in Applied Mechanics and Engineering 184 (2) (2000) 485–500. doi:10.1016/S0045-7825(99)00241-8.
URL http://www.sciencedirect.com/science/article/pii/S0045782599002418

[33] J. Makino, A Fast Parallel Treecode with GRAPE, Publications of the Astronomical Society of Japan 56 (3) (2004) 521–531. arXiv:http://oup.prod.sis.lan/pasj/article-pdf/56/3/521/17448885/pasj56-0521.pdf, doi:10.1093/pasj/56.3.521.
URL https://doi.org/10.1093/pasj/56.3.521

[34] G. Sutmann, Multi-Level Load Balancing for Parallel Particle Simulations, Proceedings of VI International Conference on Particle-Based Methods. Fundamentals and Applications (2019) 80–92.
URL https://congress.cimne.com/particles2019/frontal/doc/Ebook_Particles_2019.pdf