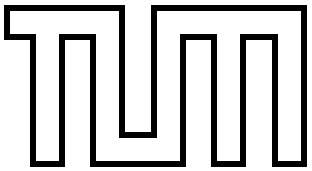


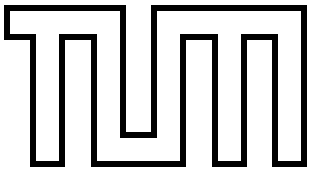
SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Rebuild Frequency Estimation and Autotuning for Dynamic Containers in Particle Simulations

Yasmine Farah



SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Rebuild Frequency Estimation and Autotuning for Dynamic Containers in Particle Simulations

Author:	Yasmine Farah
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisor:	Manish Mishra, M.Sc.
Date:	21.08.2025

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

Munich, 21.08.2025 Yasmine Farah

Acknowledgements

I express my sincere gratitude to my thesis supervisor, Manish, for his continuous support and guidance throughout the research. More importantly, I would like to express my appreciation for the time that he invested in providing answers and clarifications. I also thank Prof. Dr. Hans-Joachim Bungartz and the chair of scientific computing for providing the opportunity and the necessary resources to work on the project. Lastly, I would like to express a heartfelt thank you to my family and friends, without whom I would never have been able to complete this journey.

Abstract

Particle simulations are widely used in the natural sciences for a broad range of applications—from replicating real-life experiments to investigate physical properties, to observing the evolution of system states. These simulations can yield insights that are otherwise difficult or impossible to obtain using conventional experimental methods. As a result, various simulation techniques have been developed, each tailored to specific scenarios. For simulations involving a large number of particles, the computational cost increases significantly, thereby necessitating the development of optimized algorithms.

AutoPas, an open-source C++ performance library for node-level particle simulations, addresses this challenge by dynamically selecting suitable algorithmic configuration based on the current simulation state. The selection of the optimal configuration begins with a tuning phase, during which the runtime of a given number of samples for each configuration is measured. A weighted average of these samples is then used to balance the influence of iterations involving a container rebuild versus regular iterations, where the weighting depends on the rebuild frequency, defined as the number of iterations until the next rebuild is triggered.

To manage particles, AutoPas employs both static and dynamic containers. Unlike static containers, dynamic containers trigger a rebuild only when necessary, rather than at fixed iteration intervals. This thesis investigates a method for accurately estimating the rebuild frequency in dynamic containers within AutoPas. This estimation is integral to runtime prediction and plays a key role in the selection process for determining the optimal simulation configuration.

Contents

Part I.

Introduction and Background

1. Introduction

Particle simulations are widely used across various scientific disciplines, including chemical physics, materials science, geomechanics, and many others [4, 5]. Depending on the field of application, different methods have been developed to simulate particle interactions. These methods include smoothed particle hydrodynamics (SPH)—primarily used to simulate continuum fluid flow in a field-based model (see Figure 1.1)—the Discrete Element Method (DEM), Molecular Dynamics (MD), among others [12, 7, 4]. Molecular Dynamics is widely applied in drug discovery and in the investigation of drug resistance. For instance, it has been employed to study the effects of an antiviral compound against SARS-CoV-2 [26].

This thesis focuses on Molecular Dynamics, both as the basis for theoretical exploration and for conducting the experimental simulations.

To simulate interactions between particles—especially in systems containing a large number of them—substantial computational power is required. Well-established libraries such as ls1 mardyn and LAMMPS aim to reduce the computational burden by employing optimized algorithms and frameworks that are statically determined at the start of the simulation [24, 31].

AutoPas is a C++ library designed to achieve optimal node-level performance for particle simulations by employing an algorithm selection technique (referred to as AutoTuning), which dynamically selects the most suitable configuration based on the current simulation state [13]. The auto-tuning process is triggered periodically after a predefined number of simulation iterations.

AutoPas offers various options for particle traversal, primarily based on the well-known Linked Cells and Verlet Lists algorithms. In both methods—neighbor lists in the case of Verlet Lists, and particle cells in the case of Linked Cells—data structures must be periodically updated. This update process is referred to as *rebuilding* in the context of AutoPas. During the tuning phase, a predefined number of runtime samples is collected, with each sample representing the execution time of a single iteration. Some of these iterations include additional overhead due to a rebuild. To estimate the expected runtime of a configuration, AutoPas computes a weighted average of the collected samples. This average accounts for the rebuild frequency—the number of iterations per rebuild—by assigning different weights to iterations with and without rebuild overhead. A lower rebuild frequency results in a higher overall runtime due to the increased number of costly rebuild operations.

In AutoPas, dynamic containers may be used during simulation. Unlike regular containers, dynamic containers do not rely on a fixed rebuild frequency. Instead, a rebuild is triggered only when necessary—specifically, whenever any particle moves more than half the length of an extra buffer region called the skin [8]. This approach reduces the rebuild overhead in many cases. However, as a consequence, the rebuild frequency cannot be directly calculated, which makes it challenging to compare different configurations accurately.

This thesis investigates methods for estimating the mean rebuild frequency during non-tuning phases in dynamic containers. We begin by examining the theoretical background

and related work, followed by a discussion of the technical aspects involved in conducting experiments and extracting relevant data. Subsequently, we evaluate the current approach, showing that it often fails to select the optimal configuration. We also examine a simple simulation-based method and highlight its limitations. To address this limitation, we propose two alternative methods aimed at improving estimation accuracy: the *Velocity Method*, the *Extrapolation Method*. These methods are assessed in terms of both accuracy and performance. Finally, we identify potential areas for improvement and suggest directions for future work.

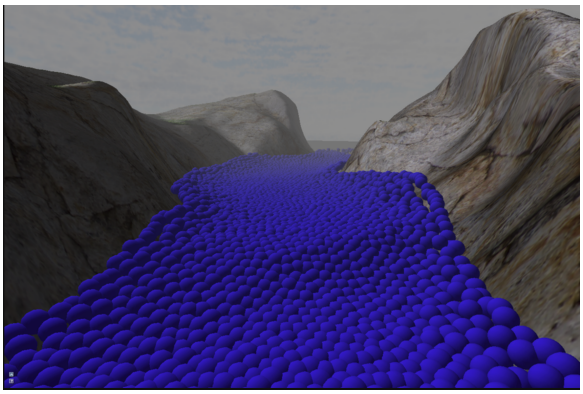


Figure 1.1.: Shallow water simulation using SPH. Source: [29]

2. Background

2.1. Molecular Dynamics

Molecular Dynamics (MD) is a computational simulation method used for the simulation of atoms and molecules, providing insights into their physical motion and dynamic behavior over time [4]. The method relies on time discretization to compute particle trajectories by numerically solving Newton's equations of motion. A simplified overview of the basic Molecular Dynamics algorithm is presented in Algorithm 1. Additionally, thermostats—temperature control mechanisms—and boundary conditions can be applied at each iteration to model various physical environments.

Algorithm 1: Basic Molecular Dynamics (MD) algorithm used to compute particle trajectories over time by iteratively solving Newton's equations of motion.

Input: Initial positions $x(t_0)$, velocities $v(t_0)$, number of iterations k

```
for i ← 0 to k − 1 do
    Compute f(t_i);                                   // Evaluate forces
    Integrate Newton's equations to obtain x(t_{i+1}), v(t_{i+1});
    Update particle states: x(t_i) → x(t_{i+1}), v(t_i) → v(t_{i+1});
```

2.1.1. Newton's Equations of Motion

From Newton's second law of motion, we obtain a system of ordinary differential equations (ODEs) that can be integrated to compute the new positions and velocities of particles:

$$f = ma \tag{2.1}$$

$$v(t) = v(t_i) + \int a \, \delta t \tag{2.2}$$

$$x(t) = x(t_i) + \int v \, \delta t \tag{2.3}$$

Where:

- f is the force acting on a particle,
- v is the velocity,
- x is the position,
- a is the acceleration,
- t is the time.

From the equations above, we derive the following relationship:

$$a = \frac{d^2x}{dt^2} = \frac{f}{m},$$

where the force f is typically a function of the particle's position.
To compute updated positions, this second-order differential equation must be integrated over time. Since analytical solutions are often computationally expensive or infeasible in complex systems, numerical methods such as Leapfrog or Störmer–Verlet are commonly used instead.

2.2. Calculation of Forces

In N-body simulations, forces act between every pair of particles. To compute the total force acting on a given particle i, we sum the pairwise forces exerted by all other particles:

$$\vec{f_i} = \sum_{j \neq i} \vec{f_{ij}} \tag{2.4}$$

Here, $\vec{f_{ij}}$ denotes the force exerted on particle i by particle j. The direction and magnitude of this force depend on the nature of the interaction potential and the relative positions of the particles.

2.2.1. Particle-Pair Potentials

To simulate the forces acting between pairs of particles, different potential functions are employed depending on the nature of the system. In astrophysical simulations, where particles represent celestial bodies with large masses, Newton's law of universal gravitation is typically used.

In contrast, Molecular Dynamics focuses on atomic and molecular interactions, where intermolecular forces play a significant role. In such cases, the Lennard-Jones potential is commonly used, as it effectively models the balance between attractive and repulsive forces at the atomic scale.

Newton's Law of Universal Gravitation

The gravitational force between two point masses is given by Newton's law of universal gravitation:

$$F = G\frac{m_1 m_2}{r^2} \tag{2.5}$$

where:

- F is the magnitude of the gravitational force,
- m_1 and m_2 are the masses of the two particles,
- r is the distance between them,
- G is the gravitational constant.

The gravitational constant G has the approximate value:

$$G \approx 6.67430 \times 10^{-11}\,\mathrm{m}^3\,\mathrm{kg}^{-1}\,\mathrm{s}^{-2}$$

Lennard-Jones Potential

$$V(r) = 4\varepsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right] \tag{2.6}$$

- ε is the depth of the potential well (i.e., the strength of the attraction),
- σ is the finite distance at which the inter-particle potential is zero,
- r is the distance between the centers of the two particles.

The Lennard-Jones potential captures the essential properties of intermolecular interactions: two particles repel each other at short distances, attract each other at intermediate distances, and stop interacting at infinite distance [18]. Figure 2.1 illustrates the intermolecular potential energy as a function of the distance between two particles. The force between two particles can be derived as the negative gradient of the Lennard-Jones potential.

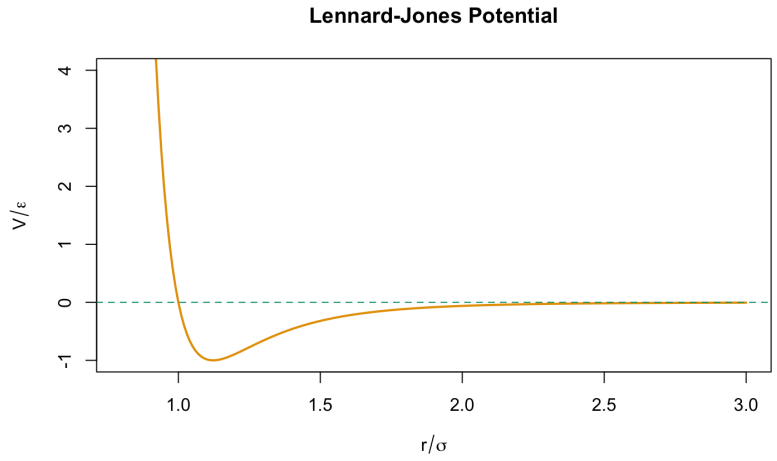


Figure 2.1.: Lennard-Jones potential: Intermolecular potential energy as a function of distance r, with parameters ε and σ. Both the distance r and the potential V are normalized by σ and ε, respectively. The potential features a short-range repulsive region and a longer-range attractive region. The equilibrium distance, where the net force is zero, corresponds to the minimum of the curve.

2.2.2. Ranges of Potentials

When calculating the pairwise interactions between particles, we distinguish between two types of potentials: short-range potentials and long-range potentials. While short-range potentials have a limited effective range, long-range potentials act over an infinite range. In AutoPas, simulations for long-range potentials are not supported [14]. Therefore, this thesis focuses exclusively on algorithms for short-range potentials.

Short-Range Potentials

The number of operations required to calculate the pairwise interactions of N particles is given by:

$$N(N-1) \in \mathcal{O}(N^2) \tag{2.7}$$

(If Newton's third law is exploited, this number can be halved.)

As the number of particles increases, this computation becomes increasingly expensive. To address this issue, a cutoff radius r_c is introduced. Interactions between particles separated by a distance greater than r_c are considered negligible and are therefore excluded from the calculation.

Figure 2.2 illustrates a possible choice of cutoff radius r_c. Values of the potential beyond r_c are close to zero and are therefore omitted from the computation.

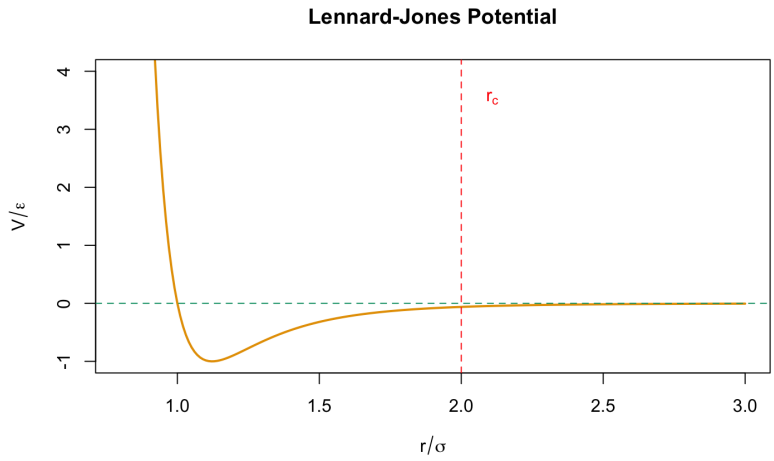


Figure 2.2.: Lennard-Jones potential with cutoff r_c (red).

2.2.3. Algorithms for Short-Range Potentials

The most computationally demanding part of most simulations is the pairwise iteration over particles, which has led to the development of optimized algorithms to handle this task efficiently. By exploiting the fact that, due to the cutoff radius, only the closest neighbors of a particle need to be considered, these algorithms focus on efficiently identifying such neighbors.

The Linked Cell Method

The Linked Cell method addresses the task of finding neighboring particles within the cutoff radius by first decomposing the domain into square cells of user-specified size $r_{\text{cell}} \times r_{\text{cell}}$, where $r_{\text{cell}} \geq r_c$. When searching for neighboring particles, only directly adjacent cells are considered, reducing the complexity to $\mathcal{O}(N)$ [1]. Figure 2.3 illustrates the domain decomposition, where only particles in the neighboring cells (blue) are considered. The

particles that fall within the cutoff radius (inside the red circle) are ultimately included in the calculation.

The main disadvantage of the Linked Cell method lies in the high percentage of unnecessary distance calculations. If one computes the probability of finding a particle within the cutoff distance r_c in a 3D volume, the ratio is given by [13]:

$$\frac{\text{Cutoff Volume}}{\text{Search Volume}} = \frac{\frac{4}{3}\pi r_c^3}{27 r_c^3} \approx 0.15 \tag{2.8}$$

Here, the cutoff volume refers to the volume of a sphere with radius r_c, and the search volume is the combined volume of 27 cubes with edge length r_c.

According to this formula, approximately 85% of the distance calculations are unnecessary.

Verlet Lists Method

The Verlet Lists method addresses the problem by maintaining a list of potential neighbors for each particle. This list includes all particles within a radius of $r_c + r_{\text{skin}}$, where r_{skin} serves as an additional margin that allows the list to be reused for the next n time steps. The larger the skin, the larger n can be. There is no loss of accuracy as long as r_{skin} is sufficiently large so that no particle can move beyond this margin within the next n time steps [32].

To illustrate the benefit of this approach, consider the case where $r_{\text{skin}} = 0.2 r_c$. The probability of finding a particle within the cutoff distance is:

$$\frac{\text{Cutoff Volume}}{\text{Search Volume}} = \frac{\frac{4}{3}\pi r_c^3}{\frac{4}{3}\pi (1.2 r_c)^3} = \frac{1}{1.2^3} \approx 0.579 \tag{2.9}$$

This means that approximately 42% of the distance calculations are unnecessary. A percentage much lower than that for Linked Cells. A disadvantage of this method is the need to maintain and periodically rebuild the neighbor lists. Since rebuilding can be computationally expensive, it is crucial to select a rebuild frequency that strikes a balance between accuracy and performance.

To this end, different strategies for rebuilding neighbor lists can be employed. These strategies may be static, using a predefined number of iterations, or dynamic, where rebuilding occurs when a particle traverses half the skin length. Previous experiments have shown that applying the skin criterion can yield performance improvements of up to 50% in cases with changing particle velocities [11].

Verlet Cluster Lists Method

Based on the observation that neighboring particles often have very similar neighbor lists, Páll and Hess developed the Verlet Cluster Lists method to address some of the drawbacks of traditional Verlet Lists [25]. Instead of maintaining a neighbor list for each individual particle, a fixed number n of particles are grouped into a cluster that shares a single neighbor list. The value of n can be chosen as a multiple of the processor's vector length to enable efficient vectorization [14]. The method reduces the total number of lists by a factor of $\frac{1}{n}$. Furthermore, the neighbor list of a cluster contains other clusters rather than individual particles.

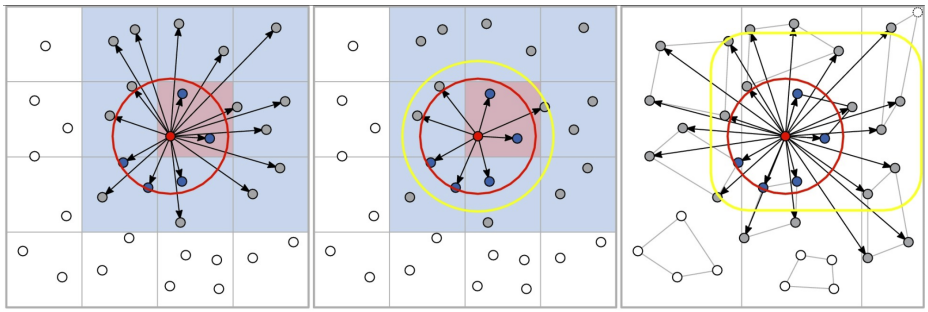


Figure 2.3.: Algorithms for short-range potentials. The cutoff radius is shown in red, and the skin radius is shown in yellow. Arrows depict the particles considered for computation, where only blue particles are included in the end. Source: [14]

2.3. AutoPas

AutoPas is an open-source C++ node-level performance library designed for use in arbitrary N-body simulations. It provides interfaces for applying pairwise forces as well as for accessing particles [13]. In the following, we present an overview of the features and algorithms supported by AutoPas. Some features will be introduced only briefly, as they are not the focus of this thesis. It is important to note that the following information is based on the paper [14], which provides a comprehensive overview of the features of AutoPas.

2.3.1. Newton's Third Law of Motion

According to Newton's third law of motion, if two bodies exert forces on each other, these forces are equal in magnitude but opposite in direction [23]. This principle allows us to reduce the number of pairwise force calculations by half. Consider two particles i and j, and the force acting between them:

$$F_{ij} = -F_{ji} \tag{2.10}$$

During iteration, F_{ij} is calculated once, then added to particle i and subtracted from particle j.

This optimization is not always applicable, as it depends on the type of traversal and the chosen force-calculation algorithm.

2.3.2. Data Layout

Data layout refers to the way particle data is stored in memory. Two options are available: AoS (Array of Structures) and SoA (Structure of Arrays).

In AoS, each particle is represented as an object containing multiple attributes (e.g., x, y, z positions), and these objects are stored in a vector. This layout is more advantageous when accessing multiple attributes of a single particle.

In SoA, each attribute is stored in a separate array, where each entry corresponds to the value of that attribute for one particle. This layout is more efficient for vectorization.

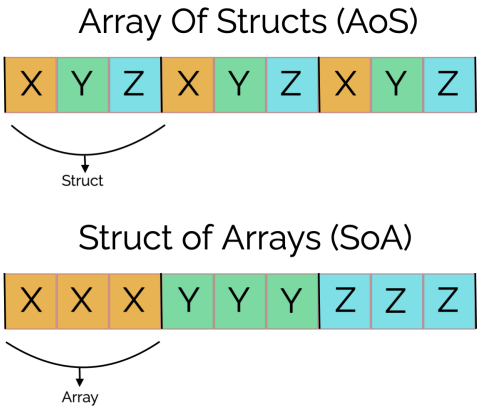


Figure 2.4.: Array of structures (AoS) vs Structure of Arrays (SoA).

2.3.3. Load Estimator

The purpose of load estimators is to evenly distribute work across all processing units. Different load estimators can be used, such as Squared Particles per Cell or Neighbor List Length[9]. The feature can also be disabled entirely.

2.3.4. Cell Size Factor

There are certain trade-offs between choosing a smaller or larger cell size factor. A smaller factor results in a larger number of cells and, consequently, greater memory overhead, while a larger factor leads to more distance calculations. Therefore, selecting a suitable factor is a matter of optimization.

2.3.5. Particle Containers

In AutoPas, particles are stored in various containers, where particle pairs for force calculation are determined based on the container type[13]. AutoPas implements the well-established algorithms introduced earlier—or versions of them—as well as some additional containers.

Direct Sum

In the Direct Sum container, all particle pairs are stored in a single vector. This means that all particles are considered, but only those with a distance smaller than r_c are ultimately included in the calculation. The advantage of this container is its straightforward implementation and lack of additional storage overhead. The drawback, however, is its $\mathcal{O}(N^2)$ runtime complexity.

Linked Cells

This container consists of a vector of cells, where each cell contains its own vector of particles. The Linked Cells container implements the previously described algorithm. In simple terms, it only considers particle pairs located in neighboring cells. Although this container reduces

the time complexity to $\mathcal{O}(N)$, it may still iterate over empty cells or cells that contain particles outside the cutoff radius.

Verlet Lists

The Verlet Lists container implements the algorithm previously described. Each particle maintains a list of potential neighbors considered during force calculations. To efficiently construct these neighbor lists, an instance of the Linked Cells container is used to store the particles. The neighbor lists are implemented as vectors of pointers to particles within the Linked Cells instance. Additionally, maps are used to associate each particle pointer with its corresponding neighbor list.

AutoPas also provides the option to use Var Verlet Lists, a generalized version of the Verlet Lists container. This container offers interfaces that allow custom implementations of neighbor list structures and their generation.

Verlet Lists Cells

The previous implementation of Verlet Lists does not retain any information about the spatial locations of the particles associated with the neighbor lists. To optimize this, the Verlet Lists Cells container links neighbor lists to the cells in which the corresponding particles are stored.

Verlet Cluster Lists

This container is organized as a two-dimensional grid of towers laid out in the xy-plane. Inside each tower, particles are stored in a single vector, ordered by their position along the z-axis, along with a SoABuffer. Each cluster contains only a pointer to its first particle, a SoAView referencing the same buffer, and a list of pointers to neighbor clusters.

2.3.6. Static vs Dynamic Containers

AutoPas has recently introduced a dynamic rebuild criterion for its containers. Previously, rebuilds could only be triggered after a fixed number of iterations—a metric referred to as the *rebuild frequency*. Containers that support the dynamic rebuild criterion are referred to as *dynamic* containers, whereas containers with a fixed rebuild frequency are referred to as *static* containers.

In dynamic Verlet Lists-based containers, neighbor lists are not rebuilt based on a fixed user-defined frequency. Instead, they are updated when any particle moves more than half the skin length. This approach accounts for the worst-case scenario, where two particles move toward each other in opposite directions. This is illustrated by Figure 2.5. Initially, the distance between them is $r_c + r_{\text{skin}}$. Once each particle has moved a distance of $\frac{r_{\text{skin}}}{2}$, their neighbor lists become invalid. Checking for this case ensures that all other movement scenarios are covered.

Dynamic Linked Cells containers follow a similar principle. Here, the skin length is added as a margin to the cell size. When a particle travels a distance of $\frac{r_{\text{skin}}}{2}$, the particle-cell assignments are updated accordingly.

In AutoPas, dynamic containers are enabled by default. This behavior can be modified during compilation.

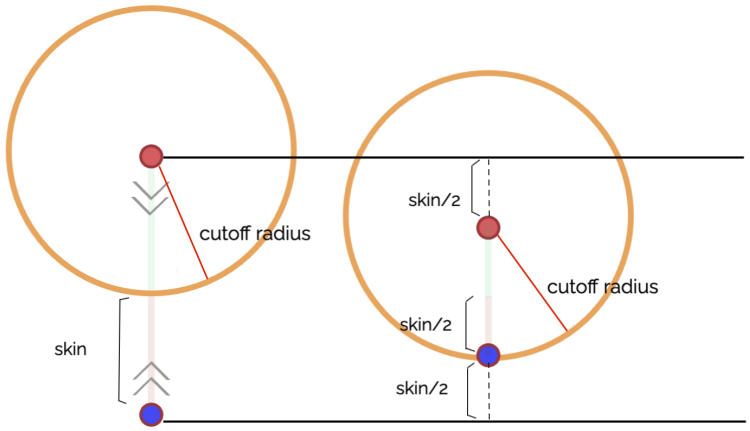


Figure 2.5.: Visualization of dynamic rebuild criterion for Verlet Lists in AutoPas. Here, the worst case scenario of two particles moving towards each other is displayed.

2.3.7. Traversal Patterns

Traversal patterns define how particles are iterated over during force calculations and are optimized based on the container type[14]. The choice of traversal also determines whether certain optimizations, such as Newton's third law (Newton 3), can be applied.

Direct Sum Traversals

AutoPas provides a single traversal option for the Direct Sum container: `ds_sequential`. This option processes all node-local particle pairs sequentially.

Linked Cells Traversals

The Linked Cells method consists of two levels of iteration. It first iterates over the cells in the domain and then over the particles within each cell. Therefore, Linked Cells traversals are grouped into two categories that reflect this structure. Table 2.1 provides an overview of the traversal strategies and their description in Autopas. The traversals that occur inside a cell are referred to as the base step.

Verlet Lists Traversals

Similar to Direct Sum, AutoPas does not support parallel traversals for Verlet Lists when Newton 3 is enabled. The only available traversal in this case is `vl_list_iteration`.

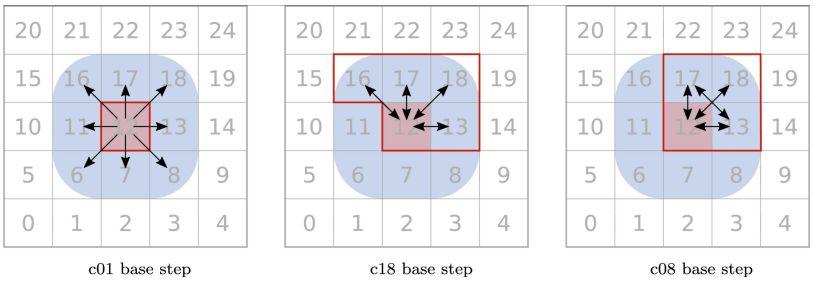


Figure 2.6.: Base step traversals. Source: [14]

Traversal(s)	Description
`c01`, `c08`, `c18`	Base steps for in-cell iterations
`lc_c01`, `lc_c08`, `lc_c18`	Linked cell traversals
`lc_c01_combined_SoA`, `lc_c04_combined_SoA`	Optimized for SoA layout
`lc_c04`, `lc_c04HCP`	Specialized variants
`lc_sliced`	Domain-slicing traversal

Table 2.1.: Traversal variants for Linked Cells in AutoPas and their descriptions

Var Verlet Lists Traversals

The only traversal pattern available for this container is `vvl_as_built`. This adaptive traversal method replicates the traversal pattern used to build the Verlet Lists. It records which thread creates which neighbor list during each phase of the traversal. This information is then used to apply the same scheduling and traversal pattern in the subsequent iteration.

Verlet Lists Cells Traversals

This container includes three traversal options: `vlc_c01` and `vlc_c18`, which process all neighbor lists stored in a cell in a manner similar to the base step traversals of Linked Cells `c01` and `c18`, respectively. The `vlc_sliced` traversal operates similarly to its counterpart in Linked Cells, dividing the domain into slices based on the number of threads.

Verlet Cluster Lists Traversals

This container also includes three traversal options. `vcl_clusterIteration` schedules towers to threads, and each thread processes the clusters in each tower. In `vcl_c01Balanced`, Newton 3 is disabled, resulting in a parallel traversal of clusters. A load-balanced scheduling is also employed. In `vcl_c06` traversal, a domain coloring approach is used, with the pattern depending on whether the Newton 3 optimization is enabled.

2.3.8. Auto-Tuning

An important feature of AutoPas is Auto-Tuning: a periodic assessment of the optimal configuration depending on performance metrics. A configuration is a permutation of the different tunable parameters, which include: container type, traversal, data layout, Newton 3, load estimator, and cell size factor.

Sampling

To be able to compare different configurations, AutoPas collects multiple samples for each configuration. Each sample is the time measured for one iteration with a specific configuration. The values of these measurements are reduced to one based on different methods, e.g., the minimum sample.

Optimum Selection

To calculate the optimum configuration, the sample values are reduced to a single value. First, the samples are classified into two categories: rebuild samples and regular samples. The rebuild samples are the iterations in which a rebuild took place, while the regular samples are the normal pairwise force-calculation iterations without the additional rebuild overhead. Each category is then reduced to a single value using a given mechanism, e.g., minimum, average. Assuming the reduced values for each category are described as C_{rebuild} and C_{regular}, we calculate the final value using the following formula:

$$\frac{C_{\text{rebuild}} + (rf - 1)C_{\text{regular}}}{rf} \tag{2.11}$$

Where rf is the rebuild frequency. The latter formula reflects a weighting of rebuild iterations and regular iterations according to the rebuild frequency.

Dynamic Rebuild Frequency for Different Configurations

An important observation is that, in dynamic containers, all configurations of a simulation share the same rebuild frequency, provided that the skin remains unchanged and thus the rebuild criterion is identical across configurations.

To verify this, we ran the simulation defined in `fallingDrop.yaml` using four different configurations, and compared the observed rebuild frequencies. The results confirmed our assumption. The output files of the simulations can be found on Github (Link in Appendix).

Tuning Strategies

Multiple strategies are available for selecting the optimal configuration. An overview of these strategies and their descriptions is provided in Table 2.2. In addition to the strategies currently implemented in AutoPas, other approaches employing machine learning concepts have also been investigated. In particular, the *Random Forest* strategy proposed by Newcome et al. [22] has shown promising results. However, it is important to note that the latter method was evaluated only on static containers with a rebuild frequency of 10.

Techniques for early stopping of the tuning process have also been explored. In some cases, when a sample of a particular configuration exhibits significantly worse performance than the best-known configuration, it can be beneficial to terminate further sampling of that configuration [19].

Strategy	Description
`FullSearch`	Evaluates all possible configurations
`RandomSearch`	Randomly tests a provided number of configurations
`PredictiveTuning`	Uses extrapolation of previous measurements to estimate the runtime of each configuration and evaluates configurations with the best predictions.
`BayesianSearch`	Assumes that the stochastic distribution of the execution time corresponds to a Gaussian Process. Based on that, uses previous measurements to select the next configuration to evaluate.
`RuleBasedTuning`	Uses expert knowledge to exclude undesirable configurations, where the knowledge is encoded as rules in a rule file.
`FuzzyTuning`	Uses fuzzy logic to predict performance of configurations in a fashion similar to RuleBasedTuning.

Table 2.2.: Tuning strategies in AutoPas and their descriptions

3. Related Work

The unique nature of dynamic containers makes it difficult to find closely related work. Most other software designed for particle simulations does not use Verlet lists with a dynamic rebuild feature (such as the $\frac{r_{\text{skin}}}{2}$ approach), let alone evaluate approximate rebuild frequencies for tuning purposes. Below is an overview of several particle simulation software packages:

- **ls1 mardyn**: an open-source program for molecular dynamics simulations restricted to rigid molecules and constant volume ensembles[24]. It is optimized for massively parallel execution on supercomputing architectures and is thus highly scalable. The program was employed in different research fields such as chemical and process engineering due to its ability to simulate a large number of particles[15]. Internally, the program relies on an adaptive method of Linked Cells for memory efficiency and load balancing methods to support challenging heterogeneous configurations[24]. More recent developments of the program include the introduction of a plugin framework, which allows for both easier maintainability and extendability and integration of user-code, as well as the integration of AutoPas to enable Auto-Tuning[27].
- **Lammps:** an open-source program for modeling particle interactions written in C++ that was designed for distributed-memory parallelism using MPI. It is primarily restricted to short-range interactions, and particle densities are moderately bounded[31]. In LAMMPS, the user is able to choose which rebuild criteria is employed for Verlet Lists. Typically, neighbor lists are rebuilt after a particle moves a distance of half a skin length [17]. LAMMPS is used actively in material science[21, 20].
- **OpenFPM:** an open and scalable framework that provides an abstraction layer for numerical simulations using particles, meshes and their combination. It provides software components that handle all parallelization and memory, reducing program development times and making parallel computing accessible to users without much knowledge in the field. For iterating through the particles, the user may use both Verlet Lists and Cell Lists[3, 16]. To update the lists, OpenFPM provides function like updateCellList(), updateVerlet()[30].
- **CoPA Cabana:** a performance portable library for particle-based simulations. Cabana builds on Kokkos, where it is used for on-node parallelism, enabling simulation on multi-core CPU and GPU architectures, and MPI for GPU-aware, multi-node communication[28]. Cabana aims to provide high-performance scientific codes that are applicable across different domains. Unlike AutoPas, the decisions are left to the user in choosing optimal configurations. For particle iterations, Cabana focuses on Verlet Lists. A Linked Cells interface is also available.

Despite the limited use of rebuild-frequency estimation for tuning purposes, there is existing literature on how to select a static rebuild frequency or skin length for Verlet lists to ensure optimal or accurate simulations [6, 2].

In LAMMPS, a short simulation can be run using a specific command to count the number of neighbor list rebuilds. This number, combined with the total number of iterations, can be used to estimate the rebuild frequency [10].

Another approach for estimating a static rebuild frequency is based on the equation provided by Verlet in his original paper [32]:

$$r_{\text{skin}} \geq (rf - 1) * \delta t * \bar{v} \tag{3.1}$$

where δt is the time step and $\bar{v}$ is the root-mean-square velocity.

However, it is important to note that using an average velocity does not prevent some particles from crossing the skin within the estimated number of time steps. Therefore, this method does not yield a conservative estimate of the rebuild frequency rf [6].

4. Technical Background

In this chapter, we present an overview of the AutoPas structure. We also describe the technical procedures for conducting experiments and extracting results, as these details are essential for interpreting the findings.

4.1. Overview of Software Design

AutoPas provides an interface that allows users to implement both the particles to be simulated and the pairwise force interactions, referred to as *functors*. To support this, AutoPas is accompanied by example simulations (e.g., `md-flexible`) that employ the AutoPas library and demonstrate how these simulations can utilize its features.

The main point of interaction for the user is the `AutoPas` class, which offers various functions that can be integrated into the simulation workflow.

Container-related internal logic is handled by the `LogicHandler` class, which ensures that all containers remain in a valid and consistent state. This class is also where we implement the mean rebuild frequency estimation method described later, since it is responsible for setting a global container rebuild frequency across all containers.

The logic of the AutoPas auto-tuning mechanism is managed by the `AutoTuner` class. This component is responsible for tasks such as collecting samples, managing the search space and configurations, and other tuning-related operations. It also provides utility functions that supply information about the tuning process, which our method relies on.

Finally, additional classes are available to handle specialized tasks such as logging tuning data or implementing different tuning strategies. A comprehensive overview of the AutoPas structure and its classes can be found in the official documentation.

4.2. Tuning Logs and Simulation Summary

We begin by analyzing the tuning logs, which record the results of the tuning phase for each configuration in a CSV file at the end of the simulation. Tuning loggers are enabled at build time and follow a specific format. In the example below, the tunable parameters *Cell Size Factor* and *Load Estimator* have been omitted for brevity.

It is also possible to have multiple tuning phases by configuring the tuning interval, which defines the number of iterations before a new tuning phase is triggered. The results of each tuning phase are appended to the same log file, providing a comprehensive view across the simulation run.

In the above table, the *Weighted* value is calculated as follows: First, samples that include rebuild overhead—*Sample 0* in our case—are aggregated into a single value using the specified reduction method, which is *minimum* as defined in the input file. The same

Container	Traversal	DL	Newton 3	Sample 0	Sample 1	Sample 2	Weighted
LinkedCells	lc_c01	AoS	disabled	6609635526	6081618008	6004369853	6009413733
LinkedCells	lc_c01	SoA	disabled	3171897828	2302624478	2329252282	2309868422
		...					
VerletClusterLists	vcl_c06	AoS	disabled	53905480168	23956226646	24037097271	24205803758
		...					
VerletListsCells	vlc_c01	AoS	disabled	14435479391	2318367261	2342268382	2419343195
VerletListsCells	vlc_sliced_c02	AoS	enabled	10004501112	1564730462	1588122509	1635061884
		...					

Table 4.1.: The table displays the layout of the csv files printed at the end of the simulation. Different permutations of tunable parameters are chosen to be evaluated by taking three samples from each. The samples are then weighted according to rebuild frequency and reduced to one value (Last column). The first Sample 0 is always a sample with rebuild overhead since it is the first iteration of the simulation.

reduction method is applied to the samples without rebuild overhead—*Sample 1* and *Sample 2* in this example.

The resulting values are then substituted into Equation 2.11 as C_{rebuild} and C_{regular}, respectively, to compute the final weighted value for each configuration.

A summary of the simulation is also printed at the end. It includes the number of particles, their type, the duration of each simulation phase along with its percentage of the total simulation time, and most importantly: the *mean rebuild frequency* of the last non-tuning phase. The latter is obtained by dividing the total number of iterations by the number of rebuilds in the final non-tuning phase of the simulation. Our objective is to develop a method that estimates the mean rebuild frequency for each non-tuning phase. When inserted into Equation 2.11 as rf, the mean rebuild frequency yields a correct estimate of the runtime of a single iteration in the subsequent non-tuning phase.

Assume the following definitions:

- N_{rebuild}, N_{regular}: the total number of iterations with and without rebuilds in the non-tuning phase, respectively,
- $N_{\text{iteration}}$: the total number of iterations in the non-tuning phase,
- C_{rebuild}, C_{regular}: the estimated runtimes of one iteration with and without rebuild, respectively.

Then, using the mean rebuild frequency as rf in $\frac{C_{\text{rebuild}}+(rf-1)C_{\text{regular}}}{rf}$, we obtain

$$\frac{C_{\text{rebuild}} + \left(\frac{N_{\text{iteration}}}{N_{\text{rebuild}}} - 1\right) C_{\text{regular}}}{\frac{N_{\text{iteration}}}{N_{\text{rebuild}}}} = \frac{N_{\text{rebuild}} \cdot C_{\text{rebuild}}}{N_{\text{iteration}}} + \left(1 - \frac{N_{\text{rebuild}}}{N_{\text{iteration}}}\right) C_{\text{regular}} \tag{4.1}$$

$$= \frac{N_{\text{rebuild}} \cdot C_{\text{rebuild}}}{N_{\text{iteration}}} + \frac{N_{\text{regular}}}{N_{\text{iteration}}} \cdot C_{\text{regular}} \tag{4.2}$$

$$= \frac{N_{\text{rebuild}} \cdot C_{\text{rebuild}} + N_{\text{regular}} \cdot C_{\text{regular}}}{N_{\text{iteration}}} = \tag{4.3}$$

Estimated runtime of one iteration in the non-tuning phase.

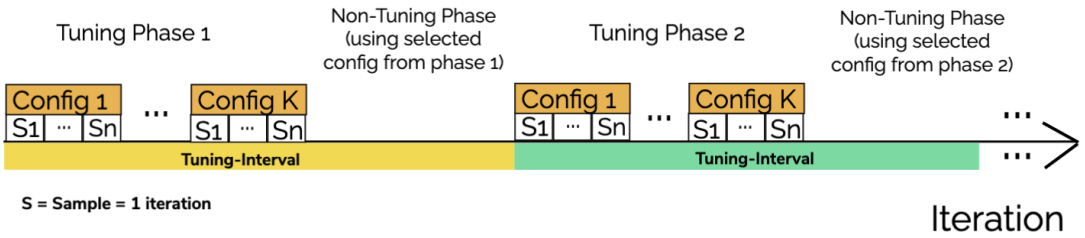


Figure 4.1.: Visualization of the simulation timeline, where n is the specified number of samples, K is the number of configurations considered. K is dependent on the tuning-strategy.

5. Motivation

In the following, we evaluate the current implementation through a series of experiments. Additionally, we investigate a simple alternative method and highlight the necessity of developing more effective approaches.

5.1. Assessment of the Current Approach

The current method uses the user-provided rebuild frequency as an initial estimate of the parameter rf in Equation 2.11 during the first tuning phase. This value also serves as an upper bound for the rebuild frequency during the entire simulation for reasons regarding the implementation of a buffer that are beyond the scope of the thesis. In subsequent tuning phases, the mean rebuild frequency calculated from **previous** non-tuning phases is employed, providing a more accurate estimate of the actual value.

It is evident that the rf value used in the first tuning phase can be highly inaccurate—particularly when the user's initial estimate deviates significantly from the true rebuild frequency. To illustrate this, we conducted a series of experiments where an extreme estimate for rf (e.g., 120) was intentionally provided, and the simulation was executed with only one tuning phase followed by a single non-tuning phase. After completion, the actual mean rebuild frequency observed during that phase was substituted back into the equation to evaluate its effect on the tuning results.

5.1.1. Experiment 1 (Falling Drop)

We ran the simulation defined in `fallingDrop.yaml` for 4000 iterations using a rebuild frequency of 120, enabling only a single tuning phase. According to the tuning loggers, the following configuration was selected as optimal: `[LinkedCells, lc_sliced_c02, SoA, (Newton 3) enabled]`. The full ranking of all configurations based on their weighted values is available on GitHub.

The simulation summary at the end reports a mean rebuild frequency of approximately 62.8548 during the non-tuning phase. Substituting $rf = 62.8548$ into Equation 2.11 and recomputing the weighted values for all configurations yields not only the same optimal configuration but also an identical overall ranking.

In this particular case, it appears that the ranking of samples—with or without rebuild overhead—remains unchanged. This suggests that, for this example, the ranking is invariant under different weighting ratios applied to the samples.

5.1.2. Experiment 2 (Spinodal Decomposition Equilibration)

We ran the simulation defined in `SpinodalDecomposition_equilibration.yaml` for 4000 iterations using a rebuild frequency of 120, enabling a single tuning phase. The follow-

ing configuration was selected as optimal: `[LinkedCells, lc_sliced, SoA, (Newton 3) enabled]`.

Similar to the previous experiment, we replaced the rebuild frequency in Equation 2.11 with the actual mean rebuild frequency reported in the summary, $rf = 20.45$. This resulted in a different optimal configuration: `[LinkedCells, lc_sliced_balanced, SoA, (Newton 3) enabled]`.

However, the weighted values for these two configurations were very close. Re-running the simulation yielded slightly different results, likely due to minor fluctuations caused by system noise, with the optimal configuration again being `[LinkedCells, lc_sliced_balanced, SoA, (Newton 3) enabled]` for both values of rf.

In this particular case, the reason for the identical optimal configuration lies in the fact that both components of the weighted value (i.e., the samples with rebuild overhead $C_{rebuild}$ and the samples without $C_{regular}$) are minimal for the same configuration. Therefore, the choice of rf for weighing the contributions becomes irrelevant.

Nevertheless, ranking the configurations based on their weighted values for different rf values results in non-identical orderings. This implies that if the pool of available configurations is reduced, different optimal configurations may be selected. For example, consider the two configurations in table 5.1.

Strategy	**Weighted** ($rf = 120$)	**Weighted** ($rf = 20.45$)
VerletListsCells,vlc_sliced,SoA	1,625,302,755	2,381,490,279
VerletListsCells,vlc_sliced_balanced,AoS	1,626,859,536	1,956,884,521

Table 5.1.: Weighted value when using $rf = 120$ vs. $rf = 20.45$.

Clearly, changing the value of rf has a noticeable impact on the relative ranking of configurations. In this case, the order of the two configurations is reversed. This observation underscores the fact that relying solely on a user-provided estimate for rf does not always yield the optimal configuration. Therefore, a more accurate estimation method is crucial for reliable tuning results.

5.1.3. Experiment 3 (Exploding Liquid)

We run the simulation `explodingLiquid.yaml` for 4000 iterations. Using a rebuild frequency of 120, we get the following optimal configuration: `[VerletListsCells, vlc_sliced_c02, AoS, (Newton 3) enabled]`.

Here again, inserting the mean rebuild frequency $rf = 14.186$, we obtain a different ranking of configurations. Although the optimal configuration remains unchanged for the different rf values, the second and third configurations switch ranks.

Further analysis shows that the C_{regular} component of Equation 2.11 for the optimal configuration is not the minimum. This indicates that there exists a user-provided rf value for which the optimal configuration differs from the one obtained using the actual mean rebuild frequency.

The best way to illustrate this is by plotting the weighted value of the following two configurations as a function of rf:

- **Configuration 1:** `[LinkedCells, lc_c04, SoA, (Newton 3) enabled]`

- **Configuration 2:** `[VerletListsCells, vlc_sliced_c02, AoS, (Newton 3) enabled]`

We chose Configuration 1 because it has the minimal C_{rebuild} component, and Configuration 2 because it has the minimal C_{regular} component—making both configurations potential candidates for optimality.

From the graph in Figure 5.1, we can infer that there exists a critical value of rf—any value below approximately 6—at which Configuration 1 becomes more optimal than Configuration 2. However, because the user-specified input also serves as an upper bound for the rebuild frequency, choosing such a small value for the next experiment forces the mean rebuild frequency to coincide with this upper bound, thereby making the optimal configuration invariant.

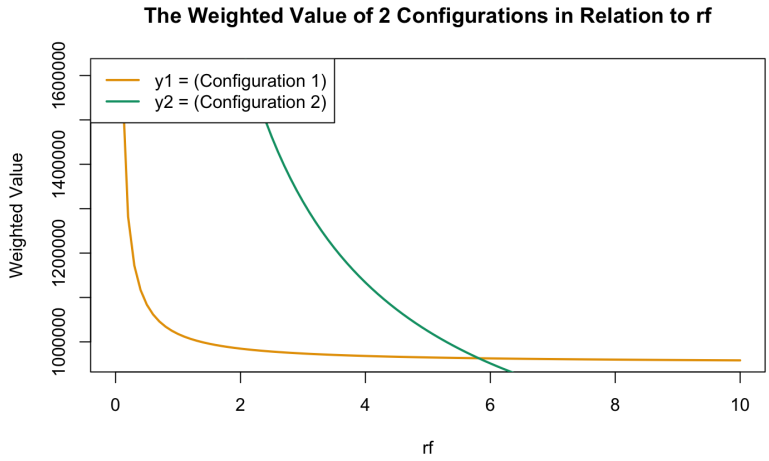


Figure 5.1.: Weighted values based on collected samples for two candidate configurations plotted against varying rebuild frequencies. The plot illustrates how the optimal configuration may change depending on the chosen rebuild frequency.

5.1.4. Experiment 4 (Heating Sphere)

We run the simulation `heatingSphere.yaml` for 4000 iterations and obtain the following optimal configuration for $rf = 120$: `[VerletListsCells, vlc_sliced_c02, AoS, (Newton 3) enabled]`.

When we substitute the actual mean rebuild frequency, $rf = 38.9362$, we find that the optimal configuration remains unchanged. However, the overall ranking of the configurations differs, indicating that the choice of rf still influences the ordering of non-optimal configurations, which may matter when filtering the configuration space.

5.1.5. Experiment 5 (Rayleigh–Taylor)

Finally, we run the simulation `rayleighTaylor.yaml` for 4000 iterations using $rf = 120$. The tuning process selects the following as the optimal configuration: `[VerletListsCells, vlc_c01, SoA, (Newton 3) disabled]`.

However, when we substitute the actual mean rebuild frequency from the summary, $rf = 3.82445$, into Equation 2.11, the optimal configuration changes to: `[LinkedCells, lc_c08, SoA, (Newton 3) enabled]`.

This example demonstrates a clear case in which an inaccurate initial estimate for rf leads to a suboptimal configuration being selected.

5.2. Assessment of the Intuitive Approach: Simulation Method

the *Simulation Method* builds upon the intuitive assumption that running the simulation for a few iterations and recording the number of rebuilds may be sufficient to be able to estimate the mean rebuild frequency for the first non-tuning phase.

The idea behind the method is to estimate the mean rebuild frequency by running a mock simulation before the actual one for a predetermined number of iterations and counting the number of rebuilds that occur. Alternatively, one may fix the number of rebuilds and count how many iterations are required to reach that count. In both approaches, the estimated mean rebuild frequency is computed by dividing the total number of iterations by the number of rebuilds.

This method is intended solely for estimating the mean rebuild frequency of the first non-tuning phase. Subsequent phases may simply reuse the estimated value from earlier ones.

A major drawback of this approach is the potential for significant overhead from the additional simulation iterations, particularly in the fixed-rebuild-count case, where reaching the target number may require many iterations. Furthermore, because the optimal configuration is not known prior to the tuning process, the mock simulation may be run using a suboptimal configuration. This can result in even greater overhead.

There is also a risk that the estimated mean rebuild frequency does not accurately reflect the true behavior of the first non-tuning phase. This is particularly relevant for simulations in which the maximum velocity varies drastically in the early iterations.

To illustrate this limitation, consider Figure 5.2a, which shows the mean rebuild frequency for the Falling Drop scenario without any tuning intervals (by specifying only one configuration in the input file).

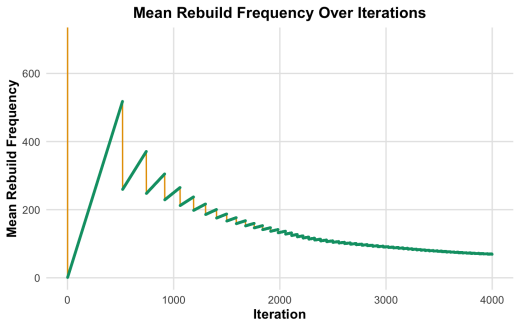
Using the *Simulation Method*, we aim to estimate the mean rebuild frequency for the first non-tuning phase—the period after tuning until the end of the simulation. The data in the figure allows us to evaluate the method's accuracy.

If the mock simulation is run for 1000 iterations, the estimated mean rebuild frequency is approximately 225, while the actual value (recorded at iteration 4000) is closer to 67.

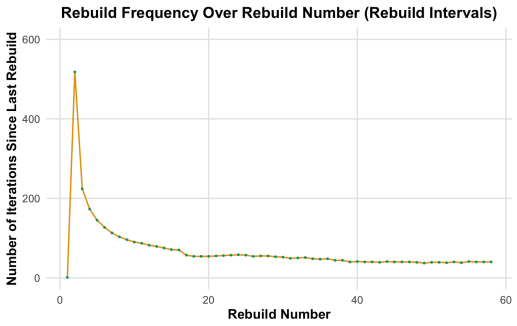
Alternatively, fixing the number of rebuilds requires examining the rebuild intervals in Figure 5.2b. The first rebuild occurs only after more than 400 iterations—far from an accurate representation of the mean rebuild frequency.

Another important detail is that the previously conducted simulation does not include any tuning phases, which results in a slightly different mean rebuild frequency compared to the case with one tuning phase at the beginning. This difference arises because the iterations that would otherwise belong to the initial tuning phase—and thus be excluded from the mean calculation—are now included. This highlights another limitation of the method: depending on the number of chosen mock iterations, they may overlap with the

tuning phase in the real simulation, causing the calculated mean to rely on data that does not reflect the non-tuning phase, particularly the data from early iterations.



(a) The mean rebuild frequency ($N_{iterations}/N_{rebuilds}$) at each iteration in the Falling Drop simulation for the first 4000 iterations. This simulation was run without any tuning phases.



(b) The number of iterations between successive rebuilds (rebuild intervals) in the first 4000 iterations of the Falling Drop simulation. This simulation was run without any tuning phases.

5.3. Reflection and Conclusion

We conclude this chapter by emphasizing the need for more suitable methods that address the previously discussed limitations—an objective that forms the core focus of this thesis.

Both the current implementation and the *Simulation Method* fail to provide an estimation mechanism that effectively balances performance and accuracy.

The current implementation, while incurring no significant overhead, relies solely on user-provided estimates for the first non-tuning phase, which are often unreliable. The *Simulation Method*, in contrast, can require a large number of iterations to achieve an accurate estimate, undermining the core purpose of auto-tuning, which is to improve performance.

6. Implementation

As stated in the related work section, most of the literature concerned with the assessment of the rebuild frequency of Verlet Lists aims to guide either the user or the program in selecting a suitable static rebuild frequency for a given skin. Although such approaches are not directly applicable to our case—since the rebuild is triggered dynamically when the displacement criterion is met—they still offer valuable inspiration for estimating rebuild frequencies in a principled manner.

One such inspiration is drawn from Verlet's original paper [32], where a formula is proposed to estimate the rebuild frequency. We refer to this as the *Velocity Method*, which will be described in detail later. This approach is implemented due to its simplicity and promising performance in preliminary evaluations.

In addition to the Velocity Method, we discuss another strategy that builds upon it: the *Extrapolation Method*. While this method will not be implemented, it is presented as a theoretical alternative that could be explored in future work.

6.1. Velocity Method

This approach builds on the observation that the magnitude of the maximum particle velocity in the container can be used to estimate the number of iterations required for the fastest particle to traverse half the skin length, under the assumption that this maximum velocity remains relatively constant over the subsequent iterations. The resulting value provides a reliable local estimate and, in cases where the maximum velocity is nearly constant, an accurate estimate of the **mean** rebuild frequency.

The following formula expresses this relationship:

$$\frac{r_{\text{skin}}}{2} = (rf - 1)\delta t\, v_{\max} \tag{6.1}$$

where:

- r_{skin} is the skin length,
- δt is the time step,
- $v_{\max}$ is the magnitude of the maximum velocity in a given iteration,
- rf is the rebuild frequency.

This equation is based on the requirement that, between two rebuilds, no particle should travel more than half the skin length—a conservative criterion applied in dynamic containers in AutoPas. The equality holds since a rebuild is triggered when a particle has traversed half the skin length.

Solving for rf yields an expression that allows us to compute an estimate of the rebuild frequency:

$$rf = \frac{r_{\text{skin}}}{2\,\delta t\,v_{\max}} + 1 \tag{6.2}$$

Since we have previously established that all configurations share the same rebuild frequency (given identical skins), the resulting estimate can be used for all configurations within the same simulation.

An important detail of this method is the underlying assumption that the maximum velocity observed at the beginning of the simulation is more or less representative of the following non-tuning phase. However, it is worth mentioning that this assumption does not hold true for all simulations. We will demonstrate this by conducting experiments across different simulation files with varying initial conditions.

The advantage of this method lies in the predictive nature of a constant velocity: By recording the maximum velocity once, we can estimate the number of iterations required for the fastest particle to traverse half the skin length, thereby obtaining a reliable **local** estimate of the rebuild frequency for the subsequent iterations. In contrast, the *Simulation Method* requires running the simulation for multiple iterations to achieve a similar estimate. Consider, for instance, a simulation with a rebuild frequency of 30 that remains relatively constant at the beginning. To estimate this value using the *Simulation Method*, the mock simulation must be run for at least 30 iterations. In contrast, the *Velocity Method* requires only recording the velocity in a single early iteration to obtain a reasonably accurate estimate.

A key decision in the implementation is determining which of the maximum velocities from the tuning iterations should be used in Equation 6.2. Intuitively, the further into the simulation we are, the more representative the observed maximum velocity becomes of subsequent iterations. This suggests that the velocity from the last tuning iteration should be used.

However, due to the current software implementation, calculating and setting the rebuild frequency estimate during the final tuning iteration would affect only the last configuration. All previous configurations would still rely on the initial user-defined rebuild frequency. This limitation cannot be circumvented, as many tuning strategies require the rebuild frequency value to be available for every configuration. Therefore, the latest point at which a uniform estimate can be applied across all configurations is the final tuning sample of the first configuration.

An overview of the algorithm used is shown in 2. In short, the method first determines the magnitude of the maximum velocity during the selected tuning iteration (i.e., the last sample of the first configuration). It then computes the estimated rebuild frequency using Equation 6.2. If the estimate exceeds the user-defined upper bound for the rebuild frequency, the upper bound is applied. (The user is expected to specify a sufficiently large upper bound to avoid unnecessary rebuild overhead.) Otherwise, the estimated value is used.

6.1.1. Results and Reflection — First Non-Tuning Phase

We test our method by conducting different experiments with various simulation files. We specify a user-provided upper-bound of 4000, and run all simulations for 4000 iterations with one tuning phase at the beginning. Table 6.1 summarizes the estimated and actual mean rebuild frequencies observed during the first non-tuning phase.

Algorithm 2: The Velocity Method used to calculate an estimated mean rebuild frequency for the following non-tuning phase. The estimate will be used to choose an optimal configuration.

Input : Skin s, Time step Δt, User-provided rebuild frequency rf_{user}

if *autoTuner.inFirstConfigurationLastSample()* **then**
  $v_{\max} \leftarrow$ getMaximumVelocityInTheContainer()
  $rf_{\text{est}} \leftarrow \left\lfloor \frac{s}{v_{\max} \cdot \Delta t \cdot 2} + 1 \right\rfloor$
  if $rf_{est} > rf_{user}$ **then**
    autoTuner.setRebuildFrequency(rf_{user})
  else
    autoTuner.setRebuildFrequency(rf_{est})

Scenario	Est. Mean rf	Real Mean rf	Relative Error (%)
Falling Drop	4000	67.19	5853.2%
Spinodal D.E.	24.14	23.31	3.5%
Exploding Liquid	260.92	14.42	1709.4%
Heating Sphere	128.13	41.8	206.5%
Rayleigh Taylor	4.45	3.89	14.3%

Table 6.1.: Comparison of the estimated and observed mean rebuild frequencies in the first non-tuning phase using the Velocity Method across different simulation scenarios.

Experiment 1 (Falling Drop)

We begin by testing our method on the `fallingDrop.yaml` simulation. The result of the estimation yields $rf = 4000$, which corresponds to the user-provided upper bound. This suggests that the value computed by the equation is excessively large—more precisely, that the maximum velocity captured at the sampled iteration was too small and not representative of the actual mean maximum velocity during the first non-tuning phase.

In this particular simulation, particles are subject to a global downward force, resulting in constant acceleration. Initially, the particles are at rest with zero velocity. Over time, they gradually accelerate due to the gravitational force acting upon them.

This result highlights a limitation of the velocity-based approach: the assumption that particle velocities remain relatively constant throughout the simulation does not always hold, especially in cases like this, where particles start from rest and undergo sustained acceleration.

Global Force Modification

In the case of `fallingDrop.yaml`, the configuration specifies a global force acting on all particles. One might consider modifying the *Velocity Method* to account for this force by explicitly computing the resulting acceleration using Newton's second law and updating particle velocities accordingly. Specifically, the adjusted velocity for each particle can be calculated as:

$$\vec{v}_{\text{new}} = \vec{v} + \frac{\vec{f}_{\text{global}}}{m} \cdot \delta t \tag{6.3}$$

where $\vec{v}$ is the current velocity, $\vec{f}_{\text{global}}$ is the global force, m is the particle mass, and δt is the duration of acceleration. The new value of $v_{\max}$ is then taken as the maximum magnitude among all updated velocities.

However, this approach introduces a new challenge: it is unclear for how many iterations this acceleration should be assumed to continue. Acceleration depends on multiple factors such as boundary conditions and domain size, making it difficult to determine a universally valid duration. As a result, this proposed modification adds complexity without significantly improving accuracy and is therefore considered unnecessary.

Experiment 2 (Spinodal Decomposition Equilibration)

We apply the estimation method to the simulation file for the Spinodal Decomposition Equilibration scenario. The resulting estimate is approximately $rf = 24.1389$, while the actual mean rebuild frequency is approximately $rf = 23.3121$, representing a deviation of about 3.5% from the real value.

The accuracy of this estimate suggests that the maximum particle velocity remains relatively constant throughout the simulation. This makes the velocity-based estimation method particularly suitable for simulations of this type.

Experiment 3 (Exploding Liquid) and Experiment 4 (Heating Sphere)

In the cases of *Exploding Liquid* and *Heating Sphere*, the estimated rebuild frequencies are again significantly larger than the actual values. This likely indicates that the particle velocities were continuously changing (i.e. gaining velocity due to external force, or increased random velocity to due to increase in temperature) during the tuning iterations, leading to underestimation of the true maximum velocity. As a side note, we ran *Exploding Liquid* both with and without MPI and observed the same mean rebuild frequency.

Experiment 5 (Rayleigh Taylor)

Similar to the *Spinodal Decomposition Equilibration* case, this simulation suggests a relatively constant maximum velocity, as the estimated rebuild frequency closely matches the actual value.

As a result of the accurate estimation, applying the Velocity Method to *Rayleigh Taylor* yields the optimal configuration, in contrast to the currently implemented approach.

6.1.2. Results and Reflection – Second Non-Tuning Phase

We evaluate our method during the second non-tuning phase by running all simulations for 8000 iterations—twice the number of iterations as before—and setting the tuning interval to 4000. This results in two tuning phases.

The outcomes are summarized in Table 6.2.

Scenario	Real Mean rf	Mean rf (Current)	Mean rf (Velocity)	(%)
Falling Drop	32.21	67.21	41.27	+80.53%
Spinodal D.E	23.46	23.77	23.24	+0.38%
Exploding Liquid	14.08	14.19	15.21	-7.24%
Heating Sphere	23.61	40.23	29.59	+45.07%
Rayleigh Taylor	4.00	3.83	4.37	-5.00%

Table 6.2.: Comparison of the Velocity Method and the Current approach for the second non-tuning phase. Improvements (Last Column) are computed relative to the real value. (Velocity Method over Current Approach

Compared to the first non-tuning phase, the estimates for the second phase are noticeably more accurate. This can be attributed to the fact that, in the later stages of the simulation—when samples for the second non-tuning phase are collected—particle velocities have become largely constant. As a result, the assumption of constant velocity underlying the Velocity Method becomes more valid, leading to more precise rebuild frequency estimates.

Comparing with the current approach, we observe that the Velocity Method offers significantly more accurate estimates in several scenarios. In particular, for simulations like *Falling Drop* and *Heating Sphere*, our method yields results that are much closer to the real rebuild frequencies. Conversely, in the few cases where the current implementation performs better—such as *Rayleigh Taylor*—the difference in accuracy is relatively minor.

A closer look at *Falling Drop* helps explain this trend. The estimate from the current implementation relies on the mean rebuild frequency from the first non-tuning phase. However, this includes data from the early stages of the simulation, which are not be representative of the simulation behavior during the second non-tuning phase. In contrast, the Velocity Method bases its estimation on the maximum velocity near the transition into the second non-tuning phase, providing a more relevant and timely snapshot of the system's dynamics.

6.1.3. Performance

For our implemented method, we evaluate the performance overhead caused by iterating through the particles in the container and performing the associated calculations. Since this iteration occurs only once at the beginning of each tuning phase, the runtime overhead depends primarily on the number of particles in the container, the tuning interval, and the total number of iterations.

To quantify this overhead, we measured the time taken during the four simulations, each running for 4000 iterations with tuning performed only once at the start. When measuring the overhead, we account for the `if`-statements that check whether the current iteration corresponds to the last sample of the first configuration (a single `if` statement). Specifically, we record the time immediately before entering the `if`-statement and immediately after exiting its body during each iteration. The measured times are accumulated in a global variable to obtain the total overhead. The results are summarized in Table 6.3 below.

From the table, we can infer that, across all conducted simulations, the contribution of our method to the total runtime is negligible.

To investigate the relationship between the number of particles in the container and the

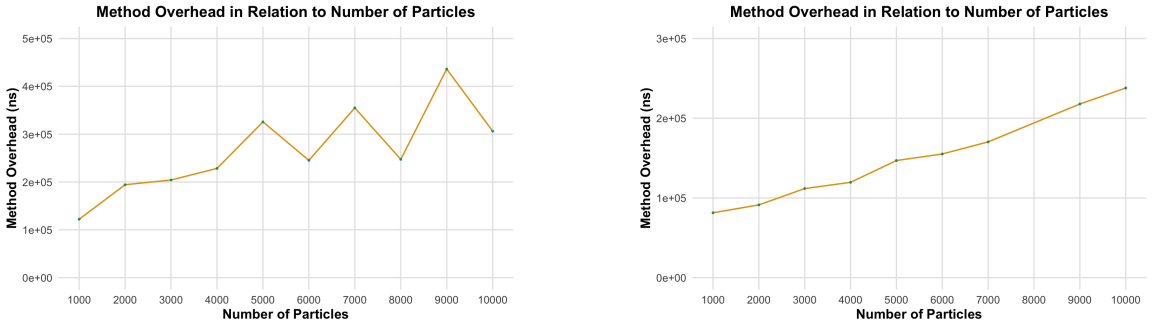
Scenario	Overhead (ns)	Total Runtime (ns)	Overhead (%)
Falling Drop	437186	52849013749	0.0008%
Spinodal D.E.	1288355	173644360895	0.0007%
Exploding Liquid	142116	6441877537	0.0022%
Heating Sphere	2583833	231310457539	0.0011%
Rayleigh Taylor	1200887	298098174462	0.0004%

Table 6.3.: Total runtime and Velocity Method overhead in nanoseconds. The overhead is given in nanoseconds and as a percentage of the total runtime.

overhead introduced by the Velocity Method, we plot the overhead as a function of the number of particles, as shown in Figure 6.1a. To obtain the graph, we used the cluster.

We expect the overhead to grow linearly, since the estimation method performs a single iteration over the particles in the container, with each additional particle contributing a constant processing time. Generally, the graph displays an increasing overhead if we ignore the fluctuations.

Running the simulations again on our local computer multiple times we obtain a slightly different graph (as shown in 6.1b), for which the linearity expectation is confirmed: ignoring minor fluctuations, the increase in runtime for every additional 1000 particles is approximately uniform.



(a) Cluster

(b) Local

Figure 6.1.: Overhead introduced by the Velocity Method as a function of the number of particles. The data is obtained by running the Spinodal Decomposition Equilibration simulation for 4000 iterations with a single tuning interval.

Using the same dataset from the cluster, we also examine the overhead as a percentage of the total runtime across different particle counts. As shown in Table 6.2, the percentage contribution slightly decreases, which could be due to the total runtime increasing at a rate slightly faster than linear, or due to minor runtime fluctuations. In all cases, the overhead remains negligible.

As mentioned above, another factor that influences the overhead is the number of tunings performed. To investigate this, we conduct similar experiments while fixing the number of particles to 1000 and varying the number of tunings. As shown in Figure 6.3, the overhead increases linearly with the number of tunings. This behavior is expected, as each additional tuning step involves one full traversal of the particle container, and with a constant number

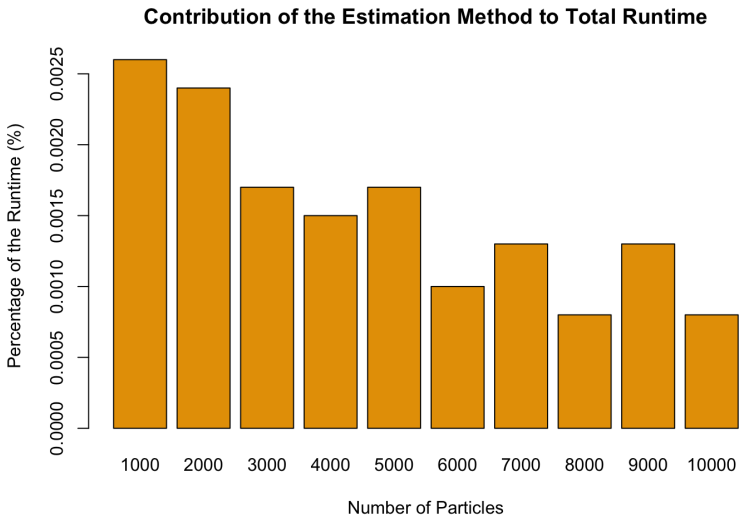
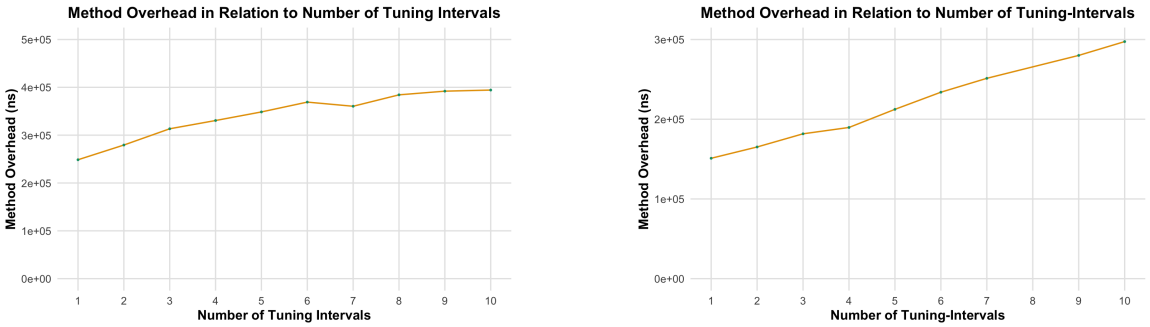


Figure 6.2.: Contribution of the estimation method to the total runtime for different numbers of particles.

of particles, each traversal contributes approximately the same overhead.



(a) Cluster (b) Local

Figure 6.3.: Overhead of the Velocity Method as a function of the number of tunings. The data was obtained by running the Spinodal Decomposition Equilibration simulation for 8000 iterations with a constant number of particles (1000).

In a similar manner, we analyze the percentage of the total runtime attributable to the Velocity Method using data obtained from the cluster. The results are summarized in Figure 6.4. No clear pattern is observed, which may be due to the total runtime depending on the configurations selected during the tuning process.

6.2. Reflections and Analysis

Overall, the proposed Velocity Method was able to accurately predict the actual mean rebuild frequency in cases where the simulation maintained relatively constant particle

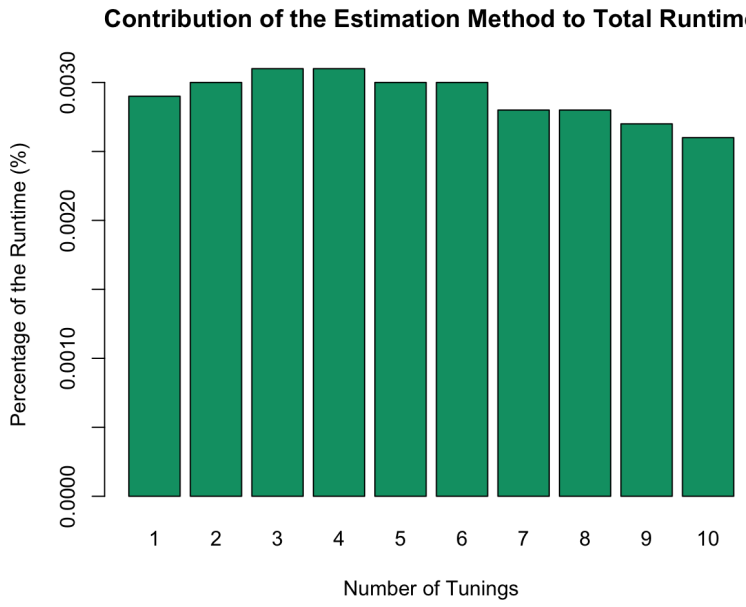


Figure 6.4.: The contribution of the estimation method to the total runtime for different numbers of tunings.

velocities. This was particularly evident in the second non-tuning phase, during which the simulation began to reach a constant state without extreme changes. Additionally, the method poses no disadvantage compared to the currently implemented approach, as it relies on the user-provided value as an upper bound.

To further evaluate and improve the method, we conducted a series of experiments aimed at better understanding the behavior of the maximum velocity and the mean rebuild frequency throughout the simulation.

For each of the previously discussed simulations, we present the following plots:

- The mean rebuild frequency over time.
- Rebuild intervals (i.e., the number of iterations since the previous rebuild, plotted as a function of the rebuild count).
- Maximum velocity over time.
- The estimated mean rebuild frequency using the Velocity Method, applied to the maximum velocity at each iteration.

6.2.1. Falling Drop

Figure 6.5a illustrates the mean rebuild frequency over the course of the Falling Drop simulation. This value is computed by dividing the current total number of iterations by the number of rebuilds that have occurred (excluding tuning iterations). At the beginning of the simulation, the mean rebuild frequency fluctuates significantly, only stabilizing at a later stage. This behavior is related to the way the mean is calculated: each green line represents

a collection of iterations with the same total number of rebuilds, while the vertical orange lines indicate the iterations at which rebuilds actually occurred.

Although the Velocity Method produced a relatively inaccurate estimate of the mean rebuild frequency in this case, it still offers a reasonable estimate for the early stages of the simulation, during which the maximum velocity remained low. This can be verified by analyzing the so-called *rebuild intervals*, which represent the number of iterations between successive rebuilds and can be interpreted as the actual rebuild frequencies throughout the simulation.

As shown in Figure 6.5b, each point represents the number of iterations since the previous rebuild. Rebuilds that occurred during the initial tuning phase (with intervals of 3 iterations) can be ignored. After tuning ends, the first few rebuild intervals exceed 400 iterations—a relatively large value.

Another important observation is that the rebuild intervals gradually decrease and begin to stabilize around the 50^{th} rebuild, where they consistently range between 40 and 50 iterations. However, the mean rebuild frequency in the first non-tuning phase is approximately 67.

As shown in Figure 6.5c, the maximum particle velocity increases approximately linearly at the beginning, then begins to stabilize. This trend clearly violates the assumption of a relatively constant maximum velocity throughout the simulation.

We apply the Velocity Method to the maximum velocity in each iteration and obtain the results shown in Figure 6.5d. The most accurate estimate of the rebuild frequency (approximately 67) appears around iteration 2000. This observation can be explained by the fact that the maximum velocity at that point closely matches the overall mean maximum velocity across the simulation. A deeper connection between the mean maximum velocity and the mean rebuild frequency will be established later when we introduce the *Extrapolation Method*.

6.2.2. Spinodal Decomposition Equilibration

If we examine the Spinodal Decomposition Equilibration simulation in Figure 6.6a, we observe a different trend compared to the Falling Drop case. The graph indicates a relatively constant rebuild frequency from the early stages of the simulation. As discussed previously, our method performs accurately for such scenarios, as the underlying assumption of a nearly constant maximum velocity is fulfilled.

Regarding the rebuild intervals shown in Figure 6.6b, we note that this simulation is not affected by outliers that could significantly skew the mean. This results in a more reliable and representative average rebuild frequency.

To verify the trend of constant values of the maximum velocity throughout the simulation, we refer to Figure 6.6c. The data confirms that the maximum velocity remains relatively constant, varying only between values of 5 and 7. The corresponding velocity-based estimates of the rebuild frequency, obtained by applying the Velocity Method to these values, are shown in Figure 6.6d. For our estimate, we used the maximum velocity observed in the third iteration (since we collect 3 samples), which yields an estimated rebuild frequency of approximately 24—a slight deviation from the actual value.

6.2.3. Exploding Liquid

The graph in Figure 6.7a, which illustrates the mean rebuild frequency for the Exploding Liquid simulation, appears somewhat similar in shape to that of the Spinodal Decomposition Equilibration simulation. However, unlike the latter, the estimated mean rebuild frequency in this case was significantly inaccurate. To understand the reason behind this discrepancy, we examine both the maximum velocity graph in Figure 6.7c and the corresponding rebuild frequency estimation in Figure 6.7d based on the proposed method.

At first glance, it is evident that during the initial iterations, the maximum velocity increases steadily before abruptly stabilizing at a certain point, after which it remains nearly constant until the end of the simulation.

The inaccurate estimate can be attributed to the fact that the maximum velocity recorded from the last sample of the first configuration was still relatively low and in the process of increasing. This resulted in an overestimated rebuild frequency. As indicated in Figure 6.7d, any velocity sample taken after approximately the 100^{th} iteration would have yielded a more accurate estimate of the mean rebuild frequency.

Furthermore, the rebuild-interval graph in Figure 6.7b shows that—excluding the tuning iterations—the rebuild intervals tend to cluster around 14 to 15 iterations. This suggests that despite the initial variation in velocity, the rebuild frequency itself stabilizes fairly early in the simulation.

6.2.4. Heating Sphere

The graph of the Heating Sphere simulation in Figure 6.8a depicts a gradually increasing mean rebuild frequency, which shows signs of stabilization towards the end of the simulation.

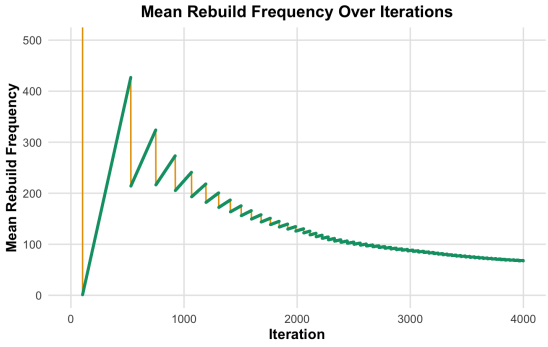
Regarding the rebuild intervals in Figure 6.8b, it can be observed that by the end of the simulation the rebuild frequency converges toward a value close to 30, while the overall mean rebuild frequency remains around 40. In this case, the discrepancy is not due to outliers influencing the mean, but rather a smooth and natural decline in the rebuild frequency over time.

The maximum velocity plot in Figure 6.8c reveals a linearly increasing trend that begins to plateau toward the end, a behavior reminiscent of the Falling Drop simulation.

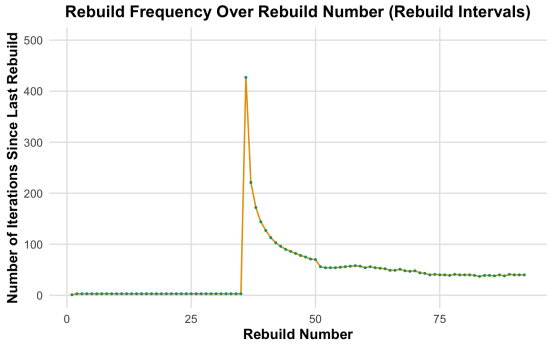
Referring to Figure 6.8d, we find that the *Velocity Method* estimate of the rebuild frequency using the maximum velocity at iteration 4000 is close to 30, which aligns well with the observed rebuild interval towards the end.

6.2.5. Rayleigh Taylor

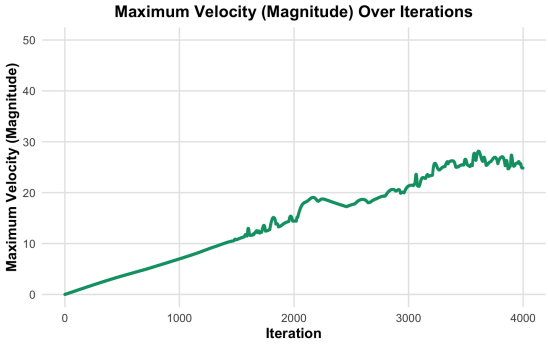
The graph of the maximum velocity for the Rayleigh–Taylor simulation, shown in Figure 6.9c, indicates that the values remain within the range of 28 to 34 throughout the simulation. These velocities correspond to estimated mean rebuild frequencies between 4.0 and 4.6, as illustrated in Figure 6.9d. Regardless of the iteration from which the velocity sample is taken, the resulting estimate remains accurate and consistent with the actual rebuild frequency.



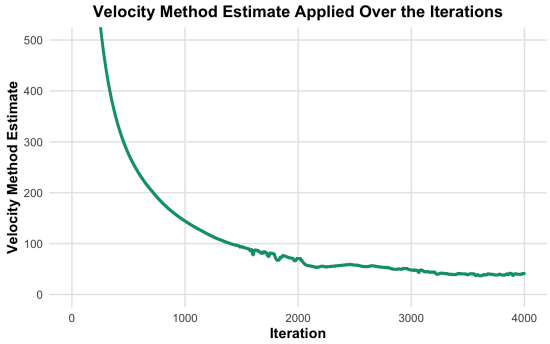
(a) The mean rebuild frequency at each iteration in the Falling Drop simulation, calculated by dividing the current iteration number by the number of rebuilds that have occurred (both excluding tuning iterations).



(b) The number of iterations between successive rebuilds (rebuild intervals) in the first 4000 iterations of the Falling Drop simulation. The first set of rebuilds, which occurred during the tuning phase, can be ignored.

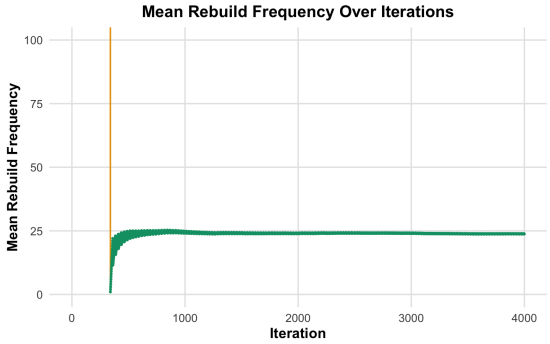


(c) The maximum velocity of the particles in the container at each iteration in the Falling Drop simulation. Data collected over 4000 iterations.

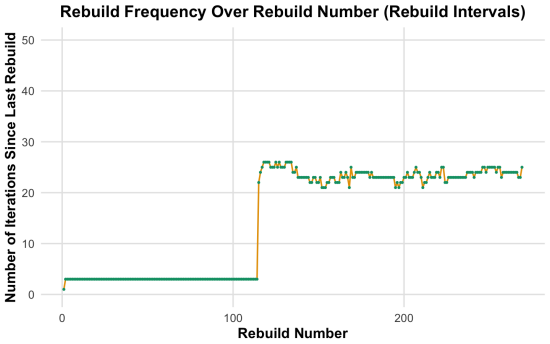


(d) The estimated rebuild frequency using the Velocity Method applied to the maximum velocity at each iteration in the Falling Drop simulation.

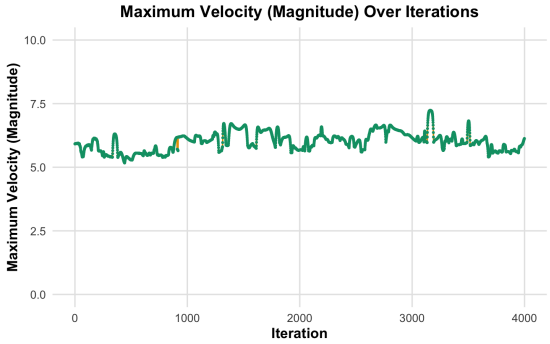
Figure 6.5.: Results from the Falling Drop simulation: (a) mean rebuild frequency, (b) rebuild intervals, (c) maximum velocity, (d) estimated rebuild frequency using the Velocity Method.



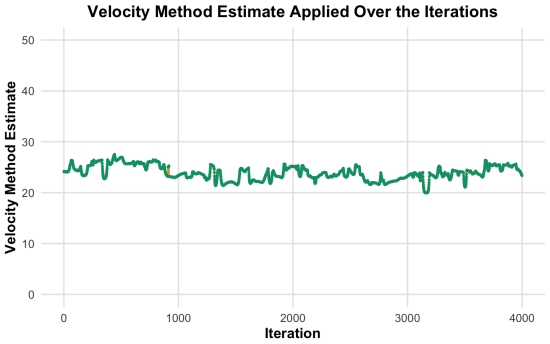
(a) The mean rebuild frequency at each iteration in the Spinodal Decomposition Equilibration simulation, calculated by dividing the current iteration number by the number of rebuilds that have occurred (both excluding tuning iterations).



(b) The number of iterations between successive rebuilds (rebuild intervals) in the first 4000 iterations of Spinodal Decomposition Equilibration. The first set of rebuilds, which occurred during the tuning phase, can be ignored.

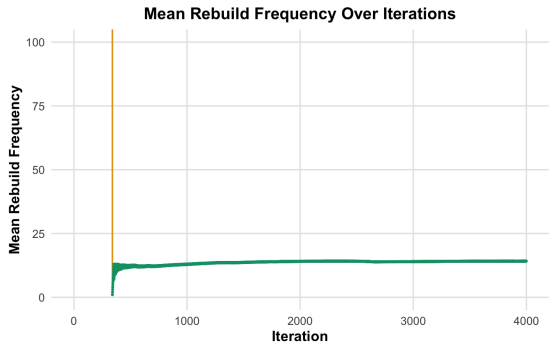


(c) The maximum velocity of the particles in the container at each iteration in the Spinodal Decomposition Equilibration simulation. Data collected over 4000 iterations.

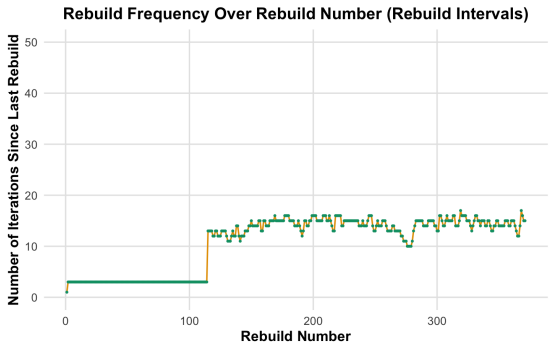


(d) The estimated rebuild frequency using the Velocity Method applied the maximum velocity at each iteration in the Spinodal Decomposition Equilibration simulation.

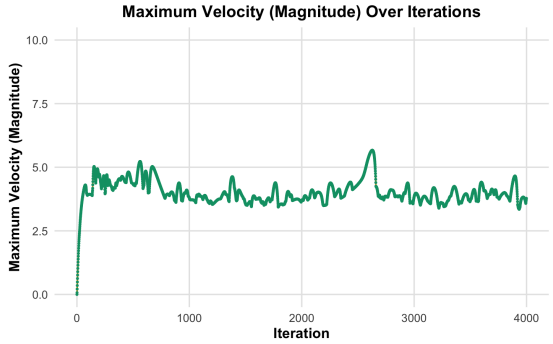
Figure 6.6.: Results from the Spinodal Decomposition Equilibration simulation: (a) mean rebuild frequency, (b) rebuild intervals, (c) maximum velocity, (d) estimated rebuild frequency using the Velocity Method.



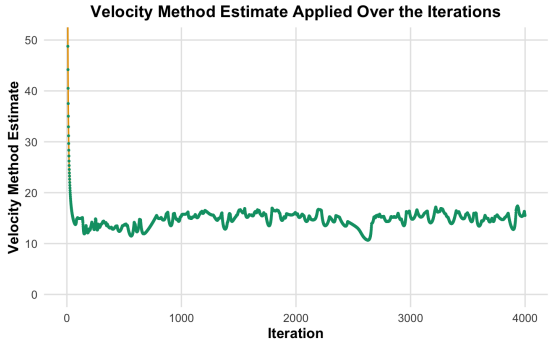
(a) The mean rebuild frequency at each iteration in the Exploding Liquid simulation, calculated by dividing the current iteration number by the number of rebuilds that have occurred (both excluding tuning iterations).



(b) The number of iterations between successive rebuilds (rebuild intervals) in the first 4000 iterations of Exploding Liquid. The first set of rebuilds, which occurred during the tuning phase, can be ignored.

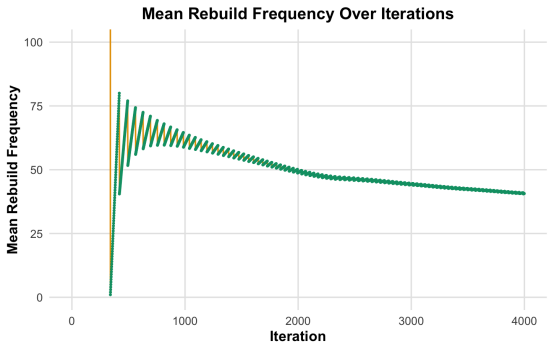


(c) The maximum velocity of the particles in the container at each iteration in the Exploding Liquid simulation. Data collected over 4000 iterations.

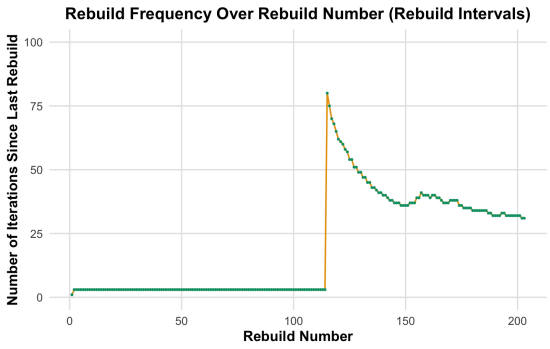


(d) The estimated rebuild frequency using the Velocity Method applied to the maximum velocity at each iteration in the Exploding Liquid simulation.

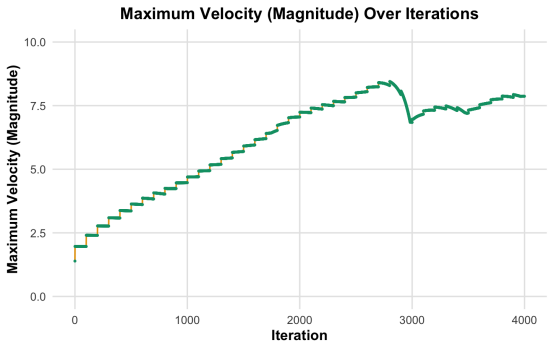
Figure 6.7.: Results from the Exploding Liquid simulation: (a) mean rebuild frequency, (b) rebuild intervals, (c) maximum velocity, (d) estimated rebuild frequency using the Velocity Method.



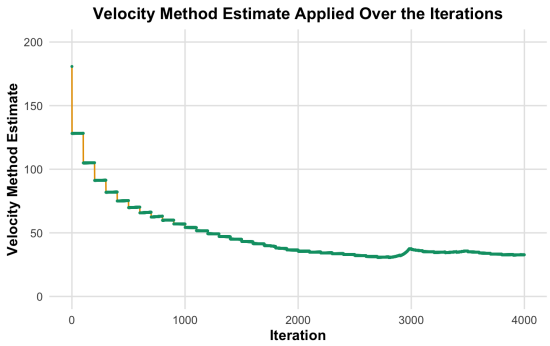
(a) The mean rebuild frequency at each iteration in the Heating Sphere simulation, calculated by dividing the current iteration number by the number of rebuilds that have occurred (both excluding tuning iterations).



(b) The number of iterations between successive rebuilds (rebuild intervals) in the first 4000 iterations of the Heating Sphere. The first set of rebuilds, which occurred during the tuning phase, can be ignored.

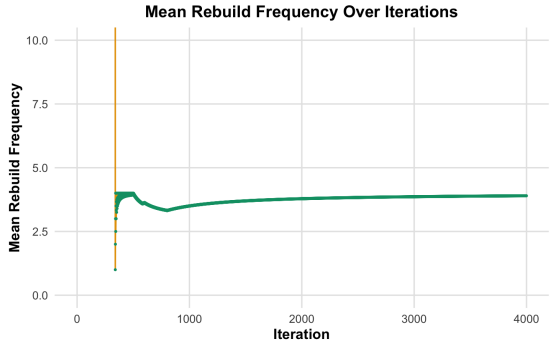


(c) The maximum velocity of the particles in the container at each iteration in the Heating Sphere simulation. Data collected over 4000 iterations.

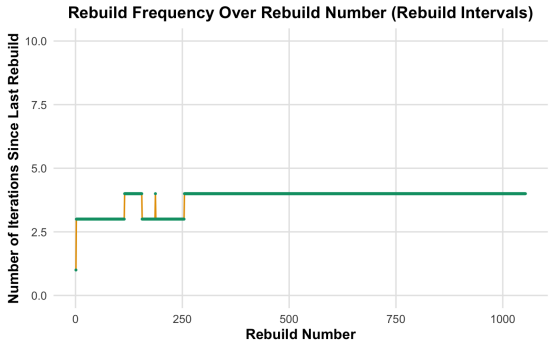


(d) The estimated rebuild frequency using the Velocity Method applied to the maximum velocity at each iteration in the Heating Sphere simulation.

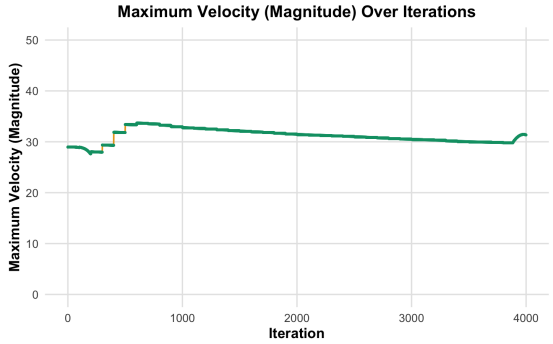
Figure 6.8.: Results from the Heating Sphere simulation: (a) mean rebuild frequency, (b) rebuild intervals, (c) maximum velocity, (d) estimated rebuild frequency using the Velocity Method.



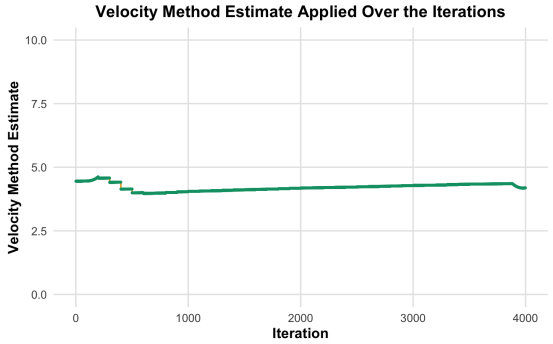
(a) The mean rebuild frequency at each iteration in the Rayleigh Taylor simulation, calculated by dividing the current iteration number by the number of rebuilds that have occurred (both excluding tuning iterations).



(b) The number of iterations between successive rebuilds (rebuild intervals) in the first 4000 iterations of the Rayleigh Taylor. The first set of rebuilds, which occurred during the tuning phase, can be ignored.



(c) The maximum velocity of the particles in the container at each iteration in the Rayleigh Taylor simulation. Data collected over 4000 iterations.



(d) The estimated rebuild frequency using the Velocity Method applied the maximum velocity at each iteration in the Rayleigh Taylor simulation.

Figure 6.9.: Results from the Rayleigh–Taylor simulation: (a) mean rebuild frequency, (b) rebuild intervals, (c) maximum velocity, (d) estimated rebuild frequency using the Velocity Method.

7. Other Methods

The Velocity Method provided an efficient way to estimate the mean rebuild frequency and yielded accurate results for some simulations, using information obtained during the tuning process.

In this chapter, we explore an alternative method that aims to address the limitations of the Velocity Method—particularly its sensitivity to accelerating velocities. This method adopts different techniques that are slightly more complex to implement but offer potential improvements in robustness and accuracy.

7.1. Extrapolation Method

As mentioned previously, some unrealistic assumptions in the Velocity Method—particularly the assumption of constant velocities—led to estimates that significantly deviated from the actual rebuild frequency. To address this issue, we now drop the constant velocity assumption and instead aim to estimate the **mean** maximum velocity, $\bar{v}_{max}$, of the non-tuning phase by extrapolating the available tuning data. After that, we employ the following relation which yields an estimate for the mean rebuild frequency. The mathematical intuition behind the estimate will be presented later:

$$\frac{r_{\text{skin}}}{2\Delta t \bar{v}_{max}} \approx \frac{N_{\text{iterations}}}{N_{\text{rebuilds}}} = rf_{mean}. \tag{7.1}$$

where $\bar{v}_{max}$ is the mean maximum velocity in the non-tuning phase, and $N_{\text{iterations}}$ and N_{rebuilds} are the total number of iterations and total number of rebuilds in the non-tuning phase, respectively.

In simple terms, we predict the evolution of the maximum velocity during the following non-tuning phase based on its behavior during the tuning phase, and calculate the mean maximum velocity of this predicted evolution.

For this extrapolation, we adopt a linear approach to prioritize computational efficiency. Leveraging linearity, the mean value can be computed by evaluating the predicted velocity at exactly half the total time (i.e., half the iterations in our case).

As in the Velocity Method, the latest iteration from which we can extract data is the iteration that collects the last sample of the first configuration. This constraint ensures that all configurations share an identical rebuild frequency during tuning. Relaxing this would not be feasible, as certain tuning strategies require the rebuild frequency value to be directly available for the configuration under consideration.

In the previous chapter, we experimentally demonstrated a correlation between the mean maximum velocity and the mean rebuild frequency. In the following, we provide a mathematical justification for this relationship.

7.1.1. Relationship Between Mean Velocity and Mean Rebuild Frequency

In the following, we demonstrate Equation 7.1 for a **single** particle, where the maximum velocity at any point in the non-tuning phase corresponds to that particle. Therefore, we use $\bar{v}$ to denote the mean velocity of the particle during this phase.

According to the definition, the mean velocity of a particle $\bar{v}$ is the total displacement divided by the total time. This is given by

$$\bar{v} = \frac{\Delta x}{N_{\text{iterations}} \cdot \Delta t}, \tag{7.2}$$

where:

- Δx is the total displacement during the non-tuning phase,
- $N_{\text{iterations}}$ is the total number of iterations of the non-tuning phase,
- Δt is the time step.

An important observation is that since a rebuild is triggered every time the particle traverses half the skin length, the number of rebuilds can be computed as:

$$\left\lfloor \frac{\Delta x}{r_{\text{skin}}/2} \right\rfloor = N_{\text{rebuilds}}, \tag{7.3}$$

where r_{skin} is the skin.

Substituting these into the expression for the mean rebuild frequency (rf_{mean_est}), we obtain:

$$\begin{aligned} rf_{mean\_est} &= \frac{r_{\text{skin}}}{2\Delta t \bar{v}} && (7.4)\\ &= \frac{r_{\text{skin}}}{2\Delta t \left(\frac{\Delta x}{N_{\text{iterations}} \cdot \Delta t} \right)} && (7.5)\\ &= \frac{r_{\text{skin}}}{2 \cdot \frac{\Delta x}{N_{\text{iterations}}}} && (7.6)\\ &= \frac{r_{\text{skin}}}{2\Delta x} \cdot N_{\text{iterations}} && (7.7)\\ &= \frac{N_{\text{iterations}}}{N_{\text{rebuilds}}} = rf_{mean}. && (7.8) \end{aligned}$$

An important remark is that the equations above are valid only for a single particle. In the case of multiple particles, the maximum velocity may belong to different particles at different iterations, making the area under the curve of the maximum velocity a combination of displacements from different particles. Consequently, Equation 7.3 becomes inapplicable in this scenario.

While this approach is not guaranteed to be fully accurate in all scenarios, we may still apply it to the maximum velocity observed among all particles in the container in order to estimate the mean rebuild frequency. This approximation is particularly effective when the maximum velocity consistently corresponds to the same particle throughout the simulation.

7.1.2. Technical Details of the Implementation

We first note that most simulations can be assigned to one of two categories: the first category consists of simulations that are initially in a state of relatively constant velocity, while the second consists of simulations that reach constant velocity only at a **later** stage.

In the estimation process, it is essential to account for the simulation's category. This distinction is crucial for simulations with nearly constant velocities. In such cases, applying a linear extrapolation using only two samples may falsely interpret a small change in velocity as ongoing acceleration, leading to a significant deviation from the real value when extrapolated over a long period.

For a simpler implementation, we use only two maximum velocity samples: one from the first sample and one from the last sample of the first configuration. The method works by recording the maximum velocity of the particles in the container at these two stages. Based on these values, we calculate the extrapolation slope, which in this case corresponds to the change in maximum velocity per iteration.

To determine the category, we calculate the relative difference between the two maximum velocity samples by dividing their absolute difference by the first sample. If the relative difference is small, we infer that the simulation has already reached a relatively constant maximum velocity. Otherwise, we assume that the system is still accelerating.

After collecting the samples, the next step is to compute an estimate of the mean maximum velocity for the upcoming non-tuning phase. If the simulation is classified as one with constant velocity (i.e., the relative difference is below a certain threshold), we take the mean of the two samples as the estimated mean maximum velocity. Otherwise, we compute the regression line and extrapolate it to half the tuning interval. This gives us an estimate for the maximum velocity at the midpoint, which is also the mean for linear functions.

Finally, we substitute the estimated mean maximum velocity into Equation 7.1 to obtain an estimate for the mean rebuild frequency. As before, the user-provided rebuild frequency is used as an upper bound.

An overview of the algorithm is presented in 3.

7.1.3. Results for First Non-Tuning Phase

In the following, we apply the *Extrapolation Method* to the five simulation scenarios for 4000 iterations and one tuning phase at the beginning, with the results summarized in Table 7.1.

As shown in the results, the estimation was able to provide an improvement over the *Velocity Method* for three out of the five simulations.

For *Spinodal Decomposition Equilibration* and *Rayleigh Taylor*, the estimated rebuild frequencies closely match the actual values, which can be attributed to the relatively constant value of the maximum velocity throughout the simulation. This was correctly identified using the relative change criterion described earlier.

However, in the other scenarios, the maximum velocity changed significantly during the simulation, and the extrapolation method was unable to capture this behavior accurately. For instance, in the case of *Falling Drop*, the method underestimated the mean maximum velocity. The estimated value was 11.4741, whereas the actual mean—approximated by visually computing the area under the curve in Figure 6.5c and dividing it by the total number of iterations—was closer to 15. Similarly, the method failed to adequately predict

Algorithm 3: The Extrapolation Method used to calculate an estimated mean rebuild frequency for the following non-tuning phase. The estimate is used to select an optimal configuration.

Input : Skin s, Time step Δt, User-provided rebuild frequency rf_{user}, Number of samples N_{samples}, Tuning interval k

Global Variables: Maximum velocity in the first sample $v_{\text{maxSampleFirst}}$, Threshold $t = 0.15$;

if *autoTuner.inFirstConfigurationFirstSample()* **then**
  $v_{\text{maxSampleFirst}} \leftarrow$ getMaximumVelocityInTheContainer()
if *autoTuner.inFirstConfigurationLastSample()* **then**
  $v_{\text{maxSampleLast}} \leftarrow$ getMaximumVelocityInTheContainer()
  $diff \leftarrow \dfrac{v_{\text{maxSampleLast}} - v_{\text{maxSampleFirst}}}{v_{\text{maxSampleFirst}}}$
  if $diff > t$ **then**
    $v_{\text{final}} \leftarrow v_{\text{maxSampleFirst}} + \dfrac{k}{2} \cdot \dfrac{v_{\text{maxSampleLast}} - v_{\text{maxSampleFirst}}}{N_{\text{samples}} - 1}$
  else
    $v_{\text{final}} \leftarrow \dfrac{v_{\text{maxSampleLast}} + v_{\text{maxSampleFirst}}}{2}$
  $rf_{\text{est}} \leftarrow \left\lfloor \dfrac{s}{v_{\text{final}} \cdot \Delta t \cdot 2} + 1 \right\rfloor$
  if $rf_{est} > rf_{user}$ **then**
    autoTuner.setRebuildFrequency(rf_{user})
  else
    autoTuner.setRebuildFrequency(rf_{est})

the trajectory of the maximum velocity in both *Exploding Liquid* and *Heating Sphere*, leading to rebuild frequency estimates that diverged significantly from the true values.

Nevertheless, the method was able to deliver better results, particularly for the case of *Falling Drop*, without introducing any disadvantages compared to the *Velocity Method*. Even in cases where the *Extrapolation Method* did not offer an improvement, its drawback was negligible.

7.1.4. Results for Second Non-Tuning Phase

In the second non-tuning phase, we ran the same five simulations for a total of 8000 iterations, using a tuning interval of 4000 iterations. The results are summarized in Table 7.2.

Although the results for the second tuning phase were slightly more accurate overall, this improvement can largely be attributed to the velocities converging to some value over time. In these conditions, the computed relative difference fell below the threshold, causing the method to behave similarly to the Velocity Method. As such, the method's performance improved not due to more accurate extrapolation, but rather because the velocity curve had become nearly constant.

Scenario	Real Mean rf	Mean rf (Velocity)	Mean rf (Extrap.)	(%)
Falling Drop	67.19	4000	88.15	+5822.07%
Spinodal D.E	23.31	24.14	24.16	-0.09%
Exploding Liquid	14.42	260.92	1.26	+1618.16%
Heating Sphere	41.80	128.13	1.43	+109.95%
Rayleigh Taylor	3.89	4.45	4.45	0.00%

Table 7.1.: Comparison of the Velocity Method and the Extrapolation approach for the first non-tuning phase. Improvements are computed relative to the real value (Extrapolation over Velocity).

Scenario	Real Mean rf	Mean rf (Velocity)	Mean rf (Extrap.)	(%)
Falling Drop	32.21	41.27	41.26	+0.03%
Spinodal D.E	23.46	23.24	23.73	-0.21%
Exploding Liquid	14.08	15.21	15.30	-0.63%
Heating Sphere	23.61	29.59	25.24	+18.42%
Rayleigh Taylor	4.00	4.37	3.99	+9.00%

Table 7.2.: Comparison of the Velocity Method and the Extrapolation Method for the second non-tuning phase. Improvements are computed relative to the real value (Extrapolation over Velocity).

7.1.5. Reflection

The Extrapolation Method did not offer a significant improvement over the Velocity Method. The only notable advantage lies in its ability to predict the correct mean rebuild frequency in scenarios where the maximum velocity increases linearly until the start of the next tuning phase, like *Falling Drop*.

While the *Extrapolation Method* did not exhibit any disadvantages compared to the *Velocity Method*, its applicability still needs to be further investigated across a wider range of simulations. Moreover, the impact of the chosen threshold value requires a more thorough analysis.

In general, extrapolating a graph based on only two data points is unlikely to produce accurate predictions. To achieve better results, it would be necessary to collect more data. This could be accomplished either by increasing the number of velocity samples or by modifying the implementation to allow for a delayed setting of the rebuild frequency.

In the latter case, a more robust approach—such as linear regression using matrix operations—could be employed. However, depending on the number of samples and the simulation overhead, this added computational cost may not be justifiable.

8. Future Work

Future improvements may focus on incorporating more samples into the estimation process, thereby improving the accuracy of the prediction. A promising enhancement of the Extrapolation Method is to collect a fixed number of samples—e.g., 10 samples from the first configuration—and apply linear regression on the collected data to obtain a more reliable velocity trend. The computational cost of this approach is low due to the relatively small number of samples.

We may validate the feasibility of this method using existing simulation data without altering the codebase. For example, one can perform linear regression on the maximum velocities from the first 10 iterations of a simulation using a statistical tool such as R.

As illustrated in Figure 8.1a, the regression line fitted to the first 10 iterations of the *Falling Drop* scenario predicts a velocity of approximately 15 at iteration 2000, which closely matches the observed mean maximum velocity, obtained by approximating the area under the curve and dividing it by the total number of iterations. This suggests that linear regression with a small number of samples can significantly enhance prediction accuracy.

Applying the same technique to the *Spinodal Decomposition Equilibration* scenario (Figure 8.1b) highlights the importance of distinguishing between simulations with constant and non-constant velocities. In this case, the regression line exhibits a slight positive slope, despite the velocities being nearly constant. To account for such behavior and to avoid overestimations in long tuning intervals, it may be useful to introduce a slope threshold. This ensures that the method provides accurate results for simulations with constant velocities.

Some simulations, such as Exploding Liquid (Figure 8.1c), did not benefit significantly from this method. In those cases, the velocities exhibit strong acceleration during the first few iterations and abruptly converge soon after. In such scenarios, extrapolation based on early data points becomes unreliable. A similar pattern can be observed in Heating Sphere (Figure 8.1c).

Table 8.1 presents a comparison between extrapolation using 2 points versus 10 points, omitting simulations with constant velocities since extrapolation is not applied to them due to the threshold classification. The mean rebuild frequencies for the 10-point extrapolation are computed by inserting the predicted mean maximum velocities, shown in Figure 8.1, into Equation 7.1.

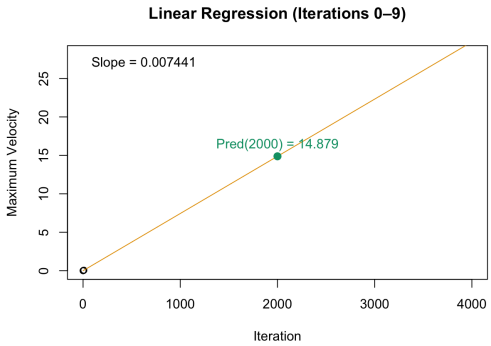
Generally, more sophisticated extrapolation strategies may be required to ensure better results for longer tuning intervals. One possible approach is to place more weight on later samples rather than earlier ones. For instance, instead of using the first and last samples, performing regression on the last two or three samples might yield better estimates, especially in cases with late stabilization of velocities.

Another area for investigation is the use of rebuild frequency as a mechanism to decide whether runtime samples of different configurations should be recomputed or if samples from previous tuning phases can be reused. The underlying assumption is that the rebuild frequency may indicate how drastically the simulation state changes; if the change is minor,

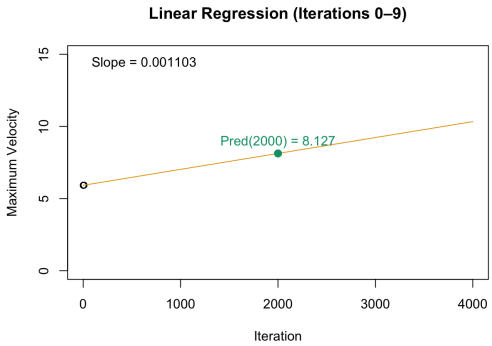
Scenario	Real Mean rf	Mean rf (Extrap. 2)	Mean rf (Extrap. 10)	(%)
Falling Drop	67.19	88.15	68.20	+29.69%
Exploding Liquid	14.42	1.26	1.23	-0.20%
Heating Sphere	41.80	1.43	4.87	+8.22%

Table 8.1.: Comparison of Extrapolation with 2 points versus 10 points for the first non-tuning phase. Improvements are computed relative to the real value (Extrapolation 10 over Extrapolation 2).

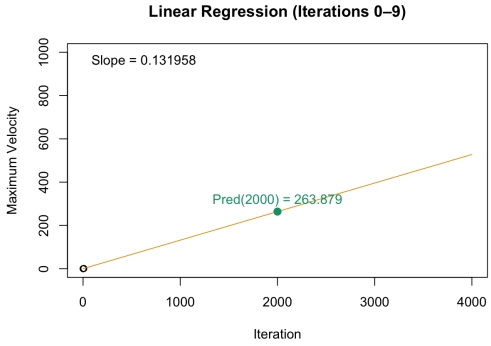
the old samples could still be valid. This approach can be implemented by introducing an *rf-relative-change threshold*, such that if the change in rebuild frequency lies below the threshold, the previous samples are reused, thereby saving runtime.



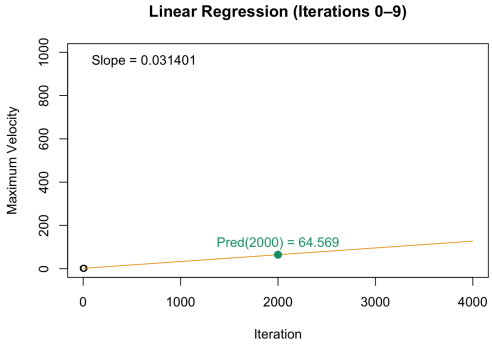
(a) Falling Drop



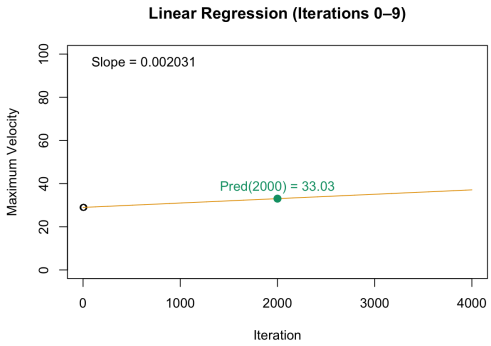
(b) Spinodal Decomposition



(c) Exploding Liquid



(d) Heating Sphere



(e) Rayleigh–Taylor

Figure 8.1.: Linear regression lines based on the first 10 iterations of each simulation: (a) Falling Drop, (b) Spinodal Decomposition, (c) Exploding Liquid, (d) Heating Sphere, (e) Rayleigh–Taylor.

Conclusion

This thesis explored three different methods for estimating the mean rebuild frequency of a non-tuning phase, a value used during tuning to select the optimal configuration.

Through experiments, we demonstrated that the currently implemented approach—which relies on a user-provided value for the first non-tuning phase and the mean rebuild frequency from previous phases for subsequent ones—does not always deliver an optimal configuration. Additionally, we presented the *Simulation Method*, which involves running a short mock simulation to empirically determine the rebuild frequency. Although this approach may yield more realistic estimates in theory, it introduces unnecessary overhead and, in many cases, inaccurate results. These observations motivated the investigation of alternative estimation techniques.

The first proposed method, the *Velocity Method*, estimates the mean rebuild frequency using the last observed maximum velocity sample of the first configuration, under the assumption that velocities remain relatively constant during the non-tuning phase. This method achieved high accuracy in simulations where the maximum velocity remained nearly constant. Importantly, since the user-provided rebuild frequency is still enforced as an upper bound, the method introduces no risk compared to the current approach. Additionally, the overhead of the method is negligible, making it a practical and efficient choice.

To address simulations with non-constant velocities at the beginning, we proposed the *Extrapolation Method*. This method linearly extrapolates the maximum velocity using two collected samples. While it did not consistently improve estimation accuracy across all scenarios, it showed improved performance in cases where the maximum velocity increases linearly. Nonetheless, its simplicity may limit its effectiveness in more complex simulations.

In summary, the *Velocity Method* stands out due to its simplicity, negligible overhead, and accurate performance in relatively constant-velocity simulations. For implementation purposes, it offers a safe and effective improvement over the currently deployed strategy.

For future work, we suggest exploring more sophisticated extrapolation techniques that may yield improved predictions in simulations with non-constant velocities, such as linear regression with additional samples or weighted extrapolation based on later velocity values. Additionally, the rebuild frequency could be employed as an indicator of whether the simulation state has changed significantly; if not, previous runtime samples could be reused, thereby reducing computational overhead.

Overall, this work contributes a practical approach to improving simulation tuning by providing reliable and efficient estimates of rebuild frequency, which is essential for optimizing performance in particle-based simulations.

Part II.

Appendix

A. All Material on Github

All collected data and input files can be found on Github https://github.com/YasperFa/Dynamic-RF-Estimation.git

B. Hardware Specification

Performance measurements were performed on a Linux Cluster using cm4_tiny. Some data was also measured locally on MacBook Air 2024, Apple M3 chip.

List of Figures

List of Tables

Bibliography

[1] *The Linked Cell Method for Short-Range Potentials*, pages 37–111. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. ISBN 978-3-540-68095-6. doi: 10.1007/978-3-540-68095-6_3. URL https://doi.org/10.1007/978-3-540-68095-6_3.

[2] Daniel Asch. Auto-tuning verlet list skin lengths in autopas. Master's thesis, Technical University of Munich, Mar 2023.

[3] Omar Awile, Ferit Büyükkeçeci, Sylvain Reboux, and Ivo F. Sbalzarini. Fast neighbor lists for adaptive-resolution particle simulations. *Computer Physics Communications*, 183(5):1073–1081, 2012. ISSN 0010-4655. doi: https://doi.org/10.1016/j.cpc.2012.01.003. URL https://www.sciencedirect.com/science/article/pii/S0010465512000057.

[4] Mohammad Badar, Shazmeen Shamsi, Jawed Ahmed, and Afshar Alam. *Molecular Dynamics Simulations: Concept, Methods, and Applications*. 08 2020. ISBN 978-3-030-94650-0. doi: 10.1007/978-3-030-94651-7_7.

[5] Ha H. Bui and Giang D. Nguyen. Smoothed particle hydrodynamics (sph) and its applications in geomechanics: From solid fracture to granular behaviour and multiphase flows in porous media. *Computers and Geotechnics*, 138:104315, 2021. ISSN 0266-352X. doi: https://doi.org/10.1016/j.compgeo.2021.104315. URL https://www.sciencedirect.com/science/article/pii/S0266352X2100313X.

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

[7] P. A. Cundall and O. D. L. Strack. A discrete numerical model for granular assemblies. *Géotechnique*, 29(1):47–65, 03 1979. ISSN 0016-8505. doi: 10.1680/geot.1979.29.1.47. URL https://doi.org/10.1680/geot.1979.29.1.47.

[8] AutoPas Developers. Dynamic containers in autopas. https://github.com/AutoPas/AutoPas/pull/821.

[9] AutoPas Developers. Alloptions.yaml - md-flexible example input, 2024. URL https://github.com/AutoPas/AutoPas/blob/master/examples/md-flexible/input/AllOptions.yaml. Accessed: 2025-05-08.

[10] EPCC. Measuring and improving lammps performance. https://epcced.github.io/archer2-advanced-use-of-lammps/02-lammps-performance/index.html. Accessed: 2025-05-31.

[11] Luis Gall. An exploration of different approaches for implementing verlet lists in autopas. Master's thesis, Technical University of Munich, Aug 2023.

[12] R. A. Gingold and J. J. Monaghan. Smoothed particle hydrodynamics: theory and application to non-spherical stars. *Monthly Notices of the Royal Astronomical Society*, 181(3):375–389, 12 1977. ISSN 0035-8711. doi: 10.1093/mnras/181.3.375. URL https://doi.org/10.1093/mnras/181.3.375.

[13] Fabio Alexander Gratl, Steffen Seckler, Nikola Tchipev, Hans-Joachim Bungartz, and Philipp Neumann. Autopas: Auto-tuning for particle simulations. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 748–757, 2019. doi: 10.1109/IPDPSW.2019.00125.

[14] Fabio Alexander Gratl, Steffen Seckler, Hans-Joachim Bungartz, and Philipp Neumann. N ways to simulate short-range particle systems: Automated algorithm selection with the node-level library autopas. *Computer Physics Communications*, 273:108262, 2022. ISSN 0010-4655. doi: https://doi.org/10.1016/j.cpc.2021.108262. URL https://www.sciencedirect.com/science/article/pii/S001046552100374X.

[15] Martin Horsch, Jadran Vrabec, Martin Bernreuther, Sebastian Grottel, Guido Reina, Andrea Wix, Karlheinz Schaber, and Hans Hasse. Homogeneous nucleation in supersaturated vapors of methane, ethane, and carbon dioxide predicted by brute force molecular dynamics. *The Journal of Chemical Physics*, 128(16):164510, 04 2008. ISSN 0021-9606. doi: 10.1063/1.2907849. URL https://doi.org/10.1063/1.2907849.

[16] Pietro Incardona, Antonio Leo, Yaroslav Zaluzhnyi, Rajesh Ramaswamy, and Ivo F. Sbalzarini. Openfpm: A scalable open framework for particle and particle-mesh codes on parallel computers. *Computer Physics Communications*, 241:155–177, 2019. ISSN 0010-4655. doi: https://doi.org/10.1016/j.cpc.2019.03.007. URL https://www.sciencedirect.com/science/article/pii/S0010465519300852.

[17] LAMMPS Developers. Parallel neighbor list construction. https://docs.lammps.org/Developer_par_neigh.html, 2025. Accessed: 2025-08-08.

[18] Johannes Lenhard, Simon Stephan, and Hans Hasse. On the history of the lennard-jones potential. *Annalen der Physik*, 536(6):2400115, 2024. doi: 10.1002/andp.202400115. URL https://onlinelibrary.wiley.com/doi/full/10.1002/andp.202400115.

[19] Manuel Lerchner. Auto-tuning with early stopping in autopas. Munich, Germany, 2025. Technical University of Munich.

[20] H B Maisun, E Latifah, and Y A Laksono. Investigating melting point and crystal structure of mg-ca alloys using parallel molecular dynamics simulations: Lammps and ovito approach. *Journal of Physics: Conference Series*, 2900(1):012017, nov 2024. doi: 10.1088/1742-6596/2900/1/012017. URL https://dx.doi.org/10.1088/1742-6596/2900/1/012017.

[21] Arief Maulana, Artoto Arkundato, Sutisna, and Herri Trilaksana. Mechanical properties of fe, ni and fe-ni alloy: Strength and stiffness of materials using lammps molecular

dynamics simulation. *AIP Conference Proceedings*, 2314(1):020008, 12 2020. ISSN 0094-243X. doi: 10.1063/5.0034046. URL https://doi.org/10.1063/5.0034046.

[22] Samuel James Newcome, Fabio Alexander Gratl, Manuel Lerchner, Abdulkadir Pazar, Manish Kumar Mishra, and Hans-Joachim Bungartz. Algorithm selection in short-range molecular dynamics simulations. In Michael H. Lees, Wentong Cai, Siew Ann Cheong, Yi Su, David Abramson, Jack J. Dongarra, and Peter M. A. Sloot, editors, *Computational Science – ICCS 2025*, pages 292–299, Cham, 2025. Springer Nature Switzerland. ISBN 978-3-031-97635-3.

[23] Isaac Newton. *The Principia: Mathematical Principles of Natural Philosophy*. University of California Press, 1999.

[24] Christoph Niethammer, Stefan Becker, Martin Bernreuther, Martin Buchholz, Wolfgang Eckhardt, Alexander Heinecke, Stephan Werth, Hans-Joachim Bungartz, Colin W. Glass, Hans Hasse, Jadran Vrabec, and Martin Horsch. ls1 mardyn: The massively parallel molecular dynamics code for large systems. *Journal of Chemical Theory and Computation*, 10(10):4455–4464, 10 2014. doi: 10.1021/ct500169q. URL https://doi.org/10.1021/ct500169q.

[25] Szilárd Páll and Berk Hess. A flexible algorithm for calculating pair interactions on simd architectures. *Computer Physics Communications*, 184(12):2641–2650, 2013. ISSN 0010-4655. doi: https://doi.org/10.1016/j.cpc.2013.06.003. URL https://www.sciencedirect.com/science/article/pii/S0010465513001975.

[26] Priyanka Purohit, Madhusmita Panda, Jules Tshishimbi Muya, Pradipta Bandyopadhyay, and Biswa Ranjan Meher. Theoretical insights into the binding interaction of nirmatrelvir with sars-cov-2 mpro mutants (c145a and c145s): Md simulations and binding free-energy calculation to understand drug resistance. *Journal of Biomolecular Structure and Dynamics*, 42(17):8865–8884, 2024. doi: 10.1080/07391102.2023.2248519. URL https://doi.org/10.1080/07391102.2023.2248519. PMID: 37599474.

[27] Steffen Seckler, Fabio Gratl, Nikola Tchipev, Matthias Heinen, Jadran Vrabec, Hans-Joachim Bungartz, and Philipp Neumann. Load balancing and auto-tuning for heterogeneous particle systems using ls1 mardyn. In Wolfgang E. Nagel, Dietmar H. Kröner, and Michael M. Resch, editors, *High Performance Computing in Science and Engineering '19*, pages 523–536, Cham, 2021. Springer International Publishing. ISBN 978-3-030-66792-4.

[28] Stuart Slattery, Samuel Temple Reeve, Christoph Junghans, Damien Lebrun-Grandié, Robert Bird, Guangye Chen, Shane Fogerty, Yuxing Qiu, Stephan Schulz, Aaron Scheinberg, Austin Isner, Kwitae Chong, Stan Moore, Timothy Germann, James Belak, and Susan Mniszewski. Cabana: A performance portable library for particle-based simulations. *Journal of Open Source Software*, 7(72):4115, 2022. doi: 10.21105/joss.04115. URL https://joss.theoj.org/papers/10.21105/joss.04115.

[29] Barbara Solenthaler, Peter Bucher, Nuttapong Chentanez, Matthias Müller, and Markus Gross. Sph based shallow water simulation. pages 39–46, 01 2011. doi: 10.2312/PE/vriphys/vriphys11/039-046.

[30] OpenFPM Development Team. Openfpm vector examples, n.d. URL `https://openfpm.mpi-cbg.de/vector-example/`. Accessed: 2025-05-05.

[31] Aidan P. Thompson, H. Metin Aktulga, Richard Berger, Dan S. Bolintineanu, W. Michael Brown, Paul S. Crozier, Pieter J. in 't Veld, Axel Kohlmeyer, Stan G. Moore, Trung Dac Nguyen, Ray Shan, Mark J. Stevens, Julien Tranchida, Christian Trott, and Steven J. Plimpton. Lammps - a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales. *Computer Physics Communications*, 271:108171, 2022. ISSN 0010-4655. doi: https://doi.org/10.1016/j.cpc.2021.108171. URL `https://www.sciencedirect.com/science/article/pii/S0010465521002836`.

[32] Loup Verlet. Computer "experiments" on classical fluids. i. thermodynamical properties of lennard-jones molecules. *Phys. Rev.*, 159:98–103, Jul 1967. doi: 10.1103/PhysRev.159.98. URL `https://link.aps.org/doi/10.1103/PhysRev.159.98`.