TUM SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY

TECHNCAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Special Handling of Fast Particles in AutoPas to Reduce Rebuild Time

Xhulia Jasimi

TUM SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY

TECHNCAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Special Handling of Fast Particles in AutoPas to Reduce Rebuild Time

Author:	Xhulia Jasimi
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisor:	Samuel James Newcome, M.Sc.
Submission Date:	17.02.2025

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

Munich, 17.02.2025 Xhulia Jasimi

Acknowledgments

I would like to express my sincere gratitude to my advisor, Samuel James Newcome, for his continuous support throughout this thesis. His guidance during our meetings and his willingness to answer every question—no matter how many emails I sent—were invaluable.

I would also like to thank the Chair of Scientific Computing and Prof. Dr. Hans-Joachim Bungartz for providing the opportunity and resources to conduct this research.

A heartfelt thank you to my family for always supporting me in every journey I embark on. Finally, I am deeply grateful to my friends, who patiently listened to me talk about my thesis for four months straight.

Abstract

Simulating particle interactions is a fundamental challenge in computational physics, molecular dynamics, and engineering applications. Efficiently managing these simulations is particularly important in large-scale systems, where millions of particles interact over extended periods of time. A key aspect of these simulations is the identification of particles which are eligible for pairwise force calculations. In algorithms like Verlet Lists, these particles are stored in neighbor lists, which are periodically rebuilt to maintain accuracy.

This thesis investigates the Fast-Particle-Buffer mechanism as an alternative approach to optimize the neighbor list rebuild process in AutoPas, a performance-optimized C++ library for short-range particle simulations. Instead of triggering a full rebuild whenever fast-moving particles exceed a predefined threshold, these particles are temporarily stored in a buffer, allowing the neighbor lists to be used longer before rebuilding.

Through various test scenarios, this approach is evaluated in terms of computational efficiency and its impact on overall simulation performance. The findings suggest that while the Fast-Particle-Buffer can reduce neighbor list rebuild costs, its effectiveness depends on the simulation's characteristics.

Contents

1. Introduction

Simulating particle interactions plays a crucial role in computational physics, molecular dynamics, and engineering applications. In large-scale systems, where millions of particles interact over extended time periods, these simulations require significant computational resources. As a result, optimizing runtime is essential to improve efficiency, performance, and scalability. [2]

AutoPas is an auto-tuning C++ library designed to optimize short-range particle simulations by selecting the most efficient algorithm for a given simulation. The library implements various algorithms. However, since each simulation is unique, there is no single best algorithm that performs optimally in all cases. To address this, AutoPas uses auto-tuning, which dynamically selects the most efficient combination of internal algorithms during run-time [17].

One of the internal algorithms provided by AutoPas is Verlet Lists. A key challenge in Verlet Lists is the computational overhead caused by frequent neighbor list rebuilds. As particles move dynamically within the simulation domain, neighbor lists—which store information about neighboring particles within a defined search region—become outdated and must be recomputed periodically. This rebuilding is computationally expensive, impacting overall simulation performance.

This thesis investigates an alternative approach: the Fast-Particle-Buffer mechanism. Instead of rebuilding neighbor lists whenever a fast-moving particle enters the search region, the buffer temporarily stores these particles and avoids neighbor lists rebuilding. The goal is to analyze whether this strategy can reduce computational overhead and improve overall simulation performance.

The structure of this thesis is as follows: Chapter 2 provides a theoretical background on molecular dynamics and intorduces AutoPas. Chapter 3 presents related projects, while Chapter 4 details the implementation of the Fast-Particle-Buffer mechanism. Chapter 5 presents experimental results, analyzing the efficiency of the proposed approaches across various scenarios. Finally, Chapter 6 discusses potential future optimizations, followed by Chapter 7, which summarizes the key findings and conclusions of this work.

2. Background

2.1. Molecular Dynamics (MD) Simulations

Molecular dynamics (MD) is a computational method used to analyze the interactions and movements of atoms and molecules. It offers better understanding of dynamic processes at atomic scale, such as diffusion, chemical reactions, and phase changes, making it an essential tool in physics, bioinformatics, molecular biology, materials science, and more. MD simulations are widely used to study protein folding, predict material properties, discover drugs, and analyze system behavior under various conditions [13] [2].

MD simulations begin with an initial configuration of particles, with specified positions and velocities. Forces acting on each particle are calculated using a force field, which defines the potential energy of the system. These forces are then translated into velocities and movements. Over discrete time steps, the positions and velocities of particles are updated, providing a dynamic view of the system's evolution.

2.1.1. Computational Challenges and Optimizations

While MD is a powerful tool, it is computationally expensive, requiring a significant amount of resources to deliver accurate and reliable results. The main challenges include handling large system sizes, modeling long simulation times, and more.

Short-Range Simulations

To balance computational efficiency and accuracy of results, approximations are commonly introduced. While the distance between two particles increases, the pairwise forces between the two start converging to zero, and can therefore be ignored. For these types of potentials, called short-range potentials, a cutoff radius (r_c) is introduced. This approximation assumes that interactions beyond r_c are negligible and can be ignored. By doing so, the computational complexity is reduced from $O(N^2)$ to $O(N)$, where N is the number of particles. [9]

Lennard-Jones Potential (LJ)

The Lennard-Jones 12-6 potential is a widely used function in molecular dynamics simulations to model interactions between particles. It describes the balance between short-range repulsion and long-range attraction forces. This balance is given by:

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

where r is the distance between two particles, ϵ is the depth of the potential well, representing the strength of attraction, and σ is the distance at which the potential changes sign. [19]

The first term, $(\sigma/r)^{12}$, models the steep repulsive forces that dominate at very short distances, while the second term, $(\sigma/r)^{6}$, represents the weaker attractive van der Waals forces. The potential reaches its minimum value at $r = 2^{1/6}\sigma$, which corresponds to the equilibrium distance between particles.

The Lennard-Jones potential is particularly suited for short-range interactions due to its rapid convergence to zero as r increases. [11]

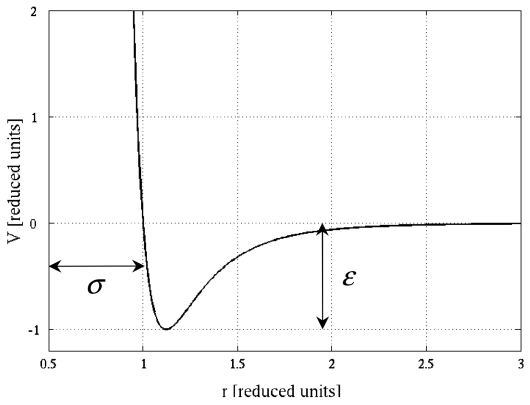


Figure 2.1.: Lennard-Jones Potential [6]

Newton's Third Law

An essential optimization in MD simulations is the application of Newton's third law, which states that every action has an equal and opposite reaction; or, the mutual actions of two bodies upon each other are always equal, and directed to contrary parts [7]. In particle simulations, this means the force that particle p_1 exerts on p_2 is equal and opposite to the force p_2 exerts on p_1. Using this symmetry the number of force

calculations is cut in half, significantly improving computational efficiency. In AutoPas, this optimization is implemented as the Newton3-optimization [9].

2.2. AutoPas Software

AutoPas is a C++ library specifically designed to optimize short-range particle simulations through dynamic algorithm selection. It acts as a black box, where users provide the specifications while the library handles the choice of the best suitable algorithm through auto-tuning. By periodically evaluating various algorithms, AutoPas ensures that the most efficient configuration is applied as simulation conditions change. The library includes example applications, such as the md-flexible framework, which will be used in the course of this thesis [10].

2.2.1. Data layouts

AutoPas supports two primary data layouts for storing particle information in memory: Array of Structures (AoS) and Structure of Arrays (SoA). In the AoS layout, each particle is represented as an object containing all its properties, such as position and force, stored together in memory. This arrangement allows for efficient cache utilization when accessing individual particles, but can limit vectorization efficiency. On the other hand, the SoA layout separates particle properties into individual arrays, with each array containing data for all particles, leading to better vectorization by aligning data contiguously in memory. However, this layout may lead to less efficient cache usage when accessing properties of a single particle [10].

2.2.2. Neighbor identification algorithms

During auto-tuning AutoPas has to decide how to manage and store the particles of the simulation, and most importantly how to identify the neighboring particles to efficiently compute the pairwise forces. As shown in section 2.1.1, for each particle, the particles within the cut-off radius should be found, and the rest will be ignored. This process is repeated for every single particle in the container, and choosing the right Neighbor identification algorithm, is crucial when it comes to performance. Below, the four algorithms used in AutoPas are presented. These algorithms are implemented as containers in AutoPas, managing neighbor identification and the overall particle organization, including the selection of the data layout.

Direct Sum

The straightforward, and thus naive, approach, is to calculate the distances from one particle to all other particles without utilizing any additional data structures. Instead, all particles are stored in a single cell, and for each particle, distances to all other particles are evaluated to determine whether they fall within the cutoff radius. Forces are only computed for pairs of particles within this radius. As illustrated in Figure 2.2a, the red particle represents the current particle, for which forces are being calculated, the red circle denotes the cutoff radius, and the arrows depict the interactions between the current particle and the others.

This method, while eliminating the complexity and overhead associated with bigger data structures, is computationally inefficient. The calculation of pairwise forces results in a time complexity of $O(n^2)$, where n is the number of particles [9]. Although simple, this approach is only practical for simulations with a very small number of particles.

Linked Cells

This approach extends the Direct Sum method by dividing the simulation domain into cells, with each cell containing the particles within its boundaries. The cell dimensions are set to at least the cutoff radius (r_c). This ensures that short-range interactions are computed only between particles in the current cell and its eight neighboring cells, as depicted with blue in Figure 2.2b.

Because the cell size matches the cutoff radius, particles outside these neighboring cells are guaranteed to lie beyond the interaction range and can be excluded from force calculations. For homogeneous particle distributions, this reduces the computational complexity from $O(n^2)$ to $O(n)$ [12]. Furthermore, Linked Cells improve cache efficiency by storing particles within the same cell contiguously in memory.

Nonetheless, there is still computational overhead, as around 84.5% of the particles in the search region are outside the cutoff radius, leading to redundant distance calculations. The probability of finding a pair of particles within the cutoff radius in a 3D simulation, is approximately 15.5% and can be estimated using the following equation [10]:

$$\frac{\textit{Cutoff volume}}{\textit{Search volume}_{LC}} = \frac{\frac{4}{3}\pi r_c^3}{(3r_c)^3}$$

Verlet Lists

The idea behind Verlet Lists is to precompute and store neighbor lists for each particle. This allows the AutoPas to efficiently track all nearby particles within a particle's cutoff

radius, making pairwise interaction calculations faster by reducing the number of distance checks needed in each iteration.

Since particles move during the simulation, these lists eventually become outdated and need to be rebuilt whenever a new particle enters or leaves the cutoff region. To reduce how often these lists need to be rebuilt, an additional safety margin is introduced, known as the Verlet skin factor s. The cutoff radius r_c is scaled by this factor, expanding the search radius to $r_c \cdot s$, as shown with yellow in Figure 2.2c. This extended region allows the neighbor lists to remain valid for multiple iterations, reducing the frequency of rebuilds.

With this approach, the probability of finding a particle in the cutoff region in a 3D simulation domain is given by [10]:

$$\frac{\textit{Cutoff volume}}{\textit{Search volume}_{VL}} = \frac{\frac{4}{3}\pi r_c^3}{\frac{4}{3}\pi (r_c \cdot s)^3} = \frac{1}{s^3}.$$

Selecting an appropriate skin factor is crucial, as it affects both the number of unnecessary calculations and how often the neighbor lists are rebuilt. For instance, a skin factor of 1.2 reduces the number of unnecessary distance calculations by half compared to Linked Cells.

Throughout the simulation, as particles move, two cases arise:

- If a particle moves away from another, it may remain in the neighbor list longer than necessary, increasing storage costs and unnecessary force calculations.
- If a particle moves closer to another, it may enter its cutoff radius but not be included in the neighbor list, leading to incorrect force calculations.

To avoid these issues, AutoPas uses a fixed rebuild frequency parameter, ensuring that all neighbor lists are rebuilt every N iterations to keep them compact and accurate.

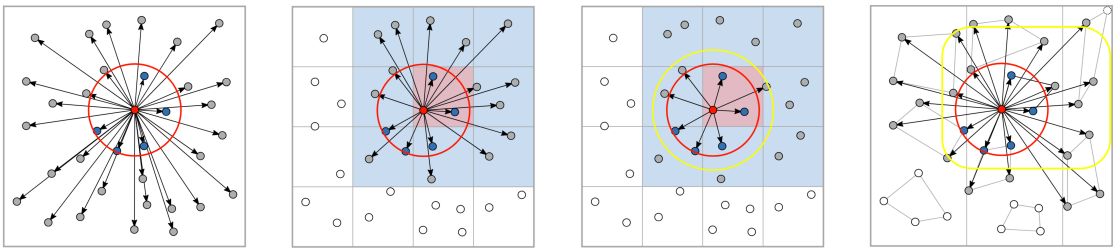
On top of that, when using dynamic containers as described in [8], the maximum particle displacement per iteration is used as a metric of when to rebuild. In this case the displacement of each particle from its last rebuild position is tracked in every iteration. This displacement is then compared to a threshold, which is chosen as half of the Verlet skin size [18]. If at least one particle exceeds this threshold, it could have entered the cutoff region of another particle, and therefore the neighbor lists are rebuilt.

To improve efficiency, AutoPas builds the Verlet Lists algorithm on top of Linked Cells. Therefore, the neighbor lists are constructed by assessing only particles in neighboring cells. This allows the computational complexity to be reduced from $O(N^2)$ (brute-force search) to $O(N)$ [20].

Verlet Lists are particularly effective for systems with high particle densities. However, their lack of spatial locality can lead to inefficient memory access, resulting in poor cache performance and reduced vectorization efficiency [9].

Verlet Cluster Lists

Verlet Cluster Lists are built upon regular Verlet Lists by grouping particles into clusters rather than maintaining individual neighbor lists for each particle Figure 2.2d. This approach reduces memory overhead, as a single neighbor list is created for each cluster instead of one for every particle. The algorithm uses the observation that neighboring particles often share similar neighbor lists, allowing M particles to be combined into a cluster. When two clusters are close, all interactions between the particles within these clusters are calculated. This optimization decreases the number of neighbor lists by a factor of $\frac{1}{M}$ [9], and enhances computational efficiency by enabling better vectorization. However, the increased search radius, can lead to additional distance calculations. Despite this, Verlet Cluster Lists are well-suited for large systems with high particle densities.



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

Figure 2.2.: Neighbor Identification Algorithms in AutoPas [9]

2.2.3. Traversals

Traversals play a crucial role in AutoPas, as they determine the order in which particle interactions are computed. AutoPas supports a variety of traversal strategies, each specific to the container being used. These strategies are designed to optimize performance by parallelizing the traversal process while avoiding race conditions and minimizing the need for schedulers or locking mechanisms [10].

Traversal strategies in AutoPas utilize different cell strategies as their base step. The goal of these base steps is to divide and color the cells of the domain, such that particles in cells of the same color can be processed independently by separate threads. Currently, AutoPas implements three types of base steps: c01, c18, and c08.

2.2.4. Base Steps in AutoPas

c01 Base Step

The c01 base step uses only a single color, as illustrated in Figure 2.3a, and calculates interactions for all neighboring cells without utilizing the Newton3 optimization. In this approach, each cell is assigned to a thread, which calculates the interactions with particles in the neighboring cells. This strategy is highly parallelizable.

c18 Base Step

The c18 base step assigns one of 18 colors to each cell, ensuring that no two neighboring cells share the same color. By utilizing the Newton3 optimization, the number of interactions to be calculated is reduced by half. Instead of processing all neighbors, it focuses only on neighbors with a higher index (see Figure 2.3b). This reduces the overall number of computations, however increases the area where race conditions need to be managed, limiting parallel execution.

c08 Base Step

As the name suggests, the c08 base step uses 8 colors. It is similar to c18 but reduces the locked area further by limiting diagonal interactions and focusing on only four cells (see Figure 2.3c). This adjustment allows for better parallel processing and improved cache utilization.

2.2.5. Traversal Strategies

This thesis focuses on the traversals `lc_c08`, `vlc_c08`, and `vcl_c06`, as they were extensively used in the experiments conducted. A detailed explanation of these strategies is provided below. For information on all traversal strategies, refer to [9].

`lc_c08`

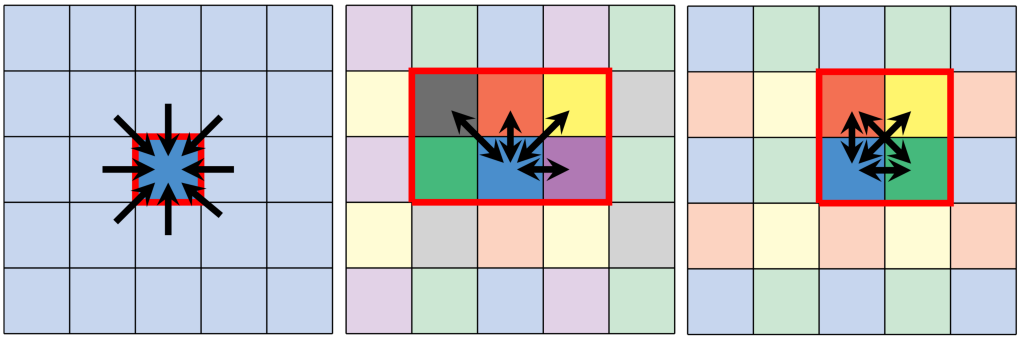
This traversal utilizes `c08` as its base step, and it is designed for the Linked Cells container. It divides a 2D domain into four colors and a 3D domain into eight colors. The dependency of the number of colors C on the number of dimensions D follows $C = 2^D$ [9].

`vlc_c08`

This traversal applies the `c08` base step to each individual cell. To ensure thread safety, a domain coloring consisting of eight colors is used. For each cell, all neighbor lists are processed. Depending on whether the lists were built with Newton3, the base step used is either c01 or c08. [5]

`vcl_c06`

This traversal is used for VerletClusterLists and it employs a 2D coloring scheme with a stride of $x = 3$ and $y = 2$, resulting in six colors ($3 \times 2 = 6$). Interactions within clusters are always computed using Newton3, regardless of whether Newton3 is enabled or disabled for the overall traversal. [5]



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

Figure 2.3.: Base Steps Approaches [14]

2.2.6. Auto-Tuning

One of the key features of AutoPas is its auto-tuning capability. Choosing the optimal combination of container and traversal for a simulation can be challenging, especially since different parts of the simulation may benefit from different configurations. AutoPas automates this process, relieving the user of the need to determine the best setup manually.

At the beginning of a simulation, AutoPas tests various configurations over a few iterations, measuring their performance. To reduce overhead, configurations using the same container are tested consecutively. This approach assumes that the state of the simulation remains relatively stable over consecutive time steps, making the performance results comparable. Importantly, the results of these initial tests are not discarded but are used to guide the following iterations [10].

After evaluating multiple configurations, AutoPas selects the best-performing setup and uses it for a user-defined number of iterations. Following this period, the system enters a re-tuning phase, where new configurations may be selected based on how the simulation has evolved.

3. Related Work

As discussed in the previous chapters, a significant portion of the computational time in molecular dynamics simulations is spent on calculating pairwise force interactions between particles. A common optimization technique is to store the nearby particles, which in the case of Verlet Lists involves maintaining neighbor lists, as previously explained.

Since rebuilding these neighbor lists is computationally expensive, several works have focused on reducing this overhead. One such approach was presented in a previous thesis by Luis Gall [8].

Gall's work primarily focused on reducing the time spent on neighbor list rebuilds. To achieve this, he proposed a dynamic rebuild criterion that adapts based on particle movement, triggering updates only when necessary. This approach was advantageous for scenarios with a high velocity difference over time.

Furthermore, Gall focused on a partial rebuild strategy, where only the neighbor lists affected by particle movement were updated. This method proved to be advantageous in large simulation domains, for a small Verlet skin size.

Beyond AutoPas, several molecular dynamics simulators also implement the neighbor indetification algorithms described in Section 2.2.2. Examples include:

- LS1 MarDyn - A Linked Cells-based simulator optimized for large-scale parallel molecular dynamics simulations. [15]
- LAMMPS - A molecular dynamics software package that employs Verlet Lists. [18]
- GROMACS - A high-performance molecular dynamics package that utilizes Verlet Cluster Lists. [1]

Unlike AutoPas, which dynamically selects the best suited algorithm based on runtime performance using tuning, these pieces of software rely on fixed algorithms that are optimized for specific types of simulations. While this allows them to be highly efficient in their intended use cases, they lack the flexibility to adapt to varying simulation conditions.

4. Implementation

4.1. Initial Problem

Simulations in AutoPas are configured using parameters specified in a `.yaml` file, which defines the conditions of the simulation. Examples of such files can be found in the appendix A. The key fields relevant to this thesis are:

```
container                       : # List of containers to choose from
verlet-rebuild-frequency        : # Frequency of neighbor list rebuilds
verlet-skin-radius              : # Distance within which a particle is
                                  # stored in the neighbor list
data-layout                     : # Data storage format (AoS or SoA)
traversal                       : # List of traversals to choose from
newton3                         : # Boolean value determining if Newton's
                                  # third law is enabled
iterations                      : # Number of iterations
```

Rebuilding neighbor lists is computationally expensive, so it is advantageous to reuse these lists for as many iterations as possible. Neighbor lists are generated for each particle, containing references to all particles within the cutoff range. To enable extended use, particles slightly beyond the cutoff radius are included in the neighbor list by extending the radius with a region called the "Verlet skin," which is also configurable in the `.yaml` file.

If particles move faster than half the defined Verlet skin distance, they may enter the cutoff region of other particles, invalidating the neighbor lists. Even if a single particle among hundreds of thousands exceeds this threshold, all neighbor lists must be rebuilt, incurring high computational costs.

This thesis investigates whether storing fast-moving particles in a buffer temporarily, instead of rebuilding the neighbor lists immediately, can reduce computational overhead. The objective is to evaluate the potential of this approach for future optimization and research.

4.2. Particle Buffer

4.2.1. Particle Buffer Mechanism

In AutoPas, each iteration involves a call to the `updateContainer()` function, which performs several tasks, including:

- Removing particles that no longer belong in the container.
- Checking whether the neighbor lists are invalid, based on criteria such as the `verlet-rebuild-frequency`, tuning iterations, or the presence of fast particles.

The particle buffer is implemented as a `std::vector<FullParticleCell<Particle»`, where each thread has its own buffer, enabling efficient multithreading. This buffer temporarily stores particles that should not yet be added back to the container. A corresponding halo particle buffer stores particles not owned by the current AutoPas object.

The function `checkNeighborListsInvalidDoDynamicRebuild()` is responsible for checking fast particles and is called within `updateContainer()`. It operates by iterating through owned particles in the container. For each particle, the displacement squared relative to the Verlet skin squared is calculated. If the displacement exceeds the threshold, the particle is marked for buffering. A copy of the particle is added to the buffer, and the original is marked as deleted to maintain neighbor list integrity. Marked particles, referred to as "dummy particles," are ignored in computations and eventually removed from the container.

This mechanism mitigates the cost of frequent neighbor list rebuilds by deferring the integration of fast particles until the next scheduled rebuild. During a rebuild, the buffer is cleared, and its particles are integrated into the neighbor lists.

```
1  template <typename Particle>
2  void LogicHandler<Particle>::checkNeighborListsInvalidDoDynamicRebuild() {
3
4   AUTOPAS_OPENMP(parallel reduction(or : _neighborListInvalidDoDynamicRebuild))
5    for (auto iter = this->begin(IteratorBehavior::owned | IteratorBehavior::
         containerOnly); iter.isValid(); ++iter) {
6      const auto distance = iter->calculateDisplacementSinceRebuild();
7      const double distanceSquare = utils::ArrayMath::dot(distance, distance);
8
9        if (distanceSquare >= halfSkinSquare) {
10
11       Particle& particle = *iter;
12       Particle particleCopy = particle;
13
```

```
14          _particleBuffer[autopas_get_thread_num()].addParticle(particleCopy);
15          internal::markParticleAsDeleted(particle);
16
17          _particleNumber++;
18        }
19    }
20    ....
21  }
```

4.2.2. Interaction Computation

During the simulation, interactions are computed in two distinct stages:

1. **Container Interactions**: The main function `computeInteractions(&traversal)` calculates interactions for particles within the container. This involves iterating over particle pairs, triplets, or higher multiples, ensuring efficient resolution of regular particle interactions.
2. **Buffer Interactions**: Interactions for particles in the buffers are computed separately using the `computeRemainderInteractions(functor, newton3)` function. This step ensures that particles in the buffer interact correctly with container particles and among themselves. The following types of interactions are handled:
   - **Particle Buffer** $\leftrightarrow$ **Container**: Interactions between buffer particles and container particles.
   - **Halo Particle Buffer** $\rightarrow$ **Container**: Interactions from halo particle buffers to container particles.
   - **Particle Buffer** $\leftrightarrow$ **Particle Buffer**: Interactions among buffer particles.
   - **Halo Particle Buffer** $\rightarrow$ **Particle Buffer**: Interactions from halo particle buffers to buffer particles.

This two-stage computation ensures accurate interaction handling for all particles, including those in the buffers. Explicit handling of buffer particles allows the simulation to avoid unnecessary neighbor list rebuilds, maintaining computational efficiency.

4.3. DeleteFunction

Initially, in `checkNeighborListsInvalidDoDynamicRebuild()`, particles were marked as deleted in the container, and a copy was created and added to the buffer. While

this approach works for containers like Verlet Lists, as it preserves the integrity of the neighbor lists, it introduces overhead for containers such as LinkedCells. In the case of LinkedCells, every fast-moving particle results in the creation of a dummy particle within the container, depending on the simulation, potentially leading to a significant accumulation of these dummies until they are disregarded.

This issue could become problematic when simulations involve combined usage of LinkedCells and VerletLists. To address this, the hypothesis was to minimize the overhead caused by dummy particles when using containers like LinkedCells.

The available delete functions at the time were:

```
bool deleteParticle(size_t cellIndex, size_t particleIndex);
bool deleteParticle(Particle &particle);
```

These functions could not be directly used within the parallel loop. For certain containers, the delete operation employs a swap-delete mechanism, where the particle to be deleted is swapped with the last particle in the cell before being removed. This implementation, while efficient, is not thread-safe within a parallel loop.

One proposed solution was to gather all particles marked for deletion into a separate buffer and process them sequentially after the parallel loop. However, this approach proved infeasible for the first function, as the indices of remaining particles change dynamically when deletions occur, making it too time-consuming to update and track these indices. For the second function, attempts to store references or pointers to the particles encountered issues when multiple particles were deleted from the same vector. This led to invalid references, as deletions could shift the positions of particles in memory.

To resolve these issues, a new delete function was implemented:

```
deleteParticle(int id, size_t cellIndex);
```

4.3.1. Updated Code

The updated implementation ensures safe handling of deletions within a parallel loop by temporarily storing particle IDs and their cell indices in a buffer. After the loop, a sequential process iterates through this buffer to perform deletions:

```
template <typename Particle>
void LogicHandler<Particle>::checkNeighborListsInvalidDoDynamicRebuild() {

AUTOPAS_OPENMP(parallel reduction(or : _neighborListInvalidDoDynamicRebuild))
  for (auto iter = this->begin(IteratorBehavior::owned | IteratorBehavior::
      containerOnly); iter.isValid(); ++iter) {
    ...
```

```
7
8       if (distanceSquare >= halfSkinSquare) {
9
10        Particle& particle = *iter;
11        Particle particleCopy = particle;
12
13       _particleBuffer[autopas_get_thread_num()].addParticle(particleCopy);
14
15        size_t cellIndex = iter.getVectorIndex();
16        toDelete[autopas_get_thread_num()].push_back(std::make_tuple(particle.
              getID(), cellIndex));
17
18        _particleNumber ++;
19      }
20    }
21
22    for (const auto& t : toDelete) {
23      for (auto p : t) {
24        int id = std::get<0>(p);
25        size_t cellIndex = std::get<1>(p);
26        _containerSelector.getCurrentContainer().deleteParticle(id, cellIndex);
27      }
28    }
29
30    ....
31  }
```

4.3.2. Implementation Details

The new delete function was implemented in several container classes, including `DirectSum.h`, `LinkedCells.h`, `LinkedCellsReferences.h`, `VerletClusterLists.h`, `VerletListsLinkedBase.h`, and `Octree.h`. Below are examples from two of these classes:

In `LinkedCells.h`

The swap-delete method is used here to efficiently manage particle deletions:

```
1  bool deleteParticle(int id, size_t cellIndex) override {
2    auto &particleVec = this->_cells[cellIndex]._particles;
3
4    for (auto &particle : particleVec) {
5      if (particle.getID() == id) {
```

```
      const bool isRearParticle = &particle == &particleVec.back();

      particle = particleVec.back();
      particleVec.pop_back();

      return !isRearParticle;
    }
  }
}
```

In `VerletListsLinkedBase.h`

Here, particles are marked as deleted to avoid interfering with the internal structures of the Verlet Lists:

```
bool deleteParticle(int id, size_t cellIndex) override {
  auto &particleVec = _linkedCells.getCells()[cellIndex]._particles;
  for (auto &particle : particleVec) {
    if (particle.getID() == id) {
      internal::markParticleAsDeleted(particle);
      return false;
    }
  }
}
```

The remaining implementations for other containers follow a similar structure. The full code can be found on `GitHub`.

4.4. CSV File

By enabling the CMake flag `LOG_ITERATIONS`, the simulation generates a `.csv` file containing valuable information for each iteration. This data provides insights into the performance and configuration of the simulation. The header fields included in the file are as follows:

`Date`	`CellSizeFactor`
`Iteration`	`Traversal`
`Functor`	`Load Estimator`
`inTuningPhase`	`Data Layout`
`Interaction Type`	`Newton 3`
`Container`	`computeInteractions[ns]`

```
remainderTraversal[ns]          computeInteractionsTotal[ns]
rebuildNeighborLists[ns]        tuning[ns]
```

Among these, the fields *computeInteractions[ns]*, *remainderTraversal[ns]*, and *rebuildNeighborLists[ns]* are of particular importance for tracking the computational time spent on interaction calculations in the container and in the fast particle buffer.

To facilitate further analysis of performance, three additional fields were introduced:

- **numberOfParticlesInContainer**: Records the total number of particles currently in the container.
- **numberFastParticles**: Tracks the number of fast-moving particles identified in each iteration.
- **particleBufferSize**: Indicates the size of the particle buffer.

These new fields allow us to quantify the behavior of fast particles, evaluate the efficiency of the buffer, and compare the number of fast particles with the total particles in the container.

5. Performance

This chapter explores whether the use of the fast particle buffer provides measurable advantages by presenting the results of various experiments conducted under different scenarios. It details the experimental setup, including the working environment, explains the rationale behind the selection of specific experiments, and provides an analysis of the observed outcomes. Due to the high number of experiments conducted, only the most relevant graphs and data are presented here. However, the complete dataset, along with all scripts used for plotting and analysis, is available in the accompanying `GitHub repository`.

5.1. Test System Specifications

The initial experiments were conducted on the CoolMUC2 Linux cluster at the Leibniz Supercomputing Centre (LRZ), TUM. This cluster operated on the SLES15 SP1 Linux OS and consisted of 812 nodes, each equipped with 28 cores running at a nominal frequency of 2.6 GHz with two hyperthreads per core [3]. The `cm2_tiny` partition was used for these experiments. Unfortunately, due to a severe hardware failure, CoolMUC2 was decommissioned during the course of this work.

Subsequent experiments were performed on CoolMUC4, the successor to CoolMUC2. CoolMUC4 features Intel® Xeon® Platinum 8380 CPUs (Ice Lake) with 112 cores per node, 512 GB of RAM, and a nominal core frequency of 3.0 GHz (ranging from 0.8 to 4.2 GHz). It operates on the SLES15 SP6 Linux OS [4]. The experiments were run on the `cm4_tiny` partition.

The code was compiled with GCC 11.2.0 in CoolMuc2 and GCC 12.2.0 in CoolMuc4.

5.2. Scenario descriptions

For the experiments, four different simulations were used, each chosen to assess the impact of the fast particle buffer under various conditions and provide a comprehensive evaluation of its performance.

5.2.1. Falling Drop

This experiment involves two objects: `CubeClosestPacked`, which represents a bed of particles, and a sphere of particles. At the start of the simulation, the sphere is accelerated by gravity and falls into the basin. Upon collision, the particles from the sphere mix with those in the basin. This experiment uses reflective boundaries and the Lennard-Jones AVX functor, containing over 15,000 particles. The default YAML configuration for this experiment is set to run for 15,000 iterations; however, as will be discussed later, this parameter is adjusted for specific tests. The initial and final states of the simulation are depicted below 5.1, illustrating the transition from the sphere's descent to the equilibrium state of the mixed particles.

5.2.2. Exploding Liquid

The second scenario describes an exploding liquid consisting of a highly compressed and heated liquid film suddenly exposed to a vacuum. This exposure causes the film to rapidly expand and disintegrate into thin filaments and droplets as it destabilizes [17]. This experiment consists of approximately 3,800 particles and, by default, runs for 12,000 iterations. Unlike the first scenario, this simulation uses periodic boundaries. A visualization of the system before and after the explosion is provided in Figure 5.2,

5.2.3. Constant Velocity Cube

This scenario features a cube that moves at a nearly constant velocity and, by default, runs for 5000 iterations. The cube consists of approximately 50,000 particles, which do not interact with each other as dynamically as in the previous two scenarios. Reflective boundaries are employed in this experiment. Due to the minimal interaction between particles, this scenario was selected to examine the effects of a structured, uniform motion. The goal was to evaluate how the particle buffer performs in such conditions. A visual representation of the cube's motion over time can be found in Figure 5.3.

5.2.4. Spinodal Decomposition Equilibration

The fourth scenario examines an equilibration simulation, where a GridBlock object populates the domain with particles. The simulation then runs for 100,000 iterations, allowing the particles to reach an equilibrium state. This scenario involves over four million particles and is characterized by high particle interaction dynamics, resulting in a substantial number of fast-moving particles. The evolution of the system to equilibrium is depicted in Figure 5.4.

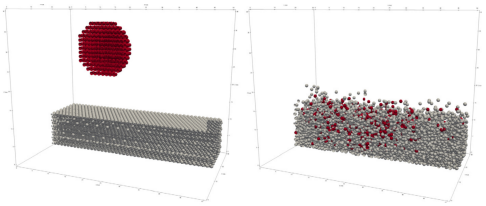


Figure 5.1.: Falling Drop [9]

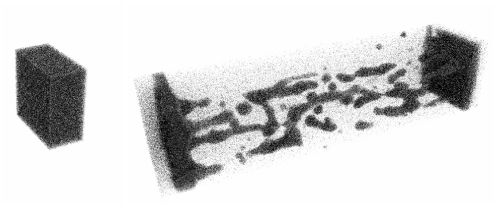


Figure 5.2.: Exploding Liquid [17]

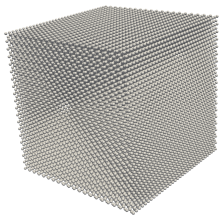


Figure 5.3.: Constant Cube

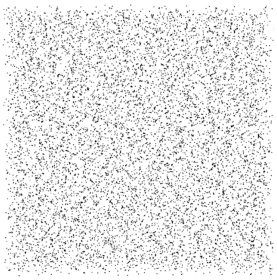


Figure 5.4.: Equilibration [16]

5.3. Experimental Strategies and Findings

The primary objective was to evaluate the performance of the fast particle buffer by systematically modifying specific variables and observing its behavior across different scenarios. Additionally, the experiments aimed to assess the impact of these variables when applied in modified configurations of the same scenarios.

The variables selected for modification included:

- **Verlet Rebuild Frequency**: This variable was adjusted to identify an optimal frequency that maximizes the reuse of neighbor lists across iterations without compromising runtime performance. The goal was to determine how much the rebuild frequency could be extended to achieve a performance gain.
- **Number of Particles**: The particle count was varied to analyze its influence on the buffer's efficiency, specifically examining whether smaller scenarios benefit from the buffer or if the improvements are primarily noticeable in larger configurations.
- **Iteration Count**: By increasing the number of iterations, the experiments aimed to evaluate whether extending the runtime reduces the need for frequent neighbor

list rebuilds in scenarios where only a few fast particles remain after the dynamic phase.

The initial approach involved running a broad range of experiments, systematically varying these parameters to identify patterns and correlations. Key metrics of interest included the number of particles in the container versus the buffer and the time spent in `computeInteractions` compared to `remainderTraversal`.

5.3.1. Initial Test Series

The initial series of experiments focused on analyzing frequency and iteration behavior across the first three scenarios.

Frequency Tests

Frequency tests were conducted by varying the rebuild frequency between 10 and 15,810. The objective was to evaluate the runtime behavior for typical low frequencies and higher frequencies exceeding the total number of iterations (effectively disabling rebuilds apart from tuning). A logarithmic distribution of frequencies was chosen to emphasize the analysis of larger frequencies. The following Python function illustrates the frequency selection logic:

```
def get_step_size(freq):
    log_freq = math.log10(freq)
    step_diff = max_step - min_step
    step_size = min_step + (log_freq * step_diff / math.log10(end_freq))
    return int(step_size)
```

For each frequency experiment, four types of graphs were generated to help interpret the results more effectively:

1. A frequency vs. runtime graph comparing the performance of the parent branch `DynamicVLMerge` with the `Fast Particle Buffer` branch.
2. A graph highlighting the differences in `computeInteractions` runtime between the two branches.
3. A graph illustrating the differences in `remainderTraversal`.

The results consistently showed that the fast particle buffer performed worse across all scenarios and frequencies:

- For the Falling Drop scenario, the runtime ranged from 1.1 to 6 times slower. 5.5a

- For the Exploding Liquid scenario, the runtime ranged from 1.1 to 4 times slower. 5.5b
- For the Constant Velocity Cube scenario, the runtime ranged from 1.3 to 11.2 times slower. 5.5c

To investigate the cause of this performance degradation, the focus was shifted to analyzing the graphs highlighting computeInteractions and remainderTraversal. However, these graphs were challenging to interpret due to significant spikes caused by the tuning phase. Tuning is the most computationally expensive part of the simulation, and its periodic spikes (determined by the tuning frequency) overshadowed the subtler runtime variations, making it difficult to analyze the true behavior of the buffer.

Refined Tests Without Tuning Spikes

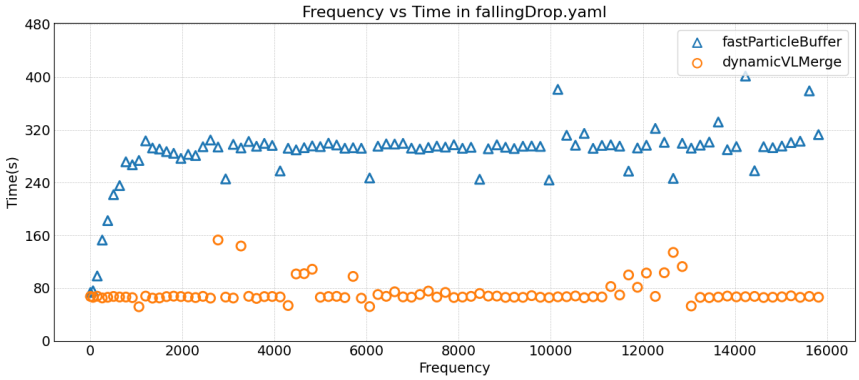
To better understand the program's behavior, the tuning phase was effectively eliminated by specifying exact configurations for the container and traversal. This ensured that the simulation used fixed setups without requiring tuning. Since testing all possible configurations was impractical, three key traversal-container combinations were selected:

- **Traversal**: `vlc_c08 / vlp_c08`, **Container**: VerletListsCells, **Data Layout**: AoS, **Newton3**: Enabled
- **Traversal**: `vcl_c06`, **Container**: VerletClusterLists, **Data Layout**: AoS, **Newton3**: Enabled
- **Traversal**: `lc_c08`, **Container**: LinkedCellsReferences, **Data Layout**: SoA, **Newton3**: Enabled

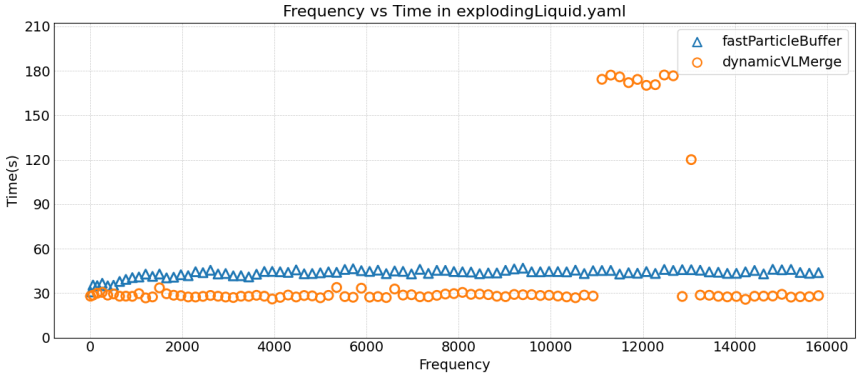
Frequency tests were then repeated for the first three scenarios using these fixed configurations. This approach eliminated tuning-related noise, allowing for a clearer analysis of how the buffer performed under different traversal and container setups. By isolating these configurations, it was possible to focus on the individual behaviors of each traversal and container, leading to more thorough insights into the implementation's performance.

Upon examining the individual frequency versus time graphs that compare the different configurations across various scenarios (Figures A.1 A.2 A.3), the implementation with the fast particle buffer generally underperformed relative to the implementation without it.

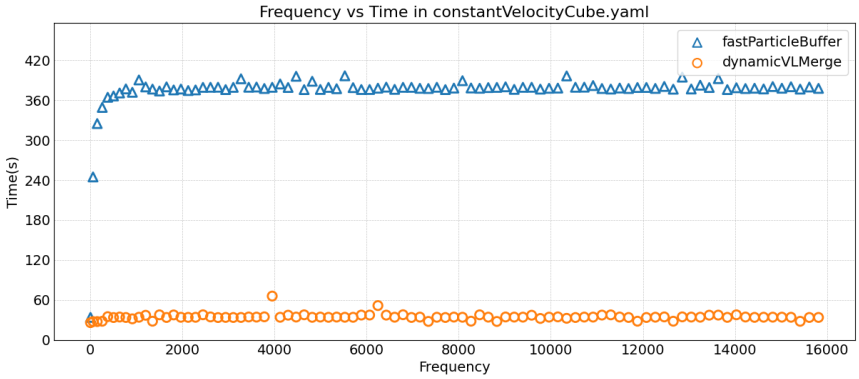
A unique case, however, is observed at frequency 10, where the Fast-Particle-Buffer branch performs nearly as well as the parent branch. This behavior occurs because, at



(a) Falling Drop



(b) Exploding Liquid



(c) Constant Velocity Cube

Figure 5.5.: Comparison of Frequency vs Time with Tuning

such a low frequency, there are almost no fast particles; the neighbor lists are rebuilt so frequently that particles do not have sufficient time to accumulate in the buffer.

Falling Drop: The runtime of the Fast-Particle-Buffer branch across all configurations was up to four times longer than that of the parent branch, Dynamic-VL-Merge (see Figure A.1). This raises the question: what is the underlying cause of this difference in performance?

To investigate, we focus on the relationship between compute interactions, remainder traversal (Fig.5.6), and neighbor list rebuild times per iteration (Fig.5.7), as well as the number of particles stored in the buffer per iteration (Fig.5.8).

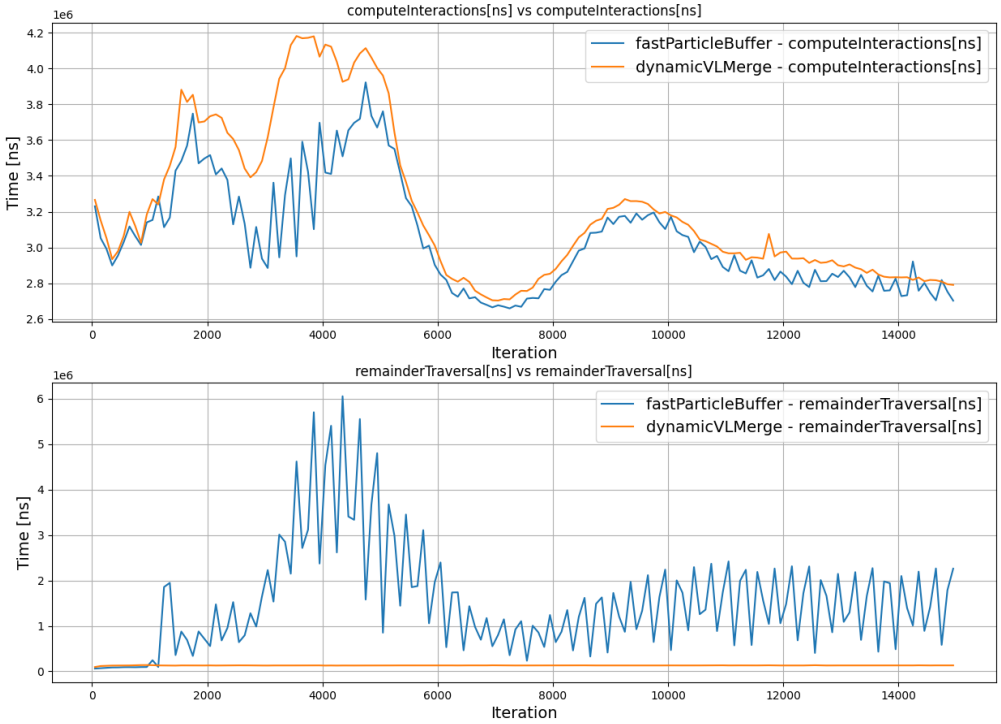


Figure 5.6.: Compute interactions versus Remainder traversal Falling Drop Vlc_c08

Between iterations 2000 and 5000, there is a gradual accumulation of particles in the buffer, peaking at approximately 1798 particles around iteration 3850. Following this peak, the number of particles in the buffer starts to decline. This fluctuation is directly reflected in the remainder traversal time, which increases as the buffer fills up and decreases once particles start leaving the buffer.

While there is a reduction in the time spent on `computeInteractions` within the Fast-Particle-Buffer branch (due to particles being removed from the container), this saving

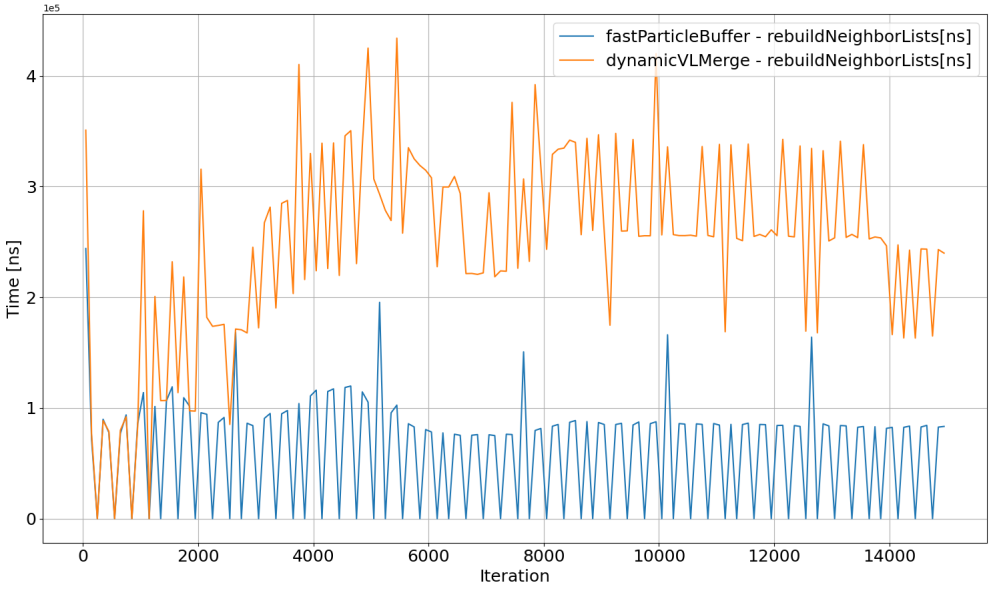


Figure 5.7.: Time spent rebuilding neighbor lists in Falling Drop Vlc_c08

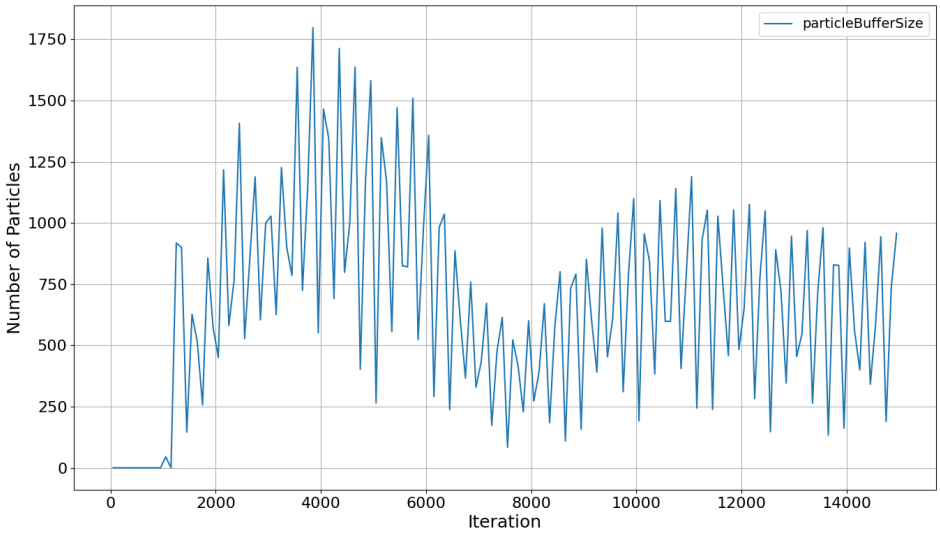


Figure 5.8.: Buffer Size throughout Iterations in Falling Drop Vlc_c08

is overshadowed by the significant increase in the time required for remainder traversal calculations. At its peak, the remainder traversal phase in the Fast-Particle-Buffer branch takes more time than the `computeInteractions` phase itself in the Dynamic-VL-Merge branch, where all particles are being considered.

One noticeable advantage of the Fast-Particle-Buffer approach is the reduction in neighbor list rebuild time, which takes an average four times less time compared to the Dynamic-VL-Merge branch (Fig.5.7). However, the saved time does not compensate for the excessive time spent in remainder traversal. The combined savings from both reduced `computeInteractions` and neighbor list rebuilds are significantly smaller than the additional time required for remainder traversal.

At the end of the experiment, the recorded execution times for key operations in both branches are as follows:

- **Fast-Particle-Buffer:**
  - `computeInteractions[ns]`: 45.74 s
  - `remainderTraversal[ns]`: 23.40 s
  - `rebuildNeighborLists[ns]`: 0.92 s
- **Dynamic-VL-Merge:**
  - `computeInteractions[ns]`: 48.23 s
  - `remainderTraversal[ns]`: 1.95 s
  - `rebuildNeighborLists[ns]`: 3.73 s

The higher the frequency, the more time is saved from avoiding neighbor list rebuilds. However, the rate at which remainder traversal time increases significantly outweighs the benefits. To illustrate, at frequency 15,810 (where rebuilding no longer occurs, as all particles are in the buffer), the theoretical maximum time savings from avoiding rebuilds is approximately 3.5×10^5 ns per iteration. However, the average remainder traversal time per iteration reaches approximately 1.38×10^7 ns, demonstrating that the overhead introduced by the buffer is vastly greater than the savings obtained from eliminating neighbor list rebuilds.

Constant Velocity Cube: The Constant Velocity Cube scenario exhibited similar behavior, where the fast particle buffer consistently underperformed compared to the counterpart (Fig. A.3). Performance differences ranged from:

- 6 times slower for `vlp_c08` (selected instead of `vlc_c08`, as `vlp_c08` was frequently chosen during tuning experiments due to its optimization for this scenario).

- Up to 18 times slower for `lc_c08`.
- Approximately 4 times slower for `vcl_c06`.

The reasoning for these results remains consistent: the time saved from infrequent neighbor list rebuilds is outweighed by the excessive computation time in remainder traversals.

Exploding Liquid: In the Exploding Liquid scenario, the traversals `lc_c08` and `vlc_c08` demonstrated similar performance trends to those observed in other scenarios (Fig. A.2). However, an exception was noted in the `vcl_c06` traversal (Fig.A.2c). During this traversal, the Dynamic VL Merge branch exhibited an average runtime nearly twice as long as the other two traversals, whereas the runtime of the Fast-Particle-Buffer branch increased only slightly in comparison.

Analyzing frequencies between 10 and 2451, the Fast-Particle-Buffer branch outperformed the Dynamic VL Merge branch. The shortest runtime achieved by the Fast-Particle-Buffer branch was 37.88 seconds at frequency 265, whereas the lowest runtime for the Dynamic VL Merge branch within `vcl_c06` was 39.2 seconds in iteration 9963.

A deeper analysis of the `computeInteractions` and `remainderTraversal` times provides insight into the underlying reason for this speedup. Unlike in other traversals, `computeInteractions` in the parent branch consumed significantly more time using `vcl_c06`, whereas the remainder traversal time in the Fast-Particle-Buffer branch experienced changes that were not as drastic (Fig. A.4). This discrepancy led to a unique case where the increased computational cost of `computeInteractions` in the Dynamic VL Merge branch effectively offset the additional remainder traversal time in the Fast-Particle-Buffer branch. Consequently, for `vcl_c06`, the Fast-Particle-Buffer branch gained a net advantage by reducing the cost of neighbor list rebuilds.

However, while this might initially appear as a speedup, it is important to put the results into context. The lowest overall runtime across all traversal methods was not achieved by the Fast-Particle-Buffer branch but rather by the Dynamic VL Merge branch in other traversal configurations. This suggests that, although the Fast-Particle-Buffer branch outperformed the Dynamic VL Merge branch specifically for `vcl_c06`, it was not the most efficient approach across all tested traversal methods. Therefore, while the buffer mechanism provided a performance gain in this isolated case, it did not lead to an overall improvement in execution time across the entire experiment.

Iteration Tests

For the iteration tests, a base frequency of 100 was maintained for the first two scenarios and 1000 for the third scenario. The number of iterations varied between 500 and 30,000, with a step size of 500 iterations. The goal of these experiments was to observe the behavior of fast particles at different stages of the simulation by increasing the number of iterations. This approach aimed to identify any correlations between simulation progression and the performance of the particle buffer implementation, as well as to expand the range of experiments conducted.

Initially, as in the frequency tests, the first three scenarios were executed with tuning enabled. However, this approach did not give us a clear enough understanding of the performance of the fast particle branch. The results of these initial tests are presented in A.5.

Individual Test Results

Falling Drop Scenario: For the `vlc_c08` traversal, no performance improvement was observed. Across all iteration counts, the fast particle buffer consistently performed worse than the Dynamic VL Merge branch.

The same trend was observed for `lc_c08`. The remainder traversal phase required excessive time, negating any potential gains from avoiding neighbor list rebuilds. Thus, the time saved from fewer rebuilds was insufficient to offset the overhead introduced by the remainder traversal.

On the other hand, the `vcl_c06` traversal displayed similar performance in both branches, with the fast particle buffer occasionally running 1-2 seconds faster. This behavior follows the same reasoning as in 5.3.1, that being the pronounced increase in runtime for `computeInteractions`. Specifically, for `vcl_c06`, `computeInteractions` required nearly four times more runtime than in `vcl_c08`. While the fast particle buffer also experienced a runtime increase, it was less pronounced than in the dynamic VL merge, explaining why the overall performance remained similar. While these results suggest some efficiency gains in this particular experiment, they do not indicate a significant overall performance improvement. The corresponding graphs can be found in A.6.

Exploding Liquid Scenario: The results for this scenario followed a pattern similar to Falling Drop. However, anomalies were observed in the behavior of `vlc_c08` and `vcl_c06`, as indicated in the A.7. Missing plotting points in the graphs suggest that the experiment failed for certain iteration counts in both branches. Although there were spikes in `vlc_c08` (A.7a), where the fast particle buffer branch appeared to outperform the Dynamic VL Merge, the anomalies in the experiments make these results unreliable.

The `lc_c08` traversal exhibited stable behavior with no failed runs; however, the results were fairly routine and did not provide significant new insights into the topic.

Constant Velocity Cube Scenario: In this scenario, no performance improvement was observed for the fast particle buffer. Instead, the performance difference between the two branches was more pronounced compared to Falling Drop. For `lc_c08`, the fast particle buffer was up to 16 times slower; for `vcl_c06`, it was 4 times slower; and for `vlp_c08`, it was around 3 times slower. This poor performance is due to the large number of particles accumulating in the buffer. For instance, in the experiment with 30,000 iterations, more than half of the total particles—around 28,000 out of 50,000—were in the buffer. This high number of particles in the buffer significantly increased the time required for remainder traversal. This is illustrated in Fig. 5.9, where the runtime scale for the remainder traversal is $e8$, while `computeInteractions` operates on the $e7$ scale, highlighting a clear difference in time consumption.

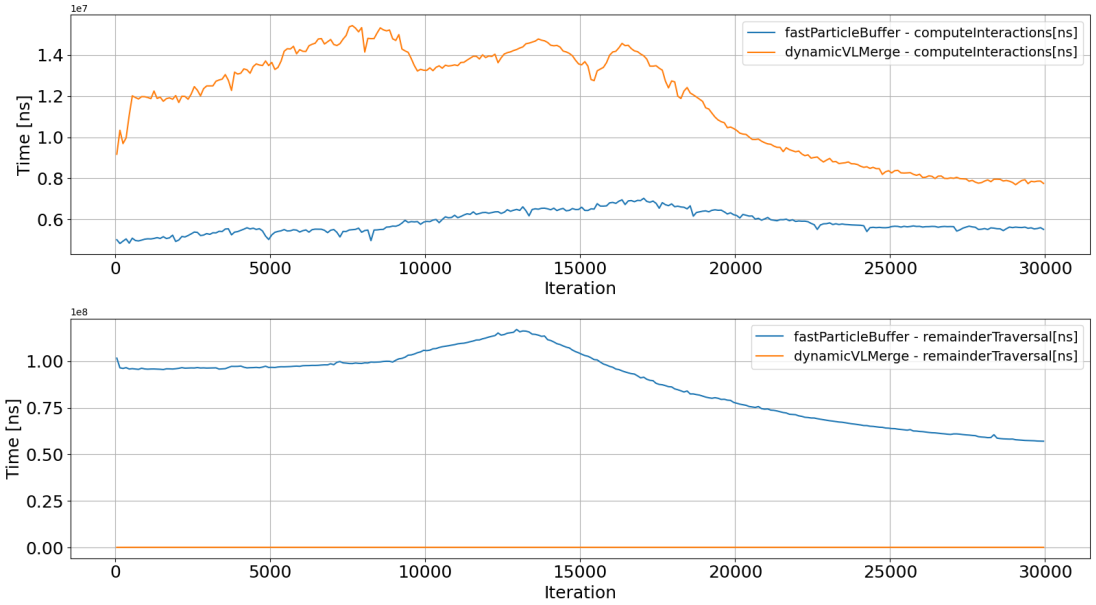


Figure 5.9.: Compute Interactions vs Remainder Traversal in Constant Velocity Cube `vlp_c08`.

5.3.2. Percentage Experiments

The primary challenge with the fast particle buffer implementation is that the time required for `remainderTraversal` significantly outweighs the time saved from reducing the frequency of neighbor list rebuilds. To address this issue, a percentage-based threshold was introduced, limiting the number of particles that can accumulate in

the buffer to a fixed percentage of the total number of particles in the container. The objective was to reduce the time spent in remainder traversal, while still benefiting from fewer neighbor list rebuilds.

Given that `rebuildFrequency` is directly tied to the particle buffer's efficiency, these experiments focus on adjusting the rebuild frequency rather than the number of iterations. In earlier experiments, the iteration count was varied to see how performance evolved over longer runtimes. However, since rebuild frequency determines when particles move in and out of the buffer, varying it gives a more precise understanding of the buffer's efficiency, making it the primary focus of this set of experiments.

Initially, experiments were conducted across frequencies ranging from 10 to 1900, based on the previous findings that smaller frequencies performed better. Threshold values of 10%, 15%, 20%, and 30% were tested, meaning that if the buffer contained the specified percentage of the container's particles, a rebuild was triggered.

For frequencies above 100, the performance gap between the Fast-Particle-Buffer branch and Dynamic-VL-Merge increased, with the latter performing better. However, for frequencies below 100, certain thresholds—particularly the 10% threshold—showed slight performance improvements of a few seconds. The complete set of graphs supporting these findings is available in the provided `repository`.

To further analyze these trends, the Falling Drop scenario was selected as it displayed consistent behavior across different configurations. The traversal `vlc_c08` was chosen as it was one of the better-performing traversals in previous experiments.

Given that performance improvements remained minor—mostly 1 to 2 seconds—the experiments were repeated four times per configuration to improve accuracy. Moreover, smaller threshold values were introduced: 5%, 1%, 0.5%, and 0.1%, along with ultra-small thresholds of 0.01%, 0.001%, and 0%, to analyze how performance evolves as the Fast-Particle-Buffer branch converges toward the Dynamic-VL-Merge branch. The primary goal of this extension was to confirm that any observed performance gains were genuinely due to the buffer mechanism.

The experimental data is presented as a box plot in Fig.5.10, where red dots represent the mean runtime for each experiment. Only frequencies from 20 to 300 are shown, as including all frequencies would make the graph appear too cluttered. However, these selected frequencies cover the main bulk of the data and are the most representative ones. The rest of the data can be found in the provided `repository`. The key findings are as follows:

- On average, thresholds between 0.1% and 1% provided the best performance.
- Extremely small thresholds (0.01%, 0.001%, 0%) sometimes performed worse than the Dynamic-VL-Merge branch. This can be explained by the overhead introduced

by continuously scanning the container to check whether the threshold condition was met.

- Although some configurations achieved minor performance improvements, these gains were marginal and nearly insignificant compared to the overall runtime.

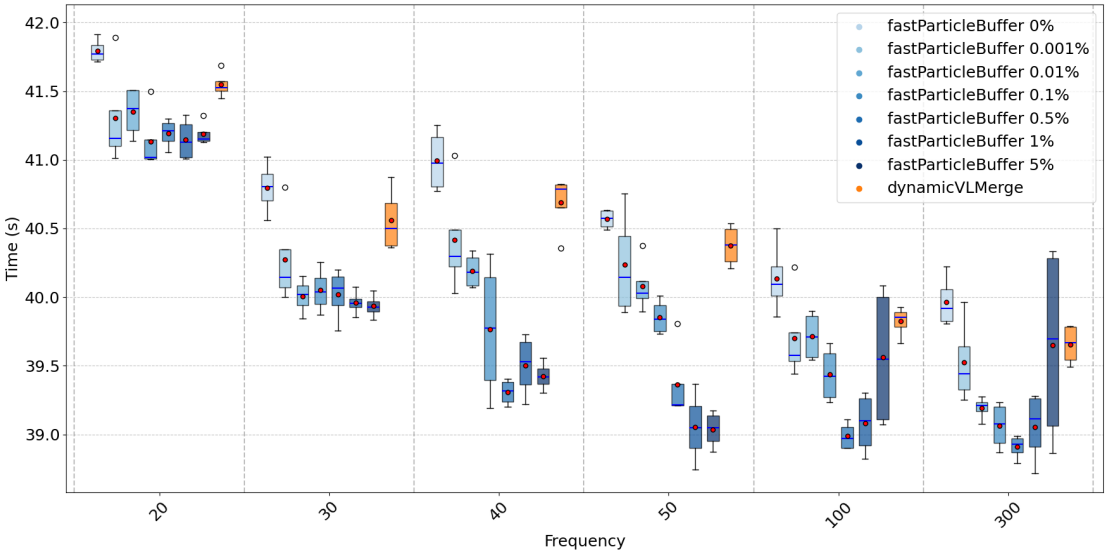


Figure 5.10.: Percentage Experiments in Falling Drop `vlc_c08`

5.3.3. Equilibration

Based on the earlier experiments, the observed performance improvement was minimal, leading to the question of whether the selected test cases were limiting the analysis. Instead of focusing on short simulations lasting around a couple of minutes or less, a more dynamic scenario with a significantly larger number of particles was needed. For this reason, the equilibration scenario was chosen.

The hypothesis was that, due to its highly dynamic nature and large particle count (4 million), this scenario would be better suited to reveal potential performance benefits of the fast particle buffer.

From prior frequency experiments, it was observed that the particle buffer performed better at smaller frequencies. Therefore, this experiment focused on frequencies ranging from 10 to 50 with a step size of 10. Additionally, as established earlier, smaller thresholds tended to yield better results. Considering the large particle count in the equilibration scenario, initial tests were conducted with thresholds of 1% and 5% of the container's particles.

It is important to note that 1% and 5% of 4 million correspond to 40,000 and 200,000 particles, respectively, which exceed the total particle count of the other scenarios.

Results for `vlc_c08`: Throughout this scenario the average runtime for the Dynamic VL Merge branch consisted of 9 hours and 59 minutes (Fig.5.11). When the threshold was set to 5%, the fast particle buffer demonstrated a performance improvement of 4.2% (equivalent to saving 25 minutes) at frequency 30. At frequency 20, the improvement was 2.5% (saving 15 minutes). For frequency 10, the runtime was marginally worse by 1%. However, for larger frequencies, such as 40 and 50, there was a significant slowdown of 44% and 111%, respectively.

For the 1% threshold, fluctuations were less pronounced compared to the 5% threshold. The runtime for all frequencies was closer to that of the Dynamic VL Merge branch. A performance improvement of 2.2% (equivalent to 13 minutes) was observed at frequency 20, and 3% (18 minutes) at frequency 30. For frequencies 10 and 50, the runtime was nearly identical to the parent branch, differing by only ±1 minute. At frequency 40, there was a minor slowdown of 1% (6 minutes), which is relatively insignificant.

The fastest runtime in this experiment was achieved by the Fast-Particle-Buffer with a 5% threshold at frequency 30, completing in 9 hours and 33 minutes. The second-fastest was the Fast-Particle-Buffer with a 1% threshold, which took 6 minutes longer.

For the Dynamic VL Merge branch, the shortest runtime was 9 hours and 47 minutes, recorded at frequency 40. Comparing the best results from both approaches, the Fast-Particle-Buffer provided a speedup of 14 minutes, equivalent to a 2.74% improvement over the fastest Dynamic VL Merge run.

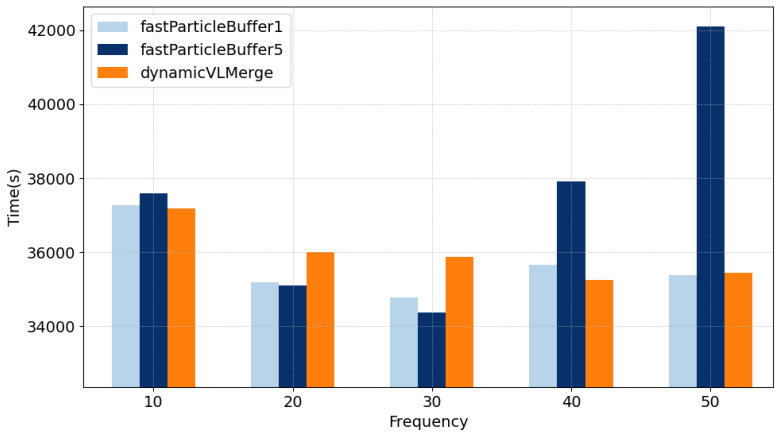


Figure 5.11.: Frequency vs Time for Equilibration vlc_c08

Results for `lc_c08`: Using LinkedCellsReferences with traversal `lc_c08` (Fig.5.12), the average runtime for the Dynamic VL Merge branch was 14 hours and 27 minutes, approximately 4 hours and 30 minutes longer than the `vlc_c08` configuration.

At a 5% threshold, a 1.94% performance improvement (16 minutes) was observed at frequency 30, with a smaller 0.9% improvement (7 minutes) at frequency 20. However, for all other frequencies, no improvement was recorded. In contrast, at frequencies 40 and 50, the runtime of the Fast-Particle-Buffer branch increased significantly. At frequency 50, the performance degraded to the point where the simulation timed out.

For the 1% threshold, a more notable improvement was observed at frequency 30, with a 2.3% improvement (20 minutes). For the remaining frequencies, the runtime was very similar to the parent branch, fluctuating between -0.96% and 0.29% of the respective Dynamic VL Merge runtimes.

For this container-traversal combination, the fastest experiment overall was achieved using the Fast-Particle-Buffer with a 1% threshold at frequency 30, resulting in a runtime of 14 hours and 4 minutes. This corresponds to a 12-minute improvement (1.43%) compared to the Dynamic VL Merge branch at frequency 20, which recorded the fastest runtime for that branch.

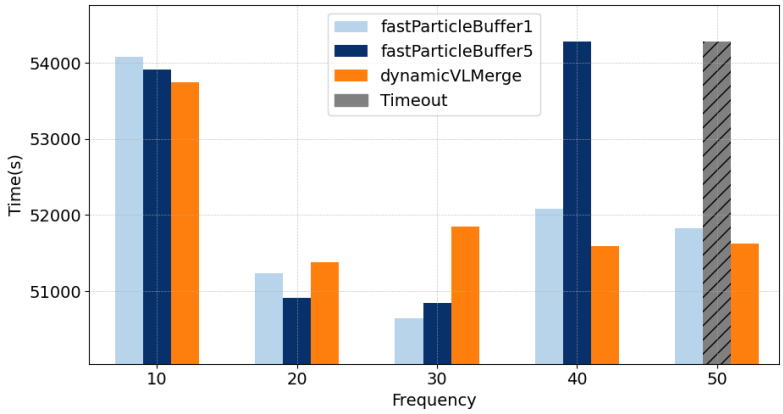


Figure 5.12.: Frequency vs Time for Equilibration lc_c08

Equilibration with Increased Temperature

To further amplify the number of fast particles and make the experiment more dynamic, the temperature of the system in the equilibration scenario was increased. This adjustment resulted in molecules with higher kinetic energy, thereby increasing their speed. The focus of this experiment remained on the same frequencies as in previous tests, with the addition of a new frequency, 5, to further investigate performance trends at lower rebuild intervals. The YAML file used for this setup can be found in the appendix [**link YAML file in the appendix**]. For this experiment, the `vlc_c08` traversal configuration was used, as the `lc_c08` configuration proved too inefficient, resulting in timeouts at the maximum allowed runtime of 16 hours.

In this configuration, the fast particle buffer with a threshold of 1% achieved an 8.6% performance improvement at frequency 10. This translates to a time saving of 60 minutes from a total runtime of 11 hours and 42 minutes. Similarly, the fast particle buffer with a threshold of 5% exhibited a 7.8% performance improvement at the same frequency, resulting in a 55-minute reduction from the same total runtime.

For the remaining frequencies, neither version of the fast particle buffer outperformed the Dynamic VL Merge branch. However, the overall fastest experiment was the Fast-Particle-Buffer with a 1% threshold, achieving a 41-minute improvement compared to the fastest version of the Dynamic VL Merge branch at frequency 40. This corresponds to a 6% performance boost, the largest improvement observed in the experiments so far. The results are summarized in Figure 5.13, which visualizes the performance trends across different configurations.

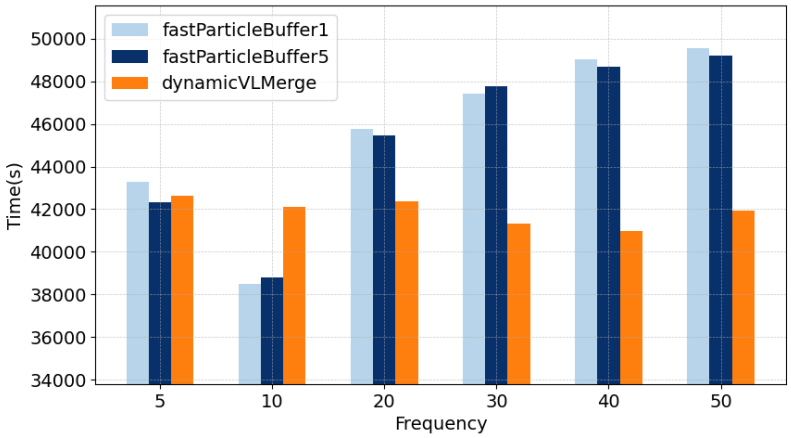


Figure 5.13.: Frequency vs Time for Equilibration with Increased Temperature

first of all there are like 300k halo aprticles, so there is already time being spent on remainder traversals bacuse of them, hence the difference in not that much as in the other experiments where there were no halo particles. also for 1 and 5 percent for freq like 30 the 1 percent doesnt even get fullfileed. so why is there a performance difference between 1% and 5% threshold??????????

for the conclusion: if there are halo nice use buffer, if no use it only for non dynamic spots.

u can either play with the threshold, or choose an appropriate frequency which is big enough, but a threshld is still recommended to cover those times where there is a boom of fast particles in the buffer suddendly. so there should be a sweet spot. at aorund 1% of particles

5.4. Checkpoint Experiments

Another approach to further investigate the benefits of the particle buffer was to apply it selectively during the least dynamic phase of the simulation. Since remainder traversal is one of the most computationally expensive aspects of the buffer mechanism, the idea was to store particles in the buffer only when the number of fast particles was relatively low.

During the most dynamic phase of the simulation, a large number of fast particles are generated in each iteration, rapidly increasing the number of particles stored in the buffer. Even with a threshold in place, the high number of buffered particles causes remainder traversal times to grow significantly, reducing the overall benefit of avoiding frequent neighbor list rebuilds. However, in the less dynamic phase, where fewer fast particles appear, the remainder traversal overhead should be considerably lower. The hypothesis was that enabling the buffer only during these less dynamic phases could still save neighbor list rebuild time while keeping remainder traversal costs manageable.

To test this hypothesis, the Falling Drop scenario was selected with a 30,000-iteration simulation. Observations showed that after iteration 15,000, the simulation became less dynamic. Based on this, the simulation was divided into two phases:

- First 15,000 iterations: The simulation ran without using the buffer.
- After iteration 15,000: The buffer was enabled with a 1% threshold.

To enable the transition between the two phases, the last VTK output file from the first phase was used as a checkpoint to initialize the second phase. A VTK output file is a structured XML-based format that stores simulation data, including particle positions, velocities, forces and more.

Figure 5.14 illustrates the results of this experiment. To assess the potential improvement, the simulation was Additionally run with a 1% threshold for the entire 30,000 iterations, allowing for comparison with both the two-phase approach and the Dynamic VL Merge branch.

As shown in the figure, the two-phase approach outperformed the standard 1% threshold Fast-Particle-Buffer, and ultimately, it also achieved a slightly faster runtime than the Dynamic VL Merge branch, although the difference was approximately 1 to 2 seconds. While this improvement may seem small, it suggests that for larger simulations running for several hours, the benefit could become more significant. The effect would likely be more pronounced in longer-running experiments, where reducing remainder traversal costs over extended periods could deliver greater performance gains.

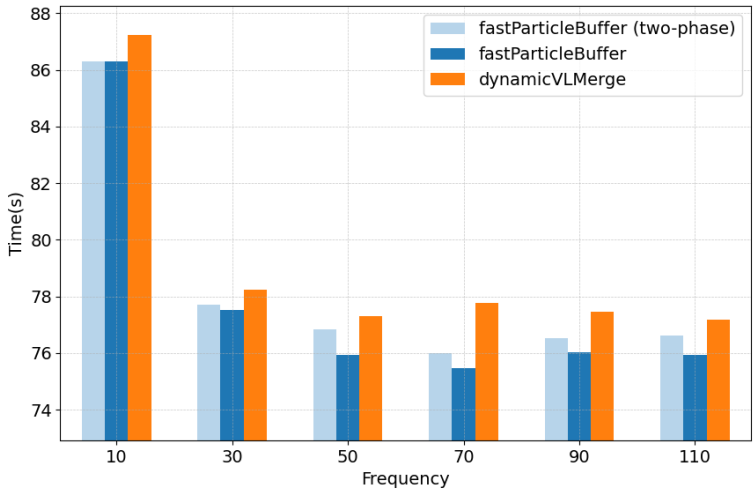


Figure 5.14.: Frequency vs Time for Falling Drop vlc_c08 in Chekcpoint

6. Future Work

While the Fast-Particle-Buffer showed potential in larger simulations, it did not provide a significant performance improvement in smaller ones. Further research can focus on different aspects to enhance its efficiency.

One possible direction is studying how different hardware setups affect the buffer's performance. Running tests on various processors and memory architectures could help understand whether certain hardware configurations benefit more from the buffer approach.

Another improvement could involve combining the Fast-Particle-Buffer with Gall's adaptive neighbor list rebuild strategy. Since both methods aim to reduce the cost of rebuilding neighbor lists, merging them could lead to better overall performance.

Additionally, an alternative approach to identifying fast particles could be explored. Instead of classifying each particle individually, a group-based approach could be tested. The current definition of fast particles is based on how much a particle moves relative to its previous position, determining whether it has potentially left or entered another particle's search region. However, this classification does not consider relative motion within a larger system of particles.

If a group of particles moves as a unit, their positions relative to one another remain unchanged, meaning that no particle has actually moved into or out of another's search region. In this case, classifying them as fast particles and adding them to the buffer would be unnecessary because their relative interactions have not changed. By incorporating a group motion detection method, the buffer mechanism could avoid unnecessary classifications of fast particles, reducing remainder traversal costs.

Testing the buffer in other larger and longer-running simulations could also be beneficial. While small simulations showed only minor improvements, a 6% performance boost was observed in one of the longer simulations. This suggests that the effectiveness of the buffer may become more apparent in extended simulations. Running additional tests on different long-running scenarios could help determine whether if even greater performance gains can be achieved.

Finally, optimizing how buffer traversal is handled could further reduce remainder traversal time. Since remainder traversal is currently a major performance bottleneck, improving its efficiency could reveal the full advantages of the buffer approach.

7. Conclusion

This thesis investigated the Fast-Particle-Buffer as a method to reduce the computational cost of frequent neighbor list rebuilds in AutoPas. The approach aimed to improve simulation efficiency by temporarily storing fast particles instead of immediately rebuilding neighbor lists.

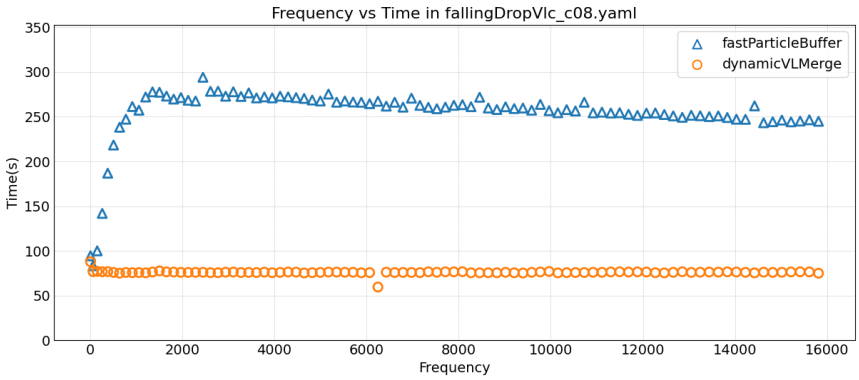
Through a series of experiments, the performance of the buffer was evaluated under different scenarios, thresholds, and frequencies. The results showed that while the buffer did not significantly improve performance in smaller simulations, it provided measurable benefits in longer-running simulations, with a maximum observed speedup of 6%.

Further analysis revealed that remainder traversal time remained a key limiting factor, often negating the benefits of reduced neighbor list rebuilds. Additionally, the effectiveness of the buffer depended on the characteristics of the simulation.

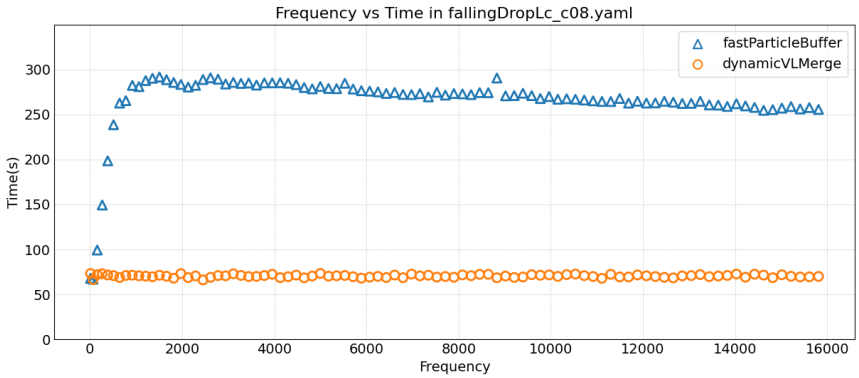
Future work could explore hardware-dependent optimizations, integrating adaptive rebuild criteria, and exploring group-based fast particle classification. Additionally, optimizing buffer traversal could further reduce remainder traversal overhead, making the approach more performative.

Overall, while the Fast-Particle-Buffer presents a good strategy for improving simulation performance, its benefits are highly scenario-dependent. With further refinements, it could become a useful optimization technique in AutoPas.

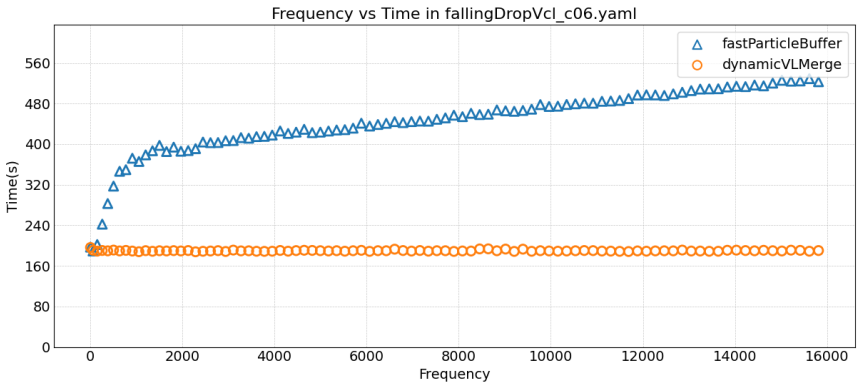
A. Appendix



(a) Falling Drop vlc_c08

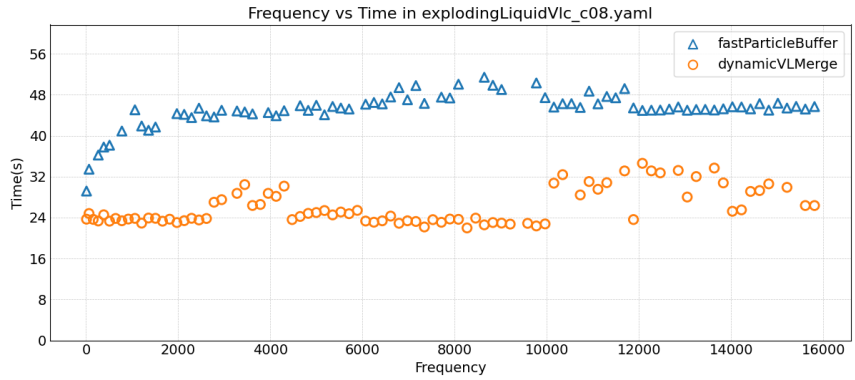


(b) Falling Drop lc_c08

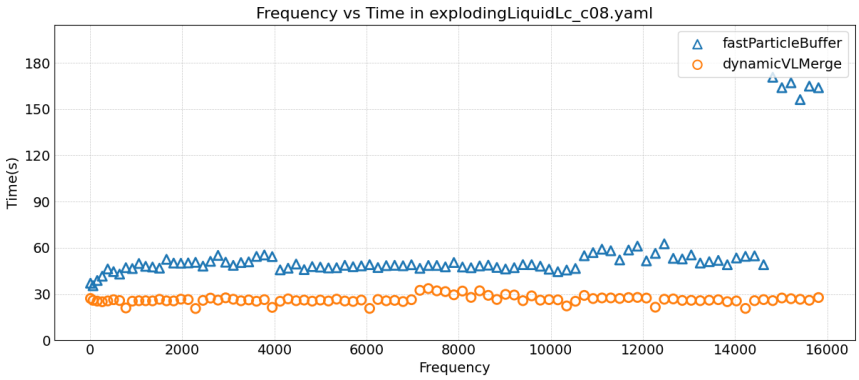


(c) Falling Drop vcl_c06

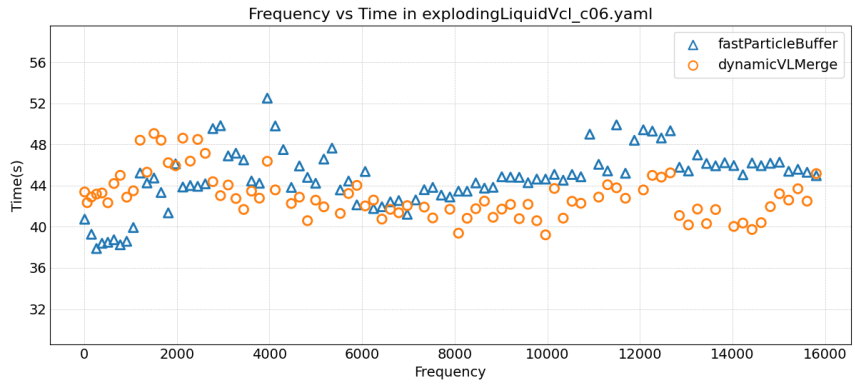
Figure A.1.: Comparison of Frequency vs Time for Falling Drop Experiments



(a) Exploding Liquid vlc_c08

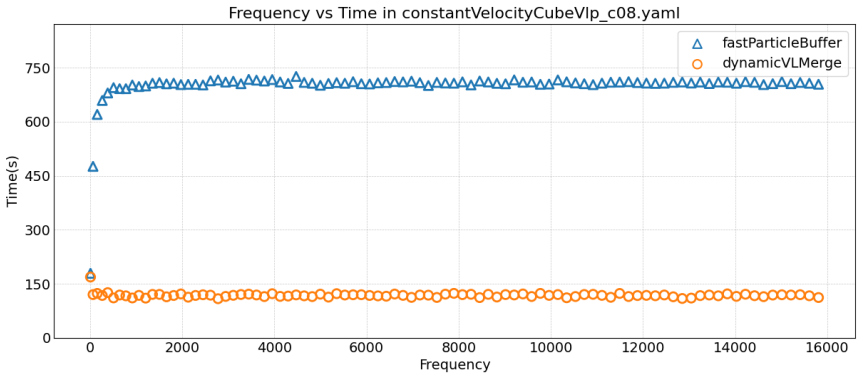


(b) Exploding Liquid lc_c08

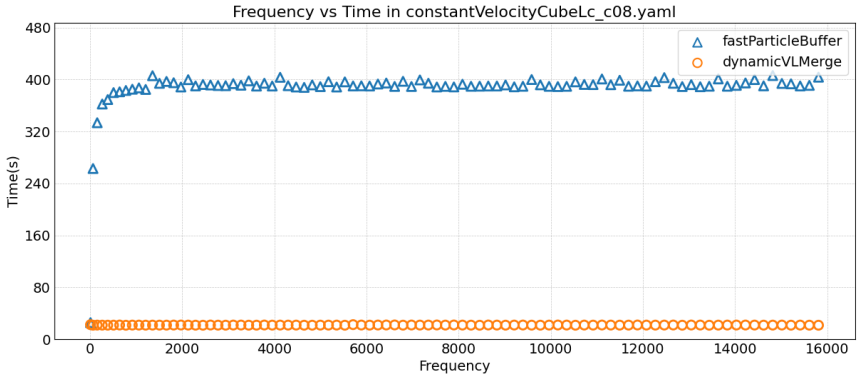


(c) Exploding Liquid vcl_c06

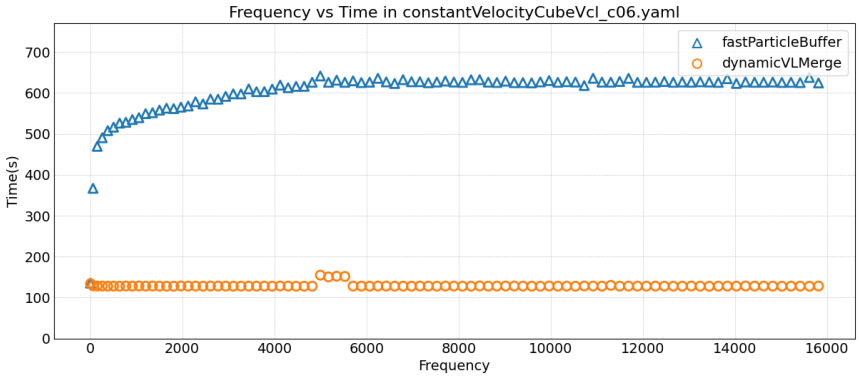
Figure A.2.: Comparison of Frequency vs Time for Exploding Liquid Experiments



(a) Constant Velocity Cube vlp_c08

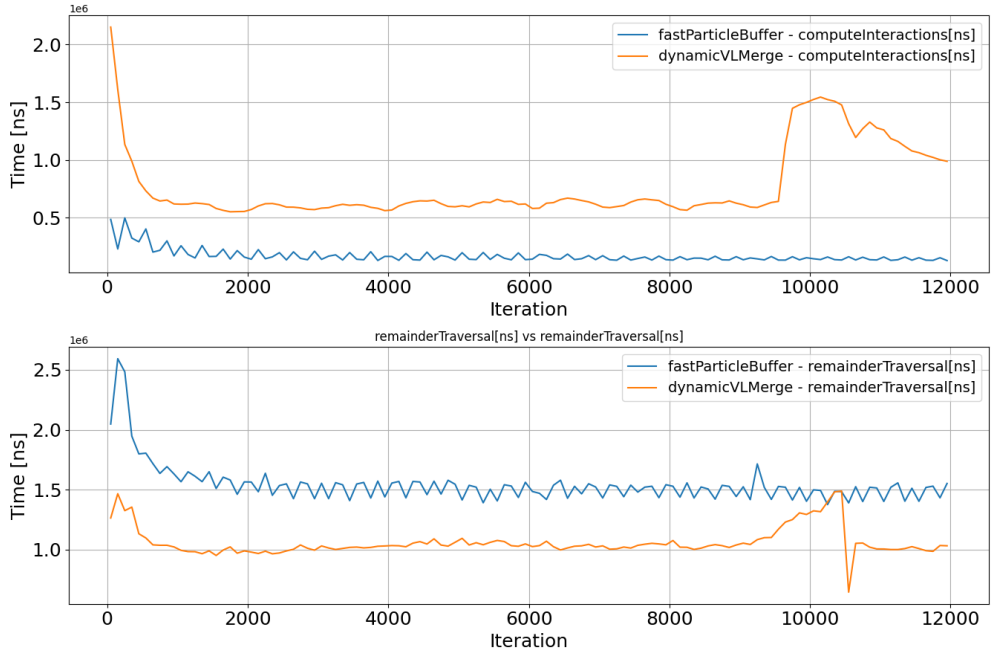


(b) Constant Velocity Cube lc_c08

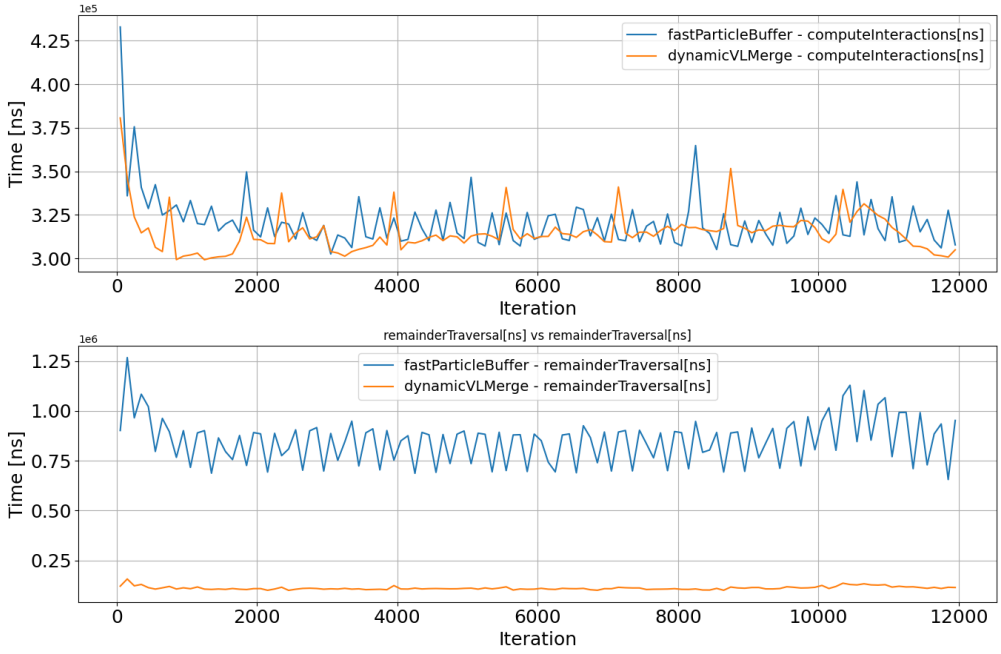


(c) Constant Velocity Cube vcl_c06

Figure A.3.: Comparison of Frequency vs Time for Constant Velocity Cube Experiments

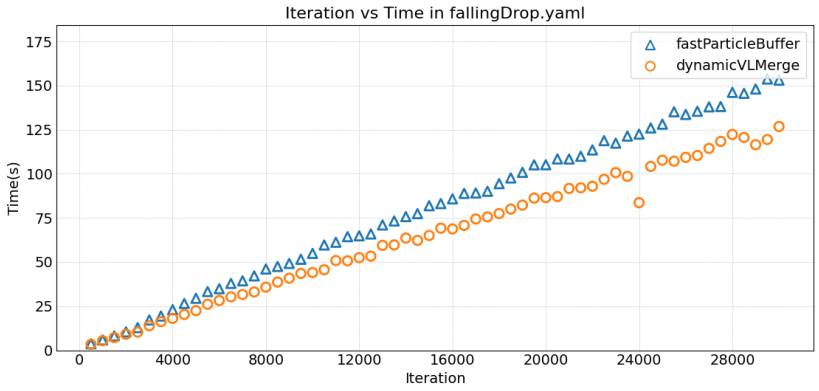


(a) Exploding Liquid Vcl_c06

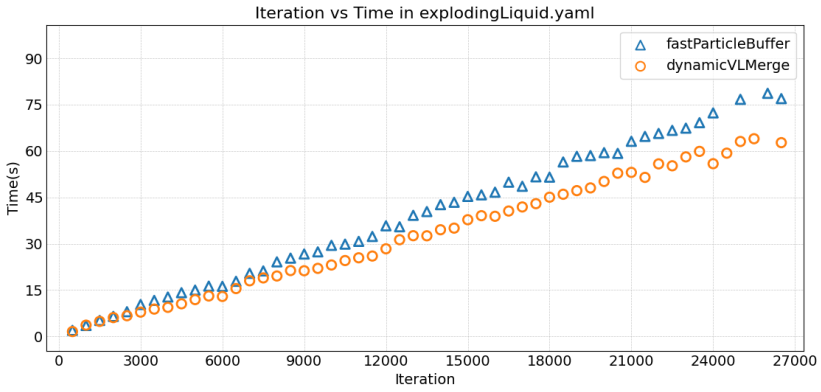


(b) Exploding Liquid Vlc_c08

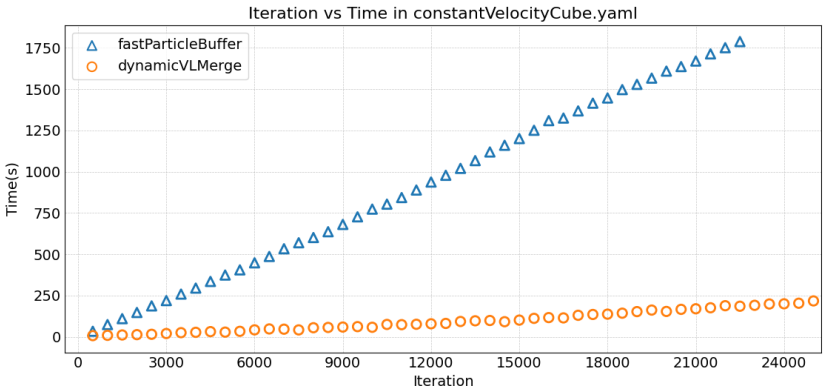
Figure A.4.: Comparison of Compute Interactions and Remainder Traversal for Exploding Liquid Experiments



(a) Falling Drop

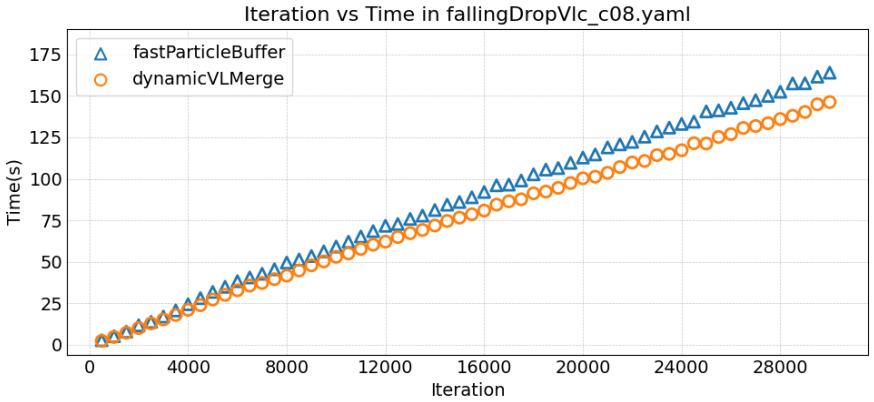


(b) Exploding Liquid

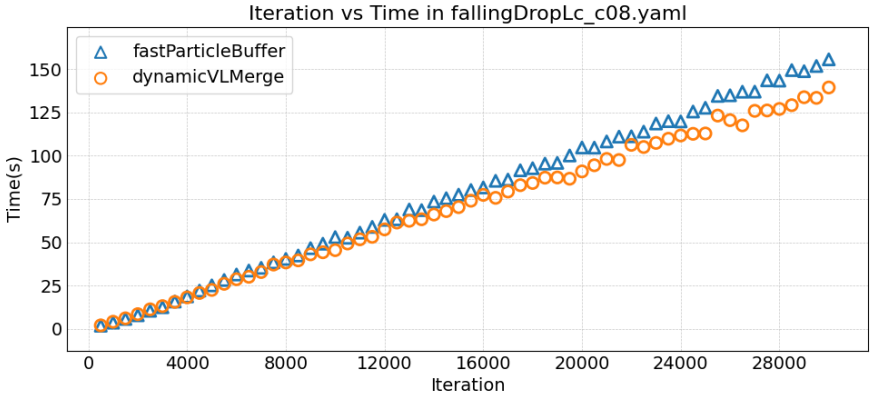


(c) Constant Velocity Cube

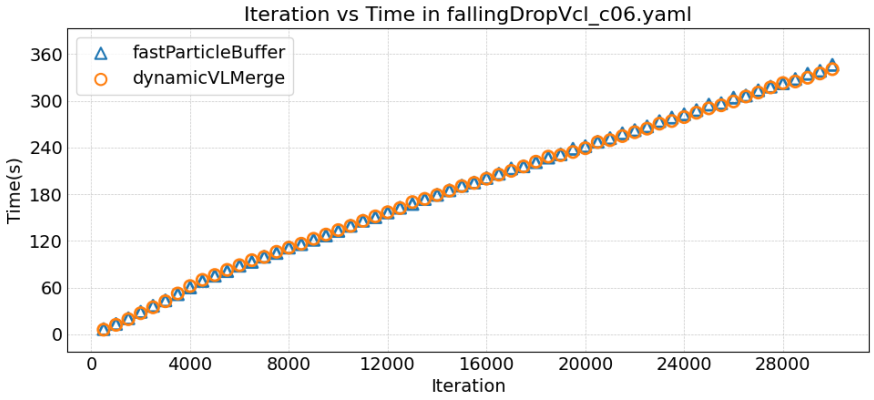
Figure A.5.: Comparison of Frequency vs Iterations with Tuning



(a) Falling Drop vlc_c08

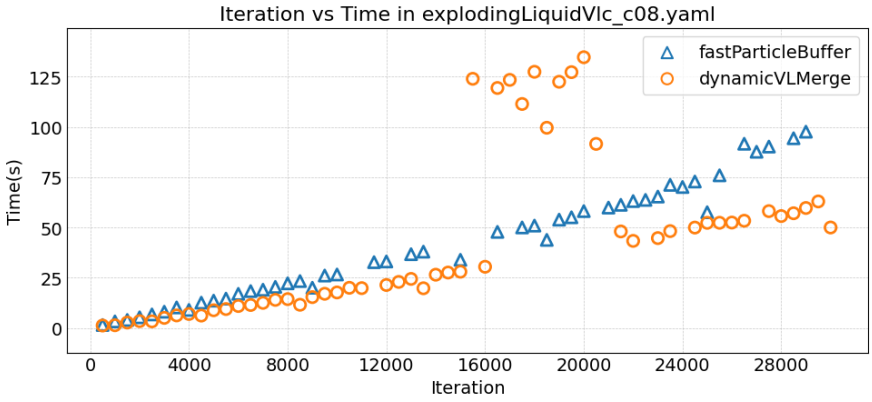


(b) Falling Drop lc_c08

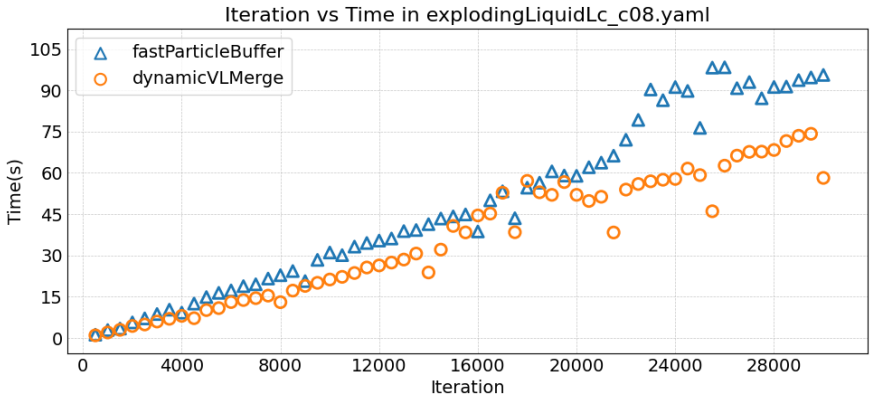


(c) Falling Drop vcl_c06

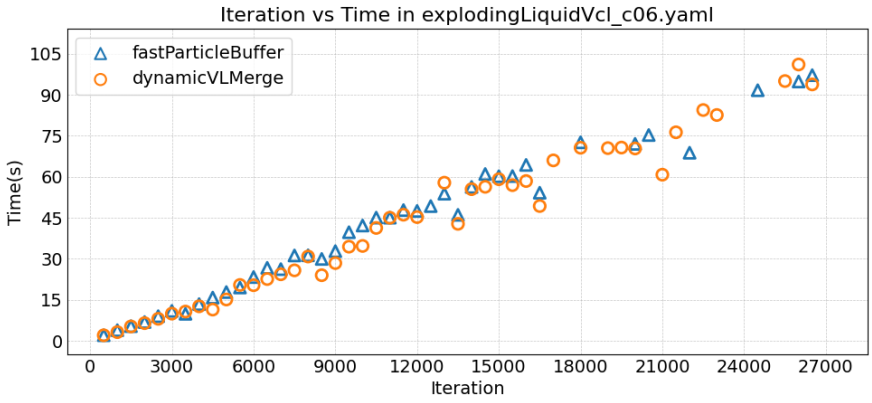
Figure A.6.: Comparison of Iteration vs Time for Falling Drop Experiments



(a) Exploding Liquid vlc_c08

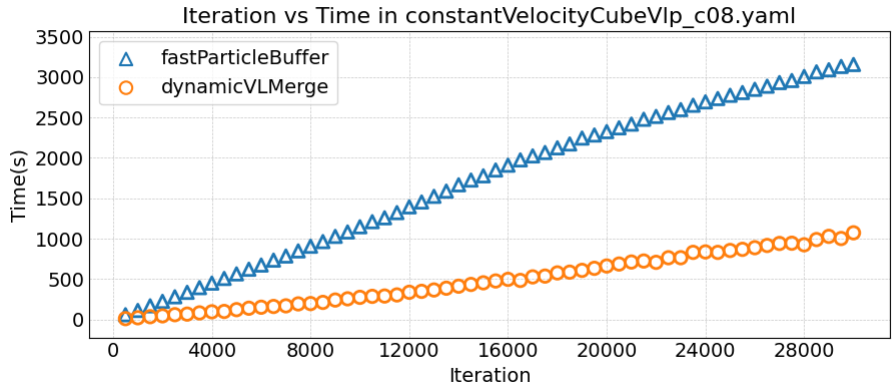


(b) Exploding Liquid lc_c08

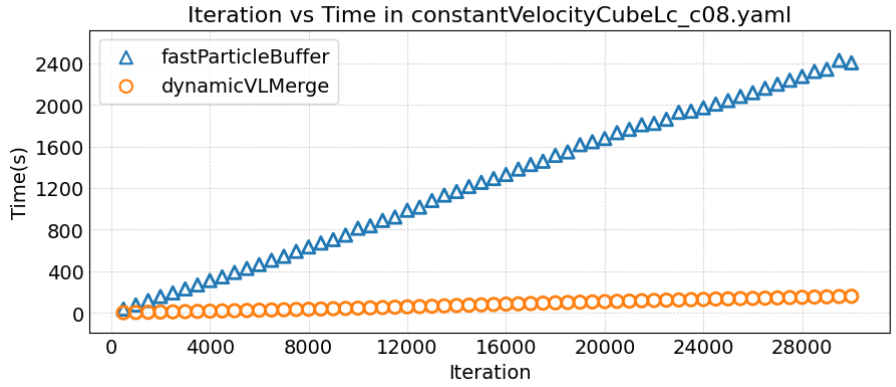


(c) Exploding Liquid vcl_c06

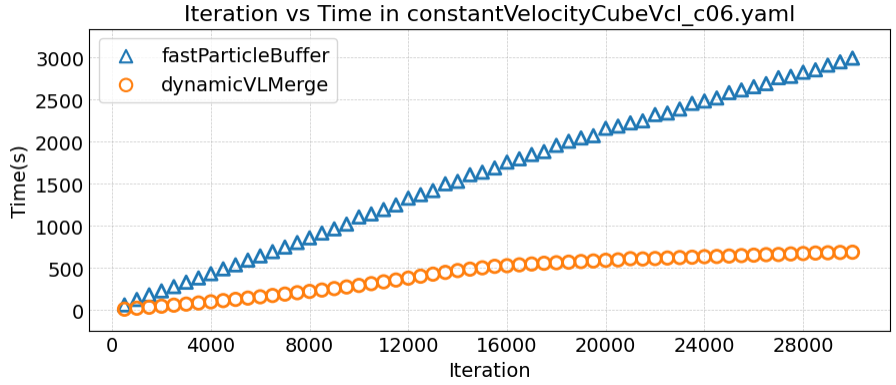
Figure A.7.: Comparison of Iteration vs Time for Exploding Liquid Experiments



(a) Constant Velocity Cube vlp_c08



(b) Constant Velocity Cube lc_c08



(c) Constant Velocity Cube vcl_c06

Figure A.8.: Comparison of Iteration vs Time for Constant Velocity Cube Experiments

List of Figures

List of Tables

Bibliography

[1] M. J. Abraham, T. Murtola, R. Schulz, S. Páll, J. C. Smith, B. Hess, and E. Lindahl. "GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers." In: SoftwareX 1 (2015), pp. 19–25.

[2] H. M. Aktulga, J. C. Fogarty, S. A. Pandit, and A. Y. Grama. "Parallel reactive molecular dynamics: Numerical methods and algorithmic techniques." In: parallel computing 38.4-5 (2012), pp. 245–259.

[3] CoolMUC-2 Documentation. Leibniz Supercomputing Centre (LRZ). 2024. URL: https://doku.lrz.de/coolmuc-2-11484376.html.

[4] CoolMUC-4 Documentation. Leibniz Supercomputing Centre (LRZ). 2024. URL: https://doku.lrz.de/coolmuc-4-1082337877.html.

[5] A. Developers. AutoPas Documentation. https://autopas.github.io/doxygen_documentation/git-master/. 2024.

[6] S. Eder, G. Vorlaufer, S. Ilincic, and G. Betz. "Simulation of Wear Processes Using Molecular Dynamics (MD) Simulations." In: Oct. 2007. DOI: 10.13140/2.1.4149.5680.

[7] S. C. Frautschi. The mechanical universe: Mechanics and heat. Cambridge University Press, 1986.

[8] L. Gall. "An Exploration of Different Approaches for Implementing Verlet Lists in AutoPas." In: (2023).

[9] F. A. Gratl, S. Seckler, H.-J. Bungartz, and P. Neumann. "N ways to simulate short-range particle systems: Automated algorithm selection with the node-level library AutoPas." In: Computer Physics Communications 273 (2022), p. 108262.

[10] F. A. Gratl, S. Seckler, N. Tchipev, H.-J. Bungartz, and P. Neumann. "Autopas: Auto-tuning for particle simulations." In: 2019 IEEE International Parallel and Distributed Processin IEEE. 2019, pp. 748–757.

[11] J. E. Jones. "On the determination of molecular fields.—II. From the equation of state of a gas." In: Proceedings of the Royal Society of London. Series A, Containing Papers of a Ma 106.738 (1924), pp. 463–477.

[12] M. G. S. Knapek and G. Zumbusch. Numerical Simulation in Molecular Dynamics, Numerics, Algor 2007.

[13] A. Kukol et al. Molecular modeling of proteins. Vol. 443. Springer, 2008, pp. 3–4.

[14] S. J. Newcome, F. A. Gratl, P. Neumann, and H.-J. Bungartz. "Towards auto-tuning Multi-Site Molecular Dynamics simulations with AutoPas." In: Journal of Computational and Applie 433 (2023), p. 115278.

[15] C. Niethammer, S. Becker, M. Bernreuther, M. Buchholz, W. Eckhardt, A. Heinecke, S. Werth, H.-J. Bungartz, C. W. Glass, H. Hasse, et al. "ls1 mardyn: The massively parallel molecular dynamics code for large systems." In: Journal of chemical theory and comp 10.10 (2014), pp. 4455–4464.

[16] M. Praus. "Energy-Efficient Molecular Dynamics Simulations: Implementing Hardware-Agnostic Energy Measurement and Evaluating Runtime-Energy Tradeoffs." en. MA thesis. Technical University of Munich, Oct. 2024.

[17] S. Seckler, F. Gratl, M. Heinen, J. Vrabec, H.-J. Bungartz, and P. Neumann. "AutoPas in ls1 mardyn: Massively parallel particle simulations with node-level auto-tuning." In: Journal of Computational Science 50 (2021), p. 101296.

[18] A. P. Thompson, H. M. Aktulga, R. Berger, D. S. Bolintineanu, W. M. Brown, P. S. Crozier, P. J. In't Veld, A. Kohlmeyer, S. G. Moore, T. D. Nguyen, et al. "LAMMPS-a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales." In: Computer Physics Communications 271 (2022), p. 108171.

[19] X. Wang, S. Ramírez-Hinestrosa, J. Dobnikar, and D. Frenkel. "The Lennard-Jones potential: when (not) to use it." In: Physical Chemistry Chemical Physics 22.19 (2020), pp. 10624–10633.

[20] Z. Yao, J.-S. Wang, G.-R. Liu, and M. Cheng. "Improved neighbor list algorithm in molecular simulations using cell decomposition and data sorting method." In: Computer physics communications 161.1-2 (2004), pp. 27–35.